Building Skills in Object-Oriented Design

Release 3.1

Steven F. Lott

December 05, 2015

Step-by-Step Construction of A Complete Application

Legal Notice

This work is licensed under a Creative Commons License. You are free to copy, distribute, display, and perform the work under the following conditions:

- **Attribution**. You must give the original author, Steven F. Lott, credit.
- **Noncommercial**. You may not use this work for commercial purposes.
- **No Derivative Works**. You may not alter, transform, or build upon this work.

For any reuse or distribution, you must make clear to others the license terms of this work.

Part I

Front Matter

CHAPTER

ONE

PREFACE

1.1 Why Read This Book?

One way to master OO design is do design and implementation of a large number of classes. This book guides you through exercises to design dozens of classes that must work together to create a large – and reasonably complete – application. We'll discuss the design in some detail to reveal an approach to object-oriented design.

While it is also important to read examples of good design, a finished product doesn't reveal the designer's decision-making process. Our goal in this book is to help programmer understand the design process that leads to a final product.

Everything is Design It's important to note that all software development involves considerable skill in design. In some circles, there is an attempt to distinguish between architects (or designers) and coders. The idea is that someone can be able to work with a language – coding – but not quite ready to do design work. The coders can be given a detailed "specification" from which they can write code.

This distinction between designers and coders doesn't really exist. It's unhelpful to try and make this distinction. All programming involves design at some level.

If we attempt to write a design specification so detailed that someone else can transform it into code without them having to make any design decisions, the specification is isomorphic to code, but written in some non-technical language like English. The design has all the details of code. An automated translation could produce code from this magically complete with fewer errors than a person.

If we write specifications at somewhat more abstract level, we're demanding that the "coder" makes some design decisions to move from the abstraction to concrete code. The more abstract the specification, the more design work must be done. The amount of design work is non-zero.

The point of this book is to build skills in object-oriented design prior to a project with a fixed budget and a looming deadline.

1.2 What You'll Do

This book provides a sequence of interesting and moderately complex exercises in OO design. The exercises are not hypothetical, but must lead directly to working programs.

The applications we will build are a step above trivial, and will require some careful thought as part of creating a workable design. Further, because the applications are largely recreational in nature, they are interesting and engaging. This book allows the reader to explore the processes and artifacts of OO design before project deadlines make good design seem impossible.

The first part will be quite detailed. The third part will summarize the designs at a higher level of abstraction.

1.3 Audience

Our primary audience includes programmers who are new to OO programming.

Knowledge of the Python language is essential. Since the focus is on OO techniques, some exposure to class definitions is important. We will provide exercises that have four key features:

- complex enough to require careful design work,
- fun enough to be engaging,
- easy enough that results are available immediately, and
- can be built in simple stages.

We'll provide a few additional details on language features. We'll mark these as "Tips". For more advanced students, these tips will be review material. We will not provide a thorough background in any programming language. The student is expected to know the basics of the language and tools.

Helpful additional skills include using one of the various unit test and documentation frameworks available. We've included information in the appendices.

Classroom Use. Instructors are always looking for classroom projects that are engaging, comprehensible, and focus on perfecting language skills. Many real-world applications require considerable explanation of the problem domain; the time spent reviewing background information detracts from the time available to do the relevant programming. While all application programming requires some domain knowledge, the idea behind these exercises is to pick a domain that many people know a little bit about. This allows an instructor to use some or all of these exercises without wasting precious classroom time on incidental details required to understand the problem.

Skills. This book assumes an introductory level of skill in the Python language. We'll focus on Python 3.4 as a minimum, with references to features of Python 3.5

Student skills we expect include the following. If you can't do these things, this book may be too advanced.

- Create source files, compile and run application programs. While this may seem obvious, we don't discuss any integrated development environment (IDE). We have to assume these basic skills are present.
- Use of the core procedural programming constructs: variables, statements, exceptions, functions. We will not, for example, spend any time on design of loops that terminate properly.
- Some exposure to class definitions and subclasses. This includes managing the basic features of inheritance, as well as overloaded method names.
- Some exposure to the various kinds of built-in collections.
- Optionally, some experience with a unit testing framework. See the appendices for supplemental exercises if you aren't familiar with Python's `unittest` or `doctest`.
- Optionally, some experience writing formal documentation. For Python programmers, this means writing docstrings and using a tool like **Epydoc** or **sphinx**. See the appendices for supplemental exerises if you aren't familiar with formal, deliverable documentation.

1.4 Organization of This Book

This book presents a series of exercises to build simulations of the common, popular casino table games: Roulette, Craps and Blackjack. Each simulation can be extended to include variations on the player's betting system. With a simple statistical approach, we can show the realistic expectations for any betting system. Each of these games has a separate part in this book. Each part consists of a number of individual exercises to build the entire simulation. The completed project results in an application that can provide simple tabular results that shows the average losses expected from each betting strategy.

The interesting degree of freedom in each of the simulations is the player's betting strategy. The design will permit easy adaptation and maintenance of the player's strategies. The resulting application program can be extended by inserting additional betting systems, which allows exploration of what (if any) player actions can minimize the losses.

Roulette. For those who've never been in a casino, or seen movies that have casinos in them, Roulette is the game with the big wheel. They spin the wheel and toss in a marble. When the wheel stops spinning, the bin in which the marble rests defines the winning outcomes.

People who bet on the right things get money. People who bet on the wrong things lose money.

Starting in *Roulette*, we proceed slowly, building up the necessary application one class at a time. Since this is the simplest game, the individual classes reflect that simplicity. We focus on isolation of responsibilities, creating a considerable number of classes. The idea is to build skills in object design by applying those skills to a number of classes.

The first chapter of the part provides details on the game of Roulette and the problem that the simulation solves. The second chapter is an overview of the solution, setting out the highest-level design for the application software. This chapter includes a technique for doing a *"walk-through"* of the design to be confident that the design will actually solve the problem.

Each of the remaining sixteen chapters is a design and programming exercise to be completed by the student. Plus or minus a Frequently Asked Questions (FAQ) section, each chapter has the same basic structure: an overview of the components being designed, some design details, and a summary of the deliverables to be built. The overview section presents some justification and rationale for the design. This material should help the student understand why the particular design was chosen. The design section provides a more detailed specification of the class or classes to be built. This will include some technical information on Java or Python implementation techniques.

Craps. For those who've never been in a casino, or seen the play "Guys and Dolls", Craps is the game with the dice. A player rolls ("shoots") the dice. Sometimes there's a great deal of shouting and clapping. A throw of the dice may – or may not – resolve bets. Additionally, a throw of the dice may also change the state of the game. A casino provides a number of visual cues as to the state of the game and the various bets.

In *Craps*, we build on the design patterns from Roulette. Craps, however, is a stateful game, so there is a more sophisticated design to handle the interactions between dice, game state and player. We exploit the **State** design pattern to show how the design pattern can be applied to this simple situation.

The first chapter is background information on the game of Craps, and the problem that the simulation solves. The second chapter is an overview of the solution, setting out the highest-level design for the application software. This chapter also provides a "walk-through" of the design.

Each of the remaining eleven chapters is an exercise to be completed by the student. Each chapter has the same basic structure: an overview of the component being designed, some design details, and a summary of the deliverables to be built.

Blackjack. For those who've never been in a casino, or seen a movie with Blackjack, Blackjack is a game with cards. The dealer deals two cards to themselves and each player. One of the dealer's card is up and one is down, providing a little bit of information on the dealer's hand. The players may ask for additional cards, or keep the hand they've got.

The idea is to build a hand that's close to 21 points, but not more than 21. In Craps and Roulette there are a lot of bets, but few player decisions. In Blackjack, there are few bets, but really complex player decisions.

In *Blackjack*, the game states are more sophisticated than Craps or Roulette. In casino gift shops, you can buy small summary cards that enumerate all possible game states and responses. The more advanced student can tackle these sophisticated playing strategies. For the less advanced student we will simplify the strategies down to a few key conditions.

The first two chapters are background information on the game of Blackjack, the problem that the simulation solves, and an overview of the solution, setting out the highest-level design for the application software. Each of the remaining six chapters is an exercise to be completed by the student. Since this is more advanced material, and builds on previous work, this part has many simple deliverables compressed into the individual chapters.

Fit and Finish. We include several fit-and-finish issues in *Fit and Finish*. This includes more information and examples on unit testing and documentation.

Additionally, we cover some "main program" issues required to knit all of the software components together into a finished whole.

1.5 Why This Subject?

Casino table games may seem like an odd choice of subject matter for programming exercises. We find that casino games have a number of advantages for teaching OO design and OO programming.

- Casino games have an almost ideal level of complexity. If they were too simple, the *house edge* would be too obvious and people would not play them. If they were too complex, people would not enjoy them as simple recreation. Years (centuries?) of experience in the gaming industry has fine-tuned the table games to fit nicely with the limits of our human intellect.
- Simulation of discrete phenomena lies at the origin of OO programming. We have found it easier to motivate, explain and justify OO design when solving simulation problems. The student can then leverage this insight into other applications of OO programming for more common transactional applications.
- The results are sophisticated but easy to interpret. Probability theory has been applied by others to develop precise expectations for each game. These simulations should produce results consistent with the known probabilities. This book will skim over the probability theory in order to focus on the programming. For a few exercises, the theoretical results will be provided to serve as checks on the correctness of the student's work.
- They're more fun than most other programming problems.

This book does not endorse casino gaming. Indeed, one of the messages of this book is that all casino games are biased against the player. Even the most casual study of the results of the exercises will allow the student to see the magnitude of the *house edge* in each of the games presented.

1.6 Conventions Used in This Book

Here is how we might present code.

Typical Python Example

```
from collections import defaultdict
combo = defaultdict(int)
for i in range(1,7):
    for j in range(1,7):
        roll= i+j
        combo[roll] += 1
for n in range(2,13):
    print( "{0:d} {1:.2%}".format( n, combo[n]/36.0 ) )
```

1. We create a Python dictionary, a map from key to value. We use the `collections.defaultdict` so that missing keys are created in the dictionary with an initial value created by the `int()` function.
2. We iterate through all combinations of two dice, using variables `i` and `j` to represent each die.
3. We sum the dice to create a roll. We increment the value in the dictionary based on the roll.
4. Finally, we print each member of the resulting dictionary. We've used a sophisticated format string that interpolates a decimal value and a floating-point percentage value.

The output from the above program will be shown as follows:

```
2 2.78%
3 5.56%
4 8.33%
5 11.11%
6 13.89%
7 16.67%
8 13.89%
9 11.11%
10 8.33%
11 5.56%
12 2.78%
```

We will use the following type styles for references to a specific `Class`, `method()`, or `variable`.

Tip: Tips Look Like This

There will be design tips, and warnings, in the material for each exercise. These reflect considerations and lessons learned that aren't typically clear to starting OO designers.

CHAPTER

TWO

FOUNDATIONS

We'll set our goal by presenting several elements that make up a complete *Problem Statement*: a *context* in which the problem arises, the *problem*, the *forces* that influence the choice of solution, the *solution* that balances the forces, and some *consequences* of the chosen solution.

Based on the problem statement, we'll present the high-level use case that this software implements. This will be *Our Simulation Application*.

In *Solution Approach* we'll summarize the approach to the solution, describing the overall strategy that we will follow. This is a kind of overall design pattern that we'll use to establish some areas of responsibility.

We'll look at some issues in *Methodology, Technique and Process* that are not technical in nature. They're more procedural and provide some direction on development, testing and writing documentation.

In *Additional Topics: Non-Functional Requirements* we'll look at issues like Quality, Rework, Reuse, and Design Patterns.

Finally, in *Deliverables_*, we'll address the kinds of deliverables that should be produced. These kinds of deliverables form the basis for each chapter.

2.1 Problem Statement

We'll start with a big-picture overview of our problem. We'll present the context in which the problem arises, a summary of the problem, and a "business use case". This will show how our application is used.

We can then dig into the details of our application.

Important: Fools Rush In

It's important not to rush in to programming.

Be sure you understand the problem being solved and how software solves that problem.

Context. Our context is the "classic" casino table games played against the house, including Roulette, Craps and Blackjack. We want to explore the consequences of various betting strategies for these casino games.

Questions include "How well does the Cancellation strategy work?" "How well does the Martingale strategy works for the Come Line odds bet in Craps?" "How well does this Blackjack strategy I found on the Internet compare with the strategy card I bought in the gift shop?"

A close parallel to this is exploring variations in rules and how these different rules have an influence on outcomes. Questions include "What should we do with the 2x and 10x odds offers in Craps?" "How should we modify our play for a single-deck Blackjack game with 6:5 blackjack odds?"

Our context does not include exploring or designing new casino games. Our context also excludes multi-player games like poker. We would like to be able to include additional against-the-house games like Pai Gow Poker, Caribbean Stud Poker, and Baccarat.

Problem. Our problem is to answer the following question: *For a given game, what player strategies produce the best results?*

Forces. There are a number of forces that influence our choice of solution. First, we want an application that is relatively simple to build. Instead of producing an interactive user interface, we will produce raw data and statistical summaries. If we have little interaction, a command-line interface will work perfectly. We can have the user specify a player strategy and the application respond with a presentation of the results. If the results are tab-delimited or in comma-separated values (CSV) format, they can be pasted into a spreadsheet for further analysis.

Another force that influences our choice of solution is the need to be platform and language agnostic. In this case, we have selected an approach that works well on POSIX-compliant operating systems (i.e., Linux, MacOS, and all of the proprietary UNIX variants), and also works on non-compliant operating systems (i.e., all of the Windows versions).

We also need to strike a balance between interesting programming, probability theory, and statistics. On one hand, the simplicity of these games means that complete analyses have been done using probability theory. However, that's not a very interesting programming exercise, so we will ignore the pure probability theory route in favor of learning OO design and programming.

Another force is the desire to reflect actual game play. While a long-running simulation of thousands of invidual cycles of play will approach the theoretical results, people typically don't spend more than a few hours at a table game. If, for example, a Roulette wheel is spun once each minute, a player is unlikely to see more that 480 spins in an eight-hour evening at a casino. Additionally, many players have a fixed budget, and the betting is confined by table limits. Finally, we need to address the subject of "money management": a player may elect to stop playing when they are ahead. This structures our statistical analysis: we must simulate sessions of play that are limited in time, the amount lost and the amount won.

High-Level Use Case. The high-level (or "business") use case is an overall cycle of investigation . From this overall view, the actor's goal is to find an optimal strategy for a given game.

Here's the scenario we're imagining.

Business Use Case

1. **Actor**. Researches alternative strategies. Uses IDE to build new classes for a simulator.
2. **IDE**. Creates new classes for the simulator.
3. **Actor**. Runs the simulator with selection of game and strategy.
4. **Simulator**. Responds with statistical results.
5. **Actor**. Evaluates the results. Uses a spreadsheet or other tool for analysis and visualization.

Consequences. We're going build the simulator application that supports this high-level (or "business") use case.

We're not going to build the IDE to build the new classes. Any IDE should work.

Additionally, we won't address how to analyze the results.

One of the most important consequences of our solution is that we will build an application into which new player betting strategies can be inserted. Clever gamblers invent new strategies all the time.

We will not know all of the available strategies in advance, so we will not be able to fully specify all of the various design details in advance. Instead, we will find ourselves reworking some parts of the solution, to support a new player betting strategy. This forces us to take an *Agile* approach to the design and implementation.

2.2 Our Simulation Application

The previous section was a fluffy overview of what we're trying to accomplish. It sets some goals and provides a detailed context for who's using this application and why.

Armed with that information, we can look at the simulation application we're going to write.

Our simulation application will allow a programmer to experiment with different casino game betting strategies. We'll build a simple, command-line simulator that provides a reliable, accurate model of the game. We need to be able to easily pick one of a variety of player betting strategies, play a number of simulated rounds of the game, and produce a statistical summary of the results of that betting strategy.

This leads us to a small *essential use case*. There is a single *actor*, the "investigator". The actor's goal is to see the expected results of using a particular strategy for a particular game. The typical scenario is the following.

Essential Use Case

1. **Actor**. Specifies which game and betting strategy to test.

   The game may require additional parameters, like betting limits.

   The strategy may need additional parameters, like an initial budget, or *stake*.

2. **System**. Responds with a statistical summary of the outcomes after a fixed number of cycles (spins, or throws or hands). The number of cycles needs to be small (on the order of 200, to reflect only a few hours of play).

On Simplicity. Yes, this use case is very simple. It's a command-line application: it's supposed to be simple.

The point is to explore OO design, not development of a fancy GUI or web application.

Simplicity is a virtue. We can add a fancy GUI or web presentation of the results later. First, create some results.

2.3 Soapbox on Use Cases

We feel that the use case technique is abused by some IT organizations. Quoting from *[Jacobson95]*. "A use case is a sequence of transactions in a system whose task is to yield a result of measurable value to an individual actor of the system."

A use case will clearly identify an actor, define the value created, and define a sequence of transactions. A use case will be a kind of system test specification. A use case will define the system's behavior, and define why an actor bothers to interact with it.

Use cases are often simplified to user stories: "As a <role>, I need <feature> so that <business value>".

A use case is not a specification, and does not replace ordinary design. We have had experiences with customers who simply retitle their traditional procedural programming specifications as "use cases". We hypothesize that this comes from an unwillingness to separate problem definition from solution definition. The consequence is a conflation of use case, technical background, design and programming specifications into gargantuan documents that defy the ability of programmers or users to comprehend them.

There are a number of common problems with use cases that will make the design job more difficult. Each of these defects should lead to review of the use case with the authors to see what, if anything, they can do to rework the use case to be more complete.

- **No Actor**. Without an actor, it's impossible to tell who is getting value from the interaction. A catch-all title like "the user" indicates that the use case is written from the point of view of a database or the application software, not an actual person.

  An actor can be an interface with other software, in which case, the actual software needs to be named. Without knowing the actor, we have trouble deciding which classes are clients and which classes provide the lower-level services of the application.

- **No Value Proposition**. There are two basic kinds of value: information for decision-making or actions taken as the result of decision-making. People interact with software because there are decisions the software cannot make or there are actions the actor cannot take. Some use cases include

value-less activities like logging in, or committing a transaction, or clicking “Okay” to continue. These are parts of operating *scenarios*, not statements of value that show how the actor is happier or more successful. Without a value proposition, we have no clue as to what problem the software solves, or what it eventually does for the actor.

- **No Interactions**. If the entire body of the use case is a series of steps the application performs, we are suspicious of the focus. We prefer a use case to emphasize interaction with the actor. Complex algorithms or interface specifications should be part of an appendix or supplemental document. Without any interaction, it isn’t clear how the actor uses the software.

We also try to make a distinction between detailed operating scenarios and use cases. We have seen customers write documents they call “detailed use cases” that describe the behavior of individual graphical user interface widgets or panels. We prefer to call these scenarios, since they don’t describe measurable business value, but instead describe technical interactions.

2.4 Solution Approach

From reading the problem and use case information, we can identify at least the following four general elements to our application.

- The game being simulated. This includes the various elements of the game: the wheel, the dice, the cards, the table, and the bets.
- The player being simulated. This includes the various decisions the player makes based on the state of the game, and the various rules of the betting system the player is following.
- The statistics being collected.
- An overall control component which processes the game, collects the statistics, and writes the details or the final summary.

When we look at common design patterns, the **Model-View-Control** pattern often helps to structure applications. A more sophisticated, transactional application may require a more complex structure. However, in this case, the game, the player, and the statistics are the **model**. The command line selection of player and the reporting of raw data is the **view**. A **control** component creates the various objects to execute the simulation and write the results.

While interesting, we will not pursue the design of a general-purpose simulation framework. Nor will we use any of the available general frameworks. While these are handy and powerful tools, we want to focus on developing application software “from scratch” (or *de novo*) as a learning exercise.

Our solution will depend heavily on desktop integration: the actor will use their IDE to create a strategy and build a new version of the application program. Once the application is built, the actor can run the application from the command line, collecting the output file. The statistical results file can be analyzed using a spreadsheet application. There are at least three separate application programs involved: the IDE (including editor and compiler), the simulator, the spreadsheet used for analysis.

A typical execution of the simulator will look like the following example.

Sample Execution

```
python3 -m casino.craps --Dplayer.name="Player1326" >details.log
```

1. We select the main simulator control using the package `casino` and the module `craps`.
2. We define the player to use, `player.name="Player1326"`. The main method will use this parameter to create objects and execute the simulation.
3. We collect the raw data in a file named `details.log`.

We are intentionally limiting our approach to a simple command-line application using the default language libraries.

There are a number of more technical considerations that we will expand in *Deliverables*. These include the use of an overall application framework and an approach for unit testing.

Among the topics this book deals with in a casual – possibly misleading – manner are probability and statitics. Experts will spot a number of gaps in our exposition. For example, there isn't a compelling need for simulation of the simpler games of Craps and Roulette, since they can be completely analyzed. However, our primary objective is to study programming, not casino games, therefore we don't mind solving known problems again. We are aware that our statistical analysis has a number of deficiencies. We will avoid any deeper investigation into statistics.

2.5 Methodology, Technique and Process

We want to focus on technical skills; we won't follow any particular software development methodology too closely. We prefer to lift up a few techniques which have a great deal of benefit.

- Incremental Development. Each chapter is a "sprint" that produces some collection of deliverables. Each part is a complete release.
- Unit Testing. We don't dwell on test-driven development, but each chapter explicitly requires unit tests for the classes built. Ideally, one writes the test cases first.
- Embedded Documentation. We provide appendices on how to use Sphinx or epydoc to create usable API documents.

The exercises are presented as if we are doing a kind of iterative design with small deliverables. We present the exercises like this for a number of reasons.

1. We find design is helped by immediate feedback. While we present the design in considerable detail, we do not present the final code. Programmers new to OO design will benefit from repeated exposure to the transformation of problem statement through design to code.
2. This presentation parallels the way software is developed. A project may emphasize larger collections of deliverables. However, the actual creation of working eventually decomposes into classes, fields and methods.

For developers enamored of a strict waterfall methodology – with all design work completed before any programming work – the book can be read in a slightly different order. From each exercise chapter, read only the **overview** and **design** sections. From that information, integrate the complete design. Then proceed through the **deliverables** sections of each chapter, removing duplicates and building only the final form of the deliverables based on the complete design.

This will show how design rework arises as part of a waterfall methodology. It will show that rework doesn't go away with a Big Design Up Front (BDUF) approach. The rework is merely shifted around a bit.

2.5.1 Making Technical Decisions

Many of the chapters will include some lengthy design decisions that appear to be little more than hand-wringning over nuances. We need to emphasize our technique for doing appropriate hand-wringing over OO design. We call it "Looking For The Big Simple", and find that managers don't often permit the careful enumeration of all the alternatives and the itemization of the pros and cons of each choice.

We have worked with managers who capriciously play their "schedule" or "budget" trump cards, stopping useful discussion of alternatives. This may stem from a fundamental discomfort with the technology, and a consequent discomfort of appearing lost in front of team members and direct reports.

Our suggestion in this book can be summarized as follows:

Important: Good Design

Good OO design comes from a good process for technical decision-making.

First, admit what we don't know, and then take steps to reduce our degrees of ignorance.

A phrase like "work smarter not harder" is useless because it doesn't provide the time and budget to actually get smarter.

The learning process, as with all things, must be managed. This means there must be time budgeted for exploring the bad designs before arriving at a good design.

A little more time spent on design can result in considerable simplification, which will reduce overall development and maintenance costs.

It's also important to note that no one in the real world is omniscient. Some of the exercises include intentional dead-ends. As a practical matter, we can rarely foresee all of the consequences of a design decision.

2.6 Additional Topics: Non-Functional Requirements

We can decompose software requirements into two broad categories. The Functional Requirements are the things the software must do; the use cases should address this completely. The Non-Functional Requirements are all of the supporting ideals and principles that make good software. The number of non-functional features of software is large. We'll talk about a few of them, specifically:

- Quality, in general
- Rework
- Reuse
- Design Patterns

2.6.1 On Quality

Our approach to overall quality assurance is relatively simple. We feel that a focus on unit testing and documetation covers most of the generally accepted quality factors. The Software Engineering Institute (SEI) published a quality measures taxonomy. While officially "legacy", it still provides an exhaustive list of quality attributes. These are broadly grouped into five categories. Our approach covers most of those five categories reasonably well.

- **Need Satisfaction**. Does the software meet the need? We start with a problem statement, define the use case, and then write software which is narrowly focused on the actor's needs. By developing our application in small increments, we can ask ourself at each step, "Does this meet the actor's needs?" It's fairly easy to keep a software development project focused when we have use cases to describe our goals.
- **Performance**. We don't address this specifically in this book. However, the presence of extensive unit tests allows us to alter the implemention of classes to change the overall performance of our application. As long as the resulting class still passes the unit tests, we can develop numerous alternative implementations to optimize speed, memory use, input/output, or any other resource.
- **Maintenance**. Software is something that is frequently changed. It changes when we uncover bugs. More commonly, it changes when our understanding of the problem, the actor or the use case changes. In many cases, our initial solution merely clarifies the actor's thinking, and we have to alter the software to reflect a deeper understanding of the problem.

  Maintenance is just another cycle of the iterative approach we've chosen in this book. We pick a feature, create or modify classes, and then create or modify the unit tests. In the case of bug fixing, we often add unit tests to demonstrate the bug, and then fix our classes to pass the revised unit tests.
- **Adaptation**. Adaptation refers to our need to adapt our software to changes in the environment. The environment includes interfaces, the operating system or platform, even the number of users is part of the environment. When we address issues of interoperability with other software, portability to new operating systems, scalability for more users, we are addressing adaptation issues.

We chose Python to avoid having interoperability and portability issues; this language give admirable support for many scalability issues. Generally, a well-written piece of software can be reused. While this book doesn't focus on reuse, Python is biased toward writing reusable software.

- **Organizational**. There are some organizational quality factors: cost of ownership and productivity of the developers creating it. We don't address these directly. Our approach, however, of developing software incrementally often leads to good developer productivity.

Our approach (Incremental, Unit Testing, Embedded Documentation) assures high quality in four of the five quality areas. Incremental development is a way to focus on need satisfaction. Unit testing helps us optimize resource use, and do maintenance well. Our choices of tools and platforms help us address adaptation.

The organizational impact of these techniques isn't so clear. It is easy to mis-manage a team and turn incremental development into a quagmire of too much planning for too little delivered software. It is all too common to declare that the effort spent writing unit test code is "wasted".

Ultimately, this is a book on OO design. How people organize themselves to build software is beyond our scope.

2.6.2 On Rework

In *Problem Statement*, we described the problem. In *Solution Approach*, we provided an overview of the solution. The following parts will guide you through an incremental design process; a process that involves learning and exploring. This means that we will coach you to build classes and then modify those classes based on lessons learned during later steps in the design process. See our *Soapbox on Rework* for an opinion on the absolute necessity for design rework.

We don't simply present a completed design. We feel that it is very important follow a realistic problem-solving trajectory so that beginning designers are exposed to the decisions involved in creating a complete design. In our experience, all problems involve a considerable amount of "learn as you go".

We want to reflect this in our series of exercises. In many respects, a successful OO design is one that respects the degrees of ignorance that people have when starting to build software. We will try to present the exercises in a way that teaches the reader how to manage ignorance and still develop valuable software.

Soapbox on Rework

Important: The best way to learn is to make mistakes.
Rework is a consequence of learning.

All of software development can be described as various forms of knowledge capture. A project begins with many kinds of ignorance and takes steps to reduce that ignorance. Some of those steps should involve revising or consolidating previous learnings.
A project without rework is suspiciously under-engineered.

For some, the word *rework* has a negative connotation. If you find the word distasteful, please replace every occurance with any of the synonyms: adaptation, evolution, enhancement, mutation. We prefer the slightly negative connotation of the word rework because it helps managers realize the importance of incremental learning and how it changes the requirements, the design and the resulting software.

Since learning will involve mistakes, good management plans for the costs and risks of those mistakes. Generally, our approach is to manage our ignorance; we try to create a design such that correcting a mistake only fixes a few classes.

We often observe denial of the amount of ignorance involved in creating IT solutions. It is sometimes very difficult to make it clear that if the problem was well-understood, or the solution was well-defined there would be immediately applicable off-the-shelf or open-source solutions. The absence of a ready-to-hand solution generally means the problem is hard. It also means that there are several degrees of ignorance: ignorance of the problem, solution and technology; not to mention ignorance of the amount of ignorance involved in each of these areas.

We see a number of consequences of denying the degrees of ignorance.

- **Programmers**. For programmers, experienced in non-OO (e.g. *procedural*) environments, one consequnece is that they find learning OO design is difficult and frustrating. Our advice is that since this is new, you have to make mistakes or you won't learn effectively. Allow yourself to explore and make mistakes; feel free to rework your solutions to make them better. Above all, do not attempt to design a solution that is complete and perfect the very first time. We can't emphasize enough the need to do design many times before understanding what is important and what is not important in coping with ignorance.
- **Managers**. For managers, experienced in non-object implementation, the design rework appears to be contrary to a fanciful expectation of reduced development effort from OO techniques. The usual form for the complaint is the following: "I thought that OO design was supposed to be easier than non-OO design." We're not sure where the expectation originates, but good design takes time, and learning to do good design seems to require making mistakes. Every project needs a budget for making the necessary mistakes, reworking bad ideas to make them good and searching for simplifications.
- **Economics**. Often, management attempts the false economy of attempting to minimize rework by resorting to a *waterfall* methodology. The idea is that having the design complete before attempting to do any development somehow magically prevents design rework. We don't see that this waterfall approach minimizes rework; rather, we see it shifting the rework forward in the process. There are two issues ignored by this approach: how we grow to understand the problem domain and providing an appropriate level of design detail.

We find that any initial "high-level" design can miss details of the problem domain, and this leads to rework. Forbidding rework amounts to mandating a full understanding of the problem prior to any code.

In most cases, our users do not fully understand their problem any more than our developers understand our users. Generally, it is very hard to understand the problem, the technology, and the solution. We find that hands-on use of preliminary versions of software can help more than endless conversations about what could be built.

There was a time when programming languages were so primitive that deep, detailed design was required. In a language that lacks the built-in collection classes, a great deal of design was required just to introduce the collections required to do useful work.

Because Python comes with so many features built in, it's often more efficient to write a draft version of a class in Python than it is to write an elaborate design document prior to writing a preliminary draft.

War Story on Bad Design

In one advanced programming course, we observed the following sad scenario. The instructor provided an excellent background in how to create abstract data type (ADT) definitions for the purely mathematical objects of *scalar*, *vector* and *matrix*. The idea was to leverage the ADTs to implement more complex operations like matrix multiplication, inversion and Gaussian elimination. The audience, primarily engineers, seemed to understand how this applied to things they did every day. The first solution, presented after a week of work, began with the following statement: "Rather than think it through from the basic definitions of matrix and vector, I looked around in my drawer and found an old FORTRAN program and rewrote that, basically transliterating the FORTRAN." We think that a lack of experience in the process of software design makes it seem that copying an inappropriate solution is more effective than designing a good solution from scratch.

2.6.3 On Reuse

While there is a great deal of commonality among the three games, the exercises do not start with an emphasis on constructing a general framework. We find that too much generalization and too much emphasis on *reuse* is not appropriate for beginning object designers. See *Soapbox on Reuse* for an opinion on reuse.

Additionally, we find that projects that begin with too-lofty reuse goals often fail to deliver valuable solutions in a timely fashion. We prefer not to start out with a goal that amounts to boiling the ocean to make a pot of tea.

Soapbox on Reuse

While a promise of OO design is reuse, this needs to be tempered with some pragmatic considerations. There are two important areas of reuse: reusing a class specification to create objects with common structure and behavior, and using inheritance to reuse structure and behavior among multiple classes of objects. Beyond these two areas, reuse can create more cost than value.

The first step in reuse comes from isolating responsibilities to create classes of objects. Generally, a number of objects that have common structure and behavior is a kind of reuse. When these objects cooperate to achieve the desired results, this is sometimes called *emergent behavior*: no single class contains the overall functionality, it grew from the interactions among the various objects.

This is sometimes called *inversion of control*.

When the application grows and evolves, we can preserve some class declarations, reusing them in the next revision of the application. This reduces the cost and risks associated with software change. Class definitions are the most fundamental and valuable kind of reuse.

Another vehicle for OO resuse is inheritance. The simple subclass-superclass relationship yields a form of reuse; a class hierarchy with six subclasses will share the superclass code seven times over. This, by itself, has tremendous benefits.

We caution against any larger scope of reuse. Sharing classes between projects may or may not work out well. The complexity of achieving inter-project reuse can be paralyzing to first-time designers. Often, different projects reflect different points of view, and the amount of sharing is limited by these points of view.

As an example, consider a product in a business context. An external customer's view of the product (shaped by sales and marketing) may be very different from the internal views of the same product. Internal views of the product (for example, finance, legal, manufacturing, shipping, support) may be very different from each other. Reconciling these views may be far more challenging than a single software development project. For that reason, we don't encourage this broader view of reuse.

2.6.4 On Design Patterns

These exercises will refer to several of the "Gang of Four" design patterns in *[Gamma95]*. The Design Patterns book is not a prerequisite; we use it as reference material to provide additional insight into the design patterns used here. We feel that use of common design patterns significantly expands the programmer's repertoire of techniques. We note where they are appropriate, and provide some guidance in their implementation.

In addition, we reference several other design patterns which are not as well documented. These are, in some cases, patterns of bad design more than patterns of good design.

2.7 Deliverables

Each chapter defines the classes to be built and the unit testing that is expected. A third category of deliverable – documentation – is merely implied.

The purpose of each chapter is to write the source files for one or more classes, the source files for one or more unit tests, and assure that a minimal set of API documentation is available.

- **Source Files**. The source files are the most important deliverable. In effect, this is the working application program. Generally, we will be running this application from within the Integrated Development Environment (IDE). We can, of course, create a stand-alone program.

  In the case of Python, the "program" is simply the packages of `.py` files. There really isn't much more to deliver. The interested student might want to look at the Python `distutils` and `setuptools` to create a distribution kit, or possibly a Python `.egg` file.

- **Unit Test Files**. The deliverables section of each chapter summarizes the unit testing that is expected, in addition to the classes to be built. We feel that unit testing is a critical skill, and emphasize it throughout the inividual exercises. We don't endorse a particular technology for implementing the unit tests. There are several approaches to unit testing that are in common use.

  For formal testing of some class, `X`, we create a separate class, `TestX`, which creates instances of `X` and exercises those instances to be sure they work. The `unittest` package is the mechanism for doing formal unit tests. Additionally, many Python developers also use the `doctest` module to assure that the sample code in the docstrings is actually correct. We cover these technologies in the appendices.

- **Documentation**. The job isn't over until the paperwork is done. The internal documentation is generally built from specially formatted blocks of comments within the source itself. We can use **Epydoc** (or **sphinx**) to create documentation based on the code.

Part II

Roulette

This part describes the game of Roulette. Roulette is the game with the big wheel. They spin the wheel, toss in a marble and wait for the wheel to stop spinning.

Roulette is – essentially – a stateless game with numerous bets and a very simple process for game play.

The chapters of this part present the details on the game, an overview of the solution, and a series of sixteen exercises to build a complete simulation of the game, plus a variety of betting strategies. Each exercise chapter builds at least one class, plus unit tests; in some cases, this includes rework of previous deliverables.

CHAPTER

THREE

ROULETTE DETAILS

We'll start out by looking a the game of Roulette in *Roulette Game*. This will focus on Roulette as played in most American casinos.

We will follow this with *Available Bets in Roulette*. There are a profusion of bets available in Roulette. Most of the sophisticated betting strategies focus on just one of the even-money bets.

In *Some Betting Strategies*, we will describe some common betting strategies that we will simulate. The betting strategies are interesting and moderately complex algorithms for changing the amount that is used for each bet in an attempt to recoup losses.

We'll also include some additional topics in *Roulette Details Questions and Answers*.

3.1 Roulette Game

The game of Roulette centers around a *wheel* with thirty-eight numbered *bins*. The numbers include 0, 00 (double zero), 1 through 36. The *table* has a surface marked with spaces on which players can place *bets*. The spaces include the 38 *numbers*, plus a variety of additional bets, which will be detailed below.

After the bets are placed by the players, the wheel is spun by the house, a small ball is dropped into the spinning wheel; when the wheel stops spinning, the ball will come to rest in one of the thirty-eight numbered bins, defining the winning number. The winning number and all of the related winning bets are paid off; the losing bets are collected. Roulette bets are all paid off using *odds*, which will be detailed with each of the bets, below.

The numbers from 1 to 36 are colored red and black in an arbitrary pattern. They fit into various ranges, as well as being even or odd, which defines many of the winning bets related to a given number. The numbers 0 and 00 are colored green, they fit into none of the ranges, and are considered to be neither even nor odd. There are relatively few bets related to the zeroes. The geometry of the betting locations on the table defines the relationships between number bets: a group of numbers is given a colorful names like a street or a corner.

Note: American Rules

There are slight variations in Roulette between American and European casinos. We'll focus strictly on the American version.

3.2 Available Bets in Roulette

There are a variety of bets available on the Roulette table. Each bet has a payout, which is stated as $n : 1$ where n is the multiplier that defines the amount won based on the amount bet.

A $5 bet at 2:1 will win $10. After you are paid, there will be $15 sitting on the table, your original $5 bet, plus your $10 additional winnings.

Note: Odds

Not all games state their odds using this convention. Some games state the odds as "2 *for* 1". This means that the total left on the table after the bets are paid will be two times the original bet. So a $5 bet will win $5, there will be $10 sitting on the table.

Here's the layout of the betting area on a Roulette table.

Fig. 3.1: Roulette Table Layout

The table is divided into two classes of bets. The "inside" bets are the 38 numbers and small groups of numbers; these bets all have relatively high odds. The "outside" bets are large groups of numbers, and have relatively low odds. If you are new to casino gambling, see *Odds and Payouts* for more information on odds and why they are offered.

- A "straight bet" is a bet on a single number. There are 38 possible bets, and they pay odds of 35 to 1. Each bin on the wheel pays one of the straight bets.

- A "split bet" is a bet on an adjacent pair of numbers. It pays 17:1. The table layout has the numbers arranged sequentially in three columns and twelve rows. Adjacent numbers are in the same row or column. The number 5 is adjacent to 4, 6, 2, 8; the number 1 is adjacent to 2 and 4. There are 114 of these split bet combinations. Each bin on the wheel pays from two to four of the available split bets. If the balls lands in either of two bins, the split bet is a winner.
- A "street bet" includes the three numbers in a single row, which pays 11:1. There are twelve of these bets on the table. Any of three bins make a street bet a winner.
- A square of four numbers is called a "corner bet" and pays 8:1. There are 22 of these bets available.
- At one end of the layout, it is possible to place a bet on the Five numbers 0, 00, 1, 2 and 3. This pays 6:1. It is the only combination bet that includes 0 or 00.
- A "line bet" is a six number block, which pays 5:1. It is essentially two adjacent street bets. There are 11 such combinations.

The following bets are the "outside" bets. Each of these involves a group of twelve to eighteen related numbers. None of these outside bets includes 0 or 00. The only way to bet on 0 or 00 is to place a straight bet on the number itself, or use the five-number combination bet.

- Any of the three 12-number ranges (1-12, 13-24, 25-36) pays 2:1. There are just three of these bets.
- The layout offers the three 12-number columns at 2:1 odds. All of the numbers in a given column have the same remainder when divided by three. Column 1 contains 1, 4, 7, etc., all of which have a remainder of 1 when divided by 3.
- There are two 18-number ranges: 1-18 is called *low*, 19-36 is called *high*. These are called even money bets because they pay at 1:1 odds.
- The individual numbers are colored red or black in an arbitrary pattern. Note that 0 and 00 are colored green. The bets on red or black are even money bets, which pay at 1:1 odds.
- The numbers (other than 0 and 00) are also either even or odd. These bets are also even money bets.

Odds and Payouts

Not all of the Roulette outcomes are equal probability. Let's compare a "split bet" on 1-2 and a *even money* bet on red.

- The split bet wins if either 1 or 2 comes up on the wheel. This is 2 of the 38 outcomes, or a 1/19 probability, 5.26%.
- The red bet wins if any of the 18 red numbers come up on the wheel. The is 18 of the 38 outcomes, or a 9/19 probability, 47.4%.

Clearly, the red bet is going to win almost ten times more often than the 1-2 bet. As an inducement to place bets on rare occurences, the house offers a higher payout on those bets. Since the 1-2 split bet wins is so rarely, they will pay you 17 times what you bet. On the other hand, since the red bet wins so frequently, they will only pay back what you bet.

You'll notice that the odds of winning the 1-2 split bet is 1 chance in 19, but they pay you 17 times your bet. Since your bet is still sitting on the table, it looks like 18 times your bet. It still isn't 19 times your bet. This discrepency between the actual probability and the payout odds is sometimes called the *house edge*. It varies widely among the various bets in the game of Roulette. For example, the 5-way bet has 5/38 ways of winning, but pays only 6:1. There is only a 13.2% chance of winning, but they pay you as if you had a 16.7% chance, keeping the 3.5% difference. You have a 5.26% chance to win a split bet, but the house pays as if it were a 5.88% chance, a .62% discrepency in the odds.

The smallest discrepency between actual chances of winning (47.4%) and the payout odds (50%) is available on the even money bets: red, black, even, odd, high or low. All the betting systems that we will look at focus on these bets alone, since the house edge is the smallest.

3.3 Some Betting Strategies

Perhaps because Roulette is a relatively simple game, elaborate betting systems have evolved around it. Searches on the Internet turn up a many copies of the same basic descriptions for a number of betting systems. Our purpose is not to uncover the actual history of these systems, but to exploit them for simple OO design exercises. Feel free to research additional betting systems or invent your own.

Martingale. The *Martingale* system starts with a base wagering amount, w, and a count of the number of losses, c, initially 0. Each loss doubles the bet.

Any given spin will place an amount of $w \times 2^c$ on a 1:1 proposition (for example, red). When a bet wins, the loss count is reset to zero; resetting the bet to the base amount, w. This assures that a single win will recoup all losses.

Note that the casinos effectively prevent successful use of this system by imposing a table limit. At a $10 Roulette table, the limit may be as low as $1,000. A Martingale bettor who lost six times in a row would be facing a $640 bet, and after the seventh loss, their next bet would exceed the table limit. At that point, the player is unable to recoup all of their losses. Seven losses in a row is only a 1 in 128 probability; making this a relatively likely situation.

Waiting. Another system is to wait until some number of losses have elapsed. For example, wait until the wheel has spun seven reds in a row, and then bet on black. This can be combined with the Martingale system to double the bet on each loss as well as waiting for seven reds before betting on black.

This "wait for a favorable state" strategy is based on a confusion between the outcome of each individual spin and the overall odds of given collections of spins. If the wheel has spun seven reds in a row, it's "due" to spin black.

1-3-2-6 System. Another betting system is called the *1-3-2-6* system. The idea is to avoid the doubling of the bet at each loss and running into the table limit. Rather than attempt to recoup all losses in a single win, this system looks to recoup all losses by waiting for four wins in a row.

The sequence of numbers (1, 3, 2 and 6) are the multipliers to use when placing bets after winning. At each loss, the sequence resets to the multiplier of 1. At each win, the multiplier is advanced through the sequence. After one win, the bet is now $3w$. After a second win, the bet is reduced to $2w$, and the winnings of $4w$ are "taken down" or removed from play. In the event of a third win, the bet is advanced to $6w$. Should there be a fourth win, the player has doubled their money, and the sequence resets.

Cancellation. Another method for tracking the lost bets is called the *Cancellation* system or the *Labouchere* system. The player starts with a betting budget allocated as a series of numbers. The usual example is 1, 2, 3, 4, 5, 6, 7, 8, 9.

Each bet is sum of the first and last numbers in the last. In this case 1+9 is 10. At a win, cancel the two numbers used to make the bet. In the event of all the numbers being cancelled, reset the sequence of numbers and start again. For each loss, however, add the amount of the bet to the end of the sequence as a loss to be recouped.

Here's an example of the cancellation system using 1, 2, 3, 4, 5, 6, 7, 8, 9.

1. Bet 1+9. A win. Cancel 1 and 9, leaving 2, 3, 4, 5, 6, 7, 8.
2. Bet 2+8. A loss. Add 10, leaving 2, 3, 4, 5, 6, 7, 8, 10.
3. Bet 2+10. A loss. Add 12, leaving 2, 3, 4, 5, 6, 7, 8, 10, 12.
4. Bet 2+12. A win. Cancel 2 and 12, leaving 3, 4, 5, 6, 7, 8, 10.
5. Next bet will be 3+10.

A player could use the *Fibonacci Sequence* to structure a series of bets in a kind of cancellation system. The Fibonacci Sequence is 1, 1, 2, 3, 5, 8, 13, ...

At each loss, the sum of the previous two bets – the next numbers in the sequence – becomes the new bet amount. In the event of a win, we simply revert to the base betting amount. This allows the player to easily track our accumulated losses, with bets that could recoup those losses through a series of wins.

3.4 Roulette Details Questions and Answers

Do the house limits really have an impact on play?

Of course they do, or the house wouldn't impose them.

Many betting strategies increase bets on a loss, in an effort to break even or get slightly ahead. How many losses can occur in a row?

The answer is "indefinite." The odds of a long series of losses get less and less probable, but it never becomes zero.

For a simple $P = .5$ event – like flipping a coin – we can often see three heads in a row. One head is $P(\mathrm{h}) = .5$. Two heads is $P(\mathrm{h}, \mathrm{h}) = .5 \times .5$. If we partition a sequence of coin tosses into pairs, we expect that 1 of 4 pairs will have two heads.

Three heads is $P(\mathrm{h}, \mathrm{h}, \mathrm{h}) = .5 \times .5 \times .5$. If we partition a sequence of coin tosses into threes, we expect that 1 of 8 triples will have three heads.

A table limit that's 30 times the base bet, say $300 at a $10 table, is a way of capping play at an event has a $P = \frac{1}{30} = 0.033$. Three percent is the odds of seeing five heads in a row. If we double the bet on each loss, we'd only need to see five losses in a row to reach the table limit.

This three percent limit is in line with our ways the house maintains an edge in each game.

Why are there so many bets in Roulette?

This is a universal feature of gambling. A lot of different kinds of bets allows a player to imagine that one of those bets is better than the others.

In Blackjack, for example, there's essentially one bet. However, casinos have added a few additional bets. It doesn't reach the complexity of Roulette or Craps, but there are a number of betting opportunities even in Blackjack.

Many of the betting strategies leverage the simplest of bets: a nearly even-money proposition like red, black, even, odd, high, or low. These are easy to analyze because they're (nearly) $P = .5$ and the cost of a series of losses or a series of wins is easy to compute.

When we look at Craps, the core bet – the "pass line" – is also nearly $P = .5$, allowing the use of similar betting strategies.

The idea of the various betting strategies, then, is to avoid all of the complexity and focus on just a simple bet. For the most part, our `Player` implementations can follow this approach.

However, if we implement the complete game, we can write `Players` that make a number of different kinds of bets to see how the house edge breaks the various betting strategies.

CHAPTER

FOUR

ROULETTE SOLUTION OVERVIEW

The first section, *Preliminary Survey of Classes*, is a survey of the classes gleaned from the general problem statement. Refer to *Problem Statement* as well as the problem details in *Roulette Details*. This survey is drawn from a quick overview of the key nouns in these sections.

We'll amplify this survery with some details of the class definitions in *Preliminary Roulette Class Structure*.

Given this preliminary of the candidate classes, *A Walkthrough of Roulette* is a walkthrough of the possible design that will refine the definitions, and give us some assurance that we have a reasonable architecture. We will make some changes to the preliminary class list, revising and expanding on our survey.

We will also include a number of questions and answers in *Roulette Solution Questions and Answers*. This should help clarify the design presentation and set the stage for the various development exercises in the chapters that follow.

4.1 Preliminary Survey of Classes

To provide a starting point for the development effort, we have to identify the objects and define their responsibilities. The central principle behind the allocation of responsibility is *encapsulation*; we do this by attempting to *isolate the information* or *isolate the processing* that must be done. Encapsulation assures that the methods of a class are the exclusive users of the fields of that class. It also makes each class very loosely coupled with other classes; this permits change without a ripple through the application.

For example, a class for each `Outcome` defines objecgs which can contain both the name and the payout odds. That way each `Outcome` instance can be used to compute a winning amount, and no other element of the simulation needs to share the odds information or the payout calculation.

In reading the background information and the problem statement, we noticed a number of nouns that seemed to be important parts of the game we are simulating.

- Wheel
- Bet
- Bin
- Table
- Red
- Black
- Green
- Number
- Odds
- Player
- House

One common development milestone is to be able to develop a class model in the Unified Modeling Language (UML) to describe the relationships among the various nouns in the problem statement. Building (and interpreting) this model takes some experience with OO programming. In this first part, we'll avoid doing extensive modeling. Instead we'll simply identify some basic design principles. We'll focus in on the most important of these nouns and describe the kinds of classes that you will build.

In a few cases, we will look forward to anticipate some future considerations. One such consideration is the house *rake*, also known as the *vigorish*, *vig*, or *commission*. In some games, the house makes a 5% deduction from some payouts. This complexity is best isolated in the `Outcome` class. Roulette doesn't have any need for a rake, since the presence of the 0 and 00 on the wheel gives the house a little over 5% edge on each bet. We'll design our class so that this can be added later when we implement Craps.

4.2 Preliminary Roulette Class Structure

We'll summarize some of the classes and responsibilities that we can identify from the problem statement. This is not the complete list of classes we need to build. As we work through the exercises, we'll discover additional classes and rework some of these preliminary classes more than once.

We'll describe each class with respect to the responsibility allocated to the class and the collaborators. Some collabotors are used by an object to get work done. We have a number of "uses-used by" collaborative relationships among our various classes.

Outcome Responsibilities.

A name for the bet and the payout odds. This isolates the calculation of the payout amount. Example: "Red", "1:1".

Collaborators.

Collected by `Wheel` into the bins that reflect the bets that win; collected by `Table` into the available bets for the `Player`; used by `Game` to compute the amount won from the amount that was bet.

Wheel Responsibilities.

Selects the `Outcome` s that win. This isolates the use of a random number generator to select `Outcome`s; and it encapsulates the set of winning `Outcome`s that are associated with each individual number on the wheel. Example: the "1" bin has the following winning `Outcome`s: "1", "Red", "Odd", "Low", "Column 1", "Dozen 1-12", "Split 1-2", "Split 1-4", "Street 1-2-3", "Corner 1-2-4-5", "Five Bet", "Line 1-2-3-4-5-6", "00-0-1-2-3", "Dozen 1", "Low" and "Column 1".

Collaborators.

Collects the `Outcome`s into bins; used by the overall `Game` to get a next set of winning `Outcome`s.

Table Responsibilities.

A collection of bets placed on `Outcome`s by a `Player`. This isolates the set of possible bets and the management of the amounts currently at risk on each bet. This also serves as the interface between the `Player` and the other elements of the game.

Collaborators.

Collects the `Outcome`s; used by `Player` to place a bet amount on a specific `Outcome`; used by `Game` to compute the amount won from the amount that was bet.

Player Responsibilities.

Places bets on `Outcome`s, updates the stake with amounts won and lost.

Collaborators.

Uses `Table` to place bets on `Outcome`s; used by `Game` to record wins and losses.

Game Responsibilities.

Runs the game: gets bets from *Player*, spins *Wheel*, collects losing bets, pays winning bets. This encapsulates the basic sequence of play into a single class.

Collaborators.

Uses *Wheel*, *Table*, *Outcome*, *Player*. The overall statistical analysis will play a finite number of games and collect the final value of the *Player*'s stake.

The class *Player* has the most important responsibility in the application, since we expect to update the algorithms this class uses to place different kinds of bets. Clearly, we need to cleanly encapsulate the *Player*, so that changes to this class have no ripple effect in other classes of the application.

4.3 A Walkthrough of Roulette

A good preliminary task is to review these responsibilities to confirm that a complete cycle of play is possible. This will help provide some design details for each class. It will also provide some insight into classes that may be missing from this overview.

A good way to structure this task is to do a Class-Reponsibility-Collaborators (CRC) *walkthrough*.

As preparation, get some 5" x 8" notecards. On each card, write down the name of a class, the responsibilities and the collaborators. Leave plenty of room around the responsibilities and collaborators to write notes. We've only identified five classes, so far, but others always show up during the walkthrough.

During the walkthrough, we identify areas of responsibility, allocate them to classes of objects and define any collaborating objects. An area of responsibility is a thing to do, a piece of information, a result. Sometimes a big piece of responsibility can be broken down into smaller pieces, and those smaller pieces assigned to other classes. There are a lot of reasons for decomposing, the purpose of this book is to explore many of them in depth. Therefore, we won't justify any of our suggestions until later in the book. For now, follow along closely to get a sense of where the exercises will be leading.

The basic processing outline is the responsibility of the *Game* class. To start, locate the *Game* card.

1. Our preliminary note was that this class "Runs the game." The responsibilities section has a summary of four steps involved in running the game.
2. The first step is "gets bets from *Player*." Find the *Player* card.
3. Does a *Player* collaborate with a *Game* to place bets? If not, update the cards as necessary to include this.
4. One of the responsibilities of a *Player* is to place bets. The step in the responsibility statement is merely "Places bets on *Outcome*s." Looking at the classes, we note that the *Table* contains the amounts placed on the Bets. Fix the collaboration information on the *Player* to name the *Table* class. Find the *Table* card.
5. Does a *Table* collaborate with a *Player* to accept the bets? If not, update the cards as necessary to include this.
6. What card has responsibility for the amount of the bet? It looks like *Table*. We note one small problem: the *Table* contains the *collection* of amounts bet on *Outcome*s.

   What class contains the individual "amount bet on an *Outcome*?" This class appears to be missing. We'll call this new class *Bet* and start a new card. We know one responsibility is to hold the amount bet on a particular *Outcome*.

   We know three collaborators: the amount is paired with an *Outcome*, all of the *Bet* s are collected by a *Table*, and the *Bet* s are created by a *Player*. We'll update all of the existing cards to name their collaboration with *Bet*.
7. What card has responsibility for keeping all of the *Bet*s? Does *Table* list that as a responsibility? We should update these cards to clarify this collaboration.

You should continue this tour, working your way through spinning the `Wheel` to get a list of winning `Outcome`s. From there, the `Game` can get all of the `Bet`s from the `Table` and see which are based on winning `Outcome`s and which are based on losing `Outcome`s. The `Game` can notify the `Player` of each losing `Bet`, and notify the `Player` of each winning `Bet`, using the `Outcome` to compute the winning amount.

This walkthrough will give you an overview of some of the interactions among the objects in the working application. You may uncover additional design ideas from this walkthrough. The most important outcome of the walkthrough is a clear sense of the responsibilities and the collaborations required to create the necessary application behavior.

4.4 Roulette Solution Questions and Answers

Why does the `Game` class run the sequence of steps? Isn't that the responsibility of some "main program?"

> **Coffee Shop Answer**. We haven't finished designing the entire application, so we need to reflect our own ignorance of how the final application will be assembled from the various parts. Rather than allocate too many responsibilities to `Game`, and possibly finding conflicts or complication, we'd rather allocate too few responsibilities until we know more.
>
> From another point of view, designing the main program is premature because we haven't finished designing the *entire* application. We anticipate a `Game` object being invoked from some statistical data gathering object to run one game. The data gathering object will then get the final stake from the player and record this. `Game`'s responsibilities are focused on playing the game itself. We'll need to add a responsibility to `Game` to collaborate with the data gathering class to run a number of games as a "session".
>
> **Technical Answer**. In procedural programming (especially in languages like COBOL), the "main program" is allocated almost all of the responsibilities. These procedural main programs usually contain a number of elements, all of which are very tightly coupled. This is a bad design, since the responsibilities aren't allocated as narrowly as possible. One small change in one place breaks the whole program.
>
> In OO languages, we can reduce the main program to a short list of object constructors, with the real work delegated to the objects. This level of coupling assures us that a small change to one class has no impact on other classes or the program as a whole.

Why is `Outcome` a separate class? Each object that is an instance of `Outcome` only has two attributes; why not use an array of Strings for the names, and a parallel array of integers for the odds?

> **Representation**. We prefer not to decompose an object into separate data elements. If we do decompose this object, we will have to ask which class would own these two arrays? If `Wheel` keeps these, then `Table` becomes very tightly coupled to these two arrays that should be `Wheel`'s responsibility. If `Table` keeps these, then `Wheel` is priviledged to know details of how `Table` is implemented. If we need to change these arrays to another storage structure, two classes would change instead of one.
>
> Having the name and odds in a single `Outcome` object allows us to change the representation of an `Outcome`. For example, we might replace the String as the identification of the outcome, with a collection of the individual numbers that comprise this outcome. This would identify a straight bet by the single winning number; an even money bet would be identified by an array of the 18 winning numbers.
>
> **Responsibility**. The principle of isolating responsibility would be broken by this "two parallel arrays" design because now the `Game` class would need to know how to compute odds. In more complex games, there would be the added complication of figuring the rake. Consider a game where the `Player`'s strategy depends on the potential payout. Now the `Game` and the `Player` both have copies of the algorithm for computing the payout. A change to one must be paired with a change to the other.
>
> The alternative we have chosen is to encapsulate the payout algorithm along with the relevant data items in a single bundle.

If `Outcome` encapsulates the function to compute the amount won, isn't it just a glorified subroutine?

If you're background is BASIC or FORTRAN, this can seem to be true. A class can be thought of as a glorified *subroutine library* that captures and isolates data elements along with their associated functions.

A class is more powerful than a simple subroutine library with private data. For example, classes introduce *inheritance* as a way to create a family of closely-related definitions in a simple way.

We discourage trying to mapping OO concepts back to other non-OO languages.

What is the distinction between an `Outcome` and a `Bet`?

We need to describe the propositions on the table on which you can place bets. The propositions are distinct from an actual amount of money wagered on a proposition. There are a lot of terms to choose from, including bet, wager, proposition, place, location, or outcome. We opted for using `Outcome` because it seemed to express the open-ended nature of a potential outcome, different from an amount bet on a potential outcome. We're considering the `Outcome` as an abstract possibility, and the `Bet` as a concrete action taken by a player.

Also, as we expand this simulation to cover other games, we will find that the randomized outcome is not something we can directly bet on. In Roulette, however, all outcomes are something we can be bet on, as well as a great many combinations of outcomes. We will revisit this design decision as we move on to other games.

Why are the classes so small?

First-time designers of OO applications are sometimes uncomfortable with the notion of *emergent behavior*. In procedural programming languages, the application's features are always embodied in a few key procedures. Sometimes a single procedure, named `main`.

A good OO design partitions responsibility. In many cases, this subdivision of the application's features means that the overall behavior is not captured in one central place. Rather, it emerges from the interactions of a number of objects.

We have found that smaller elements, with very finely divided responsibilities, are more flexible and permit change. If a change will only alter a portion of a large class, it can make that portion incompatible with other portions of the same class. A symptom of this is a bewildering nest of if-statements to sort out the various alternatives. When the design is decomposed down more finely, a change can be more easily isolated to a single class. A much simpler sequence of if-statements can be focused on selecting the proper class, which can then simply carry out the desired functions.

CHAPTER

FIVE

OUTCOME CLASS

In *Outcome Analysis* we'll look at the responsibilities and collaborators of Outcome objects.

In *Design Decision – Object Identity* we'll look at how we can implement the notion of object identity and object equality. This is important because we will be matching Outcome objects based on bets and spinning the Roulette wheel.

We'll look forward to some other use cases in *Looking Forward*. Specifically, we know that players, games, and tables will all share references to single outcome objects. How do we do this properly?

In *Outcome Design* we'll detail the design for this class. In *Outcome Deliverables* we'll provide a list of modules that must be built.

We'll look at a Python programming topic in *Message Formatting*. This is a kind of appendix for beginning programmers.

5.1 Outcome Analysis

There will be several hundred instances of `Outcome` objects on a given Roulette table. The bins on the wheel, similarly, collect various `Outcome`s together. The minimum set of `Outcome` instances we will need are the 38 numbers, Red, and Black. The other instances will add details to our simulation.

In Roulette, the amount won is a simple multiplication of the amount bet and the odds. In other games, however, there may be a more complex calculation because the house keeps 5% of the winnings, called the "rake". While it is not part of Roulette, it is good to have our `Outcome` class designed to cope with these more complex payout rules.

Also, we know that other casino games, like Craps, are stateful. An `Outcome` may change the game state. We can foresee reworking this class to add in the necessary features to change the state of the game.

While we are also aware that some odds are not stated as $x : 1$, we won't include these other kinds of odds in this initial design. Since all Roulette odds are $x : 1$, we'll simply assume that the denominator is always 1. We can forsee reworking this class to handle more complex odds, but we don't need to handle the other cases yet.

The issue we have, however, is comparing outcomes. They're used in various places. The idea is to compare the outcomes from a spin of the wheel against the outcomes associated with bets.

How does all this comparison work in Python?

Hint: The default rules aren't helpful.

5.2 Design Decision – Object Identity

Our design will depend on matching `Outcome` objects. We'll be testing objects for equality.

The player will be placing bets that contain `Outcome`s; the table will be holding bets. The wheel will select the winning `Outcome`s. We need a simple test to see if two objects of the `Outcome` class are the same.

Was the *Outcome* for a bet equal to the *Outcome* contained in a spin of the wheel?

It turns out that this comparison between objects has some subtlety to it.

Here's the naïve approach to class definition that doesn't include any provision for equality tests.

Naïve Class Definition

```
>>> class Outcome:
...     def __init__(self, name, odds):
...         self.name= name
...         self.odds= odds
```

This seems elegant enough. Sadly, it doesn't work out when we need to make equality tests.

In Python, if we do nothing special, the `__eq__()` test will simply compare the internal object id values. These object id values are unique to each distinct object, irrespective of the attribute values.

This default behavior of objects is shown by the following example:

Equality Test Failure

```
>>> oc1= Outcome( "Any Craps", 8 )
>>> oc2= Outcome( "Any Craps", 8 )
>>> oc1 == oc2
False
>>> id(oc1)
4334572936
>>> id(oc2)
4334573272
```

Note: Exact ID values will vary.

This example shows that we can have two objects that appear equal, but don't compare as equal. They are distinct objects with the same attribute values. This makes them not equal according to the default methods inherited from `object`. However, we would like to have two of these objects test as equal.

Actually, we want more than that.

5.2.1 More than equal

We'll be creating collections of *Outcome* objects, and we may need to create sets or maps where hash codes are used in addition to the simple equality tests.

Hash Codes?

Every object has a hash code. The hash code is simply an integer. It can be a summary of the bits that make up the object. It may simply be based on the internal id value for the object. Python computes hash codes and uses these as a quick test for set membership and dictionary keys.

If two hash codes don't match, the objects can't possibly be equal. Further comparisons aren't necessary. If two hash codes do match, then it's worth the investment of time to do use the more detailed equality comparison.

As we look forward, the Python `set` and `dict` depend on a `__hash__()` method and an `__eq__()` method of each object in the collection.

Hash Code Failure

```
>>> hash(oc1)
270386794
>>> hash(oc2)
270392959
```

Note: Exact ID values will vary.

This shows that two objects that look the same to us can have distinct hash codes. Clearly, this is unacceptable, since we want to be able to create a set of *Outcome* objects without having things that look like repeats.

5.2.2 Layers of Meaning

The issue is that we have three distinct layers of meaning for comparing objects to see if they are "equal".

- Have the same hash code. We can call this "hash equality".

  This means the `__hash__(self)()` method for several objects that represent the same *Outcome* must also have the same hash code. When we put an object into a set or a dictionary, Python uses the `hash()` function which is implemented by the `__hash__()` method.

  Sometimes the hash codes are equal, but the object attributes aren't actually equal. This is called a hash collision, and it's rare but not unexpected.

  If we don't implement this, the default version isn't too useful for creating sets of our *Outcome* obejcts.

- Compare as Equal. We can call this "attribute equality".

  This means that the `__eq__()` method returns True. When we use the **==** operator, this is evaluated by using the `__eq__()` method. This must be overridden by a class to implement attribute equality.

  If we don't implement this, the default version isn't too useful for our *Outcome* obejcts.

- Are references to the same object. We can call this "identity".

  We can test that two objects are the same by using the **is** comparison between two objects. This uses the internal Python identifier for each object. The identifier is revealed by the `id()` function.

  When we use the **is** comparison, we're asserting that the two variables are references to the same underlying object. This is the identity comparison.

5.2.3 Basics of Equality

We note that each instance of *Outcome* has a distinct `Outcome.name` value, it seems simple enough to compare names. This is one sense of "equal" that seems to be appropriate.

We can define the `__eq__()` and `__ne__()` methods work two ways:

- When comparing *Outcome* and string, it will compares the `Outcome.name` attribute. This is easy, lazy and seems to work perfectly.
- When comparing *Outcome* and *Outcome*, it can compare both name and odds. This seems like over-engineering. The odds depend on the name. The name is the key, the odds are just an attribute.

Similarly, we can compute the value of `__hash__()` using only the string name, and not the odds. This seems elegantly simple to return the hash of the string name rather than compute a hash.

The definition for `__hash__()` in section 3.3.1 of the *Language Reference Manual* tells us to do the calculation using a modulus based on `sys.hash_info.width`. This is the number of bits, the actual value we want to use is We'd use `sys.hash_info.modulus`.

5.3 Looking Forward

We'll be looking at `Outcome` objects in several contexts.

- We'll have them in bins of a wheel as winning outcomes from each spin of the wheel.
- We'll have them in bets that have been placed on the table.
- They player will have some outcomes that they prefer to bet on.

We'll be comparing table bet outcomes and bin outcomes for equality. We have a solution to that, above.

We'll be creating outcome objects, too. This bumps into an interesting problem.

> **How do we maintain** *Don't Repeat Yourself* (DRY) **when creating** `Outcome` **objects?**

We don't want to include the odds every time we create an `Outcome`. Repeating the odds would violate the DRY principle.

What are some alternatives?

- **Global Outcome Objects**. We can declare global variables for the various outcomes and use those global objects as needed.

  Generally, globals *variables* are often undesirable because changes to those variables can have unexpected consequences in a large application.

  Global *constants* are no problem at all. The pool of `Outcome` instances are proper constant values used to create bins and bets. There would be a lot of them, and they would all be assigned to distinct variables. This sounds complicated.

- **Outcome Factory**. We can create a function which is a **Factory** for individual `Outcome` objects.

  When some part of the application needs an `Outcome` object, the factory will do one of two things. If the object doesn't yet exist, the **Factory** would create it, save it, and return a reference to it. When some part of the application asked for an `Outcome` which already exists, the **Factory** would return a reference to the existing object.

  This centralizes the pool of global objects into a single object, the **Factory**.

  Further, we can identify `Outcome` instances by their names, and avoid repeating the payout odds. The function would map a name of an `Outcome` the object with all of it's details.

  As a practical matter, the **Factory** could be seeded with all outcomes. The factory function is – in effect – a global pool of constant objects.

- **Singleton Outcome Class**. A **Singleton** class creates and maintains a single instance of itself. This requires that the class have a static `instance()` method that is a reference to the one-and-only instance of the class.

  This saves us from creating global variables. Instead, each class definition contains it's own private reference to the one-and-only object of that class.

  However, this has the profound disadvantage that each distinct outcome would need to be a distinct subclass of `Outcome`. This is an unappealing level of complexity. Further, it doens't solve the DRY problem of repeating the details of each Outcome.

A **Factory** seems like a good way to proceed. It can maintain a collection, and provide values from that collection. We can use class strings to identify `Outcome` objects. We don't have to repeat the odds.

We'll look forward to this in subsequent exercises. For now, we'll start with the basic class.

5.4 Outcome Design

class `Outcome`

`Outcome` contains a single outcome on which a bet can be placed.

In Roulette, each spin of the wheel has a number of `Outcome`s with bets that will be paid off.

For example, the "1" bin has the following winning `Outcome`s: "1", "Red", "Odd", "Low", "Column 1", "Dozen 1-12", "Split 1-2", "Split 1-4", "Street 1-2-3", "Corner 1-2-4-5", "Five Bet", "Line 1-2-3-4-5-6", "00-0-1-2-3", "Dozen 1", "Low" and "Column 1". All of these bets will payoff if the wheel spins a "1".

5.4.1 Fields

`Outcome.`**`name`**

Holds the name of the `Outcome`. Examples include `"1"`, `"Red"`.

`Outcome.`**`odds`**

Holds the payout odds for this `Outcome`. Most odds are stated as 1:1 or 17:1, we only keep the numerator (17) and assume the denominator is 1.

We can use `name` to provide hash codes and do equality tests.

5.4.2 Constructors

`Outcome.`**`__init__`**`(`*self*, *name*, *odds*`)`

Parameters

- **`name`** (*str*) – The name of this outcome
- **`odds`** (*int*) – The payout odds of this outcome.

Sets the instance name and odds from the parameter name and odds.

5.4.3 Methods

For now, we'll assume that we're going to have global instances of each `Outcome`. Later we'll introduce some kind of **Factory**.

`Outcome.`**`winAmount`**`(`*self*, *amount*`)` → amount

Multiply this `Outcome`'s odds by the given amount. The product is returned.

Parameters **`amount`** (*number*) – amount being bet

`Outcome.`**`__eq__`**`(`*self*, *other*`)` → boolean

Compare the `name` attributes of `self` and `other`.

Parameters **`other`** (Outcome) – Another `Outcome` to compare against.

Returns True if this name matches the other name.

Return type bool

`Outcome.`**`__ne__`**`(`*self*, *other*`)` → boolean

Compare the `name` attributes of `self` and `other`.

Parameters **`other`** (Outcome) – Another `Outcome` to compare against.

Returns True if this name does not match the other name.

Return type bool

`Outcome.`**`__hash__`**`(`*self*`)` → int

Hash value for this outcome.

Returns The hash value of the name, `hash(self.name)`.

Return type int

A hash calculation must include all of the attributes of an object that are essential to it's distinct identity.

In this case, we can return `hash(self.name)` because the odds aren't really part of what makes an outcome distinct. Each outcome is an abstraction and a string name is all that identifies them.

The definition for `__hash__()` in section 3.3.1 of the *Language Reference Manual* tells us to do the calculation using a modulus based on `sys.hash_info.width`. That value is the number of bits, the actual value we want to use is `sys.hash_info.modulus`, which is based on the width.

`Outcome.__str__(self)` → string

Easy-to-read representation of this outcome. See *Message Formatting*.

This easy-to-read String output method is essential. This should return a `String` representation of the name and the odds. A form that looks like `1-2 Split (17:1)` works nicely.

Returns String of the form *name (odds*:1).

Return type str

`Outcome.__repr__(self)` → string

Detailed representation of this outcome. See *Message Formatting*.

Returns String of the form `Outcome(`*name, odds*`)`.

Return type str

5.5 Outcome Deliverables

There are two deliverables for this exercise. Both will have Python docstrings.

- The `Outcome` class.
- Unit tests of the `Outcome` class. This can be doctest strings inside the class itself, or it can be a separate `unittest.TestCase` class.

  The unit test should create a three instances of `Outcome`, two of which have the same name. It should use a number of individual tests to establish that two `Outcome` with the same name will test true for equality, have the same hash code, and establish that the `winAmount()` method works correctly.

5.6 Message Formatting

For the very-new-to-Python, there are few variations on creating a formatted string.

Generally, we simply use something like this.

```
def __str__( self ):
    return "{name:s} ({odds:d}:1)".format_map( vars(self) )
```

This uses the built-in `vars()` function to expose the attributes of an object as a simple dictionary that maps attribute names to values.

This is similar to using the `self.__dict__` internal dictionary.

The format string uses `:s` and `:d` as detailed specifications for the values to interpolate into the string. There's a lot of flexbility in how numbers are formatted.

There's another variation that can be handy.

```
def __repr__( self ):
    return "{class_:s}({name!r}, {odds!r})".format(
        class_=type(self).__name__, **vars(self) )
```

This exposes the class name as well as the attribute values.

We've used the `!r` to request the internal representation for each attribute. For a string, it means it will be explicitly quoted.

CHAPTER

SIX

BIN CLASS

This chapter will present the design for the `Bin` class. In *Bin Analysis* we'll look at the responsibilities and collaborators for a bin.

As part of designing a Bin, we need to choose what is the most appropriate kind of collection class to use. We'll show how to do this in *Design Decision – Choosing A Collection*.

In *Wrap vs. Extend a Collection* we'll look at two principle ways to embed a collection class in an application. We'll clarify a few additional issues in *Bin Questions and Answers*.

In the *Bin Design* section we'll provide the detailed design information. In *Bin Deliverables* we'll enumerate what must be built.

6.1 Bin Analysis

The Roulette wheel has 38 bins, identified with a number and a color. Each of these bins defines a number of closely related winning `Outcome`s. At this time, we won't enumerate each of the 38 `Bin`s of the wheel and all of the winning `Outcome`s (from two to fourteen in each `Bin`); we'll save that for a later exercise.

At this time, we'll define the `Bin` class, and use this class to contain a number of `Outcome` objects.

Two of the `Bin`s have relatively few `Outcome`s. Specifically, the 0 `Bin` only contains the basic "0" `Outcome` and the "00-0-1-2-3" `Outcome` . The 00 `Bin` , similarly, only contains the basic "00" `Outcome` and the "00-0-1-2-3" `Outcome` .

The other 36 `Bin`s contain the straight bet, split bets, street bet, corner bets, line bets and various outside bets (column, dozen, even or odd, red or black, high or low) that will win if this `Bin` is selected. Each number bin has from 12 to 14 individual winning `Outcome`s.

Some `Outcome`s, like "red" or "black", occur in as many as 18 individual `Bin`s. Other `Outcome`s, like the straight bet numbers, each occur in only a single `Bin`. We will have to be sure that our `Outcome` objects are shared appropriately by the `Bin`s.

Since a `Bin` is just a collection of individual `Outcome` objects, we have to select a collection class to contain the objects.

6.2 Design Decision – Choosing A Collection

There are five basic Python types that are a containers for other objects.

- **Immutable Sequence**: `tuple`. This is a good candidate for the kind of collection we need, since the elements of a `Bin` don't change. Howver, a `tuple` allows duplicates and retains things in a specific order; we can't tolerate duplicates, and order doesn't matter.

- **Mutable Sequence**: `list`. While handy for the initial construction of the bin, this isn't really useful because the contents of a bin don't change once they have been enumerated.
- **Mutable Mapping**: `dict`. We don't need the key-to-value mapping feature at all. A map does more than we need for representing a *Bin*.
- **Mutable Set**: `set`. Duplicates aren't allowed, membership tests are fast, and there's no inherent ordering. This looks close to what we need.
- **Immutable Set**: `frozenset`. Duplicates aren't allowed, membership tests are fast, and there's no inherent ordering. There's not changing it after it's been built. This seems to be precisely what we need.

Having looked at the fundamental collection varieties, we will elect to use a `frozenset`.

How will we use this collection?

6.3 Wrap vs. Extend a Collection

There are two general ways to use a collection.

- **Wrap**. Define a class which has an attribute that holds the collection. We're wrapping an existing data structure in a new class.

  Something like this:

```
class Bin:
    def __init__(self, outcomes):
        self.outcomes= frozenset(outcomes)
```

- **Extend**. Define a class which **is** the collection. We're extending an existing data structure.

  Something like this:

```
class Bin(fronzenset):
    pass
```

Both are widely-used design techniques. The tradeoff between them isn't clear at first.

Considerations include the following:

- When we **wrap**, we'll often need to write a lot of additional methods for representation, length, comparisons, inquiries, etc.

  In some cases, we will wrap a collection specifically so that these additional methods are **not** available. We want to completely conceal the underlying data structure.

- When we **extend**, we get all of the base methods of the collection. We can add any unique features.

In this case, extending the existing data structure seems to make more sense than wrapping a `frozenset`.

6.4 Bin Questions and Answers

Why wasn't *Bin* in the design overview?

> The definition of the Roulette game did mention the 38 bins of the wheel. However, when identifying the nouns, it didn't seem important. Then, as we started designing the *Wheel* class, the description of the wheel as 38 bins came more fully into focus. Rework of the preliminary design is part of detailed design. This is the first of several instances of rework.

Why introduce an entire class for the bins of the wheel? Why can't the wheel be an array of 38 individual arrays?

There are two reasons for introducing *Bin* as a separate class: to improve the fidelity of our object model of the problem, and to reduce the complexity of the *Wheel* class. The definition of the game describes the wheel as having 38 bins, each bin causes a number of individual *Outcome*s to win. Without thinking too deeply, we opted to define the *Bin* class to hold a collection of *Outcome*s. At the present time, we can't foresee a lot of processing that is the responsibility of a *Bin*. But allocating a class permits us some flexibility in assigning responsibilities there in the future.

Additionally, looking forward, it is clear that the *Wheel* class will use a random number generator and will pick a winning *Bin*. In order to keep this crisp definition of responsibilities for the *Wheel* class, it makes sense to delegate all of the remaining details to another class.

Isn't an entire class for bins a lot of overhead?

The short answer is no, class definitions are almost no overhead at all. Class definitions are part of the compiler's world; at run-time they amount to a few simple persistent objects that define the class. It's the class instances that cause run-time overhead.

In a system where were are counting individual instruction executions at the hardware level, this additional class may slow things down somewhat. In most cases, however, the extra few instructions required to delegate a method to an internal object is offset by the benefits gained from additional flexibility.

How can you introduce Set, List, Vector when these don't appear in the problem?

We have to make a distinction between the classes that are uncovered during analysis of the problem in general, and classes are that just part of the implementation of this particular solution. This emphasizes the distinction between *the problem* as described by users and *a solution* as designed by software developers.

The collections framework is part of a solution, and only hinted at by the definition of the problem. Generally, these solution-oriented classes are parts of frameworks or libraries that came with our tools, or that we can license for use in our application. The problem-oriented classes, however, are usually unique to our problem.

Why extend a built-in data structure? Why not simply use it?

There are two reasons for extending a built-in data structure:

- We can easily add methods by extending. In the case of a *Bin*, there isn't much we want to add.
- We can easily change the underlying data structure by extending. For example, we might have a different set-like collection that also inherits from `Collections.abc.Set`. We can make this change in just one place – our class extension – and the entire application benefits from the alternative set implementation.

What about hash and equality?

We are't going to be comparing bins against each other. The default rules for equality and hash computation will work out just fine.

6.5 Bin Design

class `Bin`(*Collections.frozenset*)

Bin contains a collection of *Outcome*s which reflect the winning bets that are paid for a particular bin on a Roulette wheel. In Roulette, each spin of the wheel has a number of *Outcome*s. Example: A spin of 1, selects the "1" bin with the following winning *Outcome*s: "1" , "Red" , "Odd" , "Low" , "Column 1" , "Dozen 1-12" , "Split 1-2" , "Split 1-4" , "Street 1-2-3" , "Corner 1-2-4-5" , "Five Bet" , "Line 1-2-3-4-5-6" , "00-0-1-2-3" , "Dozen 1" , "Low" and "Column 1" . These are collected into a single *Bin* .

6.5.1 Fields

Since this is an extension to the existing `frozenset`, we don't need to define any additional fields.

6.5.2 Constructors

We don't really need to write any more specialized constructor method.

We'd use this as follows:

Python Bin Construction

```
five= Outcome( "00-0-1-2-3", 6 )
zero= Bin( [Outcome("0",35), five] )
zerozero= Bin( [Outcome("00",35), five] )
```

1. `zero` is based on references to two objects: the "0" *Outcome* and the "00-0-1-2-3" *Outcome*.
2. `zerozero` is based on references to two objects: the "00" *Outcome* and the "00-0-1-2-3" *Outcome*.

6.5.3 Methods

We don't really **need** to write any more specialized methods.

How do we accumulate several outcomes in a single *Bin*?

1. Create a simple list, tuple, or set as an interim structure.
2. Create the *Bin* from this.

We might have something like this:

```
>>> bin1 = Bin( {outcome1, outcome2, outcome3} )
```

We created an interim set object and built the final *Bin* from that collection object.

6.6 Bin Deliverables

There are two deliverables for this exercise. Both should have Python docstrings.

- The *Bin* class.
- A class which performs a unit test of the *Bin* class. The unit test should create several instances of *Outcome*, two instances of *Bin* and establish that *Bin*s can be constructed from the *Outcome*s.

  Programmers who are new to OO techniques are sometimes confused when reusing individual *Outcome* instances. This unit test is a good place to examine the ways in which object *references* are shared. A single *Outcome* object can be referenced by several *Bin*s. We will make increasing use of this in later sections.

CHAPTER SEVEN

WHEEL CLASS

This chapter builds on the previous two chapters, creating a more complete composite object from the *Outcome* and *Bin* classes we have already defined. In *Wheel Analysis* we'll look at the responsibilities of a wheel and it's collaboration.

In the *Wheel Design* we'll provide the detailed design information. In the *Test Setup* we'll address some considerations for testing a class which has random behavior. In *Wheel Deliverables* we'll enumerate what must be built.

7.1 Wheel Analysis

The wheel has two responsibilities: it is a container for the *Bin*s and it picks one *Bin* at random. We'll also have to look at how to initialize the various *Bin*s that comprise a standard Roulette wheel.

In *The Container Responsibility* we'll look at the container aspect in detail.

In *The Random Bin Selection Responsibility* we'll look at the random selection aspects.

Based on this, the *Constructing a Wheel* section provides a description of how we can build the Wheel instance.

7.1.1 The Container Responsibility

Since the *Wheel* is 38 *Bin*s, it is a collection. We can review our survey of available collections in *Design Decision – Choosing A Collection* for some guidance here.

In this case, the choice of the winning *Bin* will be selected by a random numeric index. We need some kind of sequential collection.

> This makes an immutable `tuple` very appealing. This is a subclass of `collections.abc.Sequence` and has the features we're looking for.

One consequence of using a sequential collection is that we have to choose an index for the various *Bin*s.

The index values of 1 to 36 are logical mappings to *Bin*s. The *Bin* at index 1 would contain `Outcome("1", 35)` among several others. The *Bin* at index 2 would contain `Outcome("2", 35)`. And so on through 36.

We have a small problem, however, with 0 and 00: we need two separate indexes. While 0 is a valid index, what do we do with 00?

Enumerate some possible solutions before reading on.

Since the index of the *Bin* doesn't have any significance at all, we can assign the *Bin* that has the 00 *Outcome* to position 37. It doesn't actually matter because we'll never really use the index for any purpose other than random selection.

7.1.2 The Random Bin Selection Responsibility

In order for the *Wheel* to select a *Bin* at random, we'll need a random number from 0 to 37 that we can use an an index.

The `random` module offers a `Random.choice()` function which picks a random value from a sequence. This is ideal for returning a randomly selected *Bin* from our list of *Bin*s.

Testability. Note that testing a class using random numbers isn't going to be easy. To do testing properly, we'll need to create a non-random random number generator that we can use in place of the built-in random number generator.

To create a non-random random-number generator, we can do something like the following.

1. When testing, we can then set a specific seed value that will generate a known sequence of values.
2. Create a mock for the random number generator that returns a known, fixed sequence of values. We can leverage the `unittest.mock` module for this.

We'll address this in detail in *Review of Testability*. For now, we'll suggest using the first technique – set a specific seed value.

7.1.3 Constructing a Wheel

Each instance of *Bin* has a list of *Outcome*s. The zero ("0") and double zero ("00") *Bin* s only have two *Outcome*s. The other numbers have anywhere from twelve to fourteen *Outcome*s.

Clearly, there's quite a bit of complexity in building some of the bins.

Rather than dwell on these algorithms, we'll apply a common OO principle of *deferred binding*. We'll build a very basic wheel first and work on the bin-building algorithms later.

It's often simplest to build a class incrementally. This is an example where the overall structure is pretty simple, but some details are rather complex.

7.2 Wheel Design

class Wheel

Wheel contains the 38 individual bins on a Roulette wheel, plus a random number generator. It can select a *Bin* at random, simulating a spin of the Roulette wheel.

7.2.1 Fields

`Wheel.`**`bins`**

Contains the individual *Bin* instances.

This is a `tuple` of 38 elements. This can be built with `tuple( Bin() for i in range(38) )`

`Wheel.`**`rng`**

A random number generator to use to select a *Bin* from the *bins* collection.

Because of the central importance of this particular source of randomness, it seems sensible to isolate it from any other processing that might need random numbers. We can then set a seed value using `os.urandom()` or specific values for testing.

7.2.2 Constructors

`Wheel.__init__(self)`
Creates a new wheel with 38 empty *Bin*s. It will also create a new random number generator instance.

At the present time, this does not do the full initialization of the *Bin*s. We'll rework this in a future exercise.

7.2.3 Methods

`Bin.addOutcome(number, outcome)`
Adds the given *Outcome* to the *Bin* with the given number.

Parameters

- **bin** (*int*) – bin number, in the range zero to 37 inclusive.
- **outcome** (Outcome) – The Outcome to add to this Bin

`Bin.next()` → Bin
Generates a random number between 0 and 37, and returns the randomly selected *Bin*.

The `Random.choice()` function of the `random` module will select one of the available *Bin* s from the `bins` list.

Returns A Bin selected at random from the wheel.

Return type *Bin*

`Bin.get(bin)` → Bin
Returns the given *Bin* from the internal collection.

Parameters **bin** (*int*) – bin number, in the range zero to 37 inclusive.

Returns The requested Bin.

Return type *Bin*

7.3 Test Setup

We need a controlled kind of random number generation for testing purposes. This is done with tests that look like the following:

Test Outline

```
class GIVEN_Wheel_WHEN_next_THEN_random_choice(unittest.TestCase):
    def setUp(self):
        self.wheel= Wheel()
        self.wheel.rng.seed(1)
    def runTest(self):
        etc.
```

The values delivered from this seeded random number generator can be seen from this experiment.

Fixed pseudo-random sequence

```
>>> x = random.Random()
>>> x.seed(1)
>>> [x.randint(0,37) for i in range(10)]
[8, 36, 4, 16, 7, 31, 28, 30, 24, 13]
```

This allows us to predict the output from the `Wheel.next()` method.

7.4 Wheel Deliverables

There are three deliverables for this exercise. The new class and the unit test will have Python docstrings.

- The *Wheel* class.
- A class which performs a unit test of building the *Wheel* class. The unit test should create several instances of *Outcome*, two instances of *Bin*, and an instance of *Wheel*. The unit test should establish that *Bin*s can be added to the *Wheel*.
- A class which tests the Wheel class by selecting "random" values from a *Wheel* object using a fixed seed value.

CHAPTER

EIGHT

BIN BUILDER CLASS

We'll look at the question of filling in the Outcomes in each Bin of the Wheel. The *Bin Builder Analysis* section will address the various outcomes in details.

In *Bin Builder Algorithms* we'll look at eight algorithms for allocating appropriate outcomes to appropriate bins of the wheel.

The *BinBuilder Design* section will present the detailed design for this class. In *Bin Builder Deliverables* we'll define the specific deliverables.

In *Internationalization and Localization* we'll identify some considerations for providing local language names for the outcomes.

8.1 Bin Builder Analysis

It is clear that enumerating each `Outcome` in the 38 `Bin`s by hand is a tedious undertaking. Most `Bin`s contain about fourteen individual `Outcome`s. We need a one-time-only algorithm to do this job.

It is often helpful to create a class that is used to build an instance of another class. This is a design pattern sometimes called a **Builder**. We'll design an object that builds the various `Bin`s and assigns them to the `Wheel`. This will fill the need left open in the *Wheel Class*.

Additionally, we note that the complex algorithms to construct the `Bin`s are only tangential to the operation of the `Wheel` object. Because these are not essential to the design of the `Wheel` class, we find it is often helpful to segregate the rather complex builder methods into a separate class.

The `BinBuilder` class will have a method that enumerates the contents of each of the 36 number `Bin`s, building the individual `Outcome` instances. We can then assign these `Outcome` objects to the `Bin`s of a `Wheel` instance. We will use a number of steps to create the various types of `Outcome`s, and depend on the `Wheel` to assign each `Outcome` object to the correct `Bin`.

8.1.1 The Roulette Outcomes

Looking at the *Available Bets in Roulette* gives us a number of geometric rules for determining the various `Outcome`s that are combinations of individual numbers. These rules apply to the numbers from one to thirty-six. A different – and much simpler – set of rules applies to 0 and 00. First, we'll survey the table geometry, then we'll develop specific algorithms for each kind of bet.

- **Split Bets**. Each number is adjacent to two, three or four other numbers. The four corners (1, 3, 34, and 36) only participate in two split bets: 1-2 and 1-4, 2-3 and 3-6, 34-35 and 31-34, 35-36 and 33-36.

  The center column of numbers (5, 8, 11, ..., 32) each participate in four split bets with the surrounding numbers.

  The remaining "edge" numbers participate in three split bets.

While this is moderately complex, the bulk of the layout (from 4 to 32) can be handled with a simple rule to distinguish the center column from the edge columns. The ends (1, 2, 3, and 34, 35, 36) are a little more complex.

- **Street Bets**. Each number is a member of one of the twelve street bets.
- **Corner Bets**. Each number is a member of one, two or four corner bets. As with split bets, the bulk of the layout can be handled with a simple rule to distinguish the column, and hence the "corners".

  A number in the center column (5, 8, 11, ..., 32) is a member of four corners.

  All of the numbers along an edge are members of two corners. For example, 4 is part of 1-2-4-5, and 4-5-7-8.

  At the ends, 1, 3, and 34, 36, we see outcomes that members of just one corner each.
- **Line Bets**. Six numbers comprise a line; each number is a member of one or two lines. The ends (1, 2, 3 and 34, 35, 36) are each part of a single line. The remaining 10 rows are each part of two lines.
- **Dozen Bets**. Each number is a member of one of the three dozens. The three ranges are from 1 to 12, 13 to 24 and 25 to 36, making it very easy to associate numbers and ranges.
- **Column Bets**. Each number is a member of one of the three columns. Each of the columns has a number numeric relationship. The values are $3c + 1$, $3c + 2$, and $3c + 3$, where $0 \leq c < 12$.
- **The Even-Money Bets**. These include Red, Black, Even, Odd, High, Low. Each number on the layout will be associated with three of the six possible even money `Outcome`s.

  One way to handle these is to create the six individual `Outcome` instances. For each number, *n*, we can write a suite of if-statements to determine which of the `Outcome` objects are associated with the given number.

The `Bin`s for zero and double zero are just special cases. Each of these `Bin`s has a straight number bet `Outcome`, plus the "Five Bet" `Outcome` (00-0-1-2-3, which pays 6:1).

One other thing we'll probably want are handy names for the various kinds of odds. We might want to define a collection of constants for this.

While can define an `Outcome` as `Outcome( "Number 1", 35 )`, this is a little opaque. A slightly nicer form is `Outcome( "Number 1", RouletteGame.StraightBet )`.

The payouts are specific to a game, not general features of all `Outcome`s. They properly belong in some kind of game related class. We haven't designed the game yet, but we can provide a simplified class which only contains these odds definitions.

8.2 Bin Builder Algorithms

This section provides the algorithms for nine kinds of bets.

Note that we're going to be accumulating sets (or lists) of individual `Outcome` objects. These are interim objects that will be used to create the final `Bin` objects which are assigned to the `Wheel`.

We'll be happiest using the core Python `set` structure to accumulate these collections.

8.2.1 Generating Straight Bets

Straight bet `Outcomes` are the easiest to generate.

For All Numbers. For each number, n, such that $1 \leq n < 37$:

Create Outcome. Create an `Outcome` from the number, *n*, with odds of 35:1.

Assign to Bin. Assign this new `Outcome` to `Bin` *n*.

Zero. Create an *Outcome* from the "0" with odds of 35:1. Assign this to the *Bin* at index 0 in the *Wheel*.

Double Zero. Create an *Outcome* from the "00" with odds of 35:1. Assign this to *Bin* at index 37 in the *Wheel*.

8.2.2 Generating Split Bets

Split bet `Outcomes` are more complex because of the various cases: corners, edges and down-the-middle.

We note that there are two kinds of split bets:

- **left-right pairs**. These pairs all have the form $\{n, n+1\}$.
- **up-down paris**. These paris have the form $\{n, n+3\}$.

We can look at the number 5 as being part of 4 different pairs: $\{4, 4+1\}$, $\{5, 5+1\}$, $\{2, 2+3\}$, $\{5, 5+3\}$. The corner number 1 is part of 2 split bets: $\{1, 1+1\}$, $\{1, 1+3\}$.

We can generate the "left-right" split bets by iterating through the left two columns; the numbers 1, 4, 7, ..., 34 and 2, 5, 8, ..., 35.

For All Rows. For each row, r, where $0 \leq r < 12$:

First Column Number. Set $n \leftarrow 3r + 1$. This will create values 1, 4, 7, ..., 34.

Column 1-2 Split. Create a $\{n, n+1\}$ split *Outcome* with odds of 17:1.

Assign to Bins. Associate this object with two *Bin*s: n and $n+1$.

Second Column Number. Set $n \leftarrow 3r + 2$. This will create values 2, 5, 8, ..., 35.

Column 2-3 Split. Create a $\{n, n+1\}$ split *Outcome*.

Assign to Bins. Associate this object to two *Bin*s: n and $n+1$.

A similar algorithm must be used for the numbers 1 through 33, to generate the "up-down" split bets. For each number, n, we generate a $\{n, n+3\}$ split bet. This *Outcome* belongs to two *Bin*s: n and $n+3$.

8.2.3 Generating Street Bets

Street bet `Outcomes` are very simple.

We can generate the street bets by iterating through the twelve rows of the layout.

For All Rows. For each row, r, where $0 \leq r < 12$:

First Column Number. Set $n \leftarrow 3r + 1$. This will create values 1, 4, 7, ..., 34.

Street. Create a $\{n, n+1, n+2\}$ street *Outcome* with odds of 11:1.

Assign to Bins. Associate this object to three *Bin*s: $n, n+1, n+2$.

8.2.4 Generating Corner Bets

Corner bet `Outcomes` are as complex as split bets because of the various cases: corners, edges and down-the-middle.

Each corner has four numbers, $\{n, n+1, n+3, n+4\}$. This is two numbers in the same row, and two numbers in the next higher row.

We can generate the corner bets by iterating through the numbers 1, 4, 7, ..., 31 and 2, 5, 8, ..., 32. For each number, n, we generate a corner bets. This *Outcome* object belongs to four *Bin*s.

We generate corner bets by iterating through the various corners based on rows and columns. There is room for two corners within the three columns of the layout: one corner starts at column 1 and the other corner starts at column 2. There is room for 11 corners within the 12 rows of the layout.

For All Lines Between Rows. For each row, r, where $0 \leq r < 11$:

First Column Number. Set $r \leftarrow 3r + 1$. This will create values $1, 4, 7, ..., 31$.

Column 1-2 Corner. Create a $\{n, n + 1, n + 3, n + 4\}$ corner *Outcome* with odds of 8:1.

Assign to Bins. Associate this object to four *Bin*s: $n, n + 1, n + 3, n + 4$.

Second Column Number. Set $n \leftarrow 3r + 2$. This will create values $2, 5, 8, ..., 32$.

Column 2-3 Corner. Create a $\{n, n + 1, n + 3, n + 4\}$ corner *Outcome* with odds of 8:1.

Assign to Bins. Associate this object to four *Bin*s: $n, n + 1, n + 3, n + 4$.

8.2.5 Generating Line Bets

Line bet `Outcomes` are similar to street bets. However, these are based around the 11 lines between the 12 rows.

For lines s numbered 0 to 10, the numbers on the line bet can be computed as follows: $3s + 1, 3s + 2, 3s + 3, 3s + 4, 3s + 5, 3s + 6$. This *Outcome* object belongs to six individual *Bin*s.

For All Lines Between Rows. For each row, r, where $0 \leq r < 11$:

First Column Number. Set $n \leftarrow 3r + 1$. This will create values $1, 4, 7, ..., 31$.

Line. Create a $\{n, n + 1, n + 2, n + 3, n + 4, n + 5\}$ line *Outcome* withs odds of 5:1.

Assign to Bins. Associate this object to six *Bin*s: $n, n + 1, n + 2, n + 3, n + 4, n + 5$.

8.2.6 Generating Dozen Bets

Dozen bet `Outcomes` require enumerating all twelve numbers in each of three groups.

For All Dozens. For each dozen, d, where $0 \leq d < 3$:

Create Dozen. Create an *Outcome* for dozen $d + 1$ with odds of 2:1.

For All Numbers. For each number, m, such that $0 \leq m < 12$:

Assign to Bin. Associate this object to *Bin* $12d + m + 1$.

8.2.7 Generating Column Bets

Column bet `Outcomes` require enumerating all twelve numbers in each of three groups. While the outline of the algorithm is the same as the dozen bets, the enumeration of the individual numbers in the inner loop is slightly different.

For All Columns. For each column, c, such that $0 \leq c < 3$:

Create Column. Create an *Outcome* for column $c + 1$ with odds of 2:1.

For All Rows. For each row, r, where $0 \leq r < 12$:

Assign to Bin. Associate this object to *Bin* $3r + c + 1$.

8.2.8 Generating Even-Money Bets

The even money bet `Outcomes` are relatively easy to generate.

Create the Red outcome, with odds of 1:1.

Create the Black outcome, with odds of 1:1.

Create the Even outcome, with odds of 1:1.

Create the Odd outcome, with odds of 1:1.

Create the High outcome, with odds of 1:1.

Create the Low outcome, with odds of 1:1.

For All Numbers. For each number, n, such that $1 \leq n < 37$:

Low? If $1 \leq n < 19$, associate the **Low** `Outcome` with `Bin` n.

High? Otherwise, $19 \leq n < 37$, associate the **High** `Outcome` with `Bin` n.

Even? If $n \mod 2 = 0$, associate the **Even** `Outcome` with `Bin` n.

Odd? Otherwise, $n \mod 2 \neq 0$, associate the **Odd** `Outcome` with `Bin` n.

Red? If $n \in \{1, 3, 5, 7, 9, 12, 14, 16, 18, 19, 21, 23, 25, 27, 30, 32, 34, 36\}$, associate the **Red** `Outcome` with `Bin` n.

Black? Otherwise, associate the **Black** `Outcome` with `Bin` n.

8.3 BinBuilder Design

class `BinBuilder`

`BinBuilder` creates the `Outcome`s for all of the 38 individual `Bin` on a Roulette wheel.

8.3.1 Constructors

`BinBuilder.__init__`(*self*)

Initializes the `BinBuilder`.

8.3.2 Methods

`BinBuilder.buildBins`(*self*, *wheel*)

Creates the `Outcome` instances and uses the `addOutcome()` method to place each `Outcome` in the appropropriate `Bin` of `wheel`.

Parameters **`wheel`** (Wheel) – The Wheel with Bins that must be populated with Outcomes.

There should be separate methods to generate the straight bets, split bets, street bets, corner bets, line bets, dozen bets and column bets, even money bets and the special case of zero and double zero.

Each of the methods will be relatively simple and easy to unit test. Details are provided in *Bin Builder Algorithms*.

8.4 Bin Builder Deliverables

There are three deliverables for this exercise. The new classes should have meaningful Python docstrings.

- The `BinBuilder` class.

- A class which performs a unit test of the *BinBuilder* class. The unit test invoke each of the various methods that create *Outcome* instances.
- Rework the unit test of the *Wheel* class. The unit test should create and initialize a *Wheel*. It can use the `Wheel.getBin()` method to check selected *Bin*s for the correct *Outcome*s.

8.5 Internationalization and Localization

An an advanced topic, we need to avoid *hard coding* the names of the bets. Python offers extensive tools for localization (l10n) of programs. Since Python works with Unicode strings, it supports non-Latin characters, supporting internationalization (i18n), also.

Unicode has a number of characters for a variety of games. It does not have the Roulette numbers – properly color-coded – as glyphs.

CHAPTER NINE

ROULETTE BET CLASS

In addition to the design of the `Bet` class, this chapter also presents some additional questions and answers on the nature of an object, identity and state change. This continues some of the ideas from *Design Decision – Object Identity*.

In *Roulette Bet Analysis* we'll look at the details of a Bet. This will raise a question of how to identify the Outcome associated with a Bet.

We'll look at object identiy in *Design Decision – Create or Locate an Outcome*.

We'll provide some additional details in *Roulette Bet Questions and Answers*.

The *Roulette Bet Design* section will provide detailed design for the Bet class. In *Roulette Bet Deliverables* we'll enumerate the deliverables for this chapter.

9.1 Roulette Bet Analysis

A `Bet` is an amount that the player has wagered on a specific `Outcome`. This class has the responsibilty for maintaining an association between an amount, an `Outcome`, and a specific `Player`.

The general scenario is to have the `Player` construct a number of `Bet` instances. The `Wheel` is spun to select a winning `Bin`. Then each of the `Bet` objects will be checked to see if the hoped-for `Outcome` is in the actual set of `Outcome`s in the winning `Bin`.

Each winning `Bet` has an `Outcome` that matches one in the winning `Bin`. The winning bets will return money to the `Player`. All other bets aren't in the winning `Bin`; they are losers, which removes the money from the `Player`.

We have a design decision to make. Do we create a fresh `Outcome` object with each `Bet` or do we locate an existing `Outcome` object?

9.2 Design Decision – Create or Locate an Outcome

Building a `Bet` involves two parts: an `Outcome` and an amount. The amount is just a number. The `Outcome`, however, includes two parts: a name and payout odds.

We looked at this issue in *Looking Forward*. We'll revisit this design topic in some more depth here. The bottom line is this.

We don't want to create an `Outcome` object as part of constructing a `Bet` object. Here's what it might look like to place a $25 bet on Red:

Bad Idea

```
my_bet= Bet(Outcome("red", 1), 25)
```

The *Bet* includes an *Outcome* and an amount. The *Outcome* includes a name and the payout odds.

One unfortunate feature of this is that we have repeated the odds when creating an *Outcome* object. This violates the DRY principle.

We want to get a complete *Outcome* from just the name of the outcome. The will prevent repeating the odds information.

Problem. How do we locate an existing *Outcome* object?

Do we use a collection or a global variable? Or is there some other approach?

Forces. There are several parts to this design.

- We need to pick a collection. When we look at the collections (see *Design Decision – Choosing A Collection*) we can see that a Map from name to complete *Outcome* instance is ideal. This helps us associate a *Bet* with an *Outcome* given just the name of the *Outcome*.
- We need to identify some global object that builds the collection of distinct *Outcome*s.
- We need to identify some global object that can maintain the collection of *Outcome*s for use by the Player in building Bets.

If the builder and maintainer are the same object, then things would be somewhat simpler because all the responsibilities would fall into a single place.

We have several choices for the kind of global object we would use.

- **Variable**. We can define a variable which is a global map from name to *Outcome* instance. This could be an instance of the built-in `dict` class or some other mapping object. It could be an instance of a class we've designed that maps names to *Outcome* instances.

  A truly *variable* global is a dangerous thing. An immutable global object, however, is a useful idea.

  We might have this:

  #### Global Mapping

```
>>> some_map["Red"]
Outcome('Red', 1)
```

- **Function**. We can define a **Factory** function which will produce an *Outcome* instance as needed.

  #### Factory Function

```
>>> some_factory("Red")
Outcome('Red', 1)
```

- **Class**. We can define class-level methods for emitting an instance of *Outcome* based on a name. We could, for example, add methods to the *Outcome* class which retrieved instances from a class-level mapping.

  #### Class Method

```
>>> Outcome.getInstance("Red")
Outcome('Red', 1)
```

After creating the `BinBuilder`, we can see that this fits the overall **Factory** design for creating `Outcome` instances.

However, the class:*BinBuilder* doesn't – currently – have a handy mapping so that we can look up an `Outcome` based on the name of the outcome. Is this the right place to do the lookup?

It would look like this:

BinBuilder as Factory

```
>>> theBinBuilder.getOutcome("Red")
Outcome('Red', 1)
```

What about the `Wheel`?

Wheel as Factory

```
>>> theWheel.getOutcome("Red")
Outcome('Red', 1)
```

Alternative Solutions. We have a number of potential ways to gather all `Outcome` objects that were created by the `BinBuilder`.

- Clearly, the `BinBuilder` can create the mapping from name to each distinct `Outcome`. To do this, we'd have to do several things.

  First, we expand the `BinBuilder` to keep a simple Map of the various `Outcome`s that are being assigned via the `Wheel.add()` method.

  Second, we would have to add specific `Outcome` getters to the `BinBuilder`. We could, for example, include a `getOutcome()` method that returns an `Outcome` based on its name.

  Here's what it might look like in Python.

```
    class BinBuilder( object ):
        ...
        def apply( self, outcome, bin, wheel ):
            self.all_outcomes.add( outcome )
            wheel.add( bin, outcome )
        def getOutcome( self, name ):
            ...
```

- Access the `Wheel`. A better choice is to get `Outcome` obects from the `Wheel`. To do this, we'd have to do several things.

  First, we expand the `Wheel` to keep a simple Map of the various `Outcome`s created by the `BinBuilder`. This Map would be maintained by the `Wheel.add()`.

  Second, we would have to add specific `Outcome` getters to the `Wheel`. We could, for example, include a `getOutcome()` method that returns an `Outcome` based on its name.

  We might write a method function like the following to Wheel.

```
    class Wheel( object ):
        ...
        def add( self, bin, outcome ):
            self.all_outcomes.add( outcome )
            ...
        def getOutcome( self, name ):
            return set( [ oc for oc in self.all_outcomes if oc.name.lower().contains( name.lower() )
```

Solution. The allocation of responsibility seems to be a toss-up. We can see that the amount of programming is almost identical. This means that the real question is one of clarity: which allocation more clearly states our intention?

The `Wheel` is a first-class part of the game of Roulette. It showed up in our initial noun analysis. The `BinBuilder` was an implementation convenience to separate the one-time construction of the `Bin`s from the overall work of the `Wheel`.

Since `Wheel` is a first-class part of the problem, we should augment the `Wheel` to keep track of our individual `Outcome` objects by name.

9.3 Roulette Bet Questions and Answers

Why not update each `Outcome` with the amount of the bet?

> We are isolating the static definition of the `Outcome` from the presence or absence of an amount wagered. Note that the `Outcome` is shared by the wheel's `Bin` s, and the available betting spaces on the `Table`, possibly even the `Player` class. Also, if we have multiple `Player`, then we need to distinguish bets placed by the individual players.
>
> Changing a field's value has an implication that the thing has changed state. In Roulette, there isn't any state change in the definition of an `Outcome`. However, when we look at Craps, we will see that changes in the game's state will enable and disable whole sets of `Outcome`s.

Does an individual bet really have unique identity? Isn't it just anonymous money?

> Yes, the money is anonymous. In a casino, the chips all look alike. However, the placement of the bet, really does have unique identity. A `Bet` is owned by a particular player, it lasts for a specific duration, it has a final outcome of won or lost. When we want to create summary statistics, we could do this by saving the individual `Bet` objects. We could update each `Bet` with a won or lost indicator, then we can total the wins and losses.
>
> This points up another reason why we know a `Bet` is an object in its own right: it changes state. A bet that has been placed can change to a bet that was won or a bet that was lost.

9.4 Roulette Bet Design

class `Bet`

`Bet` associates an amount and an `Outcome`. In a future round of design, we can also associate a `Bet` with a `Player`.

9.4.1 Fields

`Bet.`**`amountBet`**
> The amount of the bet.

`Bet.`**`outcome`**
> The `Outcome` on which the bet is placed.

9.4.2 Constructors

`Bet.`**`__init__`**(*self*, *amount*, *outcome*)

> **Parameters**
>
> - **`amount`** (*int*) – The amount of the bet.
> - **`outcome`** (Outcome) – The `Outcome` we're betting on.

Create a new Bet of a specific amount on a specific outcome.

For these first exercises, we'll omit the `Player`. We'll come back to this class when necessary, and add that capability back in to this class.

9.4.3 Methods

`Bet.`**`winAmount`**`(`*self*`)` → int

Returns amount won

Return type int

Uses the `Outcome`'s `winAmount` to compute the amount won, given the amount of this bet. Note that the amount bet must also be added in. A 1:1 outcome (e.g. a bet on Red) pays the amount bet plus the amount won.

`Bet.`**`loseAmount`**`(`*self*`)` → int

Returns amount lost

Return type int

Returns the amount bet as the amount lost. This is the cost of placing the bet.

`Bet.`**`__str__`**`(`*self*`)` → str

Returns string representation of this bet with the form "*amount* on *outcome*"

Return type str

Returns a string representation of this bet. Note that this method will delegate the much of the work to the `__str__()` method of the `Outcome`.

`Bet.`**`__repr__`**`(`*self*`)` → str

Returns string representation of this bet with the form "Bet(*amount*, *outcome*)"

Return type str

9.5 Roulette Bet Deliverables

There are four deliverables for this exercise. The new classes will have Python docstrings.

- The expanded `Wheel` class which creates a mapping of string name to `Outcome`.
- Expanded unit tests of `Wheel` that confirm that the mapping is being built correctly.
- The `Bet` class.
- A class which performs a unit test of the `Bet` class. The unit test should create a couple instances of `Outcome`, and establish that the `winAmount()` and `loseAmount()` methods work correctly.

CHAPTER

TEN

ROULETTE TABLE CLASS

This section provides the design for the `Table` to hold the bets. In the section *Roulette Table Analysis* we'll look at the table as a whole.

One of the table's resposibilities seems to be to validate bets. In *InvalidBet Exception Design* we'll look at how we can design an appropriate exception.

In *Roulette Table Design* we'll look at the details of creating the table class. Then, in *Roulette Table Deliverables* we'll enumerate the deliverables for this chapter.

10.1 Roulette Table Analysis

We'll look at several topics in detail as part of the analysis of the table.

- *Winning vs. Losing* explores how we handle the payment of a bet and the receipt of the winnings.
- In *Container Implementation* we'll look at how we store bets.
- We'll look at casino betting limits in *Table Limits*.
- The *Adding and Removing Bets* section discusses some additional details of how the bet container must work.

The `Table` has the responsibility to keep the `Bet`s created by the `Player`. Additionally, the house imposes *table limits* on the minimum amount that must be bet and the maximum that can be bet. Clearly, the `Table` has all the information required to evaluation these conditions.

Note: Betting Constraints

Casinos prevent the *Martingale* betting system from working by imposing a table limit on each game. To cover the cost of operating the table game, the casino also imposes a minimum bet. Typically, the maximum is a multiplier of the minimum bet, often in the range of 10 to 50; a table with a $5 minimum might have a $200 limit, a $10 minimum may have only a $300 limit.

It isn't clear where the responsibility lies for determining winning and losing bets. The money placed on `Bet`s on the `Table` is "at risk" of being lost. If the bet is a winner, the house pays the `Player` an amount based on the `Outcome` odds and the `Bet` amount. If the bet is a loser, the amount of the `Bet` is forfeit by the `Player`. Looking forward to stateful games like Craps, we'll place the responsibility for determining winners and losers with the game, and not with the `Table` object.

We'll wait, then, until we write the game to finalize paying winning bets and collecting losing bets.

10.1.1 Winning vs. Losing

Another open question is the timing of the payment for the bet from the player's stake. In a casino, the payment to the casino – effectively – happens when the bet is placed on the table. In our Roulette simulation, this is a subtlety

that doesn't have any practical consequences. We could deduct the money as part of *Bet* creation, or we could deduct the money as part of resolving the spin of the wheel. In other games, however, there may several events and several opportunities for placing additional bets. For example, splitting a hand in blackjack, or placing additional odds bets in Craps.

Because we can't allow a player to bet more than their stake, we should deduct the payment as the *Bet* is created.

A consequence of this is a change to our definition of the *Bet* class. We don't need to compute the amount that is lost. We're not going to deduct the money when the bet resolved, we're going to deduct the money from the *Player*'s stake as part of creating the *Bet*. This will become part of the design of *Player* and *Bet*.

Looking forward a little, a stateful game like Craps will introduce a subtle distinction that may be appropriate for a future subclass of *Table*. When the game is in the **point off** state, some of the bets on the table are not allowed, and others become inactive. When the game is in the **point on** state, all bets are allowed and active. In Craps parlance, some bets are "not working" or "working" depending on the game state. This does not apply to the version of *Table* that will support Roulette.

10.1.2 Container Implementation

A *Table* is a collection of *Bet*s. We need to choose a concrete class for the collection of the bets. We can review the survey of collections in *Design Decision – Choosing A Collection* for some guidance here.

In this case, the bets are placed in no particular order, and are simply visited in an arbitrary order for resolution. Bets don't have specific names.

Since the number of bets varies, we can't use a Python `tuple`; a `list` will have to do. We could also use a set because duplicate bets don't make any sense.

10.1.3 Table Limits

Table limits can be checked by providing a public method `isValid()` that compares the total of all existing *Bet*s against the table limit. This should be used by the *Game* to confirm that bets are legal before proceeding.

In the unlikely event of the *Player* object creating an illegal *Bet*, this will raise an exception to indicate that we have a design error that was not detected via unit testing. This should be a subclass of `Exception` that has enough information to debug the problem with the *Player* that attempted to place the illegal bet.

Each individual bet must meet the table minimum. This is a separate rule that can be checked each time a bet is placed.

10.1.4 Adding and Removing Bets

A *Table* contains `Bets`. Instances of *Bet* are added by a *Player*. Later, `Bets` will be removed from the *Table* by the *Game*. When a bet is resolved, it must be deleted. Some games, like Roulette resolve all bets with each spin. Other games, like Craps, involve multiple rounds of placing and resolving some bets, and leaving other bets in play.

For bet deletion to work, we have to provide a method to remove a *Bet* instance. When we look at game and bet resolution we'll return to bet deletion. It's import not to over-design this class at this time; we will often add features as we develop designs for additional use cases.

10.2 InvalidBet Exception Design

We'll raise an exception for an invalid bet. This is, in general, better than having a method which returns True for a valid bet and False for an invalid bet.

It's better because we can simply place the bet, assuming that it is valid. The processing continues along this happy path.

If the bet is not valiid, the exception interrupts processing. The only way to get an invalid bet in Roulette is to have a badly damaged implementation of the *Player* class. We really need to have the application break in a catastrophic manner.

exception InvalidBet

InvalidBet is raised when the *Player* attempts to place a bet which exceeds the table's limit.

This class simply inherits all features of its superclass.

10.3 Roulette Table Design

class Table

Table contains all the *Bet* s created by the *Player*. A table also has a betting limit, and the sum of all of a player's bets must be less than or equal to this limit. We assume a single *Player* in the simulation.

10.3.1 Fields

Table.**limit**

This is the table limit. The sum of the bets from a *Player* must be less than or equal to this limit.

Table.**minimum**

This is the table minimum. Each individual bet from a *Player* must be greate than this limit.

Table.**bets**

This is a `list` of the *Bet*s currently active. These will result in either wins or losses to the *Player*.

10.3.2 Constructors

Table.**Table**()

Creates an empty `list` of bets.

10.3.3 Methods

Table.**placeBet**(*self*, *bet*)

Parameters **bet** (Bet) – A Bet instance to be added to the table.

Raises InvalidBet

Adds this bet to the list of working bets.

We'll reserve the idea of raising an exception for an individual invalid bet. This is a rare circumstance, and indicates a bug in the *Player* more than anything else.

We might, for example, confirm that the Bet's *Outcome* exists in one of the *Bin*s. We might check that the bet amount is greater than or equal to the table minimum. We might also check the upper limit on betting will be honored by all existing bets plus this new bet.

It's not **necessary** to validate each bet as they're being placed. It's only necessary to validate the bets once prior to spinning the wheel. This is a function of the Game, and a separate interface is available for this.

For an interactive game – not a simulation – we would want to validate each bet prior to accepting it so that we can provide an immediate response to the player that the potential bet is invalid. In this case, we'd leave the table untouched when a bad bet is offered.

`Table.__iter__()` → iter

Returns an iterator over the available list of *Bet* instances. This simply returns the iterator over the list of *Bet* objects.

Note that we need to be able remove Bets from the table. Consequently, we have to update the list, which requires that we create a copy of the list. This is done with `bets[:]`.

Returns iterator over all bets

`Table.__str__()` → str

Return an easy-to-read string representation of all current bets.

`Table.__repr__()` → str

Return a representation of the form `Table( bet, bet, ... )`.

Note that we will want to segregate validation as a separate method, or sequence of methods. This is used by the Game just prior to spinning the wheel (or rolling the dice, or drawing a next card.)

`Table.isValid(self)`

Raises InvalidBet if the bets don't pass the table limit rules.

Applies the table-limit rules:

- The sum of all bets is less than or equal to the table limit.
- All bet amounts are greater than or equal to the table minimum.

If there's a problem an *InvalidBet* exception is raised.

10.4 Roulette Table Deliverables

There are three deliverables for this exercise. Each of these will have complete Python docstring comments.

- An *InvalidBet* exception class. This is a simple subclass of `Exception`.
- Since there's no unique programming here, the unit test for *InvalidBet* is pretty simple. Indeed, it can seem silly to be sure that this class works with the `raise` statement; however, failure to extend `Exception` would lead to a program that more-or-less worked until a faulty *Player* class caused the invalid bet situation.
- The *Table* class.
- A class which performs a unit test of the *Table* class. The unit test should create at least two instances of *Bet*, and establish that these *Bet* s are managed by the table correctly.

CHAPTER

ELEVEN

ROULETTE GAME CLASS

Between *Player* and *Game*, we have a *chicken-and-egg design problem*: it's not clear which we should do first. In this chapter, we'll describe the design for *Game* in detail. However, in order to create the deliverables, we have to create a version of *Player* that we can use just to get started.

In the long run, we'll need to create a sophisticated hierarchy of players. Rather than digress too far, we'll create a simple player, *Passenger57* (they always bet on black), which will be the basis for further design in later chapters.

The class that implements the game will be examined in *Roulette Game Analysis*.

Once we've defined the game, we can also define a simple player class to interact with the game. We'll look at this in *Passenger57 Design*.

After looking at the player in detail, we can look at the game in detail. We'll examine the class in *Roulette Game Design*.

We'll provide some additional details in *Roulette Game Questions and Answers*. The *Roulette Game Deliverables* section will enumerate the deliverables.

There are a few variations on how Roulette works. We'll look at how we can include these details in the *Appendix: Roulette Variations* section.

11.1 Roulette Game Analysis

The `RouletteGame`'s responsibility is to cycle through the various steps of a defined procedure. We'll look at the procedure in detail. We'll also look at how we match bets in *The Bet Matching Algorithm*. This will lead us to define the player interface, which we'll look at in *Player Interface*.

This is an *active* class that makes use of the classes we have built so far. The hallmark of an active class is longer or more complex methods. This is distinct from most of the classes we have considered so far, which have relatively trivial methods that are little more than getters and setters of instance variables.

The procedure for one round of the game is the following.

A Single Round of Roulette

1. **Place Bets**. Notify the *Player* to create *Bet*s. The real work of placing bets is delegated to the *Player* class. Note that the money is committed at this point; they player's stake should be reduced as part of creating a *Bet*.
2. **Spin Wheel**. Get the next spin of the *Wheel*, giving the winning *Bin*, w. This is a collection of individual *Outcome*'s which will be winners. We can say $w = o_0, o_1, o_2, ..., o_n$: the winning bin is a set of outcomes.
3. **Resolve All Bets**.

   For each *Bet*, b, placed by the *Player*:

(a) **Winner?** If `Bet` b `Outcome` is in the winning `Bin`, w, then notify the `Player` that `Bet` b was a winner and update the `Player`'s stake.

(b) **Loser?** If `Bet` b `Outcome` is not in the winning `Bin`, w, then notify the `Player` that `Bet` b was a loser. This allows the `Player` to update the betting amount for the next round.

11.1.1 The Bet Matching Algorithm

This `Game` class will have the responsibility for matching the collection of `Outcome`s in the `Bin` of the `Wheel` with the collection of `Outcome`s of the `Bet`s on the `Table`. We'll need to structure a loop to compare individual elements from these two collections.

- Driven by `Bin`. We could use a loop to visit each `Outcome` in the winning `Bin`.

  For each `Outcome` in the winning `Bin`, $o(w)$:

  For each `Bet` contained by the `Table`, b:

  If the `Outcome` of the `Bet` matches the `Outcome` in the winning `Bin`, this bet, b, is a winner and is paid off.

  After this examination, all `Bet`s which have not been paid off are losers.

  This is unpleasantly complex because we can't resolve a `Bet` until we've checked all outcomes in the winning `Bin`.

- Driven by `Table`. The alternative is to visit each `Bet` contained by the `Table`.

  For each `Bet` in the `Table`, b:

  If the `Outcome` of b is in the `Outcome`s in the winning `Bin`, the bet is a winner. If the `Outcome` is not in the `Bin`, the bet is a loser.

  Since the winning `Bin` is a frozenset of `Outcome`s, we can exploit set membership methods to test for presence or absence of the `Outcome` for a `Bet` in the winning `Bin`.

11.1.2 Player Interface

The `Game` and `Player` collaboration involves mutual dependencies. This creates a "chicken and egg" problem in decomposing the relationship between these classes. The `Player` depends on `Game` features. The `Game` depends on `Player` features.

Which do we design first?

We note that the `Player` is really a complete hierarchy of subclasses, each of which provides a different betting strategy. For the purposes of making the `Game` work, we can develop our unit tests with a stub for `Player` that simply places a single kind of `Bet`. We'll call this player "Passenger57" because it always bets on Black.

Once we have a simplistic player, we can define the `Game` more completely.

After we have the `Game` finished, we can then revisit this design to make more sophisticated subclasses of `Player`. In effect, we'll bounce back and forth between `Player` and `Game`, adding features to each as needed.

For some additional design considerations, see *Appendix: Roulette Variations*. This provides some more advanced game options that our current design can be made to support. We'll leave this as an exercise for the more advanced student.

11.2 Passenger57 Design

class Passenger57

Passenger57 constructs a *Bet* based on the *Outcome* named `"Black"`. This is a very persistent player.

We'll need a source for the Black outcome. We have several choices; we looked at these in *Roulette Bet Class*. We will query the *Wheel* for the needed *Outcome* object.

In the long run, we'll have to define a *Player* superclass, and make *Passenger57* class a proper subclass of *Player*. However, our focus now is on getting the *Game* designed and built.

11.2.1 Fields

`Passenger57.`**`black`**

This is the outcome on which this player focuses their betting.

This *Player* will get this from the *Wheel* using a well-known bet name.

`Passenger57.`**`table`**

The *Table* that is used to place individual *Bet*s.

11.2.2 Constructors

`Passenger57.`**`__init__`**`(`*self*, *table*, *wheel*`)`

Parameters

- **`table`** (Table) – The *Table* instance on which bets are placed.
- **`wheel`** (Wheel) – The *Wheel* isntance which defines all *Outcome*s.

Constructs the *Player* with a specific table for placing bets. This also creates the "black" *Outcome*. This is saved in a variable named *Passenger57.black* for use in creating bets.

11.2.3 Methods

`Passenger57.`**`placeBets`**`(`*self*`)`

Updates the *Table* with the various bets. This version creates a *Bet* instance from the *black Outcome*. It uses *Table* `placeBet()` to place that bet.

`Passenger57.`**`win`**`(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – The bet which won.

Notification from the *Game* that the *Bet* was a winner. The amount of money won is available via a the `winAmount()` method of `theBet`.

`Passenger57.`**`lose`**`(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – The bet which won.

Notification from the *Game* that the *Bet* was a loser.

11.3 Roulette Game Design

class Game

Game manages the sequence of actions that defines the game of Roulette. This includes notifying the *Player* to place bets, spinning the *Wheel* and resolving the *Bet*s actually present on the *Table*.

11.3.1 Fields

wheel
The *Wheel* that returns a randomly selected *Bin* of *Outcome*s.

table
The *Table* which contains the *Bet*s placed by the *Player*.

player
The *Player* which creates *Bet*s at the *Table*.

11.3.2 Constructors

We based the Roulette Game constructor on a design that allows any of the fields to be replaced. This is the **Strategy** design pattern. Each of these collaborating objects is a replaceable strategy, and can be changed by the client that uses this game.

Additionally, we specifically do not include the *Player* instance in the constructor. The *Game* exists independently of any particular *Player*, and we defer binding the *Player* and *Game* until we are gathering statistical samples.

`Game.__init__(self, wheel, table)`

Parameters

- **wheel** (Wheel) – The *Wheel* instance which produces random events
- **table** (Table) – The *Table* instance which holds bets to be resolved.

Constructs a new *Game*, using a given *Wheel* and *Table*.

11.3.3 Methods

`cycle(self, player)`

Parameters player (Player) – the individual player that places bets, receives winnings and pays losses.

This will execute a single cycle of play with a given *Player*. It will

1. call `thePlayer`'s `placeBets()` method to get bets,
2. call `theWheel`'s *next()* method to get the next winning *Bin*,
3. call `theTable`'s iterator to get an `Iterator` over the *Bet*s. Stepping through this `Iterator` returns the individual *Bet* objects. If the winning *Bin* contains the *Outcome*, call the `thePlayer win()` method otherwise call the `thePlayer lose()` method.

11.4 Roulette Game Questions and Answers

Why are *Table* and *Wheel* part of the constructor while *Player* is given as part of the *cycle()* method?

We are making a subtle distinction between the casino table game (a Roulette table, wheel, plus casino staff to support it) and having a player step up to the table and play the game. The game exists without any particular player. By setting up our classes to parallel the physical entities, we give ourselves the flexibility to have multiple players without a significant rewrite. We allow ourselves to support multiple concurrent players or multiple simulations each using a different player object.

Also, as we look forward to the structure of the future simulation, we note that the game objects are largely fixed, but there will be a parade of variations on the player. We would like a main program that simplifies inserting a new player subclass with minimal disruption.

Why do we have to include the odds with the *Outcome* ? This pairing makes it difficult to create an *Outcome* from scratch.

The odds are an essential ingredient in the *Outcome* . It turns out that we want a short-hand name for each Outcome. We have three ways to provide a short name.

- A variable name. Since each variable is owned by a specific class instance, we need to allocate this to some class. The *Wheel* or the *BinBuilder* make the most sense for owning this variable.
- A key in a mapping. In this case, we need to allocate the mapping to some class. Again, the *Wheel* or *BinBuilder* make the most sense for owning the mapping.
- A method which returns the *Outcome* . The method can use a fixed variable or can get a value from a mapping.

11.5 Roulette Game Deliverables

There are three deliverables for this exercise. The stub does not need documentation, but the other classes do need complete Python docstrings.

- The *Passenger57* class. We will rework this design later. This class always places a bet on Black. Since this is simply used to test *Game*, it doesn't deserve a very sophisticated unit test of its own. It will be replaced in a future exercise.
- The `RouletteGame` class.
- A class which performs a demonstration of the *Game* class. This demo program creates the *Wheel*, the stub *Passenger57* and the *Table*. It creates the *Game* object and cycles a few times. Note that the *Wheel* returns random results, making a formal test rather difficult. We'll address this testability issue in the next chapter.

11.6 Appendix: Roulette Variations

In European casinos, the wheel has a single zero. In some casinos, the zero outcome has a special *en prison* rule: all losing bets are split and only half the money is lost, the other half is termed a "push" and is returned to the player. The following design notes discuss the implementation of this additional rule.

This is a payout variation that depends on a single *Outcome*. We will need an additional subclass of *Outcome* that has a more sophisticated losing amount method: it would push half of the amount back to the *Player* to be added to the stake. We'll call this subclass the the `PrisonOutcome` class.

In this case, we have a kind of hybrid resolution: it is a partial loss of the bet. In order to handle this, we'll need to have a `loss()` method in *Bet* as well as a `win()` method. Generally, the `loss()` method does nothing (since the money was removed from the *Player* stake when the bet was created.) However, for the `PrisonOutcome` class, the `loss()` method returns half the money to the *Player*.

We can also introduce a subclass of *BinBuilder* that creates only the single zero, and uses this new `PrisonOutcome` subclass of *Outcome* for that single zero. We can call this the `EuroBinBuilder` . The `EuroBinBuilder` does not create the five-way *Outcome* of 00-0-1-2-3, either; it creates a four-way for 0-1-2-3.

After introducing these two subclasses, we would then adjust *Game* to invoke the `loss()` method of each losing *Bet*, in case it resulted in a push. For an American-style casino, the `loss()` method does nothing. For a European-style casino, the `los()` method for an ordinary *Outcome* also does nothing, but the `los()` for a `PrisonOutcome` would implement the additional rule pushing half the bet back to the *Player* . The special behavior for zero then emerges from the collaboration between the various classes.

We haven't designed the *Player* yet, but we would have to bear this push rule in mind when designing the player.

The uniform interface between *Outcome* and `PrisonOutcome` is a design pattern called *polymorphism*. We will return to this principle many times.

CHAPTER

TWELVE

REVIEW OF TESTABILITY

This chapter presents some design rework and implementation rework for testability purposes. While testability is very important, new programmers can be slowed to a crawl by the mechanics of building test drivers and test cases. We prefer to emphasize the basic design considerations first, and address testability as a feature to be added to a working class.

In *Test Scaffolding* we'll look at the basic software components required to build unit tests.

One approach is to write tests first, then create software that passes the tests. We'll look at this in *Test-Driven Design*.

The application works with random numbers. This is awkward for testing purposes. We'll show one approach to solving this problem in *Capturing Pseudo-Radom Data*.

We'll touch on a few additional topics in *Testability Questions and Answers*.

In *Testability Deliverables* we'll enumerate some deliverables that will improve the overall quality of our application.

We'll look a little more deeply at random numbers in *Appendix: On Random Numbers*.

12.1 Test Scaffolding

Without pausing, we charged past the elephant standing in the saloon. It's time to pause a moment a take a quick glance back at the pachyderm we ignored.

In the *Roulette Game Class* we encouraged creating a stub `Player` and building a test that integrated Game, Table, Wheel, and the stub Player into a kind of working application. This is a kind of *integration test*. We've integrated our various classes into a working whole.

While this integration reflects our overall goals, it's not always the best way to assure that the individual classes work in isolation. We need to refine our approach somewhat.

Back in *Wheel Class* we touched on the problem of testing an application that includes a *random number generator* (RNG). There are two questions raised:

1. How can we develop formalized unit tests when we can't predict the random outcomes? This is a serious testability issue in randomized simulations. This question also arises when considering interactive applications, particularly for performance tests of web applications where requests are received at random intervals.
2. Are the numbers really random? This is a more subtle issue, and is only relevant for more serious applications. Cryptographic applications may care more deeply about the randomness of random numbers. This is a large subject, and well beyond the scope of this book. We'll just assume that our random number generator is good enough.

To address the testing issue, we need to develop some scaffolding that permits more controlled testing. We want to isolate each class so that our testing reveals problems in the class under test.

There are two approaches to replacing the random behavior with something more controlled.

- One approach is to create a mocked implementation of `random.Random` that returns specific outcomes that are appropriate for a given test.

  This is called the *Mock Objects* approach. It's very important for large applications. It allows us to test classes in isolation by creating mocks of their collaborators.

- A second approach is to record the sequence of random numbers actually generated from a particular seed value and use this to define the exected test results. We suggested forcing the seed to be 1 with `wheel.rng.seed(1)`.

12.2 Test-Driven Design

Good testability is achieved when classes are tested in isolation and there are no changes to the class being tested. We have to be careful that our design for *Wheel* works with a real random number generator as well as a mocked version of a random number generator.

An important consequence of this is that we suggested making the random number generator in *Wheel* visible. Rather than have *Wheel* use the `random` module directly, we suggesting creating an instance of `random.Random` as an attribute of *Wheel*.

This design choice reveals a tension between the encapsulation principle and the testability principle.

By *Encapsulation* we mean the design strategy where we define a class to encapsulate the details of it's implementation. It's unclear if the random number generator is an implementation detail or an explicit part of the *Wheel* implementation.

Generally, for must of the normal use cases, the random number generator inside a *Wheel* object is an invisible implementation detail. However, for testing purposes, the random number generator needs to be a configurable feature of the *Wheel* instance.

One approach to making something more visible is to provide default values in the constructor for the object.

Default Initialization

```
class Wheel:
    def __init__(self, rng=None):
        self.rng= rng if rng is not None else random.Random()
        ... rest of Wheel construction ...
```

For this particular situation, this technique is noisy. It introduces a feature that we'll never use outside writing tests.

In languages like Java, with lots of compile-time type-checking, we're forced either to provide an explicit parameter or rely on a complex framework that can inject this kind of change when testing.

Because Python type checking happens at run time, it's much easier to patch this class when writing a unit test. Since we can inject anything as the random number generator in a *Wheel*, our unit tests will look like this:

Mock Object Testing

```
from unittest.mock import MagicMock, Mock
import unittest

class GIVEN_Wheel_WHEN_spin_THEN_return_bin(unittest.TestCase):
    def setUp(self):
        self.bins = [ "bin1", "bin2" ]
        self.wheel = Wheel( self.bins )
        # Patch the random number generator
        self.wheel.rng= Mock()
```

```
        self.wheel.rng.choice = Mock( return_value="bin1" )

    def runTest(self):
        value= self.wheel.next()
        self.assertEqual( value, "bin1" )
        self.wheel.rng.choice.assert_called_with( self.bins )

unittest.main()
```

12.3 Capturing Pseudo-Radom Data

The other approach – using a fixed seed – means that we need to build and execute a program that reveals the fixed sequence of spins that are created by the non-random number generator.

We can create an instance of `Wheel`. We can set the random number generator seed to a known value, like `1`.

When can call the `Wheel.next()` method six times, and print the winning `Bin` instances. This sequence will always be the result for a seed value of `1`.

This discovery procedure will reveal results needed to create unit tests for `Wheel` and anything that uses it, for example, `Game`.

12.4 Testability Questions and Answers

Why are we making the random number generator more visible? Isn't object design about encapsulation?

> Encapsulation isn't the same thing as "information hiding" . For some people, the information hiding concept can be a useful way to begin to learn about encapsulation. However, information hiding is misleading because it is often taken to exremes. In this case, we want to encapsulate the bins of the wheel and the procedure for selecting the winning bin into a single object. However, the exact random-number generator (RNG) is a separate component, allowing us to bind any suitable RNG.
>
> Consider the situation where we are generating random numbers for a cryptographic application. In this case, the built-in random number generator may not be random enough. In this case, we may have a third-party Super-Random-Generator that should replace the built-in generator. We would prefer to minimize the changes required to introduce this new class.
>
> Our initial design has isolated the changes to the `Wheel`, but required us to change the constructor. Since we are changing the source code for a class, we must to unit test that change. Further, we are also obligated unit test all of the classes that depend on this class. Changing the source for a class deep within the application forces us to endure the consequence of retesting every class that depends on this deeply buried class. This is too much work to simply replace one object with another.
>
> We do, however, have an alternative. We can change the top-level `main()` method, altering the concrete object instances that compose the working application. By making the change at the top of the application, we don't need to change a deeply buried class and unit test all the classes that depend on the changed class. Instead, we are simply choosing among objects with the same superclass or interface.
>
> This is why we feel that constructors should be made very visible using the various design patterns for *Factories* and *Builders*. Further, we look at the main method as a kind of master *Builder* that assembles the objects that comprise the current execution of our application.
>
> See our *Roulette Solution Questions and Answers* FAQ for more on this subject.
>
> Looking ahead, we will have additional notes on this topic as we add the *SevenReds Player Class* subclass of `Player`.

If setting the seed works so well, why use a mock object?

> While setting the seed is an excellent method for setting up a unit test, not everyone is comfortable with random number generators. The presence of an arbitrary but predictable sequence of values looks too much like luck for some *Quality Assurance* (QA) managers. While the algorithms for the random number generator is published and well-understood, this isn't always sufficiently clear and convincing for non-mathematicians. It's often seems simpler to build a non-random mock object than to control the existing class.

12.5 Testability Deliverables

There are two deliverables for this exercise. All of these deliverables need Python docstrings.

- Revised unit tests for *`Wheel`* using a proper Mock for the random number generator.
- Revised unit tests for *`Game`* using a proper Mock for the *`Wheel`*.

12.6 Appendix: On Random Numbers

Random numbers aren't actually "random". Since they are generated by an algorithm, they are sometimes called *pseudo-random*. The distinction is important. Pseudo-random numbers are generated in a fixed sequence from a given *seed value*. Computing the next value in the sequence involves a calculation that is expected to overflow the available number of bits of precision leaving apparently random bits as the next value. This leads to results which, while predictable, are arbitrary enough that they pass rigorous statistical tests and are indistinguishable from data created by random processes.

We can make an application less predictable by choosing a very hard to predict seed value. A popular choice for the seed is the system clock; this isn't ideal.

In most operating systems a special "device" is available for producing random seed values. In Linux this is typically `/dev/random`. In Python, we can access this through the `os.urandom()` function.

As a consequence, we can make an application more predictable by setting a fixed seed value and noting the sequence of numbers generated. We can write a short demonstration program to see the effect of setting a fixed seed. This will also give us a set of predictable answers for unit testing.

CHAPTER THIRTEEN

PLAYER CLASS

The variations on `Player`, all of which reflect different betting strategies, is the heart of this application. In *Roulette Game Class*, we roughed out a stub class for `Player`. In this chapter, we will complete that design. We will also expand on it to implement the Matingale betting strategy.

We have now built enough infrastructure that we can begin to add a variety of players and see how their betting strategies work. Each player is a betting algorithm that we will evaluate by looking at the player's stake to see how much they win, and how long they play before they run out of time or go broke.

We'll look at the player problem in *Roulette Player Analysis*.

In *Player Design* we'll expand on our previous skeleton `Player` to create a more complete implementation. We'll expand on that again in *Martingale Player Design*.

In *Player Deliverables* we'll enumerate the deliverables for this chapter.

13.1 Roulette Player Analysis

The `Player` has the responsibility to create bets and manage the amount of their stake. To create bets, the player must create legal bets from known `Outcome`s and stay within table limits. To manage their stake, the player must deduct money when creating a bet, accept winnings or pushes, report on the current value of the stake, and leave the table when they are out of money.

We'll look at a number of topics:

- Our overall goal, in *Design Objectives*.
- How we manage the budget, in *Tracking the Stake*.
- How a player interacts with table limits, in *Table Limits*.
- In *Leaving the Table* we'll look at a player retiring when they're ahead. Or broke.
- In *Creating Bets from Outcomes* we'll look at a technical question of tranforming an `Outcome` instance into a `Bet` instance.

We have an interface that was roughed out as part of the design of `Game` and `Table`. In designing `Game`, we put a `placeBets()` method in `Player` to place all bets. We expected the `Player` to create `Bet`s and use the `placeBet()` method of `Table` class to save all of the individual `Bet`s.

In an earlier exercise, we built a stub version of `Player` in order to test `Game`. See *Passenger57 Design*. When we finish creating the final superclass, `Player`, we will also revise our `Passenger57` to be a subclass of `Player`, and rerun our unit tests to be sure that our more complete design still handles the basic test cases correctly.

13.1.1 Design Objectives

Our objective is to have a new abstract class, `Player`, with two new concrete subclasses: a revision to `Passenger57` and a new player that follows the Martingale betting system.

We'll defer some of the design required to collect detailed measurements for statistical analysis. In this first release, we'll simply place bets.

There are four design issues tied up in `Player`: tracking stake, keeping within table limits, leaving the table, and creating bets. We'll tackle them in separate subsections.

13.1.2 Tracking the Stake

One of the more important features we need to add to `Player` are the methods to track the player's stake. The initial value of the stake is the player's budget. There are several significant changes to the stake.

- Each bet placed will deduct the bet amount from the `Player`'s stake. We are stopped from placing bets when our stake is less than the table minimum.
- Each win will credit the stake. The `Outcome` will compute this amount for us.
- Additionally, a push will put the original bet amount back. This is a kind of win with no odds applied.

We'll have to design an interface that will create `Bet` s, reducing the stake. and will be used by `Game` to notify the `Player` of the amount won.

Additionally, we will need a method to reset the stake to the starting amount. This will be used as part of data collection for the overall simulation.'

13.1.3 Table Limits

Once we have our superclass, we can then define the `Martingale` player as a subclass. This player doubles their bet on every loss, and resets their bet to a base amount on every win. In the event of a long sequence of losses, this player will have their bets rejected as over the table limit. This raises the question of how the table limit is represented and how conformance with the table limit is assured.

We put a preliminary design in place in *Roulette Table Class*. There are several places where we could isolate this responsibility.

1. The `Player` stops placing bets when they are over the `Table` limit. In this case, we will be delegating responsibility to the `Player` hierarchy. In a casino, a sign is posted on the table, and both players and casino staff enforce this rule. This can be modeled by providing a method in `Table` that simply returns the table limit for use by the `Player` to keep bets within the limit.
2. The `Table` provides a "valid bet" method.
3. The `Table` throws an "illegal bet" exception when an illegal bet is placed.

The first alternative is unpleasant because the responsibility to spread around: both `Player` and `Table` must be aware of a feature of the `Table`. This means that a change to the `Table` will also require a change to the `Player`. This is poor object-oriented design.

The second and third choices reflect two common approaches that are summarized as:

- Ask Permission. The application has code wrapped in `if permitted:` conditional processing.
- Ask Forgiveness. The application assumes that things will work. An exception indicates something unexpected happened.

The general advice is this:

> **It's better to ask forgiveness than to ask permission.**

Most of the time, validation should be handled by raising an exception.

Handling Game State. This raises a question about how we handle advanced games where some bets are not allowed during some game states.

There are two sources of validation for a bet.

- The *Table* may reject a bet because it's over (or under) a limit.
- The *Game* may reject a bet because it's illegal in the current state of the game.

Since these considerations are part of Craps and Blackjack, we'll set them aside for now.

13.1.4 Leaving the Table

We also need to address the issue of the *Player* leaving the game. We can identify a number of possible reasons for leaving: out of money, out of time, won enough, and unwilling to place a legal bet. Since this decision is private to the *Player* , we need a way of alerting the *Game* that the *Player* is finished placing bets.

There are three mechanisms for alerting the *Game* that the *Player* is finished placing bets.

1. Expand the responsibilities of the `placeBets()` to also indicate if the player wishes to continue or is withdrawing from the game. While most table games require bets on each round, it is possible to step up to a table and watch play before placing a bet. This is one classic strategy for winning at blackjack: one player sits at the table, placing small bets and counting cards, while a confederate places large bets only when the deck is favorable. We really have three player conditions: watching, betting and finished playing. It becomes complex trying to bundle all this extra responsibility into the `placeBets()` method.
2. Add another method to *Player* that the *Game* can use to determine if the *Player* will continue or stop playing. This can be used for a player who is placing no bets while waiting; for example, a player who is waiting for the Roulette wheel to spin red seven times in a row before betting on black.
3. The *Player* can throw an exception when they are done playing. This is an exceptional situation: it occurs exactly once in each simulation. However, it is a well-defined condition, and doesn't deserve to be called "exceptional" . It is merely a terminating condition for the game.

We recommend adding a method to *Player* to indicate when *Player* is done playing. This gives the most flexibility, and it permits *Game* to cycle until the player withdraws from the game.

A consequence of this decision is to rework the *Game* class to allow the player to exit. This is relatively small change to interrogate the *Player* before asking the player to place bets.

Note: Design Evolution

In this case, these were situations which we didn't discover during the initial design. It helped to have some experience with the classes in order to determine the proper allocation of responsibilities. While design walkthroughs are helpful, an alternative is a "prototype", a piece of software that is incomplete and can be disposed of. The earlier exercise created a version of *Game* that was incomplete, and a version of `PlayerStub` that will have to be disposed of.

13.1.5 Creating Bets from Outcomes

Generally, a *Player* will have a few *Outcome*s on which they are betting. Many systems are similar to the Martingale system, and place bets on only one of the *Outcome*s. These *Outcome* objects are usually created during player initialization. From these *Outcome*s, the *Player* can create the individual *Bet* instances based on their betting strategy.

Since we're currently using the *Wheel* as a repository for all legal *Outcome* instances, we'll need to provide the *Wheel* to the *Player*.

This doesn't generalize well for Craps or Blackjack. We'll need to revisit this design decision. In the long run, we'll need to find another kind of factory for creating proper *Outcome* instances.

13.2 Player Design

class Player

We'll design the base class of *Player* and a specific subclass, *Martingale*. This will give us a working player that we can test with.

Player places bets in Roulette. This an abstract class, with no actual body for the `placeBets()` method. However, this class does implement the basic `win()` method used by all subclasses.

13.2.1 Fields

`Player.`**`stake`**
: The player's current stake. Initialized to the player's starting budget.

`Player.`**`roundsToGo`**
: The number of rounds left to play. Initialized by the overall simulation control to the maximum number of rounds to play. In Roulette, this is spins. In Craps, this is the number of throws of the dice, which may be a large number of quick games or a small number of long-running games. In Craps, this is the number of cards played, which may be large number of hands or small number of multi-card hands.

`Player.`**`table`**
: The *Table* used to place individual *Bet*s. The *Table* contains the current *Wheel* from which the player can get `Outcomes` used to build *Bet*s.

13.2.2 Constructors

`Player.`**`__init__`**(*self*, *table*)
: Constructs the *Player* with a specific *Table* for placing *Bet*s.

  Parameters **`table`** (Table) – the table to use

  Since the table has access to the *Wheel*, we can use this wheel to extract :class'Outcome' objects.

13.2.3 Methods

`Player.`**`playing`**(*self*) → boolean
: Returns `true` while the player is still active.

`Player.`**`placeBets`**(*self*)
: Updates the *Table* with the various *Bet* s.

  When designing the *Table*, we decided that we needed to deduct the amount of a bet from the stake when the bet is created. See the Table *Roulette Table Analysis* for more information.

`Player.`**`win`**(*self*, *bet*)
: **Parameters** **`bet`** (Bet) – The bet which won

  Notification from the *Game* that the *Bet* was a winner. The amount of money won is available via `bet` method `winAmount()`.

`Player.`**`lose`**(*self*, *bet*)
: **Parameters** **`bet`** (Bet) – The bet which won

Notification from the *Game* that the *Bet* was a loser. Note that the amount was already deducted from the stake when the bet was created.

13.3 Martingale Player Design

class Martingale

Martingale is a *Player* who places bets in Roulette. This player doubles their bet on every loss and resets their bet to a base amount on each win.

13.3.1 Fields

Martingale.**lossCount**

The number of losses. This is the number of times to double the bet.

Martingale.**betMultiple**

The the bet multiplier, based on the number of losses. This starts at 1, and is reset to 1 on each win. It is doubled in each loss. This is always equal to $2^{lossCount}$.

13.3.2 Methods

Martingale.**placeBets**(*self*)

Updates the *Table* with a bet on "black". The amount bet is $2^{lossCount}$, which is the value of *betMultiple*.

Martingale.**win**(*self*, *bet*)

Parameters **bet** (Bet) – The bet which won

Uses the superclass *win()* method to update the stake with an amount won. This method then resets *lossCount* to zero, and resets *betMultiple* to 1.

Martingale.**lose**(*self*, *bet*)

Parameters **bet** (Bet) – The bet which won

Uses the superclass *lose()* to do whatever bookkeeping the superclass already does. Increments *lossCount* by 1 and doubles *betMultiple*.

13.4 Player Deliverables

There are six deliverables for this exercise. The new classes must have Python docstrings.

- The *Player* abstract superclass. Since this class doesn't have a body for the placeBets(), it can't be unit tested directly.
- A revised *Passenger57* class. This version will be a proper subclass of *Player*, but still place bets on black until the stake is exhausted. The existing unit test for *Passenger57* should continue to work correctly after these changes.
- The *Martingale* subclass of *Player*.
- A unit test class for *Martingale*. This test should synthesize a fixed list of *Outcome* s, *Bin* s, and calls a *Martingale* instance with various sequences of reds and blacks to assure that the bet doubles appropriately on each loss, and is reset on each win.

- A revised *Game* class. This will check the player's `playing()` method before calling `placeBets()`, and do nothing if the player withdraws. It will also call the player's `win()` and `lose()` methods for winning and losing bets.
- A unit test class for the revised *Game* class. Using a non-random generator for *Wheel*, this should be able to confirm correct operation of the *Game* for a number of bets.

CHAPTER

FOURTEEN

OVERALL SIMULATION CONTROL

We can now use our application to generate some more usable results. We can perform a number of simulation runs and evaluate the long-term prospects for the Martingale betting system. We want to know a few things about the game:

- How long can we play with a given budget? In other words, how many spins before we've lost our stake.
- How much we can realistically hope to win? How large a streak can we hope for? How far ahead can we hope to get before we should quit?

In the *Simulation Analysis* section, we'll look at general features of simulation.

We'll look at the resulting statistical data in *Statistical Summary*.

This will lead us to the details in *Simulator Design*. We'll note that the details of the simulator require some changes to the definition of the `Player`. We'll look at this in *Player Rework*.

We'll enumerate all of this chapter's deliverables in *Simulation Control Deliverables*.

14.1 Simulation Analysis

A `Simulator` class will be allocated a number of responsibilities:

- Create the `Wheel`, `Table` and `Game` objects.
- Simulate a number of sessions (typically 100), saving the maximum stake and length of each session.
- For each session: initialize the `Player` and `Game` , cycle the game a number of times, collect the size of the `Player` 's stake after each cycle.
- Write a final summary of the results.

We'll look at several topics:

- *Simulation Terms* will formalize some definitions.
- *Simulation Control* will look at the overall simulation process.
- We'll look at how we create a new player in *Player Initialization*.
- The most important thing is gather performance data. We'll look at this in *Player Interrogation*.

14.1.1 Simulation Terms

We'll try to stick to the following definitions. This will help structure our data gathering and analysis.

cycle

A single cycle of betting and bet resolution. This depends on a single random event: a spin of the wheel or a throw of the dice. Also known as a round of play.

session

One or more cycles. The session begins with a player having their full stake. A session ends when the play elects to leave or can no longer participate. A player may elect to leave because of elapsed time (typically 250 cycles), or they have won a statistically significant amount. A player can no longer participate when their stake is too small to make the minimum bet for the table.

game

Some games have intermediate groupings of events between an individual cycles and an entire session. Blackjack has *hands* where a number of player decisions and a number of random events contribute to the payoff. Craps has a *game*, which starts with the dice roll when the point is *off*, and ends when the point is made or the shooter gets *Craps*; consequently, any number of individual dice rolls can make up a game. Some bets are placed on the overall game, while others are placed on individual dice rolls.

14.1.2 Simulation Control

The sequence of operations for the simulator looks like this.

Controlling the Simulation

1. **Empty List of Maxima**. Create an empty maxima list. This is the maximum stake at the end of each session.
2. **Empty List of Durations**. Create an empty durations list. This is the duration of each session, measured in the number of cycles of play before the player withdrew or ran out of money.
3. **For All Sessions**. For each of 100 sessions:

    Empty List of Stake Details. Create an empty list to hold the history of stake values for this session. This is raw data that we will summarize into two metrics for the session: maximum stake and duration. We could also have two simple variables to record the maximum stake and count the number of spins for the duratioon. However, the list allows us to gather other statistics, like maximum win or maximum loss.

    While The Player Is Active.

        Play One Cycle. Play one cycle of the game. See the definition in *Roulette Game Class*.

        Save Outcomes. Save the player's current stake in the list of stake values for this session. An alternative is to update the maximum to be the larger of the current stake and the maximum, and increment the duration.

    Get Maximum. Get the maximum stake from the list of stake values. Save the maximum stake metric in the maxima list.

    Get Duration. Get the length of the list of stake values. Save the duration metric in the durations list. Durations less than the maximum mean the strategy went bust.

4. **Statistical Description of Maxima**. Compute the average and standard deviation of the values in the maxima list.
5. **Statistical Description of Durations**. Compute the average and standard deviation of values in the durations list.

Both this overall `Simulator` and the `Game` collaborate with the `Player`. The `Simulator`'s collaboration, however, initalizes the `Player` and then monitors the changes to the `Player`'s stake. We have to design two interfaces for this collaboration.

- Initialization
- Interrogation

14.1.3 Player Initialization

The *Simulator* will initialize a *Player* for 250 cycles of play, assuming about one cycle each minute, and about four hours of patience. We will also initialize the player with a generous budget of the table limit, 100 betting units. For a $10 table, this is $1,000 bankroll.

Currently, the *Player* class is designed to play one session and stop when their duration is reached or their stake is reduced to zero. We have two alternatives for reinitializing the *Player* at the beginning of each session.

1. Provide some *setters* that allow a client class like this overall simulator control to reset the `stake` and `roundsToGo` values of a *Player*.
2. Provide a **Factory** that allows a client class to create new, freshly initialized instances of *Player*.

While the first solution is quite simple, there are some advantages to creating a `PlayerFactory`. If we create an **Abstract Factory**, we have a single place that creates all *Player*s.

Further, when we add new player subclasses, we introduce these new subclasses by creating a new subclass of the factory. In this case, however, only the main program creates instances of *Player*, reducing the value of the factory. While design of a **Factory** is a good exercise, we can scrape by with adding setter methods to the *Player* class.

14.1.4 Player Interrogation

The *Simulator* will interrogate the *Player* after each cycle and capture the current stake. An easy way to manage this detailed data is to create a `List` that contains the stake at the end of each cycle. The length of this list and the maximum value in this list are the two metrics the *Simulator* gathers for each session.

Our list of maxima and durations are created sequentially during the session and summarized sequentially at the end of the session. A `list` will do everything we need. For a deeper discussion on the alternatives available in the collections framework, see *Design Decision – Choosing A Collection*.

14.2 Statistical Summary

The *Simulator* will interrogate the *Player* after each cycle and capture the current stake. We don't want the sequence of values for each cycle; we want a summary of all the cycles in the session. We can save the length of the sequence as well as the maximum of the sequence. We can then calculate aggregate performance parameters for each session.

Our objective is to run several session simulations to get averages and a standard deviations for duration and maximum stake. This means that the *Simulator* needs to retain these statistical samples. We will defer the detailed design of the statistical processing, and simply keep the duration and maximum values in lists for this first round of design.

14.3 Simulator Design

class `Simulator`

Simulator exercises the Roulette simulation with a given *Player* placing bets. It reports raw statistics on a number of sessions of play.

14.3.1 Fields

`Simulator.`**`initDuration`**

The duration value to use when initializing a *Player* for a session. A default value of 250 is a good choice here.

`Simulator.`**`initStake`**
The stake value to use when initializing a *`Player`* for a session. This is a count of the number of bets placed; i.e., 100 $10 bets is $1000 stake. A default value of 100 is sensible.

`Simulator.`**`samples`**
The number of game cycles to simulate. A default value of 50 makes sense.

`Simulator.`**`durations`**
A `List` of lengths of time the *`Player`* remained in the game. Each session of play producrs a duration metric, which are collected into this list.

`Simulator.`**`maxima`**
A `List` of maximum stakes for each *`Player`*. Each session of play producrs a maximum stake metric, which are collected into this list.

`Simulator.`**`player`**
The *`Player`*; essentially, the betting strategy we are simulating.

`Simulator.`**`game`**
The casino game we are simulating. This is an instance of *`Game`*, which embodies the various rules, the *`Table`* and the *`Wheel`*.

14.3.2 Constructors

`Simulator.`**`__init__`**`(`*self*, *game*, *player*`)`
Saves the *`Player`* and *`Game`* instances so we can gather statistics on the performance of the player's betting strategy.

Parameters

- **`game`** (Game) – The Game we're simulating. This includes the *`Table`* and *`Wheel`*.
- **`player`** (Player) – The Player. This encapsulates the betting strategy.

14.3.3 Methods

`Simulator.`**`session`**`(`*self*`)` → list

Returns `list` of stake values.

Return type list

Executes a single game session. The *`Player`* is initialized with their initial stake and initial cycles to go. An empty `List` of stake values is created. The session loop executes until the *`Player`* `playing()` returns false. This loop executes the *`Game cycle()`*; then it gets the stake from the *`Player`* and appends this amount to the `List` of stake values. The `List` of individual stake values is returned as the result of the session of play.

`Simulator.`**`gather`**`(`*self*`)`
Executes the number of games sessions in *`samples`*. Each game session returns a `List` of stake values. When the session is over (either the play reached their time limit or their stake was spent), then the length of the session `List` and the maximum value in the session `List` are the resulting duration and maximum metrics. These two metrics are appended to the *`durations`* list and the *`maxima`* list.

A client class will either display the durations and maxima raw metrics or produce statistical summaries.

14.4 Player Rework

The current design for the *`Player`* doesn't provide all the methods we need.

We'll can add two new methods: one will set the `stake` and the other will set the `roundsToGo`.

`Player.setStake(self, stake)`

Parameters **stake** (*integer*) – the Player's initial stake

`Player.setRounds(self, rounds)`

Parameters **rounds** (*integer*) – the Player's duration of play

14.5 Simulation Control Deliverables

There are five deliverables for this exercise. Each of these classes needs complete Python docstring comments.

- Revision to the `Player` class. Don't forget to update unit tests.
- The `Simulator` class.
- The expected outcomes from the non-random wheel can be rather complex to predict. Because of this, one of the deliverables is a demonstration program that enumerates the actual sequence of non-random spins. From this we can derive the sequence of wins and losses, and the sequence of `Player` bets. This will allow us to predict the final outcome from a single session.
- A unit test of the `Simulator` class that uses the non-random generator to produce the predictable sequence of spins and bets.
- A main application function that creates the necessary objects, runs the `Simulator`'s `gather()` method, and writes the available outputs to `sys.stdout`

For this initial demonstration program, it should simply print the list of maxima, and the list of session lengths. This raw data can be redirected to a file, loaded into a spreadsheet and analyzed.

CHAPTER

FIFTEEN

SEVENREDS PLAYER CLASS

This section introduces an additional specialization of the Martingale strategy. Additionally, we'll also address some issues in how an overall application is composed of individual class instances. Adding this new subclass should be a small change to the main application class.

We'll also revisit a question in the design of the overall `Table`. Should we really be checking for a minimum? Or was that needless?

In *SevenReds Player Analysis* we'll examine the general strategy this player will follow.

We'll revisit object-oriented design by composition in *Soapbox on Composition*.

In *SevenReds Design* we'll look at the design of this player. We'll need to revise the overall design for the abstract `Player`, also, which we'll look at in *Player Rework*.

These design changes will lead to other changes. We'll look at these changes in *Game Rework* and *Table Rework*.

This will lead to *SevenReds Player Deliverables*, which enumerates all of the deliverables for this chapter.

15.1 SevenReds Player Analysis

The `SevenReds` player waits for seven red wins in a row before betting black. This is a subclass of `Player`. We can create a subclass of our main `Simulator` to use this new `SevenReds` class.

We note that `Passenger57`'s betting is stateless: this class places the same bets over and over until they are cleaned out or their playing session ends.

The `Martingale` player's betting, however, is stateful. This player changes the bet based on wins and losses. The state is a loss counter than resets to zero on each win, and increments on each loss.

Our `SevenReds` player will have two states: waiting and betting. In the waiting state, they are simply counting the number of reds. In the betting state, they have seen seven reds and are now playing the Martingale system on black. We will defer serious analysis of this *stateful* betting until some of the more sophisticated subclasses of `Player`. For now, we will simply use an integer to count the number of reds.

15.1.1 Game Changer

Currently, a `Player` is not informed of the final outcome unless they place a bet. We designed the `Game` to evaluate the `Bet` instances and notify the `Player` of just their `Bet`s that were wins or losses. We will need to add a method to `Player` to be given the overall list of winning `Outcome`s even when the `Player` has not placed a bet.

Once we have updated the design of `Game` to notify `Player`, we can add the new `SevenReds` class. Note that we are can intoduce each new betting strategy via creation of new subclasses. A relatively straightforward update to our simulation main program allows us to use these new subclasses. The previously working subclasses are left in place, allowing graceful evolution by adding features with minimal rework of existing classes.

In addition to waiting for the wheel to spin seven reds, we will also follow the Martingale betting system to track our wins and losses, assuring that a single win will recoup all of our losses. This makes `SevenReds` a further specialization of `Martingale`. We will be using the basic features of `Martingale`, but doing additional processing to determine if we should place a bet or not.

Introducing a new subclass should be done by upgrading the main program. See *Soapbox on Composition* for comments on the ideal structure for a main program. Additionally, see the *Roulette Solution Questions and Answers* FAQ entry.

15.1.2 Table Changes

When we designed the table, we included a notion of a valid betting state. We required that the sum of all bets placed by a `Player` was below some limit. We also required that there be a table minimum present.

A casino has a table minimum for a variety of reasons. Most notably, it serves to distinguish casual players at low-stakes tables from "high rollers" who might prefer to play with other people who wager larger amounts.

In the rare event that a player is the only person at a roulette wheel, the croupier won't spin the wheel until a bet is placed. This is an odd thing. It's also very rare. Pragmatically, there are almost always other players, and the wheel will be spun even if a given player is not betting.

Our design for a table really should **not** have any check for a minimum bet. It's a rule that doesn't make sense for the kind of simulation we're doing.

For this particular player, it's essential that the wheel is spun even if the player has no bets in play.

15.2 Soapbox on Composition

Generally, a solution is composed of a number of objects. However, the consequences of this are often misunderstood. Since the solution is a composition of objects, it falls on the main method to do just the composition and nothing more.

Our ideal main program creates and composes the working set of objects. In this case, it should decode the command-line parameters, and use this information to create and initialize the simulation objects, then start the processing. For these simple exercises, however, we're omitting the parsing of command-line parameters, and simply creating the necessary objects directly.

A main program should, therefore, look something like the following:

```
theWheel= Wheel()
theTable= Table()
theGame= Game( theWheel, theTable )
thePlayer= SevenReds( theTable )
sim= new Simulator( theGame, thePlayer )
sim.gather()
```

We created an instance of `Wheel` which has the bins and outcomes. We created an instance of `Table` which has a place to put the bets. We've unified these two through an instance of `Game` which depends on the wheel and the table.

When we created the `thePlayer`, we could have used `Martingale` or `Passenger57`. The player object can use the table to get the wheel instance. This wheel instance provides the outcomes used to build bets.

We have assembled the overall simulation by composition of a Wheel, the Table and the specific Player algorithm.

The real work is done by `Simulater.gather()`. This relies on the game, table, and player to create some data we can analyze.

In some instances, the construction of objects is not done directly by the main method. Instead, the main method will use **Builders** to create the various objects. The idea is to avoid mentioning the class definitions directly. We can upgrade or replace a class, and also upgrade the **Builder** to use that class appropriately. This isolates change to the class hierarchy and a builder function.

15.3 SevenReds Design

class SevenReds

SevenReds is a *Martingale* player who places bets in Roulette. This player waits until the wheel has spun red seven times in a row before betting black.

15.3.1 Fields

SevenReds.**redCount**

The number of reds yet to go. This starts at 7 , is reset to 7 on each non-red outcome, and decrements by 1 on each red outcome.

Note that this class inherits betMultiple. This is initially 1, doubles with each loss and is reset to one on each win.

15.3.2 Methods

SevenReds.**placeBets**(*self*)

If *redCount* is zero, this places a bet on black, using the bet multiplier.

SevenReds.**winners**(*self*, *outcomes*)

Parameters outcomes (*Set of Outcome*) – The *Outcome* set from a *Bin*.

This is notification from the *Game* of all the winning outcomes. If this vector includes red, *redCount* is decremented. Otherwise, *redCount* is reset to 7.

15.4 Player Rework

We'll need to revise the *Player* class to add the following method. The superclass version doesn't do anything with this information. Some subclasses, however, will process this.

Player.**winners**(*self*, *outcomes*)

Parameters outcomes (*Set of Outcome*) – The set of *Outcome* instances that are part of the current win.

The game will notify a player of each spin using this method. This will be invoked even if the player places no bets.

15.5 Game Rework

We'll need to revise the *Game* class to extend the cycle method. This method must provide the winning bin's *Outcome* set.

15.6 Table Rework

We'll need to revise the *Table* class to remove any minimum bet rule. If there are no bets, the game should still proceed.

15.7 SevenReds Player Deliverables

There are six deliverables from this exercise. The new classes will require complete Python docstrings.

- A revision to the *Player* to add the *Player.winners()* method. The superclass version doesn't do anything with this information. Some subclasses, however, will process this.
- A revision to the *Player* unit tests.
- A revision to the *Game* class. This will call the `winners()` with the winning *Bin* instance before paying off the bets.
- A revision to the *Table* class. This will allow a table with zero bets to be considered valid for the purposes of letting the game continue.
- The *SevenReds* subclass of *Player*.
- A unit test of the *SevenReds* class. This test should synthesize a fixed list of *Outcome*s, *Bin*s and the call a *SevenReds* instance with various sequences of reds and blacks. One test cases can assure that no bet is placed until 7 reds have been seen. Another test case can assure that the bets double (following the Martingale betting strategy) on each loss.
- A main application function that creates the necessary objects, runs the *Simulator*'s `gather()` method, and writes the available outputs to `sys.stdout`

For this initial demonstration program, it should simply print the list of maxima, and the list of session lengths. This raw data can be redirected to a file, loaded into a spreadsheet and analyzed.

CHAPTER

SIXTEEN

STATISTICAL MEASURES

The *Simulator* class collects two Lists. One has the length of a session, the other has the maximum stake during a session. We need to create some descriptive statistics to summarize these stakes and session lengths.

This section presents two ordinary statistical algorithms: mean and standard deviation. Python 3 introduces the statistics module, which can slightly simplify some of the programming required for this chapter.

We'll present the details of how to compute these two statistics for those who are interested.

In *Statistical Analysis* we'll address the overall goal of gathering and analyzing statistics.

In *Some Foundations* we'll look at the Σ operator which is widely used for stastical calculation. We'll see how to implement this in Python.

In *Statistical Algorithms* we'll look specifically at mean and standard deviation.

Since we're apply stastical calculation to a list of integer values, we'll look at how we can extend the list in *IntegerStatistics Design*.

We'll enumerate the deliverables for this chapter in *Statistics Deliverables*.

On Statistics

In principle, we could apply basic probability theory to predict the statistics generated by this simulation. Indeed, some of the statistics we are gathering are almost matters of definition, rather than actual unknown data. We are, however, much more interested in the development of the software than we are in applying probability theory to a relatively simple casino game. As a consequence, the statistical analysis here is a little off-base.
Our goal is to build a relatively simple-minded set of descriptive statistics. We will average two metrics: the peak stake for a session and the duration of the session. Both of these values have some statistical problems. The peak stake, for instance, is already a summary number. We have to be cautious in our use of summarized numbers, since we have lost any sense of the frequency with which the peaks occured. For example, a player may last 250 cycles of play, with a peak stake of 130. We don't know if that peak occurred once or 100 times out of that 250. Additionally, the length of the session has an upper bound on the number of cycles. This forces the distribution of values to be skewed away from the predictable distribution.

16.1 Statistical Analysis

We will design a Statistics class with responsibility to retain and summarize a list of numbers and produce the average (also known as the mean) and standard deviation. The *Simulator* can then use this this Statistics class to get an average of the maximum stakes from the list of session-level measures. The *Simulator* can also apply this Statistics class to the list of session durations to see an average length of game play before going broke.

We have three design approaches for encapsulating processing:

- We can extend an existing class, or

- We can wrap an existing class, creating a whole new kind of thing, or
- We can delegate the statistical functions to a separate function. This is how thing currently stand in Python. We have a build-in list class and a separate `statitics` module.

A good approach is to extend the built-in list with statistical summary features. Given this new class, we can replace the original `List` of sample values with a `StatisticalList` that both saves the values and computes descriptive statistics.

This design has the most reuse potential. This can be used in a variety of contexts, and allows us the freedom to switch `List` implementation classes without any other changes.

The detailed algorithms for mean and standard deviation are provided in *Statistical Algorithms*.

16.2 Some Foundations

For those programmers new to statistics, this section covers the Sigma operator, Σ.

$$\sum_{i=0}^{n} f(i)$$

The Σ operator has three parts to it. Below it is a bound variable, i, and the starting value for the range, written as $i = 0$. Above it is the ending value for the range, usually something like n. To the right is some function to execute for each value of the bound variable. In this case, a generic function, $f(i)$ is shown. This is read as "sum f (i) for i in the range 0 to n".

One common definition of Σ uses a closed range, including the end values of 0 and n . However, since this is not a helpful definition for software, we will define Σ to use a half-open interval. It has exactly n elements, including 0 and n -1; mathematically, $0 \leq i < n$.

Consequently, we prefer the following notation, but it is not often used. Since statistical and mathematical texts often used 1-based indexing, some care is required when translating formulae to programming languages that use 0-based indexing.

$$\sum_{0 \leq i < n} f(i)$$

Our two statistical algorithms have a form more like the following function. In this we are applying some function, f, to each value, x_i of an list, x.

When computing the mean, as a special case, there is no function applied to the values in the list. When computing standard deviation, the function involves subtracting and multiplying.

$$\sum_{0 \leq i < n} f(x_i)$$

We can transform this definition directly into a for loop that sets the bound variable to all of the values in the range, and does some processing on each value of the List of Integers.

This is the Python implementation of Sigma. This computes two values, the sum, `s` and the number of elements, `n`.

```
s= sum( theList )
n = len( theList )
```

When computing the standard deviation, we do something like the following

```
s= sum( f(x) for x in theList )
n = len( theList )
```

Where the `f(x)` calculuation computes the measure of deviation from the average.

16.3 Statistical Algorithms

We'll look at two important algorithms:

- *mean*, and
- *standard deviation*.

16.3.1 Mean

Computing the mean of a list of values is relatively simple. The mean is the sum of the values divided by the number of values in the list. Since the statistical formula is so closely related to the actual loop, we'll provide the formula, followed by an overview of the code.

$$\mu_x = \frac{\sum_{0 \leq i < n} x_i}{n}$$

The definition of the Σ mathematical operator leads us to the following method for computing the mean:

```
sum(self)/len(self)
```

This matches the mathematical definition nicely.

16.3.2 Standard Deviation

The standard deviation can be done a few ways. We'll use the formula shown below. This computes a deviation measurement as the square of the difference between each sample and the mean.

The sum of these measurements is then divided by the number of values times the number of degrees of freedom to get a standardized deviation measurement.

Again, the formula summarizes the loop, so we'll show the formula followed by an overview of the code.

$$\sigma_x = \sqrt{\frac{\sum_{0 \leq i < n} (x_i - \mu_x)^2}{n - 1}}$$

The definition of the Σ mathematical operator leads us to the following method for computing the standard deviation:

We can use a generator expression for this.

```
m = mean(x)
math.sqrt( sum( (x-m)**2 for x in self ) / (len(self)-1) )
```

This seems to match the mathematical definition nicely.

16.4 IntegerStatistics Design

class IntegerStatistics(*list*)

IntegerStatistics computes several simple descriptive statistics of `Integer` values in a `List`.

This extends list with some additional methods.

16.4.1 Constructors

Since this class extends the built-in list, we'll leverage the existing constructor.

16.4.2 Methods

`IntegerStatistics.mean(self)`
Computes the mean of the `List` of values.

`IntegerStatistics.stdev(self)`
Computes the standard deviation of the `List` values.

16.5 Statistics Deliverables

There are three deliverables for this exercise. These classes will include the complete Python dostring.

- The `IntegerStatistics` class.
- A unit test of the `IntegerStatistics` class.

  Prepare some simple list (or tuple) of test data.

  The results can be checked with a spreadsheet
- An update to the overall `Simulator` that gets uses an `IntegerStatistics` object to compute the mean and standard deviation of the peak stake. It also computest the mean and standard deviation of the length of each session of play.

Here is some standard deviation unit test data.

Sample Value
10
8
13
9
11
14
6
4
12
7
5

Here are some intermediate results and the correct answers given to 6 significant digits. Your answers should be the same to the precision shown.

sum 99

count 11

mean 9.0

sum (x-m)2** 110.0

stdev 3.317

CHAPTER SEVENTEEN

RANDOM PLAYER CLASS

This section will introduce a simple subclass of *Player* who bets at random.

In *Random Player Analysis* we'll look at what this player does.

We'll turn to how the player works in *Random Player Design*.

In *Random Player Deliverables* we'll enumerate the deliverables for this player.

An important consideration is to compare this player with the player who always bets black and the player using the Martigale strategy to always bet black. Who does better? If they're all about the same, what does that say about the house edge in this game?

17.1 Random Player Analysis

One possible betting strategy is to bet completely randomly. This serves as an interesting benchmark for other betting strategies.

We'll write a subclass of *Player* which steps through all of the bets available on the *Wheel*, selecting one or more of the available outcomes at random. This *Player*, like others, will have a fixed initial stake and a limited amount of time to play.

The *Wheel* class can provide an `Iterator` over the collection of *Bin* instances. We could revise *Wheel* to provide a `binIterator()` method that we can use to return all of the *Bin*s. From each *Bin*, we will need an iterator we can use to return all of the *Outcome*s.

To collect a list of all possible *Outcome*s, we would use the following algorithm:

Locating all Outcomes

1. **Empty List of Outcomes**. Create an empty set of all *Outcome*s, `all_OC`.
2. **Get Bin Iterator**. Get the Iterator from the *Wheel* that lists all *Bin*s.
3. **For each Bin**.

   Get Outcome Iterator. Get the Iterator that lists all *Outcome*s.

   For each Outcome.

      Save Outcome. Add each *Outcome* to the set of all known outcomes, `all_OC`.

To place a random bet, we would use the following algorithm:

Placing a Random Bet

1. Get the size of the pool of all possible *Outcome*s, s.
2. Get a random number, u, from zero to the $s - 1$. That is, $0 \leq u < s$. This is readily available in `random.randrange()`.
3. Return element u from the pool of *Outcome*s.

17.2 Random Player Design

class PlayerRandom

PlayerRandom is a *Player* who places bets in Roulette. This player makes random bets around the layout.

17.2.1 Fields

`PlayerRandom.rng`

A Random Number Generator which will return the next random number.

When writing unit tests, we will want to patch this with a mock object to return a known sequence of bets.

17.2.2 Constructors

`PlayerRandom.__init__(table)`

This uses the `super()` construct to invoke the superclass constructor using the *Table*.

Parameters **table** (Table) – The *Table* which will accept the bests.

This will create a `random.Random` random number generator.

It will also use the wheel associated with the table to get the set of bins. The set of bins is then used to create the pool of outcomes for creating bets.

17.2.3 Methods

`PlayerRandom.placeBets(self)`

Updates the *Table* with a randomly placed bet.

17.3 Random Player Deliverables

There are five deliverables from this exercise. The new classes need Python docstrings.

- Updates to the class *Bin* to return an iterator over available *Outcome*s. Updates to unittests for the class *Bin*, also.
- Updates to the *Wheel* to return an iterator over available *Bin*s. Updates to the unittests for the class *Wheel*, also.
- The *PlayerRandom* class.
- A unit test of the *PlayerRandom* class. This should use the NonRandom number generator to iterate through all possible *Outcome*s.
- An update to the overall *Simulator* that uses the *PlayerRandom*.

CHAPTER

EIGHTEEN

PLAYER 1-3-2-6 CLASS

This section will describe a player who has a complex internal state. We will also digress on the way the states can be modeled using a group of polymorphic classes.

We'll start by examining the essential 1-3-2-6 betting strategy in *Player 1-3-2-6 Analysis*.

This will lead us to considering stateful design and how to properly handle polymorphism. This is the subject of *On Polymorphism*.

We'll address some additional design topics in *Player 1-3-2-6 Questions and Answers*.

The design is spread across several sections:

- *Player1326 State Design*,
- *Player1326 No Wins Design*,
- *Player1326 One Win Design*,
- *Player1326 Two Wins Design*, and
- *Player1326 Three Wins_*.

Once we've defined the various states, the overall player can be covered in *Player1326 Design*.

We'll enumerate the deliverables in *Player 1-3-2-6 Deliverables*.

There are some more advanced topics here. First, we'll look at alternative designs in *Advanced Exercise – Refactoring*. Then, we'll look at ways to reuse state objects in *Advanced Exercise – Less Object Creation*. Finally, we'll examine the **Factory** design pattern in *Player1326 State Factory Design*.

18.1 Player 1-3-2-6 Analysis

On the Internet, we found descriptions of a betting system called the "1-3-2-6" system. This system looks to recoup losses by waiting for four wins in a row. The sequence of numbers (1, 3, 2 and 6) are the multipliers to use when placing bets after winning.

At each loss, the sequence resets to the multiplier of 1.

At each win, the multiplier is advanced. It works like this:

- After one win, the bet advances to 3×. This is done by leaving the winnings in place, and adding to them.
- After a second win, the bet is reduced to 2×, and the winnings of 4× are taken off the table.
- In the event of a third win, the bet is advanced to 6× by putting the 4× back into play.

Should there be a fourth win, the sequence resets to 1×. The winnings from the 6× bet are the hoped-for profit.

This betting system makes our player more stateful than in previous betting systems. When designing `SevenReds`, we noted that this player was stateful; that case, the state was a simple count.

In this case, the description of the betting system seems to identify four states: no wins, one win, two wins, and three wins. In each of these states, we have specific bets to place, and state transition rules that tell us what to do next. The following table summarizes the states, the bets and the transition rules.

Table 18.1: 1-3-2-6 Betting States

Current State	Bet Amount	On loss, change to:	On win, change to:
No Wins	1	One Win	No Wins
One Win	3	Two Wins	No Wins
Two Wins	2	Three Wins	No Wins
Three Wins	6	No Wins	No Wins

When we are in a given state, the table gives us the amount to bet in the *Bet* column. If this bet wins, we transition to the state named in the *On Win* column, otherwise, we transition to the state named in the *On Loss* column. We always start in the No *Wins* state.

Design Pattern. We can exploit the **State** design pattern to implement this more sophisticated player. The design pattern suggests that we create a hierarchy of classes to represent these four states. Each state will have a slightly different bet amount, and different state transition rules. Each individual state class will be relatively simple, but we will have isolated the processing unique to each state into separate classes.

One of the consequences of the **State** design pattern is that it obligates us to define the interface between the *Player* and the object that holds the *Player*'s current state.

It seems best to have the state object follow our table and provide three methods: `currentBet()`, `nextWon()`, and `nextLost()`. The *Player* can use these methods of the state object to place bets and pick a new state.

- A state's `currentBet()` method will construct a *Bet* from an *Outcome* that the *Player* keeps, and the multiplier that is unique to the state. As the state changes, the multiplier moves between 1, 3, 2 and 6.
- A state's `nextWon()` method constructs a new state object based on the state transition table when the last bet was a winner.
- A state's `nextLost()` method constructs a new state based on the state transition table when the last bet was a loser. In this case, all of the various states create a new instance of the `NoWins` object, resetting the multiplier to 1 and starting the sequence over again.

18.2 On Polymorphism

One very important note is that we never have to check the class of any object. This is so important, we will repeat it here.

Important: We don't use `isistance()`.

We use *polymorphism* and design all subclasses to have the same interface.

Python relies on *duck typing*:

> **"if it walks like a duck and quacks like a duck, it is a duck"**

This defines class membership using a very simple rule. If an object has the requested method, then it's a member of the class.

Additionally, Python relies on the principle that it's better to seek forgiveness than ask permission.

What this translates to is an approach where an object's methods are simply invoked. If the object implements the methods, then it walked like a duck and – for all practical purposes – actually *was* a duck. If the method is not implemented, the application has a serious design problem and needs to crash.

We find that `isinstance()` is sometimes used by beginning programmers who have failed to properly delegate processing to the subclass.

Often, when a responsibility has been left out of the class hierarchy, it is allocated to the client object. The typical Pretty-Poor Polymorphism looks like the following:

Pretty Poor Polymorphism

```
class SomeClient( object ):
    def someMethod( self ):
        if isinstance(x, AClass):
            Special Case that should have been part of AClass
```

In all cases, uses of `isinstance()` must be examined critically.

Generally, we can avoid `isinstance()` tests by refactoring the special case out of the collaborating class. There will be three changes as part of the refactoring.

1. We will move the special-case the functionality into the class being referenced by `isinstance()`. In the above example, the special case is moved to `AClass`.
2. We will usually have to add default processing to the superclass of `AClass` so that all other sibling classes of `AClass` will have an implementation of the special-case feature.
3. We simply call the refactored method from the client class.

This refactoring leads to a class hierarchy that has the property of being *polymorphic*: all of the subclasses have the same interface: all objects of any class in the hierarchy are interchangable. Each object is, therefore, responsible for correct behavior. More important, a client object does not need to know which subclass the object is a member of: it simply invokes methods which are defined with a uniform interface across all subclasses.

18.3 Player 1-3-2-6 Questions and Answers

Why code the state as objects?

> The reason for encoding the states as objects is to encapsulate the information and the behavior associated with that state. In this case, we have both the bet amount and the rules for transition to next state. While simple, these are still unique to each state.
>
> Since this is a book on design, we feel compelled to present the best design. In games like blackjack, the player's state may have much more complex information or behavior. In those games, this design pattern will be very helpful. In this one case only, the design pattern appears to be over-engineered.
>
> We will use this same design pattern to model the state changes in the Craps game itself. In the case of the Craps game, there is additional information as well as behavior changes. When the state changes, bets are won or lost, bets are working or not, and outcomes are allowed or prohibited.

Isn't it simpler to code the state as a number? We can just increment when we win, and reset to zero when we lose.

> The answer to all "isn't it simpler" questions is "yes, but..." In this case, the full answer is "Yes, but what happens when you add a state or the states become more complex?"
>
> This question arises frequently in OO programming. Variations on this question include "Why is this an entire object?" and "Isn't an object over-engineering this primitive type?" See *Design Decision – Object Identity* FAQ entry on the `Outcome` class for additional background on object identity.
>
> Our goal in OO design is to isolate responsibility. First, and most important, we can unambiguously isolate the responsibilities for each individual state. Second, we find that it is very common that only one state changes, or new states get added. Given these two conditions, the best object model is separate state objects.

Doesn't this create a vast number of state objects?

Yes.

There are two usual follow-up questions: "Aren't all those objects a lot of memory overhead?" or "...a lot of processing overhead?"

Since Python removes unused objects, the old state definitions are removed when no longer required.

Object creation is an overhead that we can control. One common approach is to use the **Singleton** design pattern. In this case, this should be appropriate because we only want a single instance of each of these state classes.

Note that using the **Singleton** design pattern doesn't change any interfaces except the initialization of the `Player1326` object with the starting state.

Is Polymorphism necessary?

In some design patterns, like **State** and **Command**, it is essential that all subclasses have the same interface and be uniform, indistinguishable, almost anonymous instances. Because of this polymorphic property, the objects can be invoked in a completely uniform way.

In our exercise, we will design a number of different states for the player. Each state has the same interface. The actual values for the instance variables and the actual operation implemented by a subclass method will be unique. Since the interfaces are uniform, however, we can trust all state objects to behave properly.

There are numerous designs where polymorphism doesn't matter at all. In many cases, the anonymous uniformity of subclasses isn't relevant. When we move on to example other casino games, we will see many examples of non-polymorphic class hierarchies. This will be due to the profound differences between the various games and their level of interaction with the players.

18.4 Player1326 State Design

class `Player1326State`

`Player1326State` is the superclass for all of the states in the 1-3-2-6 betting system.

18.4.1 Fields

`Player1326State.`**`player`**

The `Player1326` player who is currently in this state. This player will be used to provide the `Outcome` that will be used to create the `Bet`.

18.4.2 Constructors

`Player1326State.`**`__init__`**`(`*self*, *player*`)`

The constructor for this class saves the `Player1326` which will be used to provide the `Outcome` on which we will bet.

18.4.3 Methods

`Player1326State.`**`currentBet`**`(`*self*`)` → Bet

Constructs a new `Bet` from the player's preferred `Outcome`. Each subclass has a different multiplier used when creating this `Bet`.

In Python, this method should return `NotImplemented`. This is a big debugging aid, it helps us locate subclasses which did not provide a method body.

`Player1326State.`**`nextWon`**`(`*self*`)` → Player1326State
Constructs the new *`Player1326State`* instance to be used when the bet was a winner.

In Python, this method should return `NotImplemented`. This is a big debugging aid, it helps us locate subclasses which did not provide a method body.

`Player1326State.`**`nextLost`**`(`*self*`)` → Player1326State
Constructs the new *`Player1326State`* instance to be used when the bet was a loser. This method is the same for each subclass: it creates a new instance of *`Player1326NoWins`*.

This defined in the superclass to assure that it is available for each subclass.

18.5 Player1326 No Wins Design

class `Player1326NoWins`

`Player1326NoWins` defines the bet and state transition rules in the 1-3-2-6 betting system. When there are no wins, the base bet value of 1 is used.

18.5.1 Methods

`Player1326NoWins.`**`currentBet`**`(`*self*`)` → Bet
Constructs a new *`Bet`* from the player's `outcome` information. The bet multiplier is 1.

`Player1326NoWins.`**`nextWon`**`(`*self*`)` → Player1326State
Constructs the new *`Player1326OneWin`* instance to be used when the bet was a winner.

18.6 Player1326 One Win Design

class `Player1326OneWin`

`Player1326OneWin` defines the bet and state transition rules in the 1-3-2-6 betting system. When there is one wins, the base bet value of 3 is used.

18.6.1 Methods

`Player1326OneWin.`**`currentBet`**`(`*self*`)` → Bet
Constructs a new *`Bet`* from the player's `outcome` information. The bet multiplier is 3.

`Player1326OneWin.`**`nextWon`**`(`*self*`)` → Player1326State
Constructs the new *`Player1326TwoWins`* instance to be used when the bet was a winner.

18.7 Player1326 Two Wins Design

class `Player1326TwoWins`

`Player1326TwoWins` defines the bet and state transition rules in the 1-3-2-6 betting system. When there are two wins, the base bet value of 2 is used.

18.7.1 Methods

Player1326TwoWins.**currentBet**(*self*) → Bet

Constructs a new *Bet* from the player's outcome information. The bet multiplier is 2.

Player1326TwoWins.**nextWon**(*self*) → Player1326State

Constructs the new *Player1326ThreeWins* instance to be used when the bet was a winner.

18.8 Player1326 Three Wins

class **Player1326ThreeWins**

Player1326ThreeWins defines the bet and state transition rules in the 1-3-2-6 betting system. When there are three wins, the base bet value of 6 is used.

18.8.1 Methods

Player1326ThreeWins.**currentBet**(*self*) → Bet

Constructs a new *Bet* from the player's outcome information. The bet multiplier is 6.

Player1326ThreeWins.**nextWon**(*self*) → Player1326State

Constructs the new *Player1326NoWins* instance to be used when the bet was a winner.

An alternative is to update the player to indicate that the player is finished playing.

18.9 Player1326 Design

class **Player1326**

Player1326 follows the 1-3-2-6 betting system. The player has a preferred *Outcome*, an even money bet like red, black, even, odd, high or low. The player also has a current betting state that determines the current bet to place, and what next state applies when the bet has won or lost.

18.9.1 Fields

Player1326.**outcome**

This is the player's preferred *Outcome*. During construction, the Player must fetch this from the *Wheel*.

Player1326.**state**

This is the current state of the 1-3-2-6 betting system. It will be an instance of a subclass of *Player1326State*. This will be one of the four states: No Wins, One Win, Two Wins or Three Wins.

18.9.2 Constructors

Player1326.**__init__**(*self*)

Initializes the state and the outcome. The *state* is set to the initial state of an instance of *Player1326NoWins*.

The *outcome* is set to some even money proposition, for example "Black".

18.9.3 Methods

`Player1326.placeBets(self)`

Updates the *Table* with a bet created by the current state. This method delegates the bet creation to *state* object's `currentBet()` method.

`Player1326.win(self, bet)`

Parameters bet (Bet) – The Bet which won

Uses the superclass method to update the stake with an amount won. Uses the current state to determine what the next state will be by calling *state*'s objects `nextWon()` method and saving the new state in *state*

`Player1326.lose(self, bet)`

Parameters bet (Bet) – The Bet which lost

Uses the current state to determine what the next state will be. This method delegates the next state decision to *state* object's `nextLost()` method, saving the result in *state*.

18.10 Player 1-3-2-6 Deliverables

There are eight deliverables for this exercise. Additionally, there is an optional, more advanced design exercise in a separate section.

- The five classes that make up the *Player1326State* class hierarchy.
- The *Player1326* class.
- A Unit test for the entire *Player1326State* class hierarchy. It's possible to unit test each state class, but they're so simple that it's often easier to simply test the entire hierarchy.
- A unit test of the *Player1326* class. This test should synthesize a fixed list of *Outcome*s, *Bin*s, and calls a *Player1326* instance with various sequences of reds and blacks. There are 16 different sequences of four winning and losing bets. These range from four losses in a row to four wins in a row.
- An update to the overall *Simulator* that uses the *Player1326*.

18.11 Advanced Exercise – Refactoring

Initially, each subclass of *Player1326State* has a unique `currentBet()` method.

This class can be simplified slightly to have the bet multiplier coded as an instance variable, `betAmount`. The `currentBet()` method can be refactored into the superclass to use the `betAmount` value.

This would simplify each subclass to be only a constructor that sets the `betAmount` multiplier to a value of 1, 3, 2 or 6.

Similarly, we could have the next state after winning defined as an instance variable, `nextStateWin`. We can initialized this during construction. Then the `nextWon()` method could also be refactored into the superclass, and would return the value of the `nextStateWin` instance variable.

The deliverable for this exercise is an alternative *Player1326State* class using just one class and distinct constructors that create each state object with an appropriate bet multipler and and next state win value.

18.12 Advanced Exercise – Less Object Creation

The object creation for each state change can make this player rather slow.

There are a few design pattens that can reduce the number of objects that need to be created.

1. Global objects for the distinct state objects.

   While truly global objects are usually a mistake, we can justify this by claiming that we're only creating objects that are shared among a few objects.

   The objects aren't global variables: they're global constants. There's no state change, so they aren't going to be shared improperly.

2. The **Singleton** design pattern.

   With some cleverness, this can be made transparent in Python. However, it's easiest to make this explicit, by defining a formal `instance()` method that fetches (or creates) the one-and-only instance of the class.

3. A **Factory** object that produces state objects. This **Factory** can retain a small pool of object instances, eliminating needless object construction.

The deliverables for this exercise are revisions to the *Player1326State* class hierarchy to implement the **Factory** design pattern. This will not change any of the existing unit tests, demonstrations or application programs.

18.12.1 Player1326 State Factory Design

`Player1326StateFactory.`**values**

This is a map from a class name to an object instance.

`Player1326StateFactory.`**__init__**(*self*)

Create a new mapping from the class name to object instance. There are only four objects, so this is relatively simple.

`Player1326StateFactory.`**get**(*self*, *name*) → Player1326State

Parameters **name** (*String*) – name of one of the subclasses of *Player1326State*.

CHAPTER

NINETEEN

CANCELLATION PLAYER CLASS

This section will describe a player who has a complex internal state that can be modeled using existing library classes.

In *Cancellation Player Analysis* we'll look at what this player does.

We'll turn to how the player works in *PlayerCancellation Design*.

In *Cancellation Player Deliverables* we'll enumerate the deliverables for this player.

19.1 Cancellation Player Analysis

One method for tracking the lost bets is called the "cancellation" system or the "Labouchere" system. The player starts with a betting budget allocated as a series of numbers.

The usual example sequence is `[ 1, 2, 3, 4, 5, 6 ]`.

Each bet will be the sum of the first and last numbers in the last.

In this example, the end values of 1+6 leads the player to bet 7.

When the player wins, the player cancels the two numbers used to make the bet. In the event that all the numbers are cancelled, the player has doubled their money, and can retire from the table happy.

For each loss, however, the player adds the amount of the bet to the end of the sequence; this is a loss to be recouped. This adds the loss to the amount bet to assure that the next winning bet both recoups the most recent loss and provides a gain. Multiple winning bets will recoup multiple losses, supplemented with small gains.

Example. Here's an example of the cancellation system using `[ 1, 2, 3, 4, 5, 6 ]`

1. Bet 1+6. A win. Cancel 1 and 6 leaving `[ 2, 3, 4, 5 ]`
2. Bet 2+5. A loss. Add 7 leaving `[ 2, 3, 4, 5, 7 ]`
3. Bet 2+7. A loss. Add 9 leaving `[ 2, 3, 4, 5, 7, 9 ]`
4. Bet 2+9. A win. Cancel 2 and 9 leaving `[ 3, 4, 5, 7 ]`
5. Next bet will be 3+7.

State. The player's state is a list of individual bet amounts. This list grows and shrinks; when it is empty, the player leaves the table. We can keep a `List` of individual bet amounts. The total bet will be the first and last elements of this list. Wins will remove elements from the collection; losses will add elements to the collection. Since we will be accessing elements in an arbitrary order, we will want to use an `ArrayList`. We can define the player's state with a simple list of values.

19.2 PlayerCancellation Design

class PlayerCancellation

PlayerCancellation uses the cancellation betting system. This player allocates their available budget into a sequence of bets that have an accelerating potential gain as well as recouping any losses.

19.2.1 Fields

PlayerCancellation.**sequence**
This List keeps the bet amounts; wins are removed from this list and losses are appended to this list. THe current bet is the first value plus the last value.

PlayerCancellation.**outcome**
This is the player's preferred *Outcome*.

19.2.2 Constructors

PlayerCancellation.**__init__**(*self*)
This uses the *PlayerCancellation.resetSequence()* method to initalize the sequence of numbers used to establish the bet amount. This also picks a suitable even money *Outcome*, for example, black.

19.2.3 Methods

PlayerCancellation.**resetSequence**(*self*)
Puts the initial sequence of six Integer instances into the *sequence* variable. These Integers are built from the values 1 through 6.

PlayerCancellation.**placeBets**(*self*)
Creates a bet from the sum of the first and last values of *sequence* and the preferred outcome.

PlayerCancellation.**win**(*self*, *bet*)

Parameters **bet** (Bet) – The bet which won

Uses the superclass method to update the stake with an amount won. It then removes the fist and last element from *sequence*.

PlayerCancellation.**lose**(*self*, *bet*)

Parameters **bet** (Bet) – The bet which lost

Uses the superclass method to update the stake with an amount lost. It then appends the sum of the first and list elements of *sequence* to the end of *sequence* as a new Integer value.

19.3 Cancellation Player Deliverables

There are three deliverables for this exercise.

- The *PlayerCancellation* class.
- A unit test of the *PlayerCancellation* class. This test should synthesize a fixed list of *Outcome*s, *Bin* s, and calls a *PlayerCancellation* instance with various sequences of reds and blacks. There are 16 different sequences of four winning and losing bets. These range from four losses in a row to four wins in a row. This should be sufficient to exercise the class and see the changes in the bet amount.
- An update to the overall *Simulator* that uses the *PlayerCancellation*.

CHAPTER

TWENTY

FIBONACCI PLAYER CLASS

This section will describe a player who has an internal state that can be modeled using methods and simple values instead of state objects.

This is a variation on the Martingale System. See *Martingale Player Design* for more information.

In *Fibonacci Player Analysis* we'll look at what this player does.

We'll turn to how the player works in *PlayerFibonacci Design*.

In *Fibonacci Player Deliverables* we'll enumerate the deliverables for this player.

20.1 Fibonacci Player Analysis

A player could use the *Fibonacci Sequence* to structure a series of bets in a kind of cancellation system. The Fibonacci Sequence is

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

At each loss, the sum of the previous two bets is used, which is the next number in the sequence. In the event of a win, we revert to the basic bet.

Example. Here's an example of the Fibonacci system.

1. Bet 1. A win.
2. Bet 1. A loss. The next value in the sequence is 1.
3. Bet 1. A loss. The next value in the sequence is 2.
4. Bet 2. A loss. The next value in the sequence will be 3
5. Bet 3. In the event of a loss, the next bet is 5. Otherwise, the bet is 1.

State. In order to compute the Fibonacci sequence, we need to retain the two previous bets as the player's state. In the event of a win, we revert to the basic bet value of `1`.

In the event of a loss, we can update the two numbers to show the next step in the sequence. The player's state is just these two numeric values.

20.2 PlayerFibonacci Design

class `PlayerFibonacci`

`PlayerFibonacci` uses the Fibonacci betting system. This player allocates their available budget into a sequence of bets that have an accelerating potential gain.

20.2.1 Fields

`PlayerFibonacci.recent`
This is the most recent bet amount. Initially, this is 1.

`PlayerFibonacci.previous`
This is the bet amount previous to the most recent bet amount. Initially, this is zero.

20.2.2 Constructors

`PlayerFibonacci.__init__(self)`
Initialize the Fibonacci player.

20.2.3 Methods

`PlayerFibonacci.win(self, bet)`

Parameters **bet** (Bet) – The bet which won

Uses the superclass method to update the stake with an amount won. It resets *recent* and *previous* to their initial values of 1 and 0.

`PlayerFibonacci.lose(self, bet)`

Parameters **bet** (Bet) – The bet which lost

Uses the superclass method to update the stake with an amount lost. This will go "forwards" in the sequence. It updates *recent* and *previous* as follows.

$$next \leftarrow recent + previous$$
$$previous \leftarrow recent$$
$$recent \leftarrow next$$

Parameters **bet** (Bet) – The *Bet* which lost.

20.3 Fibonacci Player Deliverables

There are three deliverables for this exercise.

- The *PlayerFibonacci* class.
- A unit test of the *PlayerFibonacci* class. This test should synthesize a fixed list of *Outcome*s, *Bin*s, and calls a *PlayerFibonacci* instance with various sequences of reds and blacks. There are 16 different sequences of four winning and losing bets. These range from four losses in a row to four wins in a row. This should be sufficient to exercise the class and see the changes in the bet amount.
- An update to the overall *Simulator* that uses the *PlayerFibonacci*.

CHAPTER TWENTYONE

CONCLUSION

The game of Roulette has given us an opportunity to build an application with a considerable number of classes and objects. It is comfortably large, but not complex; we have built tremendous fidelity to a real-world problem. Finally, this program produces moderately interesting simulation results.

In this section we'll look at our overall approach to design in *Exploration*.

We'll look at some design principles in *The SOLID Principles*.

In *Other Design Patterns* we'll look at some of the other design patterns we've been using.

21.1 Exploration

We note that a great many of our design decisions were not easy to make without exploring a great deal of the overall application's design. We've shown how to do this exploration: design just enough but remain tolerant of our own ignorance.

There's an idealized fantasy in which a developer design an entire, complex application before writing any software. The process for creating a complete design is still essentially iterative. Some parts are designed in detail, with tolerance for future changes; then other parts are designed in detail and the two design elements reconciled. This **can** be done on paper or on a whiteboard.

With Python, however, we can write – and revise – draft programming as easily as erasing a whiteboard. It's quite easy to do incremental design by writing and revising a working base of code.

For new designers, we can't give enough emphasis to the importance of creating a trial design, exploring the consequences of that design, and then doing rework of that design. Too often, we have seen trial designs finalized into deliverables with no opportunity for meaningful rework. In *Review of Testability*, we presented one common kind of rework to support more complete testing. In *Player Class*, we presented another kind of rework to advance a design from a stub to a complete implementation.

21.2 The SOLID Principles

There are five principles of object-oriented design. We've touched on several. For more information, see https://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

S Single responsibility principle. We've emphasized this heavily by trying to narrow the scope of responsibility in each class.

O Open/closed principle. The terminology here is important. A class is open to extension but closed to modification. We prefer to wrap or extend classes. We prefer not to modify or tweak a class. When we make a change to a class, we are careful to be sure that the ripple touches the entire hierarchy and all of the collaborators.

L Liskov substitution principle. In essence, this is a test for proper polymorphism. If classes are truly polymorphic, one can be substituted for another. We've generally focused on assuring this.

I Interface segregation principle. In most of what we're doing, we've kept interfaces as narrow as possible. When confronted with **Wrap vs. Extend** distinctions, the idea of interface segregation suggests that we should prefer wrapping a class because that tends to narrow the interface.

D Dependency inversion principle. This principle guides us toward creating common abstractions. Our *Player* is an example of this. We didn't create games and tables that depend on a specific player. We created games and tables that would work with any class that met the minimal requirements for the *Player* interface.

21.3 Other Design Patterns

We also feel compelled to point out the distinction between relatively active and passive classes in this design. We had several passive classes, like *Outcome*, *Bet*, and *Table*, which had few responsibilities beyond collecting a number of related attributes and providing simple functions.

We also had several complex, active classes, like *Game*, *BinBuilder* and all of the variations on *Player*. These classes, typically, had fewer attributes and more complex methods. In the middle of the spectrum is the *Wheel*.

We find this distinction to be an enduring feature of OO design: there are *things* and *actors*; the things tend to be passive, acted upon by the actors. The overall system behavior emerges from the collaboration among all of the objects in the system; primarily – but not exclusively – the behavior of the active classes.

Part III

Craps

This part describes parts of the more complex game of Craps. Craps is played with dice. A player throws the dice; sometimes it's an immediate win (or loss). Other times, the number on the dice become the "point", and you contine throwing dice until you make your point or crap out.

Craps is a game with two states and a number of state-change rules. It has a variety betting alternatives, some of which are quite complex.

The chapters of this part presents the details on the game, an overview of the solution, and a series of eleven exercises to build a complete simulation of the game, with a variety of betting strategies. The exercises in this part are more advanced; unlike *Roulette*, we will often combine several classes into a single batch of deliverables.

There are several examples of rework in this part, some of them quite extensive. This kind of rework reflects three more advanced scenarios: refactoring to generalize and add features, renaming to rationalize the architecture, and refactoring to extract features. Each of these is the result of learning; they are design issues that can't easily be located or explained in advance.

CHAPTER

TWENTYTWO

CRAPS DETAILS

In the first section of this chapter, *Craps Game*, we will present elements of the game of Craps. Craps is a stateful game, and we'll describe the state changes in some detail.

We'll look at the way dice work in *Creating A Dice Frequency Distribution*.

We will review some of the bets available on the Craps table in some depth in *Available Bets*. Craps offers some very obscure bets, and part of this obscurity stems from the opaque names for the various bets. Creating this wide variety of bets is an interesting programming exercise, and the first four exercise chapters will focus on this.

In *Some Betting Strategies* we will describe some common betting strategies that we will simulate. The betting strategies described in *Roulette* will be applied to Craps bets, forcing us to examine the Roulette solution for reuse opportunities.

We'll also look at an alternative set of bets in *Wrong Betting*.

22.1 Craps Game

Craps centers around a pair of *dice* with six sides. The 36 combinations form eleven numbers from 2 to 12; additionally, if the two die have equal values, this is called a *hardways*, because the number was made "the hard way".

The *table* has a surface marked with spaces on which players can place *bets*. The names of the bets are opaque, and form an obscure jargon. There are several broad classification of bets, including

- those which follow the sequence of dice throws that make up a complete *game*,
- the *propositions* based on only the next throw of the dice,
- some "hardways" bets on rolling an equal pair before the game is revolved, and,
- some unmarked bets.

These bets will be defined more completely, below, under *Available Bets*.

When the bets are placed, the dice are thrown by one of the players. This number resolves the one-roll bets that are winners or losers; it also determines the state change in the game, which may resolve any game-based winners and losers.

The first roll of the dice can be an immediate win or loss for the game, or it can establish the *point* for the following rounds of this game. Subsequent rolls either make the point and win the game, roll a seven and lose the game, or the game continues. When the game is won, the winning player continues to shoot; a losing player passes the dice around the table to the next player.

In addition to the bets marked on the table, there are some additional bets which are not marked on the table. Some of these are called *odds bets*, or *free odds bets*, and pay off at odds that depend on the dice, not on the bet itself. An odds bet must be associated with a bet called a *line bet*, and are said to be "behind" the original bet; they can be called "behind the line odds" bets. The four line bets will be defined more fully, below.

Note: Casino Variation

There are slight variations in the bets available in different casinos. We'll focus on a common subset of bets.

The dice form a frequency distribution with values from 2 to 12. An interesting preliminary exercise is to produce a table of this distribution with a small demonstration program. This frequency distribution is central to understanding the often cryptic rules for Craps. See *Creating A Dice Frequency Distribution* for a small application to develop this frequency distribution.

The Craps game has two states: *point off* and *point on*. When a player begins a game, the point is off; the initial throw of the dice is called the "come out roll". The number thrown on the dice determines the next state, and also the bets that win or lose. The rules are summarized in the following table.

Craps Game States

Table 22.1: Point is off. Only Pass and Don't Pass Bets Allowed.

Roll	Bet Resolution	Next State
2, 3, 12	"Craps": Pass bets lose, Don't Pass bets win.	Point Off
7, 11	"Winner": Pass bets win, Don't Pass bets lose.	Point Off
4, 5, 6, 8, 9, 10		Point on the number rolled, *p*.

Table 22.2: Point is on p. Additional Bets Allowed.

Roll	Bet Resolution	Next State
2, 3, 12		Point still on *p*
11		Point still on *p*
7	"Loser": **all** bets lose. The table is cleared.	Point Off
Point, *p*	"Winner": point is made, Pass bets win, Don't Pass bets lose.	Point Off
Non-*p* number	Come bets are activated on this new number	Point still on *p*

There is a symmetry to these rules. The most frequent number, 7, is either an immediate winner or an immediate loser. The next most frequent numbers are the six point numbers, 4, 5, 6, 8, 9, 10, which form three sets: 4 and 10 are high odds, 5 and 9 are middle odds, 6 and 8 are low odds. The relatively rare numbers (2, 3, 11 and 12) are handled specially on the first roll only, otherwise they are ignored. Because of the small distinction made between the *craps* numbers (2, 3 and 12), and 11, there is an obscure "C-E" bet to cover both craps and eleven.

22.2 Creating A Dice Frequency Distribution

Since Python is interpreted, you can enter the following directly in the interpreter and get the frequency of each number.

Dice Frequency I

```
from collections import defaultdict
freq= defaultdict(int)
for d1 in range(6):
    for d2 in range(6):
        n= d1+d2+2
        freq[n] += 1
print( freq )
```

We've enumerate all 36 combinations of values `(d1, d2)`. The sum of these will be a value between 2 and 12. The list `freq` is a dictionary that maps an integer sum of the dice to an integer count of the number of occurrences of that sum.

This can also be done with a counter and a generator expression.

Dice Frequency II

```
from collections import Counter
freq= Counter( (d1+d2+2) for d1 in range(6) for d2 in range(6) )
print( freq )
```

Here we've used a `Counter`. We've initialized the counter using a generator expression that – similar to the previous example – enumerates all 36 combinations of values `(d1, d2)`.

The `Counter` consumes all of the values from the generator expression. It automatically finds matching values and counts how many times the duplicate values were seen.

22.3 Available Bets

There are four broad classification of bets: - those which follow the sequence of dice throws that make up a complete *game*, - those based on just the next throw of the dice (called *proposition bets*), - the hardways bets, and - the unmarked bets.

The game bets are also called line bets, can be supplemented with additional odds bets, placed behind the line bet. There are actually two sets of game bets based on the pass line and the come line.

The following illustration shows half of a typical craps table layout. On the left are the game bets. On the right (in the center of the table) are the single-roll propositions.

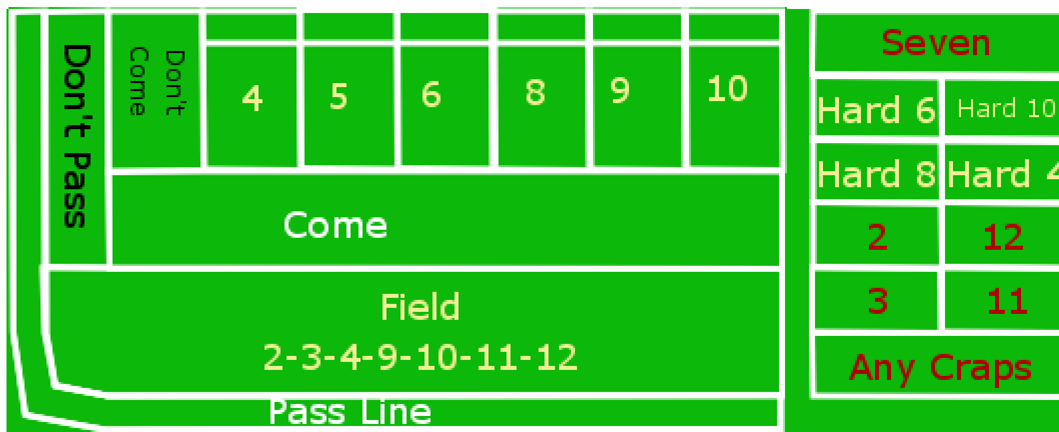

Fig. 22.1: Craps Table Layout

Game Bets. The Game bets have two states. There is a set of bets allowed for the first roll, called the "come out" roll. If a point is established a number of additional bets become available. These are the pass line bets.

In addition to the basic pass line bets, there are also come line bets, which are similar to game bets.

Come Out Roll. When the point is *off*, the player will be making the come out roll, and only certain bets are available. Once the point is *on*, all bets are available.

The bets which are allowed on the come out roll are the *Pass Line* and *Don't Pass Line* bets. These bets follow the action of the player's game. There are several possible outcomes for these two bets.

- Player throws craps (2, 3, or 12). A Pass Line bet loses. A Don't Pass Line bet wins even money on 2 or 3; however, on 12, it returns the bet. This last case is called a "push" ; this rule is noted on the table by the "bar 12" legend in the Don't Pass area.
- Player throws 7 or 11. A Pass Line bet wins even money. A Don't Pass Line bet loses.
- Player throws a point number. The point is now *on*. A large white token, marked "on" is placed in one of the six numbered boxes to announce the point. The player can now place an additional odds bet behind the line. Placing a bet behind the Pass Line is called "buying odds on the point". Placing a bet behind the Don't Pass Line is called "laying odds against the point".

Point Rolls. Once the point is on, there are three possible outcomes.

- The player makes the point before rolling 7. A Pass Line bet wins even money. A Don't Pass Line bet loses. The point determines the odds used to pay the behind the Pass Line odds bet.
- The player throws 7 before making the point. A Pass Line bet loses. A Don't Pass Line bet pays even money. The point determines the odds used to pay the behind the Don't Pass Line odds bet.
- The player throws a number other than 7 and the point: the game continues.

Come Line Bets. Once the point is on, the bets labeled *Come* and *Don't Come* are allowed. These bets effectively form a game parallel to the main game, using a come point instead of the main game point. Instead of dropping the large "on" token in a numbered square on the layout, the house will move the individual Come and Don't Come bets from their starting line into the numbered boxes (4, 5, 6, 8, 9, 10) to show which bets are placed on each point number. There are several possible outcomes for these two bets.

- Player throws craps (2, 3, or 12). A Come Line bet loses. A Don't Come bet wins even money on 2 or 3; it is a push on 12. Established come point bets in the numbered squares remain.
- Player throws 11. A Come Line bet wins even money. A Don't Come bet loses. Established come point bets in the numbered squares remain.
- Player throws a point number (4, 5, 6, 8, 9, or 10). Any come bets in that numbered box are winners – the come point was made – and the point determines the odds used to pay the behind the line odds. (Note that if this is the point for the main game, there will be no bets in the box, instead the large "on" token will be sitting there.) New bets are moved from the Come Line and Don't Come Line to the numbered box, indicating that this bet is waiting on that point number to be made. Additional behind the line odds bets can now be placed. If the main game's point was made, the "on" token will be removed (and flipped over to say "off"), and the next roll will be a come out roll for the main game. These bets (with the exception of the free odds behind a Come Line bet) are still working bets.
- The player throws 7 before making the main point. A Come bet loses. A Don't Come bet pays even money. The bets in the numbered boxes are all losers.

Propositions. There are also a number of one-roll *propositions*, including the individual numbers 2, 3, 7, 11, 12; also *field* (2, 3, 4, 9, 10, 11 or 12), *any craps* (2, 3 or 12), the *horn* (2, 3, 11 or 12), and a *hop bet*. These bets have no minimum, allowing a player to cover a number of them inexpensively.

Hardways. There are four hardways bets. These are bets that 4, 6, 8 or 10 will come up on two equal die (the hard way) before they come up on two unequal die (the easy way) or before a 7 is rolled. Hard 6, for example, is a bet that the pair (3,3) will be rolled before any of the other four combinations (1,5), (2,4), (5,1), (4,2) that total 6, or any of the 6 combinations that total 7. These bets can only be placed when the point is on, but they are neither single-roll propositions, nor are they tied to the game point.

Unmarked Bets. There are additional unmarked bets. One is to make a "place bet" on a number; these bets have a different set of odds than the odds bets behind a line bet. Other unmarked bets are to "buy" a number or "lay" a number, both of which involve paying a commission on the bet. Typically, only the six and eight are bought, and they can only be bought when they are not the point.

For programmers new to casino gambling, see *Odds and Payouts* for more information on odds payments for the various kinds of bets.

Odds and Payouts

Not all of the Craps outcomes are equal probability. Some of the bets pay fixed odds, defined as part of the bet. Other bets pay odds that depend on the point established.

The basic Pass Line, Don't Pass, Come Line and Don't Come bets are even money bets. A $5 bet wins an additional $5.

The odds of winning a Pass Line bet is 49.3%. The 1:1 payout does not reflect the actual probability of winning. The points of 4 and 10 have odds of 3 in 36 to win and 6 in 36 to lose. The points of 5 and 9 have odds of 4 in 36 to win and 6 in 36 to lose. The points of 6 and 8 have odds of 5 in 36 to win and 6 in 36 to lose. The odds, therefore are 2:1, 3:2 and 6:5 for these behind-the-line odds bets, respectively.

The combination of a Pass Line bet plus odds behind the line provides the narrowest house edge. For example, a bet on the Pass Line has an expected return of -1.414%. If we place double odds behind the line with an expected return of 0, the net expected return for the combination is -0.471%.

The various proposition bets have odds stated on the table. Note that these rarely reflect the actual odds of winning. For example, a bet on 12 is a 1/36 probability, but only pays 30:1.

There are 1/36 ways to win a hard six or hard eight bet, 10/36 ways to loose and the remaining 25 outcomes are indeterminate. While a hard 6 or hard 8 should pay 10:1, the actual payment is 9:1. Similarly for a hard 4 or hard 10: they should pay 8:1, the actual payment is 7:1.

22.4 Some Betting Strategies

All of the Roulette betting strategies apply to Craps. However, the additional complication of Craps is the ability to have multiple working bets by placing additional bets during the game. The most important of these additional bets are the behind the line odds bets. Since these are paid at proper odds, they are the most desirable bets on the layout. The player, therefore, should place a line bet, followed by a behind the line odds bet.

The other commonly used additional bets are the Come (or Don't Come) bets. These increase the number of working bets. Since the additional odds bets for these pay proper odds, they are also highly desirable. A player can place a Come bet, followed by a behind the line odds bet on the specific number rolled.

Beyond this, the bets are less interesting. All of the proposition and hardways bets have odds that are incorrect. We can simulate these bets to discover the house edge built into the incorrect odds.

Some players will count the number of throws in game, and place a "7" bet if the game lasts more than six throws. This is a form of betting against one's self. When the player has a Pass Line bet that the game will be won, they're diluted this with a 7 bet that the game will be lost. While the common rationale is that the second bet protects against a loss, it also reduces the potential win. We can simulate this kind of betting and examine the potential outcomes.

22.5 Wrong Betting

Craps has two forms of odds betting: "right" and "wrong". The Pass Line and Come Line, as well as buying odds, are all "right". The Don't Pass, Don't Come and Laying Odds are all "wrong". Sometimes wrong betting is also called "the dark side."

Right betting involves odds that provide a large reward for a small wager. The odds on a Pass Line bet for the number 4 are 2:1, you will win double your bet if the point is made. You put up $10 at risk to make $20.

Wrong betting involves odds that provide a small reward for a large wager. The odds on a Don't Pass Line bet for the number 4 are described as "laid at 2:1"; the player will win half of what was bet if the point is missed. The player put $20 at risk to make $10.

This distinction can be ignored by novice programmers. None of our simulated players are wrong bettors, since this involves putting a large amount at risk for a small payout, which violates a basic rule of gambling.

Never risk a lot to win a little.

CHAPTER TWENTYTHREE

CRAPS SOLUTION OVERVIEW

We will present a survey of the classes gleaned from the general problem statement in *Preliminary Survey of Classes*. In *Preliminary Craps Class Structure* we'll describe some potential class definitions.

Given this survey of the candidate classes, we will then do a walkthrough to refine the definitions. We'll show this in *A Walkthrough of Craps*. We can use assure ourselves that we have a reasonable architecture. We will make some changes to the preliminary class list, revising and expanding on our survey.

We will also include a number of questions and answers in *Craps Solution Questions and Answers*. This should help clarify the design presentation and set the stage for the various development exercises in the chapters that follow.

23.1 Preliminary Survey of Classes

We have divided up the responsibilities to provide a starting point for the development effort. The central principle behind the allocation of responsibility is *encapsulation*. In reading the background information and the problem statement, we noticed a number of nouns that seemed to be important parts of the Craps game.

- Dice
- Bet
- Table
- Point
- Proposition
- Number
- Odds
- Player
- House

The following table summarizes some of the classes and responsibilities that we can identify from the problem statement. This is not the complete list of classes we need to build. As we work through the exercises, we'll discover additional classes and rework some of these classes more than once.

We also have a legacy of classes available from the Roulette solution. We would like to build on this infrastructure as much as possible.

23.2 Preliminary Craps Class Structure

Outcome Responsibilities

A name for a particular betting opportunity. Most outcomes have fixed odds, but the behind the line odds bets have odds that depend on a point value.

Collaborators

Collected by *Table* into the available bets for the *Player*; used by *Game* to compute the amount won from the amount that was bet.

Dice Responsibilities

Selects any winning propositions as well as next state of the game.

Collaborators

Used by the overall *Game* to get a next set of winning *Outcome*s, as well as change the state of the *Game*.

Table Responsibilities

A collection of bets placed on *Outcome*s by a *Player*. This isolates the set of possible bets and the management of the amounts currently at risk on each bet. This also serves as the interface between the *Player* and the other elements of the game.

Collaborators

Collects the *Outcome*s; used by *Player* to place a bet amount on a specific *Outcome*; used by *Game* to compute the amount won from the amount that was bet.

Player Responsibilities

Places bets on *Outcome*s, updates the stake with amounts won and lost. This is the most important responsibility in the application, since we expect to update the algorithms this class uses to place different kinds of bets. Clearly, we need to cleanly encapsulate the *Player*, so that changes to this class have no ripple effect in other classes of the application.

Collaborators

Uses *Table* to place *Bet*s on preferred *Outcome*s; used by *Game* to record wins and losses.

Game Responsibilities

Runs the game: gets bets from *Player*, throws the *Dice*, updates the state of the game, collects losing bets, pays winning bets. This encapsulates the basic sequence of play into a single class. The overall statistical analysis is based on playing a finite number of games and seeing the final value of the *Player*'s stake.

Collaborators

Uses *Dice*, *Table*, *Outcome*, *Player*.

23.3 A Walkthrough of Craps

A good preliminary task is to review these responsibilities to confirm that a complete cycle of play is possible. This will help provide some design details for each class. It will also provide some insight into classes that may be missing from this overview. A good way to structure this task is to do a CRC walkthrough. For more information on this technique see *A Walkthrough of Roulette*.

The basic processing outline is the responsibility of the *Game* class. To start, locate the *Game* card.

1. Our preliminary note was that this class "Runs the game." The responsibilities section has a summary of five steps involved in running the game.
2. The first step is "gets bets from :class:Player'." Find the *Player* card.
3. Does a *Player* collaborate with a *Game* to place bets? Note that the game state influences the allowed bets. Does *Game* collaborate with *Player* to provide the state information? If not, add this information to one or both cards.

4. The *Game*'s second step is to throw the *Dice*. Is this collaboration on the *Dice* card?
5. The *Game*'s third step is to update the state of the game. While the state appears to be internal to the *Game*, requiring no collaboration, we note that the *Player* needs to know the state, and therefore should collaborate with *Game*. Be sure this collaboration is documented.
6. The *Game*'s fourth and fifth steps are to pay winning bets and collect losing bets. Does the *Game* collaborate with the *Table* to get the working bets? If not, update the collaborations.

Something we'll need to consider is the complex relationship between the dice, the number rolled on the dice, the game state and the various bets. In Roulette, the wheel picked a random *Bin* which had a simple list of winning bets; all other bets were losers. In Craps, however, we find that we have game bets that are based on changes to the game state, not simply the number on the dice. The random outcome is used to resolve one-roll proposition bets, resolve hardways bets, change the game state, and resolve game bets.

We also note that the house moves Come Line (and Don't Come) bets from the Come Line to the numbered spaces. In effect, the bet is changed from a generic *Outcome* to a more specific *Outcome*. This means that a *Bet* has a kind of state change, in addition to the *Game*'s state change and any possible *Player* state change.

Important: Stateful Objects

To continue this rant, we find that most interesting IT applications involve stateful objects. Everything that has a state or status, or can be updated, is stateful. Often, designers overlook these state changes, describing them as "simple updates". However, state changes are almost universally accompanied by rules that determine legal changes, events that precipitate changes, and actions that accompany a state change. Belittling stateful objects causes designers to overlook these additional details. The consequence of ignoring state is software that performs invalid or unexpected state transitions. These kinds of bugs are often insidious, and difficult to debug.

A walkthrough gives an overview of the interactions among the objects in the working application. You may uncover additional design ideas from this walkthrough. The most important outcome of the walkthrough is a clear sense of the responsibilities and the collaborations required to create the necessary application behavior.

23.4 Craps Solution Questions and Answers

Why is *Outcome* a separate class? Each object that is an instance of *Outcome* is merely a number from 2 to 12.

> Here we have complex interdependency between the dice, the game states, the bets and outcomes. An outcome has different meanings in different game states: sometimes a 7 is an immediate winner, other times it as an immediate loser. Clearly, we need to isolate these various rules into separate objects to be sure that we have captured them accurately without any confusion, gaps or conflicts.
>
> We can foresee three general kinds of *Outcome* s: the propositions that are resolved by a single throw of the dice, the hardways that are resolved periodically, and the game bets which are resolved when a point is made or missed. Some of the outcomes are only available in certain game states.
>
> The alternative is deeply nested if-statements. Multiple objects introduce some additional details in the form of class declarations, but objects have the advantage of clearly isolating responsibilities, making us more confident that our design will work properly. If-statements only conflate all of the various conditions into a tangle that includes a huge risk of missing an important and rare condition.
>
> See the discussion under *Design Decision – Object Identity* for more discussion on object identity and why each Outcome is a separate object.

What is the difference between Dice and Wheel? Don't they both represent simple collections with random selection?

> Perhaps. At the present time, the distinction appears to be in the initialization of the two collections of Bins of the Wheel or Throws of the Dice.

Generally, we are slow to merge classes together without evidence that they are really the same thing. In this case, they appear very similar, so we will note the similarities and differences as we work through the design details. There is a fine line between putting too many things together and splitting too many things apart. Generally, the mistake we see most often is putting too many things together, and resolving the differences through complex if-statements and other hidden processing logic.

CHAPTER

TWENTYFOUR

OUTCOME CLASS

This chapter will examine the *Outcome* class, and its suitability for the game of Craps. We'll present some additional code samples to show a way to handle the different kinds of constuctors that this class will need.

We'll start with *Outcome Analysis* to examine the different kinds of outcomes that are part of craps. This will lead us to some additional features, specifically *More Complex Odds* and *Commission Payments*.

In *Fractional Odds Design* we'll look at a way to design a more sophisticated set of odds. This will include *Optional Parameter Implementation Technique* to show ways to handle the optional denominator, or denominator with a default value of 1.

In *Outcome Rework* we'll revise the previous version *Outcome* to handle Craps odds. We'll enumerate the deliverables in *Outcome Deliverables*.

24.1 Outcome Analysis

For the game of Craps, we have to be careful to disentangle the random events produced by the *Dice* and the outcomes on which a *Player* creates a *Bet*. In Roulette, this relationship was simple: a *Bin* was a container of *Outcome*s; a player's *Bet* referenced one of these *Outcome*s. In Craps, however, we have one-roll outcomes, hardways outcomes, plus outcomes that aren't resolved until the end of the game. What are the varieties of outcomes?

Varietals. There is a rich variety of bet *Outcome*s. We'll itemize them so that we can confirm the responsibilities for this class.

- The Line Bets: these are the Pass Line, Don't Pass Line, Come Line, Don't Come Line. These outcomes have fixed odds of 1:1.
- The four Hardways bets: 4, 6, 8 and 10. These outcomes also have fixed odds which depend on the number.
- The various one-roll propositions. All of these have fixed odds. These are most like the original *Outcome* used for Roulette.
- The six Come-point bets (4, 5, 6, 8, 9 and 10). Each of these has fixed odds. Also, the initial line bet is moved to a point number from the Come Line bet, based on the number shown on the dice. We'll examine these is some detail, below.
- The Odds bets placed behind the Line bets. These have odds based on the point, not the outcome itself. We'll have to look at these more closely, also.
- The six Placed Numbers have odds are based on the number placed. These outcomes have fixed odds. Once the bet is placed, these bets are resolved when the number is rolled or when a game losing seven is rolled.
- The Buy and Lay bets require a commission payment, called a vigorish, when the bet is placed. The outcomes have simple, fixed odds. As with the placed number bets, these bets are resolved when the number is rolled or when a game losing seven is rolled.

Looking more closely at the bets with payout odds that depend on the point rolled, we note that the Come Line (and Don't Come) odds bets are moved to a specific number when that point is established. For example, the player places

a Come bet, the dice roll is a 4; the come bet is moved into the box labeled "4". Any additional odds bet is placed in this box, also.

This leaves us with the Pass Line and Don't Pass Line odds bet, which also depend on the point rolled. In this case, the bets are effectively moved to a specific numbered box. In a casino, a large, white, "on" token in placed in the box. The effect is that same as if the house had moved all the Pass Line bets. The odds bet, while physically behind the Pass Line, is effectively in the box identified by the point. Again, the bet is moved from the 1:1 Pass Line to one of the six numbered boxes; any odds bet that is added will be on a single `Outcome` with fixed odds.

In this case, the existing `Outcome` class still serves many of our needs. Looking forward, we will have to rework `Bet` to provide a method that will change to a different `Outcome`. This will move a line bets to one of the six numbered point boxes.

24.1.1 More Complex Odds

There are two additional responsibilities that we will need in the `Outcome` class: more complex odds and a house commission. In Roulette, all odds were stated as $n : 1$, and our `winAmount()` depended on that. In craps, many of the odds have non-unit denominators. Example odds include 6:5, 3:2, 7:6, 9:5. In a casino, the bets are multiples of $5, $6 or $10 to accomodate the fractions.

In our simulation, we are faced with two choices for managing these more complex odds: exact fractions or approximate floating-point values.

We suggest using Python's `fractions` module. We can replace the odds with a `fractions.Fraction` object. We would use something like `Outcome("Something", Fraction(2,1) )` for `2:1` odds.

We'll look at the design alternatives in the *Fractional Odds Design* section.

24.1.2 Commission Payments

The second extension we have to consider is for the bets which have a commission when they are created: buy bets and lay bets. The buy bet involves an extra 5% placed with the bet: the player puts down $21, a $20 bet and a $1 commission. A lay bet, which is a *wrong bet*, involves a risk of a large amount of money against a small win, and the commission is based on the potential winning. For a 2:3 wrong bet, the commission is 5% of the outcome; the player puts down $31 to win $20 if the point is not made.

In both buy and lay cases, the `Player` sees a price to create a bet of a given amount. Indeed, this generalizes nicely to all other bets. Most bets are simple and the price is the amount of the bet. For buy bets, however, the price is 5% of the amount of the bet; for lay bets, the price is 5% of the possible payout. The open question is the proper allocation of responsibility for this price. Is the price related to the `Outcome` or the `Bet`?

When we look at the buy and lay bets, we see that they are based on existing point number `Outcome`s and share the same odds. However, there are three very different ways create a `Bet` on one of these point number `Outcome`s:

- a bet on the Pass Line (or Don't Pass Line),
- a bet on the Come Line (or Don't Come Line), and
- a buy (or lay) bet on the number.

When we bet via the Pass Line or Come Line, the Line bet was moved to the point number, and the odds bet follows the Line bet. For this reason, the price is a feature of creating the `Bet`. Therefore, we'll consider the commission as the price of creating a bet and defer it to the `Bet` class design.

We observe that the slight error in the Line bet odds is the house's edge on the Line bet. When we put an odds bet behind the line, the more correct odds dilutes this edge. When we buy a number, on the other hand, the odds are correct and the house takes the commission directly.

24.2 Fractional Odds Design

In Roulette, all outcomes paid a multiple of the bet. For example, the “Dozen 1-12” outcome paid 2:1. We used a simple `int` value to show the multiplier.

In Craps, outcomes have more complex payouts. We might see a payout of 6:5. We can’t use a simple `int` value.

Problem. How do we represent more complex odds?

Forces. We have several choices.

- Use a `float` value. This replaces exact integer values with floating-point approximations. There’s little compelling reason for this. It can lead to displaying values that look like 3.999999999999997 instead of 4.
- Explicit numerator and denominator. The original design for outcome had an assumed denominator of `1`. It’s a small change to introduce an explicit denominator value with a default of `1`.
- Use `fractions.Fraction`. This requires very little change. The `Fraction` class works seamlessly with integer values, allowing us to use this with little change.

24.3 Optional Parameter Implementation Technique

A common technique that helps to add featurtes is to add optional parameters to a method. We make a parameter optional by providing a default value.

There’s a common confusion that can arise when using a mutable value as a default. We’ll show the immutable example first. Then we’ll show what happens when a mutable object isused.

The following example shows a simple required parameter, followed by an optional parameter. This uses an immutable object as the default value.

```
class SomeClass:
    def __init__( self, reqArg, optArg=1 ):
         initialize using reqArg and optArg
```

The optional argument’s default value is an immutable int object. Each time we evaluate `SomeClass(x)` with a single actual argument value, the default object for the `optArg` parameter is reused. Since a `1` has no internal state, sharing this object has no adverse consequences.

An instance of one of the Python mutable types (lists, sets, maps) should not be provided as default values for an initializer. This is because the single initialization object – created when the class is defined – will be shared by all instances created with the default value.

```
class BadIdea:
    def __init__( self, reqArg, optArg=[] ):
         initialize using reqArg and optArg
```

We can easily create many instances of `BadIdea` that share the single, mutable default list object.

To avoid this undesirable sharing of an instance of the default value, we have to do the following.

```
class SomeClass:
    def __init__( self, aList=None ):
        if aList is not None:
            initialize using aList
        else:
            default initialization using a fresh []
```

In this case, we’ve used an immutable `None` as an indicator that we should create a fresh, empty, mutable list object.

This is a general way to add default and optional values to a function. We can provide optional parameters with a default value of `None`.

24.4 Outcome Rework

Outcome contains a single outcome on which a bet can be placed.

24.4.1 Attributes

We won't change the existing attributes. We will, however, change the use of the `odds` attribute from an integer to a `Fraction`.

24.4.2 Constructors

We have two ways to introduce fractional odds. One is to make the numerator and denominator explicit.

`Outcome.__init__(`*name*, *numerator*, *denominator=1*`)`

Parameters

- **name** (*string*) – The name of this outcome.
- **numerator** (*int*) – the payout odds numerator
- **denominator** (*int*) – the payout odds denominator

Sets the name and odds from the parameters. This method will create an appropriate `Fraction` from the given values.

Example 1: 6:5 is a right bet, the player will win 6 for each 5 that is bet.

Example 2: 2:3 is a wrong bet, they player will win 2 for each 3 that is bet.

Here's an alternative.

`Outcome.__init__(`*name*, *odds*`)`

Parameters

- **name** (*string*) – The name of this outcome.
- **numerator** (*Fraction or int*) – the payout odds numerator

Sets the name and odds from the parameters. This method will create an appropriate `Fraction` from the given values.

- If `isinstance(odds,int)`, create a `Fraction(odds, 1)`.
- If `isinstance(odds,Fraction)`, use the given object directly.

24.4.3 Methods

`Outcome.winAmount(`*self*, *amount*`)`

Parameters **amount** (*Fraction*) – amount of the bet

Returns the product of this *Outcome*'s odds by the given amount. The result will be a Fraction, which will work nicely.

`Outcome.__str__(`*self*`)` → string

An easy-to-read String output method is also very handy. This should return a `String` representation of the name and the odds. A form that looks like `1-2 Split (17:1)` is the goal.

This requires decomposing the odds into numerator and denominator.

24.5 Outcome Deliverables

There are three deliverables for this exercise.

- The revised *Outcome* class that handles fractional odds and returns type `Fraction` from `winAmount()`.
- A class which performs a unit test of the *Outcome* class. The unit test should create a couple instances of *Outcome*, and establish that the `winAmount()` method works correctly.
- A revision to each subclass of *Player* to correctly implement the revised result from `winAmount()`. Currently, there are six subclasses of *Player* : *Passenger57*, *SevenReds*, *PlayerRandom*, *Player1326*, *PlayerCancellation*, and *PlayerFibonacci*.

CHAPTER TWENTYFIVE

THROW CLASS

In Craps, a throw of the dice may change the state of the game. This close relationship between the `Throw` and `CrapsGame` leads to another chicken-and-egg design problem. We'll design `Throw` in detail, but provide a rough stub for `CrapsGame`.

We'll try to keep the `Throw` class as a parallel with the `Bin` class in Roulette. It will hold a bunch of individual `Outcome` instances.

We'll look at the nature of a `Throw` in *Throw Analysis*.

This will lead us to look – again – at *The Wrap vs. Extend Question*. This is an important question related to how we'll use a collection class.

In *Throw Design* we'll look at the top-level superclass. There are a number of subclasses:

- *Natural Throw Design*,
- *Craps Throw Design*,
- *Eleven Throw Design*, and
- *Point Throw Design*.

Once we have the throw defined, we can examine the outline of the game in *Craps Game Design*. This will show how game and throw collaborate. In *Throw Deliverables* we'll enumerate the deliverables for this chapter.

25.1 Throw Analysis

The pair of dice can throw a total of 36 unique combinations. These are summarized into fifteen distinct outcomes: the eleven numbers from 2 to 12, plus the four hardways variations for 4, 6, 8 and 10.

In Roulette, the randomized positions on the wheel were called `Bin`s and each one had a very simple collection of winning `Outcome`s. In Craps, however, the randomized throws of the dice serve three purposes: - they resolve simple one-roll proposition bets, - they may resolve hardways bets, and - they change the game state (which may resolve game bets).

From this we can allocate three responsibilities. We'll look at each of these responsibilities individually.

One-Throw Propositions. A `Throw` of the dice includes a collection of proposition `Outcome`s which are immediate winners. This collection will be some combination of 2, 3, 7, 11, 12, Field, Any Craps, or Horn. For completeness, we note that each throw could also contain one of the 21 hop-bet `Outcome`s; however, we'll ignore the hop bets.

Multi-Throw Propositions. A `Throw` of the dice may resolve hardways bets (as well as place bets and buy bets). There are three possible conditions for a given throw:

- some bets may be winners,
- some bets may be losers, and
- some bets may remain unresolved.

This tells us that a *Throw* may be more than a simple collection of winning *Outcome*s. A *Throw* must also contain a list of losing *Outcome*s. For example, any of the two easy 8 rolls (6-2 or 5-3) would contain winning *Outcome*s for the place-8 bet and buy-8 bet, as well as a losing *Outcome* for a hardways-8 bet. The hard 8 roll (4-4), however, would contain winning *Outcome*s for the place-8 bet, buy-8 bet, and hardways-8 bet

Game State Change. Most importantly, a *Throw* of the dice can lead to a state change of the *Game*. This may resolve game-level bets. From the *Craps Game*, we see that the state changes depend on both the *Game* state plus the kind of *Throw*. The rules identify the following species of *Throw*.

- **Craps**. These are throws of 2, 3 or 12. On a come-out roll, this is an immediate loss. On any other roll, this is ignored. There are 4 of these throws.
- **Natural**. This is a throw of 7. On a come-out roll, this is an immediate win. On any other roll, this is an immediate loss and a change of state. There are 6 of these throws.
- **Eleven**. This is a throw of 11. On a come-out roll, this is an immediate win. On any other roll, this is ignored. There are 2 of these throws.
- **Point**. This is a throw of 4, 5, 6, 8, 9, or 10. On a come-out roll, this estanblishes the point, and changes the game state. On any other roll, this is is compared against the established point: if it matches, this is a win and a change of game state. Otherwise, it doesn't match and no game state change occurs. There are a total of 24 of these throws, with actual frequencies between three and five.

The state change can be implemented by defining methods in *Game* that match the varieties of *Throw*. We can imagine that the design for *Game* will have four methods: `craps()`, `natural()`, `eleven()`, and `point()`. Each kind of *Throw* will call the matching method of *Game*, leading to state changes, and possibly game bet resolution.

The game state changes lead us to design a hierarchy of *Throw* classes to enumerate the four basic kinds of throws. We can then initialize a *Dice* object with 36 *Throw* objects, each of the appropriate subclass. When all of the subclasses have an identical interface, this embodies the principle of polymorphism. For additional information, see *On Polymorphism*.

In looking around, we have a potential naming problem: both a wheel's *Bin* and the dice's *Throw* are somehow instances of a common abstraction. Looking forward, we may wind up wrestling with a deck of cards trying to invent a common nomenclature for all of these randomizers. They create random events, and this leads us to a possible superclass for *Bin* and *Throw*: *RandomEvent*.

Currently, we can't identify any features that we can refactor up into the superclass. Rather than over-engineer this, we'll hold off on complicating the design until we find something else that is common between our sources of random events.

25.2 The Wrap vs. Extend Question

Note that an instance of a *Throw* is effectively a container for a set of *Outcome*s. We have the standard **Wrap vs. Extend** question that we need to answer here.

- **Wrap**. Each *Throw* has an internal frozenset of *Outcome* objects.
- **Extend**. We base *Throw* on the class frozenset directly and add methods to add features.

We have a fairly large number of methods that are introduced here.

When we look back at Roulette, a *Bin* had no impact on the state of the game. In Craps, though, there's a need for each *Throw* to update the current state of the game. Perhaps the point is now on, perhaps the game was won, perhaps seven was thrown and the game was lost.

There's very little evidence to help make a choice. Lacking further information, we'll focus on using a **Wraps** approach, and define a *Throw* so that is has a frozenset as an attribute.

25.3 Throw Design

class Throw

Throw is the superclass for the various throws of the dice. Each subclass is a different grouping of the numbers, based on the rules for Craps.

25.3.1 Fields

Throw.outcomes

A Set of one-roll Outcomes that win with this throw.

It seems like the numbers might be handy. When we look at the design for the :class‘Dice‘ class, we may regret putting these attributes here. After all, the *Dice* reflect some pair of numbers. The *Throw* is a set of *Outcome*s that may match *Bet*s.

We'll include them now because it makes it easy to produce helpful debugging output.

Throw.d1

One of the two die values, from 1 to 6.

Throw.d2

The other of the two die values, from 1 to 6.

25.3.2 Constructors

Throw.__init__(*self*, *d1*, *d2*, * *outcomes*)

Creates this throw, and associates the given Set of *Outcome*s that are winning propositions.

Parameters

- **d1** – The value of one die
- **d2** – The value of the other die
- **outcomes** – The various outcomes for this Throw

25.3.3 Methods

Throw.hard(*self*) → boolean

Returns true if *d1* is equal to *d2*. This helps determine if hardways bets have been won or lost.

Throw.updateGame(*self*, *game*)

Parameters **game** (CrapsGame) – the Game to be updated based on this throw.

Calls one of the *Game* state change methods: craps(), natural(), eleven(), point(). This may change the game state and resolve bets.

Throw.__str__(*self*) → String

An easy-to-read String output method is also very handy. This should return a String representation of the dice. A form that looks like 1,2 works nicely.

25.4 Natural Throw Design

class NaturalThrow

Natural Throw is a subclass of *Throw* for the natural number, 7.

25.4.1 Constructors

`NaturalThrow.__init__(self, d1, d2)`

Parameters

- **d1** – The value of one die
- **d2** – The value of the other die

Creates this throw. The constraint is that $d1 + d2 = 7$. If the constraint is not satisfied, simply raise an exception.

This uses the superclass constructor to add appropriate *Outcome*s for a throw of 7.

25.4.2 Methods

`NaturalThrow.hard(self)` → boolean

A natural 7 is odd, and can never be made "the hard way". This method always returns false.

`NaturalThrow.updateGame(self, game)`

Parameters **game** (CrapsGame) – the Game to be updated based on this throw.

Calls the `natural()` method of a game *Game*. This may change the game state and resolve bets.

25.5 Craps Throw Design

class CrapsThrow

`Craps Throw` is a subclass of *Throw* for the craps numbers 2, 3 and 12.

25.5.1 Constructors

`CrapsThrow.__init__(self, d1, d2)`

Parameters

- **d1** – The value of one die
- **d2** – The value of the other die

Creates this throw. The constraint is that $d1 + d2 \in \{2, 3, 12\}$. If the constraint is not satisfied, simply raise an exception.

This uses the superclass constructor to add appropriate *Outcome*s for a throw of craps.

25.5.2 Methods

`CrapsThrow.hard(self)` → boolean

The craps numbers are never part of "hardways" bets. This method always returns false.

`CrapsThrow.updateGame(self, game)`

Parameters **game** (CrapsGame) – the Game to be updated based on this throw.

Calls the `craps()` method of a game *Game*. This may change the game state and resolve bets.

25.6 Eleven Throw Design

class ElevenThrow

`Eleven Throw` is a subclass of *Throw* for the number, 11. This is special because 11 has one effect on a come-out roll and a different effect on point rolls.

25.6.1 Constructors

`ElevenThrow.__init__(self, d1, d2)`

Parameters

- **d1** – The value of one die
- **d2** – The value of the other die

Creates this throw. The constraint is that $d1+d2 = 11$. If the constraint is not satisfied, simply raise an exception.

This uses the superclass constructor to add appropriate *Outcome*s for a throw of 11.

25.6.2 Methods

`ElevenThrow.hard(self)` → boolean

Eleven is odd and never part of "hardways" bets. This method always returns false.

`ElevenThrow.updateGame(self, game)`

Parameters **game** (CrapsGame) – the Game to be updated based on this throw.

Calls the `eleven()` method of a game *Game*. This may change the game state and resolve bets.

25.7 Point Throw Design

class PointThrow

`Point Throw` is a subclass of *Throw* for the point numbers 4, 5, 6, 8, 9 or 10.

25.7.1 Constructors

`PointThrow.__init__(self, d1, d2)`

Parameters

- **d1** – The value of one die
- **d2** – The value of the other die

Creates this throw. The constraint is that $d1 + d2 \in \{4, 5, 6, 8, 9, 10\}$. If the constraint is not satisfied, simply raise an exception.

This uses the superclass constructor to add appropriate *Outcome*s for a throw of craps.

25.7.2 Methods

`PointThrow.hard(self)` → boolean

Eleven is odd and never part of "hardways" bets. This method always returns false.

Returns `true` if `d1` is equal to `d2`. This helps determine if hardways bets have been won or lost.

`PointThrow.updateGame(self, game)`

Parameters **game** (CrapsGame) – the Game to be updated based on this throw.

Calls the `point()` method of a game *`Game`*. This may change the game state and resolve bets.

25.8 Craps Game Design

class CrapsGame

`CrapsGame` is a preliminary design for the game of Craps. This initial design contains the interface used by the *`Throw`* class hierarchy to implement game state changes.

25.8.1 Fields

`GrapsGame.point`

The current point. This will be replaced by a proper *State* design pattern.

25.8.2 Constructors

`CrapsGame.__init__(self)`

Creates this Game. This will be replaced by a constructor that uses *`Dice`* and `CrapsTable`.

25.8.3 Methods

`CrapsGame.craps(self)`

Resolves all current 1-roll bets.

If the point is zero, this was a come out roll: Pass Line bets are an immediate loss, Don't Pass Line bets are an immediate win.

If the point is non-zero, Come Line bets are an immediate loss; Don't Come Line bets are an immediate win.

The state doesn't change.

A future version will delegate responsibility to the *`craps()`* method of a current state object.

`CrapsGame.natural(self)`

Resolves all current 1-roll bets.

If the point is zero, this was a come out roll: Pass Line bets are an immediate win; Don't Pass Line bets are an immediate loss.

If the point is non-zero, Come Line bets are an immediate win; Don't Come bets are an immediate loss; the point is also reset to zero because the game is over.

Also, hardways bets are all losses.

A future version will delegate responsibility to the *`natural()`* method of a current state object.

`CrapsGame.eleven(self)`
Resolves all current 1-roll bets.

If the point is zero, this is a come out roll: Pass Line bets are an immediate win; Don't Pass Line bets are an immediate loss.

If the point is non-zero, Come Line bets are an immediate win; Don't Come bets are an immediate loss.

The game state doesn't change.

A future version will delegate responsibility to the `eleven()` method of a current state object.

`CrapsGame.point(self, point)`

Parameters **`point`** (*integer*) – The point value to set.

Resolves all current 1-roll bets.

If the point was zero, this is a come out roll, and the value of the dice establishes the point.

If the point was non-zero and this throw matches the point the game is over: Pass Line bets and associated odds bets are winners; Don't Pass bets and associated odds bets are losers; the point is reset to zero.

Finally, if the point is non-zero and this throw does not match the point, the state doesn't change; however, Come point and Don't come point bets may be resolved. Additionally, hardways bets may be resolved.

A future version will delegate responsibility to the current state's `point()` method to advance the game state.

`Throw.__str__(self)` → String
An easy-to-read String output method is also very handy. This should return a `String` representation of the current state. The stub version of this class has no internal state object. This class can simply return a string representation of the point; and the string `"Point Off"` when `point` is zero.

25.9 Throw Deliverables

There are eleven deliverables for this exercise.

- A stub class for `CrapsGame` with the various methods invoked by the throws. The design information includes details on bet resolution that doesn't need to be fully implemented at the present time. For this stub class, the change to the `point` variable is required for unit testing. The other information should be captured as comments and output statements that help confirm the correct behavior of the game.
- The `Throw` superclass, and the four subclasses: `CrapsThrow`, `NaturalThrow`, `ElventThrow`, `PointThrow`.
- Five classes which perform unit tests on the various classes of the `Throw` class hierarchy.

CHAPTER TWENTYSIX

DICE CLASS

Unlike Roulette, where a single *Bin* could be identified by the number in the bin, *Dice* use a pair of numbers.

The idea is to have the *Dice* parallel to the Roulette *Wheel*. The *Dice* is a collection of *Throw*s. The *Dice* is responsible to picking a *Throw* at random. We'll look at this in detail in *Dice Analysis*.

We'll reconsider some features of *Throw* in *Throw Rework*.

Once we've settled on the features, we'll look at the details in *Dice Design*. We'll enumerate the deliverables in *Dice Deliverables*.

We'll look at the subject of performance improvements in *Dice Optimization*.

26.1 Dice Analysis

The dice have two responsibilities: they are a container for the *Throw*s and they pick one of the *Throw*s at random.

We find that we have a potential naming problem: both a *Wheel* and the *Dice* are somehow instances of a common abstraction. Looking forward, we may wind up wrestling with a deck of cards trying to invent a common nomenclature for the classes. They create random events, and this leads us to a possible superclass: `Randomizer`. Rather than over-engineer this, we'll hold off on adding this design element until we find something else that is common among them.

Container. Since the *Dice* have 36 possible *Throw*s, it is a collection. We can review our survey of the collections in *Design Decision – Choosing A Collection* for some guidance here. In this case, we note that the choice of *Throw* can be selected by a random numeric index.

For Python programmers, this makes the a `tuple` very appealing. The collection of outcomes is fixed, and an immutable structure makes the most sense.

After selection a collection type, we must then deciding how to index each *Throw* in the *Dice* collection. Recall that in Roulette, we had 38 numbers: 1 to 36, plus 0 and 00. By using 37 for the index of the *Bin* that contained 00, we had a simple integer index for each *Bin*.

For Craps it seems better to use a two-part index with the values of two independent dice.

Index Choices. In this case, we have two choices for computing the index into the collection,

- We can rethink our use of a simple sequential structure. If we use a `Map`, we can use an object representing the pair of numbers as an index instead of a single int value.
- We have to compute a unique index position from the two dice values.

Decision Forces. There are a number of considerations to choosing between these two representations.

1. If we create an object with each unique pair of integers, we can then use that object to be the index for a `Map`. The `Map` associates this "pair of numbers" object with its *Throw*.
2. We can transform the two numeric dice values to a single index value for the sequence. This is a technique called *Key Address Transformation*; we transform the keys into the address (or index) of the data.

We create the index, i, from two dice, d_1, d_2, via a simple linear equation: $i = 6(d_1 - 1) + (d_2 - 1)$.

We can reverse this calculation to determine the two dice values from an index. $d_1 = \lfloor i \div 6 \rfloor + 1; d_2 = (i \bmod 6) + 1$. Python offers a `divmod()` function which does precisely this calculation.

This doesn't obviously scale to larger collections of dice very well. While Craps is a two-dice game, we can imagine simulating a game with larger number of dice, making this technique complex.

However, we're only doing a two-dice game. We don't need to over-engineer this.

Because of encapsulation, the choice of algorithm is completely hidden within the implementation of *Dice*.

Solution. Our recommendation is to encapsulate the pair of dice in a `tuple` instance. We can use this object as index into a `dict` that associates a `tuple` with a *Throw*.

More advanced students can create a class hierarchy for *Dice* that includes all of both implementations as alternative subclasses.

Random Selection. The random number generator in `random.Random` helps us locate a *Throw* at random.

First, we can get the `list` of keys from the `dict` that associates a `tuple` of dice numbers with a *Throw*.

Second, we use `Random.choice()` to pick one of these `tuple`s.

We use this randomly selected `tuple` to return the selected *Throw*.

26.2 Throw Rework

We need to update *Throw* to return an appropriate key object.

There are two parts to this. First, we need to calculate the key.

There are two ways to handle this.

- **Eager**. This means we calculate the key as soon as we know the two dice values. This can be an attribute which is fetched like any other.

  We need to update the *Throw* constructor to create the key when the *Throw* is being built. This will allow all parts of the application to share references to a single instance of the key.

- **Lazy**. This means we don't calculate the key until its required. We often use the `@property` decorator for methods which embody a lazy calculation that we want to appear as if it was an attribute.

  For this, We add a method to *Throw* to return the `tuple` that is a key for this *Throw*.

  `Throw.key(`*self*`)` → tuple

It's very difficult to make an *eager vs. lazy* decision until the entire application has been built and we know **all** the places where an object is used.

26.3 Dice Design

class Dice

Dice contains the 36 individual throws of two dice, plus a random number generator. It can select a *Throw* at random, simulating a throw of the Craps dice.

26.3.1 Fields

`Dice.throws`

This is a `dict` that associates a `NumberPair` with a *Throw*.

`Dice.rng`
An instance of `random.Random`

Generates the next random number, used to select a *Throw* from the *throws* collection.

26.3.2 Constructors

`Dice.__init__(self)`
Build the dictionary of *Throw* instances.

At the present time, this does not do the full initialization of all of the *Throw*s. We're only building the features of *Dice* related to random selection. We'll extend this class in a future exercise.

26.3.3 Methods

`addThrow(self, throw)`

Parameters **throw** (Throw) – The *Throw* to add.

Adds the given *Throw* to the mapping maintained by this instance of *Dice*. The key for this *Throw* is available from the `Throw.getKey()` method.

`next(self)` → Throw
Returns the randomly selected *Throw*.

First, get the `list` of keys from the `throws`.

The `random.Random.choice()` method will select one of the available keys from the the list.

This is used to get the corresponding *Throw* from the `throws Map`.

`Dice.getThrow(self, d1, d2)` → Throw

Parameters

- **d1** – The value of one die
- **d2** – The other die

This method takes a particular combination of dice, locates (or creates) a `NumberPair`, and returns the appropriate *Throw*.

This is not needed by the application. However, unit tests will need a method to return a specific *Throw* rather than a randomly selected *Throw*.

26.4 Dice Deliverables

There are three deliverables for this exercise. In considering the unit test requirements, we note that we will have to follow the design of the *Wheel* class for convenient testability: we will need a way to get a particular *Throw* from the *Dice*, as well as replacing the random number generator with one that produces a known sequence of numbers.

- The *Dice* class.
- A class which performs a unit test of building the *Dice* class. The unit test should create several instances of *Outcome*, two instances of *Throw*, and an instance of *Dice*. The unit test should establish that *Throw*s can be added to the *Dice*.
- A class which performs a demonstration of selecting non-random values from the *Dice* class. By setting a particular seed, the *Throw*s will be returned in a fixed order. To discover this non-random order, a demonstration should be built which includes the following.

1. Create several instances of *Outcome*.
2. Create two instances of *Throw* that use the available *Outcome*s.
3. Create one instance of *Dice* that uses the two *Throw*s.
4. A number of calls to the *next()* method should return randomly selected *Throw*s.

Note that the sequence of random numbers is fixed by the seed value. The default constructor for a random number generator creates a seed based on the system clock. If your unit test sets a particular seed value, you will get a fixed sequence of numbers that can be used to get a consistent result.

26.5 Dice Optimization

First, we note that premature optimization is a common trap.

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified"
>
> —Donald Knuth
>
> "Structured Programming with Goto Statements". Computing Surveys 6:4 (1974), 261-301.

The eager vs. lazy calculation of the key associated with a pair of dice is something that seems like it shoud have one "best" way. It seems like we should be able to chose between eager and lazy calculation of key values.

This decision is actually quite difficult to make.

Eager calculation seems optimal: get it done once and reuse the answer many times. However, in some cases, the calculation is rather expensive and isn't always needed. In this case, the key involves the creation of a new object, and this can be a costly operation.

We've made an effort to optimize this by thinking of the collection of *Throw*s as a fixed pool of objects, allocated once, and then never created again. It appears that they key associated with a *Throw* is only computed once.

For this example, the **Eager v. Lazy** decision seems to be moot.

In other cases, it's a significant optimization.

In all cases, we need to use a profiler to see if this particular piece of the application is slowest. We should only optimize the parts which are demonstrably slowest. Optimizing parts which aren't slow (or aren't even correct) is simply a waste of time.

We should articulate alternative designs. We should leave a note in the docstrings about alternative implementations. We should not, however, pursue each alternative until we know that it adds significant value to explore the alternatives carefully.

CHAPTER TWENTYSEVEN

THROW BUILDER CLASS

This chapter identifies some subtleties of bets in Craps, and proposes some rather involved design rework to resolve the issues that are raised.

Recall that each instance of the class `Dice` is a container for a set of `Throw`s. Each `Throw` contains a set of `Outcome`s that are the basis for bets.

One additional feature is that a `Throw` will change the state of the game. We must be sure to account for this additional responsibility.

A further problem is that the `Outcome` doesn't have a fixed payout in Craps. This will alter the design for `Outcome` yet again to handle this feature.

We'll look at this in detail in *Throw Builder Analysis*. We'll tackle the variable odds feature in *Outcomes with Variable Odds*.

This will lead us to refactor `Outcome`. We'll look at this in *Refactoring The Outcome Hierarchy*.

We'll digress into some other design considerations in *Soapbox on Subclasses*, *Soapbox on Architecture*, and *Throw Builder Questions and Answers*, and *Soapbox on Justification*.

After considering the alternatives, we'll look at two approaches to the rework:

- In *Design Light* we'll try to minimize the rework. The details will be in *Minimal Outcome Rework*.
- In *Design Heavy*, we'll acknowledge that there is no "simple" solution. We'll look at the details in:
  - *RandomEvent class*,
  - *Bin Rework*,
  - *Throw Rework*, and
  - *Outcome Rework*.

After the rework is in place, qe can then look at the common design issues. We'll cover these in *Common Design*. This will include *OutcomeField Design*, *OutcomeHorn Design*, and *ThrowBuilder Class Design*.

In *Throw-Builder Deliverables* we'll enumerate the deliverables for this chapter.

We'll present sidebars on the proper design of subclasses and the proper architecture for the packages that make up an application. Additionally, we'll provide a brief FAQ on the design issues raised.

27.1 Throw Builder Analysis

Enumerating each `Outcome` in the 36 `Throw`s could be a tedious undertaking. We'll design a *Builder* to enumerate all of the `Throw`s and their associated list of `Outcome`s. This will build the `Dice`, finishing the elements we deferred from *Dice Class*.

The 36 ways the dice fall can be summarized into 15 kinds of `Throw`, with a fixed distribution of probabilities. We have two ways to enumerate these.

- We could develop a *Builder* class that enumerates the 36 possible `Throw`s, assigning the appropriate attribute values to each object. This will create a number of duplicates: in Craps, dice showing `(1, 2)` are equivalent to dice showing `(2, 1)`
- An alternative is for a *Builder* class to step through the 15 kinds of `Throw`s, creating the proper number of instances of each kind. There is one intance of `(1, 1)`, two instances of `(1, 2)`, etc.

We looked at this in *Creating A Dice Frequency Distribution*. Beacuse of the vast number of one-off special cases (e.g. hardways outcomes), it seems simpler to examine each of the 36 pairs of dice and determine which kind of `Throw` to build.

The proposition bets define eight one-roll `Outcome`s that need to be assigned to the various `Throw` instances we are building. We will share references to the following `Outcome` objects among the `Throw`s:

- The number 2 proposition, with 30:1 odds. There's only one instance of this, `(1, 1)`.
- The number 3 proposition, with 15:1 odds. There are two instances, `(1, 2)` and `(2, 1)`.
- The number 7 proposition, with 4:1 odds. There are six ways to throw this.
- The number 11 proposition, with 15:1 odds. There are two instances, `(5, 6)` and `(6, 5)`.
- The number 12 proposition, with 30:1 odds. There's only one instance of this, `(6, 6)`.
- The "any craps" proposition, with 7:1 odds. This belongs to all of the various combinations of dice that total 2, 3, or 12.
- There are actually two "horn" proposition outcomes. One belongs to dice totalling 2 or 12, with odds of 27:4. The other belongs to dice totalling 3 or 11, with odds of 3:1. We'll address this below by reworking the `Outcome` class.
- There are two "field" proposition outcomes, also. One belongs to throws totalling 2 or 12 and pays 2:1. The other belongs to throws totalling 3, 4, 9, 10, or 11 and pays even money (1:1).

We can use the following algorithm for building the `Dice`.

Building Dice

For All Faces Of Die 1. For all d_1, such that $1 \leq d_1 < 7$:

> **For All Faces Of A Die 2**. For d_2, such that $1 \leq d_2 < 7$:
>
> > **Sum the Dice**. Compute the sum, $s \leftarrow d_1 + d_2$.
> >
> > **Craps?** If s is in 2, 3, and 12, we create a `CrapsThrow` instance. This will include a reference to one of the 2, 3 or 12 `Outcome` s, plus references to the Any Craps, Horn and Field `Outcome`s.
> >
> > **Point?** For s in 4, 5, 6, 8, 9, and 10 we will create a `PointThrow` instance.
> >
> > > **Hard?** When $d_1 = d_2$, this is a *hard* 4, 6, 8 or 10.
> > >
> > > **Easy?** Otherwise, $d_1 \neq d_2$, this is an *easy* 4, 6, 8 or 10.
> >
> > **Field?** For s in 4, 9, or 10, we include a refrence to the Field `Outcome`. Note that 2, 3, and 12 Field outcomes where handled above under **Craps**.
> >
> > **Horn?** For s in 2, 3, 11, or 12, we include a reference to Horn `Outcome`.
> >
> > **Natural?** For s of 7, we create a `NaturalThrow` instance. This will also include a reference to the 7 `Outcome`.
> >
> > **Eleven?** For s of 11, we create an `ElevenThrow` instance. This will include references to the 11, Horn and Field `Outcome`s.

At this point, the algorithm is mostly a concept. We need to examine the outcomes with variable odds, first.

27.2 Outcomes with Variable Odds

Our detailed examination of the bets has turned up an interesting fact about Field bets and Horn bets: these outcomes have payoffs that depend on the number on the dice. In the earlier chapter, *Outcome Class*, we missed this nuance, and did not provide for a `Dice.winAmount()` method that depends on the `Dice`.

We'll need to redesign the `Outcome` to handle these details.

Problem Statement. Unlike the Pass Line and Come Line bets, Field bets and Horn bets have payoffs that depend on the number currently showing on the dice. Note that Come Line bets are moved on the table from the generic Come Line to a new `Outcome` when a point is established. This is not the case for Field and Horn bets.

How do we compute the win amount for Field and Horn bets?

Context. Our design objective is to have a `Bet` reference a single `Outcome` object. Doing this allows a `Bet` to be compared with a `Set` of winning `Outcome`s associated with the current throw of the `Dice` or the current `Bin` of a `Wheel`.

We'd like to have a single horn `Outcome` object and field `Outcome` object shared by multiple instances of `Throw` to make this comparison work in a simple, general way.

As an example, the player can place a bet on the Field `Outcome`, which is shared by all of the field numbers (2, 3, 4, 9, 10, 11, and 12). The problem we have is that for 2 and 12, the outcome pays 2:1 and for the other field numbers it pays 1:1, and our design only has a single set of payout odds.

Forces. In order to handle this neatly, we have two choices.

- Have two `Outcome`s bundled into a single `Bet`. This allows us to create a `Bet` that includes both the low-odds field outcome (3, 4, 9, 10 and 11) plus the high-odds field outcome (2 and 12). One of the nice features of this is that it is a small expansion to `Bet`.

  Further research shows us that there are casino-specific variations on the field bet, including the possibility of three separate `Outcome`s for those casinos that pay 3:1 on 12. This makes construction of the `Bet` rather complex, and dilutes the responsibility for creating a proper `Bet`. Once we put multiple `Outcome`s into a `Bet`, we need to assign responsibility for keeping the bundle of Field `Outcome`s together.

  Pursuing this further, we could expand `Outcome` to follow the **Composite** design pattern. We could introduce a subclass which was a bundle of multiple `Outcome`s. This would allow us to keep `Bet` very simple, but we still have to construct appropriate composite `Outcome` instances for the purpose of creating `Bet`s. Rather than dive into allocating this responsibility, we'll look at other alternatives, and see if something turns up that doesn't add as much complexity.

- Another approach is to add an optional argument to `Outcome` that uses the current `Throw` to calculate the win amount.

  This allows us to have a single field bet `Outcome` with different odds for the various numbers in the field. This further allows us to create slightly different field bet `Outcome` class definitions for the casino-specific variations on the rules.

Solution. Our first design decision, then, is to modify `Outcome` to calculate the win amount given the current `Throw`.

Consequences. There are a number of consequences of this design decision.

- Where in the `Outcome` class hierarchy do we add this additional `winAmount()` method?
- We need to design the new `winAmount()` method so that we don't break everything we've written so far.

This leads us to two rounds of additional problem-solving.

27.3 Refactoring The Outcome Hierarchy

Consequent Problem: Class Hierarchy. While it appears simplest to add a "variable odds" subclass of *Outcome* with a new method that uses the number on the dice, we find that there are some additional considerations.

Our design depends on polymorphism among objects of the *Outcome* class: all instances have the same interface. In order to maintain this polymorphism, we need to add this new method to the superclass. The superclass version of the new `winAmount()` based on the Craps *Throw* can return an answer computed by the original `winAmount()` method. We can then override this in a subclass for Field and Horn bets in Craps.

An alternative is to break polymorphism and create a Craps-specific *Outcome* subclass. This would ripple out to *Throw*, *Bet*, *Table*, *Player*. This is an unpleasant cascade of change, easily avoided by assuring that the entire *Outcome* class hierarchy is polymorphic.

Solution. Our second design decision, then, is to insert the change at the top of the *Outcome* class hierarchy, and override this new `winAmount()` method in the few subclasses that we use to create Horn and Field *Outcome*s.

- The Horn bet `winAmount()` method applies one of two odds, based on the event's value.
- The Field bet may have any of two or three odds, depending on the casino's house rules. It is difficult to identify a lot of commonality between Horn bets and Field bets. Faced with these irreconcilable differences, we will need two different `winAmount()` methods, leading us to create two subclasses: *OutcomeField* and *OutcomeHorn*.

The differences are minor, merely a list of numbers and odds. However, our overall objective is to minimize if-statements. (Or, stated another way, we prefer to maximize the use of dependency injection; or we prefer inversion of control.) We prefer many simple classes over a single class with even a moderately complex method.

Consequent Problem: Dependencies. We've decided to add a dependency to the *Outcome.winAmount()*; specifically, we've made it dependent on a *Throw* object. While this works well for Craps, it makes no sense for Roulette.

To allow the games to evolve independently, we should not have any dependencies between games. This means that a general-purpose class like *Outcome* can't depend on a game-specific class like *Throw*. A general-purpose class has to depend on some a superclass (or interface) that encompasses the Craps-specific *Throw* as well as the Roulette-specific *Bin*.

Additional Classes. To break the dependency between a general-purposes class and a game-specific class, we need introduce a superclass that includes both *Throw* and *Bin* as subclasses. This will *Outcome* to work with either Craps and Roulette; keeping them independent of each other.

We could call the parent class a *RandomEvent*. This new class would have an integer event identifier: either the wheel's bin number or the total of the two dice. Given this new superclass, we could then rearrange both *Throw* and *Bin* to be subclasses of *RandomEvent*. This would also force us to rework parts of *Wheel* that creates the *Bin*s.

A benefit of creating a *RandomEvent* class hierarchy is that we can change the new `winAmount()` method to compute the win amount given a *RandomEvent* instead of a highly Craps-specific *Throw*. This makes the `winAmount()` method far more generally useful, and keeps Craps and Roulette separate from each other.

This technique of reworking *Throw* and *Bin* to be subclasses of a common superclass is a fairly common kind of *generalization refactoring*: we found things which needed to be unified because – after some detailed study – they're closely related.

We walk a fine line here.

There's an urge to conflate too many nearly-common features into a single class, leading to a brittle design that cannot easily be reworked. In our example, we considered lifting only one common attribute to the superclass so that a related class (*Outcome*) could operate on instances of these two classes in a uniform manner. For more information on this rework, see *Soapbox on Subclasses*.

Approaches. We will present two alternative designs paths: minimal rework, and a design that is at the fringe of over-engineering. We're forced to look at both options because we often have the urge (or are told by managers) to focus on what seems like the quickest route.

27.4 Soapbox on Subclasses

Designers new to OO techniques are sometimes uncomfortable with the notion of highly-specialized subclasses. We'll touch on two reasons why specialized subclasses are far superior to the alternatives.

One approach to creating common features is to add nested if-statements instead of creating subclasses. In our example, we might have elected to add if-statements that would determine if this was a variable-odds outcome, and then determine which of the available odds would be used. The first test (for being a variable-odds outcome) is, in effect, a determination of which subclass of `Outcome` is being processed.

If statements often imply a class structure.

Since an object's membership in a class determines the available methods, there's no reason to *test* for membership using if-statements. In most cases, the only relevant tests for membership are done at construction time. If we use that initial decision to select the subclass (with appropriate subclass-specific methods) we do not repeat that decision every time a method is invoked. This is the efficiency rationale for introducing a subclass to handle these special cases.

The more fundamental reason is that specialized subclasses usually represent distinct kinds of real-world things, which we are modeling in software-world. In contrast, a procedural view-point is that the variant behavior is a special case: a condition or situation that requires unique processing. This view often confuses the implementation of the special case (via an if-statement) with the nature of the specialization.

In our case, we have a number of distinct things, some of which are related because they have common attributes and behavior. The `Outcome` is fairly intangible, so the notion of commonality can be difficult to see. Contrast this with `Dice` and `Wheel`, which are tangible, and are obviously different things, however they have common behavior and a common relationship with a casino game.

Design Aid. Sometimes it helps to visualize this by getting pads of different-colored sticky paper, and making a mockup of the object structure on whiteboard. Each class is represented by a different color of paper. Each individual object is an individual slip of sticky paper. To show the relationship of `Dice`, `Throw` and `Outcome`, we draw a large space on the board for an instance of `Dice` which has smaller spaces for 36 individual `Throw`s.

In one `Throw` instance, we put a sticky for `Outcome` s 2, Field, Horn, and Any Craps. We use three colors of stickies to show that 2 and Any Craps are ordinary `Outcome` s, Field is one subclass and Horn is another subclass.

In another `Throw` instance, we put a sticky for `Outcome` 7, using the color of sticky for ordinary `Outcome`s.

This can help to show what the final game object will be examining to evaluate winning bets. The game object will have a list of winning `Outcome`s and bet `Outcome` s actually on the table. When a 2 is thrown, the game process will pick up each of the stickies, compare the winning `Outcome` s to the bets, and then use the method appropriate to the color of the sticky when computing the results of the bet.

27.5 Soapbox on Architecture

There are a number of advanced considerations behind the *Design Heavy* section. This is a digression on architecture and *packages of classes*. While this is beyond the basics of OO design, it is a kind of justification for the architecture we've chosen.

A good design balances a number of forces. One example of this is our use of a class hierarchy to decompose a problem into related class descriptions, coupled with the collaboration among individual objects to compose the desired solution. The desired behavior emerges from this tension between decomposition of the class design and composition of the objects to create the desired behavior.

Another example of this decomposition vs. composition is the organization of our classes into packages. We have, in this book, avoided discussion of how we package classes. It is a more subtle aspect of a good design, consequently we find it challenging to articulate sound principles behind the layers and partitions of a good collection of packages. There are some design patterns that give us packaging guidance, however.

Design Patterns. One packaging pattern is the **5-Layer Design**, which encourages us to separate our design into *layers* of *view*, *control*, *model*, *access* and *persistence*. For our current application, the view is the output log written to `System.out`, the control is the overall main method and the `Simulation` class, the model is the casino game model. We don't have any data access or data persistence issues, but these are often implemented with *JDBC* and a *relational database*.

While one of the most helpful architectural patterns, this version of the *5-Layer Design* still leaves us with some unsatisfying gaps. For example, common or *infrastructure* elements don't have a proper home. They seem to form another layer (or set of layers). Further, the model layer often decomposes into domain elements, plus elements which are specializations focused on unique features of the business, customer, vendor or product.

Another packaging pattern is the **Sibling Partition**, which encourages us to separate our application-specific elements to make them parallel *siblings* of a superclass so that we can more easily add new applications or remove obsolete applications. In this case, each casino game is a separate application of our casino game simulator. At some point, we may want to isolate one of the games to reuse just the classes of that game in another application. By making the games proper siblings of each other, and children of an abstract parent, they can be more easily separated.

General vs. Specific. Applying these layered design and application partitioning design patterns causes us to examine our casino game model more closely and further sub-divide the model into game-specific and game-independent elements. If necessary, we can further subdivide the general elements into those that are part of the *problem domain* (casino games) and those that are even more general *application infrastructure* (e.g., simulation and statistics). Our ideal is to have a tidy, short list of classes that provides a complete game simulation. We can cut our current design into three parts: Roulette, Craps and application infrastructure. This allows us to compose Roulette from the Roulette-specific classes and the general infrastructure classes, without including any of the Craps-specific classes.

The following architecture diagram captures a way to structure the packages of these applications.

<table>
<tr><td colspan="2">Simulator, Statistics</td></tr>
<tr><td>Craps Player hierarchy</td><td>Roulette Player hierarchy</td></tr>
<tr><td>Craps Game, Table, Dice, Throw</td><td>Roulette Game, Table, Wheel, Bin</td></tr>
<tr><td colspan="2">Outcome, RandomEvent, Bet, Player, any other superclasses</td></tr>
</table>

Our class definitions have implicitly followed this architecture, working from general to game- and player-specific classes. Note that our low-level classes evolved through several increments. We find this to be superior to attempting to design the general classes from the outset: it avoids any over-engineering of the supporting infrastructure. Additionally, we we careful to assure that our top-level classes contain minimal processing, and are are compositions of lower-level object instances.

Dependencies. A very good design could carefully formalize this aspect of the architecture by assuring that there are minimal references between layers and partitions, and all references are "downward" references from application-specific to general infrastructure packages. In our case, the Simulator should have access only to Player and Game layers.

Two Game partitions should be separate with no references between these packages.

Finally, we would like to assure that the Player and Game don't have invalid "upward" references to the Simulator. There are some tools that can enforce this architectural separation aspect of the design. Lacking tools, we need to exercise some discipline to honor the layers and partitions.

27.6 Throw Builder Questions and Answers

Why is do we need `RandomEvent`? Isn't this overengineering?

Clean separation between Craps and Roulette isn't necessary, but is highly desirable. We prefer not to have Roulette classes depend in any way on Craps classes. Instead of having them entangled, we factor out the entanglement and make a new class from this. This is also called reducing the coupling between classes. We prefer the term "entanglement" because it has a suitably negative connotation.

Why couldn't we spot the need for `RandomEvent` earlier in the design process?

Some experienced designers do notice this kind of commonality between `Throw` and `Bin`, and can handle it without getting badly side-tracked.

However, some designers can spend too much time searching for this kind of commonality. We prefer to wait until we are sure we've understood the problem and the solution before committing to a particular class design.

Isn't the goal to leave Roulette alone? Isn't the ideal to extend the design with subclasses, leaving the original design in place?

Yes, the goal is to extend a design via subclasses. But, this is only possible if the original design is suitable for extension by subclassing. We find that it is very difficult to create a design that both solves a problem and can be extended to solve a number of related problems.

Note that a general, extensible design has two independent feature sets. On one level it solves a useful problem. Often, this is a difficult problem in its own right, and requires considerable skill merely to ferret out the actual problem and craft a usable solution within budget, time and skill constraints.

On another, deeper level, our ideal design can be extended. This is a different kind of problem that requires us to consider the various kinds of *design mutations* that may occur as the software is maintained and adapted. This requires some in-depth knowledge of the problem domain. We need to know how the current problem is a specialization of other more general problems. We also need to note how our solution is only one of many solutions to the current problem. We have two dimensions of generalization: problem generalization as well as solution generalization.

Our initial design for roulette just barely provided the first level of solution. We didn't make any effort to plan for generalization. The "Design Heavy" solution generalizes Roulette to make it more suitable for Craps, also. Looking forward, we'll have to make even more adjustments before we have a very tidy, general solution.

27.7 Soapbox on Justification

It is very difficult to justify design rework. Some managers have a blind-spot on the amount of evidence required to justify rework. The conversations have the following form.

Architect. We need to disentangle Roulette and Craps so that Craps is not a subclass of Roulette. I've got a revised design that will take *X* hours of effort to implement.

Manager. I'll need some justification. Why do we have to fix it?

Architect. The structure is illogical: Craps isn't a special case of Roulette, they're independent specializations of something more general.

Manager. Illogical isn't a good enough justification. Our overall problem domain always contains illogical special cases and user-oriented considerations. You'll have to provide something more concrete.

Architect. Okay, in addition to being illogical, it will become too complex: in the future, we'll probably have trouble implementing other games.

Manager. How much trouble? Will it be more than *X* hours of effort?

Architect. When we include maintenance, adaptation and debugging, the potential future cost is probably larger than *2X* hours of effort. And it's illogical.

Manager. Probably larger? We can't justify rework based on probable costs. You'll need something tangible. Will it save us any lines of code?

Architect. No, it will add lines of code. But it will reduce maintenance and adaptation costs because it will be more logical.

Manager. I conclude that the change is unjustified.

In many cases, we have a manager who's mind is influenced by a schedule. A schedule that was created prior to any deep knowledge of the software being built. Someone focused on a schedule won't be swayed by facts. This is generally a symptom of an organization that is thinking-impaired. Typically, it is impossible to engage project managers further than this.

However, in some cases, conversation continues in the following vein.

Architect. Unjustified? Okay, please define what constitutes adequate justification for improvements?

Manager. Real savings of effort is the only justification for disrupting the schedule.

Architect. And future effort doesn't count?

Manager. The probability of savings in the future isn't tangible.

Architect. And more logical doesn't count?

Manager. If course not; it doesn't result in real schedule savings.

Architect. Real schedule savings? That's absurd. The schedule is a notional projection of possible effort. A possible reduction in the possible effort is just as real as the schedule.

Manager. Wouldn't it be simpler to...?

For reasons we don't fully understand, a schedule becomes a kind of established fact. Any change to the schedule requires other established facts, not the conjecture of design. For some reason, the reality that the schedule is only a conjecture, based on a previously conjectured design doesn't seem to sway managers from clinging to the schedule. This makes it nearly impossible to justifiy making a design change. The only way to accumulate enough evidence is to make the design change and then measure the impact of the change. In effect, no level of proof can ever override the precedence-setting fact of the schedule.

To continue this rant, the same kind of "inadequate evidence" issue seems to surround most technology changes. We have had conversations like the one shown above regarding object-oriented design, the use of objects in relational databases, the use of object-oriented databases, the use of open-source software, and the use of the star-schema data model for reporting and analysis. Here's the summary

> Some insist that change can only be considered based on established facts; we observe that these facts can only be established by making a change.

While this chicken-and-egg problem is easily resolved by recognizing that the current architecture or schedule is actually *not* an established fact, this is a difficult mental step to take.

As a final complaint, we note that all "wouldn't it be simpler..." gambits can be described as a *management trump card*. The point is rarely an effort to reduce code complexity, but to promote management understanding of the design. While this is a noble effort, there is sometimes a communication gap between designers and managers. It is incumbent both on designers to communicate fully, and on managers to provide time, budget and positive reinforcement for detailed communication of design considerations and consequences.

27.8 Design Light

In order to get the Craps game to work, we can minimize the amount of design. This minimal rework is a revision to `Outcome`.

This is followed by *Common Design*: the two subclasses of `Outcome` (`OutcomeField`, and `OutcomeHorn`), and the initializer for `Dice`.

This minimal design effort has one unpleasant consequence: Roulette's `Outcome` instances will depend on the Craps-specific `Throw` class. This entangles Roulette and Craps around a feature that is really a special case for Craps only. This kind of entanglement often limits our ability to successfully package and reuse these classes.

27.8.1 Minimal Outcome Rework

The class `Outcome` needs a method to compute the win amount based on a `Throw`.

In Python, it's sensible to optional parameters to achieve the same degree of flexibility.

`Outcome.`**`winAmount`**`(`*self*, *throw=None*`)` → int

Returns the product this `Outcome`'s odds numerator by the given amount, divided by the odds denominator.

Parameters **`throw`** (Throw) – An optional `Throw` that determines the actual odds to use. If not provided, this `Outcome`'s odds are used.

For Craps Horn bet and Field bets, a subclass will override this method to check the specific value of the `throw` and compute appropriate odds.

All other classes will ignore the optional `throw` parameter.

In principle, this is all we need.

What's wrong? We've hopeless entangled Roulette and Craps at a deep level. Roulette now depends on Craps details.

Sigh.

27.9 Design Heavy

In order to produce a solution that has a better architecture with more reusable components, we need to do some additional generalization. This design effort disentangles Roulette and Craps; they will not share the `Throw` class that should only be part of Craps. Instead, the highly reused `Outcome` class will depend only on a new superclass, `RandomEvent`, which is not specific to either game.

Given the new generalization, `RandomEvent`, we can rework the `Outcome` to use this for computing win amounts. We will have to rework `Bin`, `Wheel`, and `Throw` to make proper use of this new superclass.

Then we can move to the *Common Design* features: the craps-specific subclasses (`OutcomeField`, and `OutcomeHorn`), and the initializer for `Dice`.

27.9.1 RandomEvent class

class `RandomEvent`(*frozenset*)

The class *RandomEvent* is the superclass for the random events on which a player bets. This includes *Bin* of a Roulette wheel and *Throw* of Craps dice.

An event is a collection of inividiual *Outcome*s. Instances of *Bin* and a *Throw* can leverage this collection instead of leveraging frozenset directly.

Using a common class of our definition is slightly better than using a generic built-in class. The improvement is that we can extend our class to add features.

Note that there's no real implementation. We can use the `pass` statement for the body.

27.9.2 Bin Rework

The class *Bin* needs to be a subclass of *RandomEvent*.

The set of outcomes is removed from *Bin*; it's defined in *RandomEvent*.

27.9.3 Throw Rework

The class *Throw* needs to be a subclass of *RandomEvent*.

The set of outcomes is removed from *Throw*; it's defined in *RandomEvent*.

27.9.4 Outcome Rework

The class *Outcome* needs a method to compute the win amount based on a *RandomEvent*.

In Python, we use optional parameters to achieve the same degree of flexibility.

`Outcome.winAmount`(*self*, *RandomEvent=None*) → int

Returns the product this *Outcome*'s odds numerator by the given amount, divided by the odds denominator.

Parameters **event** (Throw) – An optional *RandomEvent* that determines the actual odds to use. If not provided, this *Outcome*'s odds are used.

For Craps Horn bet and Field bets, a subclass will override this method to check the specific value of the `event` and compute appropriate odds.

27.10 Common Design

Once we've finished the rework, we can design the various specialized outcomes required by Craps. We'll look at the two special cases we identified:

- *OutcomeField Design* will cover Field bets.
- *OutcomeHorn Design* will cover Horn bets.

Once we've defined all of the possible outcomes, we can move forward to building all of the thows. We'll examine this in *ThrowBuilder Class Design*.

27.10.1 OutcomeField Design

class OutcomeField

OutcomeField contains a single outcome for a field bets that has a number of different odds, and the odds used depend on a *RandomEvent*.

Methods

`OutcomeField.winAmount(self, throw=None) → int`

Returns the product this *Outcome*'s odds numerator by the given amount, divided by the odds denominator.

Parameters **throw** (Throw) – An optional *Throw* that determines the actual odds to use. If not provided, this *Outcome*'s odds are used.

`OutcomeField.__str__(self) → string`

This should return a `String` representation of the name and the odds. A form that looks like `Field (1:1, 2 and 12 2:1)` works nicely.

27.10.2 OutcomeHorn Design

class OutcomeHorn

OutcomeHorn contains a single outcome for a Horn bet that has a number of different odds, and the odds used depend on a *RandomEvent*.

Methods

`OutcomeHorn.winAmount(self, throw=None) → int`

Returns the product this *Outcome*'s odds numerator by the given amount, divided by the odds denominator.

Parameters **throw** (Throw) – An optional *Throw* that determines the actual odds to use. If not provided, this *Outcome*'s odds are used.

`OutcomeHorn.__str__(self) → string`

This should return a `String` representation of the name and the odds. A form that looks like `Horn (27:4, 3:1)` works nicely.

27.10.3 ThrowBuilder Class Design

class ThrowBuilder

ThrowBuilder initializes the 36 *Throw*s, each initialized with the appropriate *Outcome*s. Subclasses can override this to reflect different casino-specific rules for odds on Field bets.

Constructors

`ThrowBuilder.__init__(self)`

Initializes the ThrowBuilder.

Methods

ThrowBuilder.**buildThrows**(*self*, *dice*)

Creates the 8 one-roll *Outcome* instances (2, 3, 7, 11, 12, Field, Horn, Any Craps).

It then creates each of the 36 *Throw*s, each of which has the appropriate combination of *Outcome*s. The *Throw*s are assigned to `dice`.

27.11 Throw-Builder Deliverables

There are two deliverables for the light version of this exercise.

- Rework the *Outcome* class to add the new `winAmount()` method that uses a *Throw*.
- Rework the *Outcome* class unit test to exercise the new `winAmount()` method that uses a *Throw*. For all current subclasses of *Outcome*, the results of both versions of the `winAmount()` method produce the same results.

There are five deliverables for the heavy version of this exercise.

- Create the *RandomEvent* class.
- Rework the *Bin* class to be a subclass of *RandomEvent*. The existing unit tests for *Bin* should continue to work correctly.
- Rework the *Throw* class to be a subclass of *RandomEvent*. The existing unit tests should continue to work correctly.
- Rework the *Outcome* class to add the new `winAmount()` method that uses a *RandomEvent*.
- Rework the *Outcome* class unit test to exercise the new `winAmount()` method that uses a *RandomEvent*. For all current subclasses of *Outcome*, the results of both versions of the `winAmount()` method produce the same results.

There a six common deliverables no matter which approach you take.

- Create the *OutcomeField* class.
- Create a unit test for the *OutcomeField* class. Two instances of *Throw* are required: a 2 and a 3. This should confirm that there are different values for `winAmount()` for the two different *Throw* instances.
- Create the *OutcomeHorn* class.
- Create a unit test for the *OutcomeHorn* class. Two instances of *Throw* are required: a 2 and a 3. This should confirm that there are different values for `winAmount()` for the two different *Throw* instances.
- Create the *ThrowBuilder*. This was our objective, after all.
- Rework the unit test of the *Dice* class. The unit test should create and initialize a *Dice*. It can use the `getThrow()` method to check selected *Throw*s for the correct *Outcome*s.

The correct distribution of throws is as follows. This information will help confirm the results of *ThrowBuilder*.

Throw	Frequency
2	1
3	2
easy 4	2
hard 4	1
5	4
easy 6	4
hard 6	1
7	6
easy 8	4
hard 8	1
9	4
easy 10	2
hard 10	1
11	2
12	1

CHAPTER

TWENTYEIGHT

BET CLASS

This chapter will examine the *Bet* class, and its suitability for the game of Craps. We'll expand the design to handle additional complications present in real casino games.

We'll look at details of craps betting in *Bet Analysis*.

This will lead to some design changes. We'll cover these in *Bet Rework*.

We can then extend the *Bet* class hierarchy to include an additional kind of bet that's common in Craps. We'll cover the details in *CommissionBet Design*.

We'll detail the deliverables in *Bet Deliverables*.

28.1 Bet Analysis

A *Bet* is an amount that the player has wagered on a specific *Outcome*. This is a simple association of an amount, an *Outcome*, and a specific *Player*.

When considering the various line bet outcomes (Pass Line, Come Line, Don't Pass and Don't Come), we noted that when a point was established the bet was either a winner or a loser, or it was moved from the line to a particular number based on the throw of the dice. We'll need to add this responsibility to our existing definition of *Bet*. This responsibility can be implemented as a `setOutcome()` method that leaves the amount intact, but changes the *Outcome* from the initial Pass Line or Come Line to a specific point outcome.

A complexity of placing bets in Craps is the commission (or vigorish) required for Buy bets and Lay bets. This is a 5% fee, in addition to the bet amount. A player puts $21 down, which is a $20 bet and a $1 commission. We'll need to add a a commission or vig responsibility to our definition of *Bet*.

This price to place a bet generalizes nicely to all other bets. For most bets, the price is simply the amount of the bet. For Buy bets, the price is 5% higher than the amount of the bet; for Lay bets, the price depends on the odds. This adds a new method to *Bet* that computes the price of the bet. This has a ripple effect throughout our *Player* hierarchy to reflect this notion of the price of a bet. We will have to make a series of updates to properly deduct the price from the player's stake instead of deducting the amount of the bet.

There are two parts to creating a proper Craps bet: a revision of the base *Bet* to separate the price from the amount bet, and a *CommissionBet* subclass to compute prices properly for the more complex Craps bets.

28.2 Bet Rework

Bet associates an amount and an *Outcome*. The *Game* may move a *Bet* to a different *Outcome* to reflect a change in the odds used to resolve the *Bet*. In a future round of design, we can also associate a it with a *Player*.

28.2.1 Methods

Bet.**setOutcome**(*self*, *outcome*)

Parameters **outcome** (Outcome) – The new outcome for this bet amount

Sets the *Outcome* for this bet. This has the effect of moving the bet to another *Outcome*.

Bet.**price**(*self*) → int

Computes the price for this bet. For most bets, the price is the amount. Subclasses can override this to handle buy and lay bets where the price includes a 5% commission on the potential winnings.

For Buy and Lay bets, a $20 bet has a price of $21.

28.3 CommissionBet Design

class CommissionBet

CommissionBet is a *Bet* with a commission payment (or vigorish) that determines the price for placing the bet.

28.3.1 Fields

CommissionBet.**vig**

Holds the amount of the vigorish. This is almost universally 5%.

28.3.2 Methods

Bet.**price**(*self*) → int

Computes the price for this bet. There are two variations: Buy bets and Lay bets.

A Buy bet is a right bet; it has a numerator greater than or equal to the denominator (for example, 2:1 odds, which risks 1 to win 2), the price is 5% of the amount bet. A $20 Buy bet has a price of $21.

A Lay bet is a wrong bet; it has a denominator greater than the numerator (for example, 2:3 odds, which risks 3 to win 2), the price is 5% of 2/3 of the amount. A $30 bet Layed at 2:3 odds has a price of $31, the $30 bet, plus the vig of 5% of $20 payout.

28.4 Bet Deliverables

There are three deliverables for this exercise.

- The revised *Bet* class.
- The new *CommissionBet* subclass. This computes a price that is 5% of the bet amount.
- A class which performs a unit test of the various *Bet* classes. The unit test should create a couple instances of *Outcome*, and establish that the `winAmount()` and `price()` methods work correctly. It should also reset the *Outcome* associated with a *Bet*

We could rework the entire *Player* class hierarchy for Roulette to compute the *Bet*'s price in the `placeBets()`, and deduct that price from the player's stake. For Roulette, however, this subtlety is at the fringe of over-engineering, as no bet in Roulette has a commission.

CHAPTER TWENTYNINE

CRAPS TABLE CLASS

In Roulette, the table was a passive repository for *Bet*s. In Craps, however, the table and game must collaborate to accept or reject bets based on the state of the game. This validation includes rules based on the total amount bet as well as rules for the individual bets.

In *Throw Class*, we roughed out a stub version of *CrapsGame* that could be used to test *Throw*. In this section, we will extend this stub with additional features required by the table.

In *Craps Table Analysis* we'll look at two issues related to how the game works. In *Movable Bets* we'll look at how the Pass Line bet works. In *Game State and Allowed Bets* we'll look at the bet validation issue.

We'll dig into three separate design decisions:

- *Design Decision – Table vs. Game Responsibility*,
- *Design Decision – Allowable Outcomes*, and
- *Design Decision – Domain of Allowed Bets*.

After looking closely at the choices, in *Handling Working Bets* we'll create a final approach to handling the bets on the table.

This will lead to detailed design presentations in *CrapsGame Stub* and *Craps Table Design*. We'll enumerate the deliverables for this chapter in *Craps Table Deliverables*.

29.1 Craps Table Analysis

The *Table* is where the *Bet*s are placed. The money placed on *Bet*s on the *Table* is "at risk". There are several outcomes:

- A bet can win an amount based on the odds,
- A bet can lose the amount placed by the player,
- The Don't Come and Don't Pass bets may be returned, an event called a "push", and
- The Buy and Lay bets include a commission (or vigorish) to place the bet; the commission is lost money; the balance of the bet, however, may win or lose.

The responsibility for this new event called a push is something we can allocate to *Game*, the commission price belongs to *Bet*.

29.1.1 Movable Bets

Some *Bet*s (specifically Pass, Don't Pass, Come and Don't Come) may have their *Outcome* changed. The use case works like this.

1. The bet is created by the Player with one *Outcome*, for example, "Pass". The Table accepts this bet.

2. That bet may be resolved as an immediate winner or loser. The Game and Throw will determine if the Bet is a winner as placed.

   More commonly, the *Bet* may be changed to a new *Outcome*, possibly with different odds.

   In a casino, the chips initially placed on Come Line and Don't Come bets are relocated to a point number box to show this change. In the case of Pass Line and Don't Pass bets, the "On" marker is placed on the table to show an implicit movement of all of those line bets.

The change is the responsibility of the *Game*; however, the *Table* must provide an iterator over the line bets that the *Game* will move.

29.1.2 Game State and Allowed Bets

Each change to the game state changes the allowed bets as well as the active bets. When the point is off, most of the bets on the table are not allowed, and some others are inactive, or not "working".

When a point is established, all bets are allowed, and all bets are active. We'll examine the rules in detail, below.

The *Table* must be able to reject bets which are inappropriate for the current *Game* state.

29.2 Design Decision – Table vs. Game Responsibility

We've identified three responsibilities that are part of handling Craps bets:

- moving *Bet*s,
- inactivating outcomes based on game state, and
- accepting or rejecting bets based on *Game* state.

Clearly, these require additional collaboration between *Game* and *Table*. We will have to add methods to *CrapsGame* that will allow or deny some bets, as well as methods that will active or deactive some bets.

We have to choose where in the class hierarchy we will retrofit this additional collaboration.

Problem. Should we put these new responsibilities at a high-enough level that we'll add table and game collaboriation to the *Table* class used for Roulette?

Forces. If we do add this for Roulette, we could simply return `true` from the method that validates the allowed bets, since all bets are allowed in Roulette.

However, our overall application design does not depend on all subclasses of *Game* and *Table* being polymorphic; we will never mix and match different combinations of Craps Table and Roulette Game.

Solution. Because we don't need polymorphism between Craps and Roulette, we can create a subclass of *Table* with a more complex interface and leave Roulette untouched. Perhaps we'll call it `CrapsTable`.

29.3 Design Decision – Allowable Outcomes

After deciding to create a `CrapsTable` subclass, we have several consequent decisions. First, we turn the interesting question of how best to allocate responsibility for keeping the list of *Outcome*s which change with the game state.

Problem. Which class determines the valid and invalid *Outcome*s?

Forces. We can see two places to place this responsibility.

1. `CrapsTable`. This class could have methods to return the lists of *Outcome*s that are allowed or not allowed. *CrapsGame* can make a call to get the list of *Outcome*s and make the changes. Making each change would involve the `CrapsTable` a second time to mark the individual *Outcome*s. This information is then used by the `CrapsTable` to validate individual *Bet*s.
2. *CrapsGame*. This class could invoke a method of `CrapsTable` that changes a single *Outcome*'s state to makt it inactive. This information is then used by the `CrapsTable` to validate individual *Bet*s.

   A feature of this choice is to have the `validBet()` method of `CrapsTable` depend on *CrapsGame* to determine which bets are allowed or denied. In this case, *CrapsGame* has the responsibility to respond to requests from either `CrapsTable` or *Player* regarding a specific *Outcome*s.

Solution We need to place a validation method in the `CrapsTable`; but the Table simply delegates the details to the *CrapsGame*. This allows the Player to deal directly with the Table. But it centralizes the actual decision-making on the Game.

This leaves the table as a fairly passive repository for bets. The bulk of the decision-making for validity is delegated to the game.

Consequences. The game must move *Outcome*s for certain kinds of bets. Additionally, the `CrapsTable`'s `isValid()` method will use the *CrapsGame* to both check the validity of individual bets as well as the entire set of bets created by a player. The first check allows or denies individual bets, something `CrapsTable` must do in collaboration with *CrapsGame*. For the second check, the `CrapsTable` assures that the total of the bets is within the table limits; something for which only the table has the information required.

29.4 Design Decision – Domain of Allowed Bets

The rule for allowed and non-allowed bets is relatively simple. When the game state has no point (also known as the come out roll), only Pass Line and Don't Pass bets are allowed, all other bets are not allowed. When the point is on, all bets are allowed. We'll have to add an `isAllowed()` to *CrapsGame*, which `CrapsTable` will use when the player attempts to place a bet.

We have two ways to implement this:

- A "validation" function that determines if bets are allowed.
- A "what's possible" function that returns an enumeration of legal bets.

The idea of one-by-one validation might make sense in a situation where the player transactions are quite complex. For casino games, the player's alternatives are narrowly constrained.

It makes considerable sense for the *Game* to supply the domain of allowed bets to the *Table* and *Player*. In this way, a *Player* can simply extract the interesting bets from the available domain of possible bets.

29.5 Handling Working Bets

The rule for working and non-working bets adds a layer of complexity to the game state.

On the come out roll, all odds bets placed behind any of the six Come Point numbers are not working. This rule only applies to odds behind Come Point bets; odds behind Don't Come bets are always working. We'll have to add an `isWorking()` to *CrapsGame*, which `CrapsTable` will use when iterating through working bets.

The sequence of events that can lead to this condition is as follows.

1. The player places a Come Line bet, the dice roll is 4, 5, 6, 8, 9 or 10, and establishes a point. The bet is moved to one of the six come points.
2. The player creates an additional odds bet placed behind this come point bet.

3. A dice roll makes the main game point is a winner. changing the game state so the next roll is a come out roll. In this state, any additional odds behind a come point bet will be a non-working bets on the subsequent come-out roll.

As with allowed bets, we have a domain of working bets for a given game state. We can implement this as a function that responds with state information. We can also implement this as a collection of bets what are working in a given game state.

The code could look like this:

Working Bets Method

```
if theTable.is_working(some_bet):
    if theTable.winner(some_bet):
        player.win(some_bet)
    else:
        player.lose(some_bet)
```

Or, it could look like this:

Working Bets Collection

```
if some_bet in theTable.working_bets():
    if some_bet in theTable.winning_bets():
        player.win(some_bet)
    else:
        player.lose(some_bet)
```

The distinction is minor.

Also note that the examples don't include push outcomes. We'll look at the details of hanlding that in the *CrapsGame Class* section.

What's important is that we can handle these subtle cases gracefully. This elegant processing of complex rules is one of the important reasons why object-oriented programming can be more successful than procedural programming. In this case, we can isolate this state-specific processing to the `CrapsGame`. We can also provide the interface to the `CrapsTable` that makes this responsibility explicit and easy to use.

29.6 CrapsGame Stub

`CrapsGame` is a preliminary design for the game of Craps. In addition to features required by the `Throw`, this version includes features required by the `CrapsTable` class.

29.6.1 Methods

`CrapsGame.isAllowed(`*self*, *outcome*`)` → boolean

Parameters **outcome** (Outcome) – An `Outcome` that may be allowed or not allowed, depending on the game state.

Determines if the `Outcome` is allowed in the current state of the game. When the `point` is zero, it is the come out roll, and only Pass, Don't Pass, Come and Don't Come bets are allowed. Otherwise, all bets are allowed.

`CrapsGame.isWorking(`*self*, *outcome*`)` → boolean

Parameters outcome (Outcome) – An *Outcome* that may be allowed or not allowed, depending on the game state.

Determines if the *Outcome* is working in the current state of the game. When the *point* is zero, it is the come out roll, odds bets placed behind any of the six come point numbers are not working.

29.7 Craps Table Design

CrapsTable is a subclass of *Table* that has an association with a *CrapsGame* object. As a *Table*, it contains all the *Bet*s created by the *Player*. It also has a betting limit, and the sum of all of a player's bets must be less than or equal to this limit. We assume a single *Player* in the simulation.

29.7.1 Fields

CrapsTable.**game**

The *CrapsGame* used to determine if a given bet is allowed or working in a particular game state.

29.7.2 Constructors

CrapsTable.**__init__**(*self*, *game*)

Parameters game (CrapsGame) – The CrapsGame instance that controls the state of this table

Uses the superclass for initialization of the empty LinkedList of bets.

29.7.3 Methods

CrapsTable.**isValid**(*self*, *bet*) → boolean

Parameters bet (Bet) – The bet to validate.

Validates this bet by checking with the *CrapsGame* to see if the bet is valid; it returns true if the bet is valid, false otherwise.

CrapsTable.**allValid**(*self*) → boolean

This uses the superclass to see if the sum of all bets is less than or equal to the table limit. If the individual bet outcomes are also valid, return true. Otherwise, return false.

29.8 Craps Table Deliverables

There are three deliverables for this exercise.

- A revision of the stub *CrapsGame* class to add methods for validating bets in different game states. In the stub, the point value of 0 means that only the "Pass Line" and "Don't Pass Line" bets are valid, where a point value of non-zero means all bets are valid.
- The CrapsTable subclass.
- A class which performs a unit test of the CrapsTable class. The unit test should create a couple instances of *Bet*, and establish that these *Bet*s are managed by the table correctly.

  For testing purposes, it is easiest to have the test method simply set the the point variable in the *CrapsGame* instance to force a change in the game state. While public instance variables are considered by some to be a bad policy, they facilitate the creation of unit test classes.

CHAPTER THIRTY

CRAPSGAME CLASS

In *Throw Class*, we roughed out a stub version of `CrapsGame` that could be used to test `Throw`. We extended that stub in *Craps Table Class*. In this chapter, we will revise the game to provide the complete process for Craps. This involves a number of features, and we will have a state hierarchy as well as the Game class itself.

In the process of completing the design for `CrapsGame`, we will uncover another subtlety of craps: winning bets and losing bets. Unlike Roulette, where a `Bin` contained winning `Outcome`s and all other `Outcome`s where losers, Craps includes winning `Outcome`s, losing `Outcome`s, and unresolved `Outcome`s. This will lead to some rework of previously created Craps classes.

In *Craps Game Analysis* we'll examine the game of craps in detail. We'll look at four topics:

- *Game State* will address how the game transitions between point on and point off states.
- *The Game State Class Hierarchy* will look at the **State** design pattern and how it applies here.
- In *Resolving Bets* we'll look at how game, table, and player collaborate to resolved bets.
- In *Moveable Bets* we'll look at how the game collaborates with bets like the Pass Line bet which have an outcome that moves.

In *Design Decision – Win, Lose, Wait* we'll look at bet payoff issues.

In *Additional Craps Design* we'll revisit our algorithm for building `Outcome`s and `Throw`s and creating the `Dice`.

This chapter involves a large amount of programming. We'll detail this in *Craps Game Implementation Steps*.

We'll look at preliminary rework in a number of sections:

- *Throw Rework*,
- *ThrowBuilder Rework*, and
- *Bet Rework*.

Once we've cleaned up the `Throw` and `Bet` classes, we can focus on new development for the Craps game.

In *CrapsPlayer Class Stub* we'll rough out a design for a player.

We'll define the game states in the following sections:

- *CrapsGameState Class* will address the superclass,
- *CrapsGamePointOff Class* will look at the point off state, and
- *CrapsGamePointOn Class* will look at the point on state.

In *CrapsGame Design* we'll define the overall game class. We'll enumerate the deliverables for this chapter in *Craps Game Deliverables*.

We'll look at some additional game design issues in *Optional Working Bets*.

30.1 Craps Game Analysis

Craps is considerably more complex than Roulette. As noted in *Craps Details*, there is a multistep procedure that involves state changes, multiples rounds of betting, and bets that move from generic ("Come" or "Pass Line") to specific.

We can see three necessary features to the `CrapsGame`:

- *Game State*,
- *The Game State Class Hierarchy*,
- *Resolving Bets*, and
- *Moveable Bets* when a point is established.

Also, we will discover some additional design features to add to other classes.

30.1.1 Game State

A `CrapsGame` object cycles through the various steps of the Craps game; this sequence is shown in *Game State*. For statistical sampling purposes, we don't want to process complete games, since they have an arbitrary number of dice throws, and each throw offers additional betting opportunities. Because of all the betting opportunities, we will gather data from each individual throw of the dice. Since the dice are thrown at a predictable average rate, the length of a session depends on the number of throws and has little to do with the number of games.

Since we will follow the *State* design pattern, we have three basic design decisions. First, we have to design the state class hierarchy to own responsibilities for the unique processing of the individual states. Second, we have to design an interface for the game state objects to interact with the overall `CrapsGame`. Additionally, we will need to keep an object in the `CrapsGame` which contains the current state. Each throw of the dice will update the state, and possibly resolve game bets. To restart the game, we can create a fresh object for the initial point-off state.

The following procedure provides the detailed algorithm for the game of Craps.

A Single Game of Craps

Point Off State. The first throw of the dice is made with no point. This game may be resolved in a single throw of the dice, no point will be established. If a point is established the game transitions to the **Point On State**.

1. **Place Bets**. The point is off; this is the come out roll. Notify the `Player` to create `Bet`s. The real work of placing bets is delegated to the `Player` class. Only Pass and Don't Pass bets will be allowed by the current game state.
2. **Odds Bet Off?** Optional, for some casinos only. For any odds bets behind a come point, interrogate the player to see if the bet is on or off.
3. **Come-Out Roll**. Get the next throw of the `Dice`, giving the winning `Throw`, *t*. The `Throw` contains the individual `Outcome`s that can be resolved on this throw.
4. **Resolve Proposition Bets**. For each `Bet`, *b*, placed on a one-roll proposition:
   - **Proposition Winner?** If `Bet` *b*'s `Outcome` is in the winning `Throw`, *t*, then notify the `Player` that `Bet` *b* was a winner and update the `Player` 's stake. Note that the odds paid for winning field bets and horn bets depend on the `Throw`.
   - **Proposition Loser?** If `Bet` *b*'s `Outcome` is not in the winning `Throw`, *t*, then notify the `Player` that `Bet` *b* was a loser. This allows the `Player` to update their betting amount for the next round.
5. **Natural?** If the throw is a 7 or 11, this game is an immediate winner. The game state must provide the Pass Line `Outcome` as a winner.

For each `Bet`, *b* :

- **Come-Out Roll Natural Winner?** If `Bet` *b*'s `Outcome` is in the winning game state, then notify the `Player` that `Bet` *b* was a winner and update the `Player` 's stake. A Pass Line bet is a winner, and a Don't Pass bet is a loser.
- **Come-Out Roll Natural Loser?** If `Bet` *b*'s `Outcome` is not in the winning game state, then notify the `Player` that `Bet` *b* was a loser. This allows the `Player` to update the betting amount for the next round. A Pass Line bet is a loser, and a Don't Pass bet is a winner.

6. **Craps?** If the throw is a 2, 3, or 12, this game is an immediate loser. The game state must provide the Don't Pass Line `Outcome` as a winner; note that 12 is a push in this case, requiring special processing by the `Bet` or the `Outcome`: the bet amount is simply returned.

   For each `Bet`, *b*:

   (a) **Come-Out Roll Craps Winner?** If `Bet` *b* 's `Outcome` is in the winning game state and the bet is working, then notify the `Player` that `Bet` *b* was a winner and update the `Player`'s stake. If the bet is not working, it is ignored.

   (b) **Come-Out Roll Craps Loser?** If `Bet` *b* 's `Outcome` is not in the winning game state and the bet is working, then notify the `Player` that `Bet` *b* was a loser. If the bet is not working, it is ignored.

7. **Point Established**. If the throw is a 4, 5, 6, 8, 9 or 10, a point is established. The game state changes, to reflect the point being on. The Pass Line and Don't Pass Line bets have a new `Outcome` assigned, based on the point.

Point On State. While the game remains unresolved, the following steps are performed. The game is resolved when the point is made or a natural is thrown.

1. **Place Bets**. Notify the player to place any additional bets. The game state will allow all bets.
2. **Point Roll**. Get the next throw of the dice.
3. **Resolve Proposition Bets**. Resolve any one-roll proposition bets. This is the procedure described above for iterating through all one-roll propositions. See *Resolve Proposition Bets*.
4. **Natural?** If the throw was 7, the game is a loser. Resolve all bets; the game state will show that all bets are active. The game state will include Don't Pass and Don't Come bets as winners, as will any of the six point bets created from Don't Pass and Don't Come Line bets. All other bets will lose, including all hardways bets.

   This throw resolves the game, changing the game state. The point is off.
5. **Point Made?** If the throw was the main game point, the game is a winner. Resolve Pass Line and Don't Pass Line bets, as well as the point and any odds behind the point.

   Come Point and Don't Come Point bets (and their odds) remain for the next game. A Come Line or Don't Come Line bet will be moved to the appropriate Come Point.

   This throw ends the game; changing the game state. The point is off; odds placed behind Come Line bets are not working for the come out roll.
6. **Other Point Number?** If the throw was any of the come point numbers, come bets on that point are winners. Resolve the point come bet and any odds behind the point. Also, any buy or lay bets will be resolved as if they were odds bets behind the point; recall that the buy and lay bets involved a commission, which was paid when the bet was created.
7. **Hardways?** For 4, 6, 8 and 10, resolve hardways bets. If the throw was made the hard way (both dice equal), a hardways bet on the thrown number is a winner. If the throw was made the easy way, a hardways bet on the thrown number is a loser. If the throw was a 7, all hardways bets are losers. Otherwise, the hardways bets remain unresolved.

30.1.2 The Game State Class Hierarchy

We have identified some processing that is unique to each game state. Both states will have a unique list of allowed bets, a unique list of non-working bets, a unique list of throws that cause state change and resolve game bets, and throws that resolve hardways bets.

In the Craps Table (*Craps Table Analysis*), we allocated some responsibilities to `CrapsGame` so that a `CrapsTable` could validate bets and determine which bets were working.

Our further design details have shown that the work varies by state. Therefore, the methods in `CrapsGame` will delegate the real work to each state's methods. The current stub implementation checks the value of the `point` variable to determine the state. We'll replace this with simply calling an appropriate method of the current state object.

State Responsibilities. Each `CrapsGameState` subclass, therefore, will have an `isValid()` method that implements the state-specific bet validation rules. In this case, a point-off state object only allows the two Pass Line bets: Pass Line, Don't Pass Line. The point-on state allows all bets. Additionally, we've assigned to the `CrapsTable` has to determine if the total amount of all a player's bets meets or exceeds the table limits.

Each subclass of `CrapsGameState` will override an `isWorking()` method with one that validates the state-specific rules for the working bets. In this case, a point-off state object will identify the the six odds bets placed behind the come point numbers (4, 5, 6, 8, 9 and 10) as non-working bets, and all other bets will be working bets. A point-on state object will simply identify all bets as working.

The subclasses of `CrapsGameState` will need methods with which to collaborate with a `Throw` object to update the state of the `CrapsGame`.

Changing Game State. We have identified two game states: point-off (also know as the come out roll) and point-on. We have also set aside four methods that the various `Throw` objects will use to change the game state. The interaction between `CrapsGame`, the four kinds of `Throw`s and the two subclasses of `CrapsGameState`s works as follows:

1. There are 36 instances of `Throw`, one of which was selected at random to be the current throw of the dice.

   The `Game` object calls the `Throw` object's `updateGame()` method. Each of the subclasses of `Throw` have different implementations for this method.

2. The `Throw` object calls one of the `Game`'s methods to change the state. There are four methods available: `craps()`, `natural()`, `eleven()`, and `point()`. Differen subclasses of `Throw` will call an appropriate method for the kind of throw.

3. The `Game` has a current state, embodied in a `CrapsGameState` object. The Game will delegate each of the four state change methods (`craps()`, `natural()`, `eleven()`, and `point()`) to the current `CrapsGameState` object. There are two subclasses, depending on the state of the point: point-on and point-off.

4. In parallel with `Game`, each `CrapsGameState` object has four state change methods (`craps()`, `natural()`, `eleven()`, and `point()`). Each state provides different implementations for these methods. In effect, the two states and four methods create a kind of table that enumerates all possible state change rules.

Complex? At first glance the indirection and delegation seems like a lot of overhead for a simple state change. When we consider the kinds of decision-making this embodies, however, we can see that this is an effective solution.

When one of the 36 available `Throw`s has been chosen, the `CrapsGame` calls a single method to update the game state. Because the various subclasses of `Throw` are polymorphic, they all respond with unique, correct behavior.

Similarly, each of the subclasses of `Throw` simply uses one of four methods to update the `CrapsGame`, without having to discern the current state of the `CrapsGame`. We can consider `CrapsGame` as a kind of façade over the methods of the polymorphic `CrapsGameState`. Our objective is to do the decision-making once when the object is created; this makes all subsequent processing free of complex decision-making (i.e., simple) but indirect.

What's important about this design is that there are no if-statements required to make it work. Instead, objects simply invoke methods.

30.1.3 Resolving Bets

The *CrapsGame* class also has the responsibility for matching the *Outcome*s in the current *Throw* with the *Outcome*s of the *Bet*s on the CrapsTable.

In addition to matching *Outcome*s in the *Throw*, we also have to match the *Outcome*s of the current game state.

Finally, the *CrapsGame* class must also resolve hardways bets, which are casually tied to the current game state. We'll look at each of these three resolution procedures in some detail before making any final design decisions.

- **Resolving Bets on Proposition Outcomes**. We'll need a bet resolution method that handles one-roll propositions. This is similar to the bet resolution in the Roulette game class. The current *Throw* contains a collection of *Outcome*s which are resolved as winners. All other *Outcome*s will be losers. While appropriate for the one-roll propositions, we'll see that this doesn't generalize for other kinds of bets.
- **Resolving Bets on Game Outcomes**. The second, and most complex bet resolution method handles game outcomes. Bets on the game as a whole have three groups of *Outcome*s: winners, losers and unresolved.

  This "unresolved" outcome is fundamentally different from Roulette bet resolution and proposition bet resolution.

  Consider a Pass Line bet: in the point-off state, a roll of 7 or 11 makes this bet a winner, a roll of 2, 3 or 12 makes this bet a loser, all other numbers leave this bet unresolved. After a point is established, this Pass Line bet has the following resolutions: a roll of 7 makes this bet a loser, rolling the point makes this bet is a winner, all other numbers leave this bet unresolved.

  In addition to this three-way decision, we have the additional subtlety of Don't Pass Line bets that can lead to a fourth resolution: a push when the throw is 12 on a come out roll. We don't want to ignore this detail because it changes the odds by almost 3%.
- **Resolving Hardways Bets**. We have several choices for implementation of this multi-way decision. This is an overview, we'll dive into details below.
  - We can keep separate collections of winning and losing *Outcome*s in each *Throw*. This will obligate the game to check a set winners and a set of losers for bet resolution.
  - We can add a method to the *Bet* class that will return a code for the effect of a win, lose or wait for each *Outcome*. A win would add money to the *Player*; a lose would subtract money from the *Player*. This means that the game will have to decode this win-lose response as part of bet resolution.
  - We can make each kind of resolution into a **Command** class. Each subclass of BetResolution would peform the "pay a winner", "collect a loser" or "leave unresolved" procedure based on the *Throw* or class:*CrapsGameState*.

30.1.4 Moveable Bets

In the casino, the Come (and Don't Come) Line bets start on the given line. If a come point is established, the come line bet is moved to a numbered box. When you add behind the line odds bets, you place the chips directly on the numbered box for the Come Point number.

This procedure is different from the Pass (and Don't Pass) Line bet. The bet is is placed on the line. If a point is established, a large white "On" token shows the numbered box where, in effect, the behind the line odds chips belong.

Note that the net effect of both bets is identical. The pass line and behind-the-line odds bets have a payout that depends on the "On" token. The come line bets are moved and odds a place in a box on which the payout depends.

One of the things the *CrapsGame* does is change the *Outcome* of the Come and Don't Come Line bets. If a Come or Don't Come Line bet is placed and the throw is a point number (4, 5, 6, 8, 9, or 10), the bet is not resolved on the first throw; it is moved to one of the six point number *Outcome*s.

When designing the *Bet* class, in the Craps Bet *Bet Analysis*, we recognized the need to change the *Outcome* from a generic "Pass Line Odds" to a specific point with specific odds of 2:1, 3:2 or 6:5.

We'll develop a `moveToThrow()` method that accepts a *Bet* and the current *Throw* and move that bet to an appropriate new *Outcome*.

In addition to moving bets, we also need to create bets based on the currently established point. We also need to deactivate bets based on the established point.

As an example, the Pass Line Odds and Don't Pass Odds are created after the point is established. Since the point is already known, creating these bets is best done by adding a *CrapsGame.pointOutcome()* method that returns an *Outcome* based on the current point. This allows the `CrapsPlayer` to get the necessary *Outcome* object, create a *Bet* and give that *Bet* to the `CrapsTable`.

30.2 Design Decision – Win, Lose, Wait

Bet resolution in Craps is more complex than simply paying winners and collecting all other bets as losers. In craps, we have winners, losers and unresolved bets. Further, some bets have a resolution in which only the original price of the bet is returned. This is a kind of 1:1 odds special case.

Problem. What's the best way to retain a collection of *Outcome*s that are resolved as a mixture of winning, losing, unresolved, and pushes (resolved as a special case)?

Note that if we elect to introduce a more complex multi-way bet resolution, we have to decide if we should reimplement the bet resolution in Roulette. Using very different bet resolution algorithms for Craps and Roulette doesn't seem appealing. While a uniform approach is beneficial, it would involve some rework of the Roulette game to make use of a more sophisticated design.

Alternatives We'll look at three altenative responsibility assignments in some depth.

- **Winning and Losing Collections**. We could expand the *Throw* to keep separate collections of winners and losers. We could expand the `CrapsGameState` to detail the winning and losing *Outcome*s for each state. All other *Outcome*s would be left unresolved.

  This is a minor revision to *Dice* and *Throw* to properly create the two groups of *Outcome*s. Consequently *ThrowBuilder* will have to be expanded to identify losing *Outcome*s in addition to the existing winning *Outcome*s.

  This will require the *CrapsGame* to make two passes through the bets. It must match all active *Bet*s on the `CrapsTable` against the winning *Outcome*s in the current `CrapsGameState`; the matches are paid a winning amount and removed. It must also match match all active *Bet*s on the `CrapsTable` against the losing *Outcome*s in the current `CrapsGameState`; these are removed as losers.

- **Winning and Losing Codes, Evaluated by CrapsGame**. We could enumerate three code values that represent actions to take: these actions are "win", "lose", and "unresolved". The class *Game* and each subclass of `GameState` would have a resolution method that examines a *Bet* and returns the appropriate code value.

  This is a minor revision to *Dice* and *Throw* to properly associate a special code with each *Outcome*. Consequently *ThrowBuilder* will have to be expanded to identify losing *Outcome*s in addition to the existing winning *Outcome*s.

  Each *GameState* would also need to respond with appropriate codes.

  This will require the *CrapsGame* to make one pass through the *Bet*s, passing each each bet to the `GameState` resolution method. Based on the code returned, the *CrapsGame* would then have an if-statement to decide to provide bets to the *Player.win()* or *Player.lose()* method.

- **Wining and Losing Commands**. We could define a hierarchy of three subclasses. Each subclass implements winning, losing or leaving a bet unresolved.

  This is a minor revision to *Dice* and *Throw* to properly associate a special object with each *Outcome*. We would create single objects of each resolution subclass. The *ThrowBuilder* will have to be expanded to associate the loser Command or winner command with each *Outcome*. Further, the unresolved Command would have to be associated with all *Outcome*s that are not resolved by the *Throw* or *GameState*.

This will require the *CrapsGame* to make one pass through the *Bet*s, using the associated resolution object. The resolution object would then handle winning, losing and leaving the bet unresolved.

Before making a determination, we'll examine the remaining bet resolution issue to see if a single approach can cover single-roll, game and hardways outcomes.

Resolving Bets on Hardways Outcomes. In addition to methods to resolve one roll and game bets, we have to resolve the hardways bets. Hardways bets are similar to game bets. For *Throw*s of 4, 6, 8 or 10 there will be one of three outcomes:

- when the number is made the hard way, the matching hardways bet is a winner;
- when the number is made the easy way, the matching hardways bet is a loser; otherwise the hardways bet is unresolved;
- on a roll of seven, all hardways bets are losers.

Since this parallels the game rules, but applies to a *Throw*, it leads us to consider the design of *Throw* to be parallel to the design of *CrapsGame*. We can use either a collection of losing *Outcome*s in addition to the collection of winning *Outcome*s, or create a multi-way discrimination method, or have the *Throw* call appropriate methods of `CrapsTable` to resolve the bet.

Solution. A reasonably flexible design for *Bet* resolution that works for all three kinds of bet resolutions is to have *Throw* and `CrapsGameState` call specific bet resolution methods in `CrapsPlayer`.

This unifies one-roll, game and hardways bets into a single mechanism. It requires us to provide methods for win, lose and push in the `CrapsPlayer`. We can slightly simplify this to treat a push as a kind of win that returns the bet amount.

Consequences. The *CrapsGame* will iterate through the the active *Bet*s. Each *Bet* and the *Player* will be provided to the current *Throw* for resolving one-roll and hardways bets. Each *Bet* and the *Player* will also be provided to the `CrapsGameState` to resolve the winning and losing game bets.

We can further simplify this if each *Bet* carries a reference to the owning *Player*. In this way, the *Bet* has all the information necessary to notify the *Player*.

In the long run, this reflects the reality of craps table where the table operators assure that each bet has an owning player.

30.3 Additional Craps Design

We will have to rework our design for *Throw* to have both a one-roll resolution method and a hardways resolution method. Each of these methods will accept a single active *Bet*. Each resolution method could use a Set of winner *Outcome*s and a Set of loser *Outcome*s to attempt to resolve the *Bet*.

We will also need to rework our design for *Dice* to correctly set both winners and losers for both one-roll and harways bets when constructing the 36 individual *Throw* instances.

We can use the following expanded algorithm for building the *Dice*. This is a revision to *Throw Builder Analysis* to include lists of losing bets as well as winning bets.

Building Dice With Winning and Losing Outcomes

For All Faces Of Die 1. For all d_1, such that $1 \le d_1 < 7$:

For All Faces Of A Die 2. For d_2, such that $1 \le d_2 < 7$:

Sum the Dice. Compute the sum, $s \leftarrow d_1 + d_2$.

Craps? If s is in 2, 3, and 12, we create a *CrapsThrow* instance. The winning bets include one of the 2, 3 or 12 number *Outcome*, plus all craps, horn and field *Outcome*s. The losing bets include the other number *Outcome*s. This throw does not resolve hardways bets.

Point? For s in 4, 5, 6, 8, 9, and 10 we will create a *PointThrow* instance.

> **Hard way?** When $d_1 = d_2$, this is a *hard* 4, 6, 8 or 10. The appropriate hard number *Outcome* is a winner.
>
> **Easy way?** Otherwise, $d_1 \neq d_2$, this is an *easy* 4, 6, 8 or 10. The appropriate hard number *Outcome* is a loser.

Field? For s in 4, 9 and 10 we include the field *Outcome* as a winner. Otherwise the field *Outcome* is a loser. Note that 2, 3, and 12 Field outcomes where handled above under **Craps**.

Losing Propositions. Other one-roll *Outcome*s, including 2, 3, 7, 11, 12, Horn and Any Craps are all losers for this *Throw*.

Natural? If s is 7, we create a *NaturalThrow* instance. This will also include a 7 *Outcome* as a winner. It will have numbers 2, 3, 11, 12, Horn, Field and Any Craps *Outcome*s as losers for this *Throw*. Also, all four hardways are losers for this throw.

Eleven? If s is 11, we create an *ElevenThrow* instance. This will include 11, Horn and Field *Outcome*s as winners. It will have numbers 2, 3, 7, 12 and Any Craps *Outcome*s as losers for this *Throw*. There is no hardways resolution.

Craps Player Class Hierarchy. We have not designed the actual CrapsPlayer. This is really a complete tree of classes, each of which provides a different betting strategy. We will defer this design work until later. For the purposes of making the *CrapsGame* work, we can develop our unit tests with a kind of stub for CrapsPlayer which simply places a single Pass Line *Bet*. In several future exercises, we'll revisit this design to make more sophisticated players.

See *Some Betting Strategies* for a further discussion on an additional player decision offered by some variant games. Our design can be expanded to cover this. We'll leave this as an exercise for the more advanced student. This involves a level of collaboration between CrapsPlayer and *CrapsGame* that is over the top for this part. We'll address this kind of very rich interaction in *Blackjack*.

30.4 Craps Game Implementation Steps

We have identified the following things that must be done to implement the craps game.

1. Change the *Throw* to include both winning and losing *Outcome*s
2. Once we have fixed the *Throw* class, we can update the *ThrowBuilder* class to do a correct initialization using both winners and losers. Note that we have encapsulated this information so that there is no change to *Dice*.
3. We will also update *Bet* to carry a reference to the *Player* to make it easier to post winning and losing information directly to the player object.
4. We will need to create a stub CrapsPlayer for testing purposes.
5. We will also need to create our CrapsGameState class hierarchy to represent the two states of the game.
6. Once the preliminary work is complete, we can then transform the *CrapsGame* we started in *CrapsGame Stub* into a final version of *CrapsGame*. This will collaborate with a CrapsPlayer and maintain a correct CrapsGameState. It will be able to get a random *Throw* and resolve *Bet*s.

We'll address each of these separately.

30.5 Throw Rework

Throw is the superclass for the various throws of the dice. A *Throw* identifies two sets of *Outcome*s: immediate winners and immediate losers. Each subclass is a different grouping of the numbers, based on the state-change rules for Craps.

30.5.1 Fields

Throw.**win1Roll**
A set of of one-roll Outcomes that win with this throw.

Throw.**lose1Roll**
A set of one-roll Outcomes that lose with this throw.

Throw.**winHardway**
A set of hardways Outcomes that win with this throw. Not all throws resolve hardways bets, so this and the loseHardway Set may both be empty.

Throw.**loseHardway**
A set of hardways Outcomes that lose with this throw. Not all throws resolve hardways bets, so this and the winHardway Set may both be empty.

Throw.**d1**
One of the two die values, from 1 to 6.

Throw.**d2**
The other of the two die values, from 1 to 6.

30.5.2 Constructors

Throw.**__init__**(*d1*, *d2*, *winners=None*, *losers=None*)

Parameters

- **d1** (*int*) – One die value.
- **d2** (*int*) – The other die value.
- **winners** (set of *Outcome*s) – All the outcomes which will be paid as winners for this Throw.
- **losers** – All the outcomes which will be collectsd as winners for this Throw.

Creates this throw, and associates the two given Set s of *Outcome*s that are winning one-roll propositions and losing one roll propositions.

30.5.3 Methods

Throw.**add1Roll**(*self*, *winners*, *losers*)

Parameters

- **winners** (set of *Outcome*s) – All the outcomes which will be paid as winners for this Throw.
- **losers** – All the outcomes which will be collectsd as winners for this Throw.

Adds outcomes to the one-roll winners and one-roll losers Sets.

Throw.**addHardways**(*self*, *winners*, *losers*)

Parameters

- **winners** (set of *Outcome*s) – All the outcomes which will be paid as winners for this Throw.
- **losers** – All the outcomes which will be collectsd as winners for this Throw.

Adds outcomes to the hardways winners and hardways losers Sets.

`Throw.`**`hard`**`(`*self*`)` → boolean

Returns `true` if *d1* is equal to *d2*.

This helps determine if hardways bets have been won or lost.

`Throw.`**`updateGame`**`(`*self*, *game*`)`

Parameters game (CrapsGame) – CrapsGame instance to be updated with the results of this throw

Calls one of the *Game* state change methods: `craps()`, `natural()`, `eleven()`, `point()`. This may change the game state and resolve bets.

`Throw.`**`resolveOneRoll`**`(`*self*, *bet*`)`

Parameters bet – The bet to to be resolved

If this *Bet*'s *Outcome* is in the Set of one-roll winners, pay the *Player* that created the *Bet*. Return `true` so that this *Bet* is removed.

If this *Bet*'s *Outcome* is in the Set of one-roll losers, return `true` so that this *Bet* is removed.

Otherwise, return `false` to leave this *Bet* on the table.

`Throw.`**`resolveHardways`**`(`*self*, *bet*`)`

Parameters bet – The bet to to be resolved

If this *Bet*'s *Outcome* is in the Set of hardways winners, pay the *Player* that created the *Bet*. Return `true` so that this *Bet* is removed.

If this *Bet*'s *Outcome* is in the Set of hardways losers, return `true` so that this *Bet* is removed.

Otherwise, return `false` to leave this *Bet* on the table.

`Throw.`**`__str__`**`(`*self*`)` → str

This should return a `String` representation of the dice. A form that looks like `1,2` works nicely.

30.6 ThrowBuilder Rework

ThrowBuilder initializes the 36 *Throw*s, each initialized with the appropriate *Outcome*s. Subclasses can override this to reflect different casino-specific rules for odds on Field bets.

30.6.1 Methods

`ThrowBuilder.`**`buildThrows`**`(`*self*, *dice*`)`

Para dice The Dice to build

Creates the 8 one-roll *Outcome* instances (2, 3, 7, 11, 12, Field, Horn, Any Craps), as well as the 8 hardways *Outcome* instances (easy 4, hard 4, easy 6, hard 6, easy 8, hard 8, easy 10, hard 10).

It then creates each of the 36 *Throw*s, each of which has the appropriate combination of *Outcome*s for one-roll and hardways. The various *Throw*s are assigned to `dice`.

30.7 Bet Rework

Bet associates an amount, an *Outcome* and a *Player*. The *Game* may move a *Bet* to a different *Outcome* to reflect a change in the odds used to resolve the *Bet*.

30.7.1 Constructors

Bet.**__init__**(*self*, *amount*, *outcome*, *player=None*)

This replaces the existing constructor and adds an optional parameter.

Parameters

- **amount** (*int*) – The amount being wagered.
- **outcome** (Outcome) – The specific outcome on which the wager is placed.
- **player** (*CrapsPlayer*) – The player who will pay a losing bet or be paid by a winning bet.

Initialize the instance variables of this bet. This works by saving the additional player information, then using the existing `Bet.Bet()` constructor.

30.8 CrapsPlayer Class Stub

`CrapsPlayer` constructs a *Bet* based on the *Outcome* named `"Pass Line"`. This is a very persistent player.

30.8.1 Fields

CrapsPlayer.**passLine**

This is the *Outcome* on which this player focuses their betting. It will be an instance of the `"Pass Line"` *Outcome*, with 1:1 odds.

CrapsPlayer.**workingBet**

This is the current Pass Line *Bet*.

Initially this is `None`. Each time the bet is resolved, this is reset to `None`.

This assures that only one bet is working at a time.

CrapsPlayer.**table**

That *Table* which collects all bets.

30.8.2 Constructors

CrapsPlayer.**__init__**(*self*, *table*)

Parameters **table** (Table) – The *Table*.

Constructs the `CrapsPlayer` with a specific table for placing bets. The player creates a single `"Pass Line"` *Outcome*, which is saved in the *passLine* variable for use in creating *Bet*s.

30.8.3 Methods

30.9 CrapsGameState Class

`CrapsGameState` defines the state-specific behavior of a Craps game. Individual subclasses provide methods used by `CrapsTable` to validate bets and determine the active bets. Subclasses provide state-specific methods used by a *`Throw`* to possibly change the state and resolve bets.

30.9.1 Fields

`CrapsGameState.`**`game`**

The overall *`CrapsGame`* for which this is a specific state. From this object, the various next state-change methods can get the `CrapsTable` and an `Iterator` over the active *`Bet`*s.

30.9.2 Constructors

`CrapsGameState.`**`__init__`**`(`*self*, *game*`)`

Parameters **`game`** (Game) – The game to which this state applies

Saves the overall *`CrapsGame`* object to which this state applies.

30.9.3 Methods

`CrapsGameState.`**`isValid`**`(`*self*, *outcome*`)` → boolean

Parameters **`outcome`** (Outcome) – The outcome to be tested for validity

Returns true if this is a valid outcome for creating bets in the current game state.

Each subclass provides a unique definition of valid bets for their game state.

`CrapsGameState.`**`isWorking`**`(`*self*, *outcome*`)` → boolean

Parameters **`outcome`** (Outcome) – The outcome to be tested for if it's working

Returns true if this is a working outcome for existing bets in the current game state.

Each subclass provides a unique definition of active bets for their game state.

`CrapsGameState.`**`craps`**`(`*self*, *throw*`)` → CrapsGameState

Parameters **`throw`** (Throw) – The throw that is associated with craps.

Return an appropriate state when a 2, 3 or 12 is rolled. It then resolves any game bets.

Each subclass provides a unique definition of what new state and what bet resolution happens.

`CrapsGameState.`**`natural`**`(`*self*, *throw*`)` → CrapsGameState

Parameters **`throw`** (Throw) – The throw that is associated with a natural seven.

Returns an appropriate state when a 7 is rolled. It then resolves any game bets.

Each subclass provides a unique definition of what new state and what bet resolution happens.

`CrapsGameState.`**`eleven`**`(`*self*, *throw*`)` → CrapsGameState

Parameters **`throw`** (Throw) – The throw that is associated an eleven.

Returns an appropriate state when an 11 is rolled. It then resolves any game bets.

Each subclass provides a unique definition of what new state and what bet resolution happens.

`CrapsGameState.point(self, throw)` → CrapsGameState

Parameters **throw** (Throw) – The throw that is associated with a point number.

Returns an appropriate state when the given point number is rolled. It then resolves any game bets.

Each subclass provides a unique definition of what new state and what bet resolution happens.

`CrapsGameState.pointOutcome(self)` → Outcome

Returns the *Outcome* based on the current point. This is used to create Pass Line Odds or Don't Pass Odds bets. This delegates the real work to the current `CrapsGameState` object.

`CrapsGameState.moveToThrow(self, bet, throw)`

Parameters

- **bet** (Bet) – The Bet to update based on the current Throw
- **throw** (Throw) – The Throw to which the outcome is changed

Moves a Come Line or Don't Come Line bet to a new *Outcome* based on the current throw. If the value of `theThrow` is 4, 5, 6, 8, 9 or 10, this delegates the move to the current `CrapsGameState` object. For values of 4 and 10, the odds are 2:1. For values of 5 and 9, the odds are 3:2. For values of 6 and 8, the odds are 6:5. For other values of `theThrow`, this method does nothing.

`CrapsGameState.__str__(self)` → str

In the superclass, this doesn't do anything. Each subclass, however, should display something useful.

30.10 CrapsGamePointOff Class

`CrapsGamePointOff` defines the behavior of the Craps game when the point is off. It defines the allowed bets and the active bets. It provides methods used by a *Throw* to change the state and resolve bets.

All four of the game update methods (craps, natural, eleven and point) use the same basic algorithm. The method will get the `CrapsTable` from `theGame`. From the `CrapsTable`, the method gets the `Iterator` over the *Bet*s. It can then match each *Bet* against the various *Outcome*s which win and lose, and resolve the bets.

30.10.1 Constructors

`CrapsGamePointOff.__init__(self, game)`

Parameters **game** (CrapsGame) – The game to which this state applies.

Uses the superclass constructor to save the overall *CrapsGame* object.

30.10.2 Methods

`CrapsGamePointOff.isValid(self, outcome)` → boolean

Parameters **outcome** (Outcome) – The outcome to be tested for validity

There are two valid *Outcome*s: Pass Line, Don't Pass Line. All other *Outcome*s are invalid.

`CrapsGamePointOff.isWorking(self, outcome)` → boolean

Parameters **outcome** (Outcome) – The outcome to be tested to see if it's working

There are six non-working *Outcome*s: "Come Odds 4", "Come Odds 5", "Come Odds 6", "Come Odds 8", "Come Odds 9" and "Come Odds 10". All other *Outcome*s are working.

`CrapsGamePointOff.craps(self, throw)`

Parameters **throw** (Throw) – The throw that is associated with craps.

When the point is off, a roll of 2, 3 or 12 means the game is an immediate loser. The Pass Line *Outcome* is a loset. If the *Throw* value is 12, a Don't Pass Line *Outcome* is a push, otherwise the Don't Pass Line *Outcome* is a winner. The next state is the same as this state, and the method should return `this`.

`CrapsGamePointOff.`**`natural`**`(`*self*, *throw*`)`

Parameters **`throw`** (Throw) – The throw that is associated with a natural seven.

When the point is off, 7 means the game is an immediate winner. The Pass Line *Outcome* is a winner, the Don't Pass Line *Outcome* is a loser. The next state is the same as this state, and the method should return `this`.

`CrapsGamePointOff.`**`eleven`**`(`*self*, *throw*`)`

Parameters **`throw`** (Throw) – The throw that is associated an eleven.

When the point is off, 11 means the game is an immediate winner. The Pass Line *Outcome* is a winner, the Don't Pass Line *Outcome* is a loser. The next state is the same as this state, and the method should return `this`.

`CrapsGamePointOff.`**`point`**`(`*self*, *throw*`)`

Parameters **`throw`** (Throw) – The throw that is associated with a point number.

When the point is off, a new point is established. This method should return a new instance of `CrapsGame-PointOn` created with the given *Throw*'s value. Note that any Come Point bets or Don't Come Point bets that may be on this point are pushed to player: they can't be legal bets in the next game state.

`CrapsGamePointOff.`**`pointOutcome`**`(`*self*`)` → Outcome

Returns the *Outcome* based on the current point. This is used to create Pass Line Odds or Don't Pass Odds bets. This delegates the real work to the current `CrapsGameState` object. Since no point has been established, this returns `null`.

`CrapsGamePointOff.`**`__str__`**`(`*self*`)` → str

The point-off state should simply report that the point is off, or that this is the come out roll.

30.11 CrapsGamePointOn Class

`CrapsGamePointOn` defines the behavior of the Craps game when the point is on. It defines the allowed bets and the active bets. It provides methods used by a *Throw* to change the state and resolve bets.

30.11.1 Fields

`CrapsGamePointOn.`**`point`**

The point value.

30.11.2 Constructors

`CrapsGamePointOff.`**`__init__`**`(`*self*, *point*, *game*`)`

Saves the given point value. Uses the superclass constructor to save the overall *CrapsGame* object.

30.11.3 Methods

`CrapsGamePointOff.`**`isValid`**`(`*self*, *outcome*`)` → boolean

Parameters **`outcome`** (Outcome) – The outcome to be tested for validity

It is invalid to Buy or Lay the *Outcome*s that match the point. If the point is 6, for example, it is invalid to buy the "Come Point 6" *Outcome*. All other *Outcome*s are valid.

`CrapsGamePointOff.isWorking(self, outcome)` → boolean

Parameters **outcome** (Outcome) – The outcome to be tested to see if it's working

All *Outcome*s are working.

`CrapsGamePointOff.craps(self, throw)`

Parameters **throw** (Throw) – The throw that is associated with craps.

When the point is on, 2, 3 and 12 do not change the game state. The Come Line *Outcome* is a loser, the Don't Come Line *Outcome* is a winner. The next state is the same as this state, and the method should return `this`.

`CrapsGamePointOff.natural(self, outcome)`

Parameters **throw** (Throw) – The throw that is associated with a natural seven.

When the point is on, 7 means the game is a loss. Pass Line *Outcome*s lose, as do the pass-line odds *Outcome* s based on the point. Don't Pass Line *Outcome*s win, as do all Don't Pass odds *Outcome* based on the point. The Come Line *Outcome* is a winner, the Don't Come Line *Outcome* is a loser. However, all Come Point number *Outcome*s and Come Point Number odds *Outcome* are all losers. All Don't Come Point number *Outcome*s and Don't Come Point odds *Outcome*s are all winners. The next state is a new instance of the `CrapsGamePointOff` state.

Also note that the *Throw* of 7 also resolved all hardways bets. A consequence of this is that all `Bets` on the `CrapsTable` are resolved.

`CrapsGamePointOff.eleven(self, throw)`

Parameters **throw** (Throw) – The throw that is associated an eleven.

When the point is on, 11 does not change the game state. The Come Line *Outcome* is a winner, and the Don't Come Line *Outcome* is a loser. The next state is the same as this state, and the method should return `this`.

`CrapsGamePointOff.point(self, throw)`

Parameters **throw** (Throw) – The throw that is associated with a point number.

When the point is on and the value of `throw` doesn't match *point*, then the various Come Line bets can be resolved. Come Point *Outcome* s for this number (and their odds) are winners. Don't Come Line *Outcome* s for this number (and their odds) are losers. Other Come Point number and Don't Come Point numbers remain, unresolved. Any Come Line bets are moved to the Come Point number *Outcome*s. For example, a throw of 6 moves the *Outcome* of the Come Line *Bet* to Come Point 6. Don't Come Line bets are moved to be Don't Come number *Outcome*s. The method should return `this`.

When the point is on and the value of `throw` matches *point*, the game is a winner. Pass Line *Outcome*s are all winners, as are the behind the line odds *Outcome*s. Don't Pass line *Outcome*s are all losers, as are the Don't Pass Odds *Outcome*s. Come Line bets are moved to thee Come Point number *Outcome*s. Don't Come Line bets are moved to be Don't Come number *Outcome*s. The next state is a new instance of the `CrapsGamePointOff` state.

`CrapsGamePointOff.pointOutcome(self)` → Outcome

Returns the *Outcome* based on the current point. This is used to create Pass Line Odds or Don't Pass Odds bets. This delegates the real work to the current `CrapsGameState` object. For points of 4 and 10, the *Outcome* odds are 2:1. For points of 5 and 9, the odds are 3:2. For points of 6 and 8, the odds are 6:5.

`CrapsGamePointOff.__str__(self)` → str

The point-off state should simply report that the point is off, or that this is the come out roll.

30.12 CrapsGame Design

CrapsGame manages the sequence of actions that defines the game of Craps. This includes notifying the *Player* to place bets, throwing the *Dice* and resolving the *Bet* s actually present on the *Table*.

Note that a single cycle of play is one throw of the dice, not a complete craps game. The state of the game may or may not change.

30.12.1 Fields

`CrapsGame.dice`
Contains the dice that returns a randomly selected *Throw* with winning and losing *Outcome*s. This is an instance of *Dice*.

`CrapsGame.table`
The `CrapsTable` contains the bets placed by the player.

`CrapsGame.player`
The `CrapsPlayer` who places bets on the `CrapsTable`.

30.12.2 Constructors

We based this constructor on an design that allows any of these objects to be replaced. This is the *Strategy* design pattern. Each of these objects is a replaceable strategy, and can be changed by the client that uses this game.

Additionally, we specifically do not include the *Player* instance in the constructor. The *Game* exists independently of any particular *Player*, and we defer binding the *Player* and *Game* until we are gathering statistical samples.

`CrapsGame.__init__(self, dice, table)`

Parameters

- **dice** (Dice) – The dice to use
- **table** – The table to use for collecting bets
- **CrapsTable** –

Constructs a new *CrapsGame*, using a given *Dice* and `CrapsTable`.

The player is not defined at this time, since we may want to run several simulations with different players.

30.12.3 Methods

`CrapsGame.__init__(self, player)`

Parameters **player** (*CrapsPlayer*) – The player who will place bets on this game

This will execute a single cycle of play with a given *Player*.

1. It will call `thePlayer placeBets()` to get bets. It will validate the bets, both individually, based on the game state, and collectively to see that the table limits are met.
2. It will call `theDice` *next()* to get the next winning *Throw*.
3. It will use the *Throw*'s `updateGame()` to advance the game state.
4. It will then call `theTable bets()` to get an `Iterator` ; stepping through this `Iterator` returns the individual *Bet* objects.
   - It will use the *Throw*'s `resolveOneRoll()` method to check one-roll propositions. If the method returns true, the *Bet* is resolved and should be deleted.
   - It will use the *Throw*'s `resolveHardways()` method to check the hardways bets. If the method returns true, the *Bet* is resolved and should be deleted.

`CrapsGame.pointOutcome(self)` → Outcome
Returns the *Outcome* based on the current point. This is used to create Pass Line Odds or Don't Pass Odds bets. This delegates the real work to the current `CrapsGameState` object.

`CrapsGame.moveToThrow(self, bet, throw)`

Parameters

- **bet** (Bet) – The Bet to move based on the current throw
- **throw** (Throw) – The Throw to which to move the Bet's Outcome

Moves a Come Line or Don't Come Line bet to a new *Outcome* based on the current throw. This delegates the move to the current `CrapsGameState` object.

This method should – just as a precaution – assert that the value of `theThrow` is 4, 5, 6, 8, 9 or 10. These point values indicate that a Line bet can be moved. For other values of `theThrow`, this method should raise an exception, since there's no reason for attempting to move a line bet on anything but a point throw.

`CrapsGame.reset(self)`
This will reset the game by setting the state to a new instance of `GamePointOff`. It will also tell the table to clear all bets.

30.13 Craps Game Deliverables

There are over a dozen deliverables for this exercise. This includes significant rework for *Throw* and *Dice*. It also includes development of a stub `CrapsPlayer`, the `CrapsGameState` hierarchy and the first version of the *CrapsGame*. We will break the deliverables down into two groups.

Rework. The first group of deliverables includes the rework for *Throw* and *Dice*, and all of the associated unit testing.

- The revised and expanded *Throw* class. This will ripple through the constructors for all four subclasses, *NaturalThrow*, *CrapsThrow*, *ElevenThrow*, *PointThrow*.
- Five updated unit tests for the classes in the *Throw* class hierarchy. This will confirm the new functionality for holding winning as well as losing *Outcome*s.
- The revised and expanded *ThrowBuilder*. This will construct *Throw*s with winning as well as losing *Outcome*s.
- A unit test for the *Dice* class that confirms the new initializer that creates winning as well as losing *Outcome*s.

New Development. The second group of deliverables includes development of a stub `CrapsPlayer`, the `CrapsGameState` hierarchy and the first version of the *CrapsGame*. This also includes significant unit testing.

- The `CrapsPlayer` class stub. We will rework this design later. This class places a bet on the Pass Line when there is no Pass Line *Bet* on the table. One consequence of this is that the player will be given some opportunities to place bets, but will decline. Since this is simply used to test *CrapsGame*, it doesn't deserve a very sophisticated unit test of its own. It will be replaced in a future exercise.
- A revised *Bet*, which carries a reference to the *Player* who created the *Bet*. This will ripple through all subclasses of *Player*, forcing them to all add the `this` parameter when constructing a new *Bet*.
- The *CrapsGame* class.
- A class which performs a demonstration of the *CrapsGame* class. This demo program creates the *Dice*, the stub `CrapsPlayer` and the `CrapsTable`. It creates the *CrapsGame* object and cycles a few times. Note that we will need to configure the *Dice* to return non-random results.

We could, with some care, refactor our design to create some common superclasses between Roulette and Craps to extract features of *Throw* so they can be shared by *Throw* and *Bin*. Similarly, there may be more common features between `RouletteGame` and *CrapsGame*. We'll leave that as an exercise for more advanced students.

30.14 Optional Working Bets

Some casinos may give the player an option to declare the odds bet behind a come point as "on" or "off". This is should not be particularly complex to implement. There are a number of simple changes required if we want to add this interaction between `CrapsPlayer` and *`CrapsGame`*.

1. We must add a method to the `CrapsPlayer` to respond to a query from the *`CrapsGame`* that determines if the player wants their come point odds bet on or off.
2. We need to update *`Bet`* to store the *`Player`* who created the *`Bet`*. Third, the *`CrapsGame`* gets the relevant *`Bet`*s from the *`Table`*, and interrogates the *`Player`* for the disposition of the *`Bet`*.

CHAPTER

THIRTYONE

CRAPSPLAYER CLASS

The variations on `Player`, all of which reflect different betting strategies, is the heart of this application. In *Roulette Game Class*, we roughed out a stub class for `Player`, and refined it in *Player Class*. We will further refine this definition of `Player` for use in Craps.

In *Craps Player Analysis* we'll look at the general responsibilities and collaborators of a player. Since we've already examined many features of the game, we can focus on the player and revising the roughed-out version we created earlier.

We'll present the details of the design in three parts:

- *CrapsPlayer Design* covers the superclass features,
- *CrapsPlayerPass Subclass* covers a subclass which only bets the pass line, and
- *Craps Martingale Subclass* covers a player who uses Martigale betting.

In *Craps Player Deliverables* we'll detail the deliverables for this chapter.

31.1 Craps Player Analysis

We have now built enough infrastructure that we can begin to add a variety of players and see how their betting strategies work. Each player is betting algorithm that we will evaluate by looking at the player's stake to see how much they win, and when they stop playing because they've run out of time or gone broke.

The `Player` has the responsibility to create bets and manage the amount of their stake. To create bets, the player must create legal bets from known `Outcome` s and stay within table limits. To manage their stake, the player must deduct the price of a bet when it is created, accept winnings or pushes, report on the current value of the stake, and leave the table when they are out of money.

We have an interface that was roughed out as part of the design of `CrapsGame` and `CrapsTable`. In designing `CrapsGame`, we put a `placeBets()` method in `CrapsPlayer` to place all bets. We expected the `CrapsPlayer` to create `Bet` s and use the `placeBet()` method of `CrapsTable` class to save all of the individual `Bet`s.

In an earlier exercise, we built a stub version of `CrapsPlayer` in order to test `Game`. See *CrapsPlayer Class Stub*. When we finish creating the final superclass, `CrapsPlayer`, we will also revise our `CrapsPlayerStub` to be more complete, and rerun our unit tests to be sure that our more expanded design still handles the basic test cases correctly.

Our objective is to have a new abstract class, `CrapsPlayer`, with a concrete subclass that follows the Martingale system, using simple Pass Line bets and behind the line odds bets.

We'll defer some of the design required to collect detailed measurements for statistical analysis. In this first release, we'll simply place bets. Most of the `Simulator` class that we built for Roulette should be applicable to Craps without significant modification.

Some Basic Features. Our basic `CrapsPlayer` will place a Pass Line bet and a Pass Line Odds bet. This requires the player to interact with the `CrapsTable` or the *CrapsGame* to place bets legally. On a come out roll, only the Pass Line will be legal. After that, a single Pass Line Odds bet can be placed. This leads to three betting rules:

- Come Out Roll. Condition: No Pass Line Bet is currently placed and only the Pass Line bet is legal. Action: Place a Pass Line bet.
- First Point Roll. Condition: No odds bets is currently placed and odds bets are legal. Action: Place a Pass Line Odds bet.
- Other Point Roll. Condition: An odds bets is currently placed. Action: Do Nothing.

Beyond simply placing Pass Line and Pass Line Odds bets, we can use a Martingale or a Cancellation betting system to increase our bet on each loss, and decrease our betting amount on a win. Since we have two different bets in play – a single bet created on the come out roll, a second odds bet if possible – the simple Martingale system doesn't work well. In some casinos, the behind the line odds bet can be double the pass line bet, or even 10 times the pass line bet, giving us some complex betting strategies. For example, we could apply the Martingale system only to the odds bet, leaving the pass line bet at the table minimum. We'll set this complexity aside for the moment, build a simple player first.

31.2 CrapsPlayer Design

`CrapsPlayer` is a subclass of *Player* and places bets in Craps. This an abstract class, with no actual body for the *Player.placeBets()* method. However, this class does implement the basic `win()` and `lose()` methods used by all subclasses.

Since this is a subclass of the basic Roulette player, we inherit several useful features. Most of the features of *Player* are repeated here for reference purposes only.

31.2.1 Fields

`CrapsPlayer.`**`stake`**
: The player's current stake. Initialized to the player's starting budget.

`CrapsPlayer.`**`roundsToGo`**
: The number of rounds left to play. Initialized by the overall simulation control to the maximum number of rounds to play. In Roulette, this is spins. In Craps, this is the number of throws of the dice, which may be a large number of quick games or a small number of long-running games. In Craps, this is the number of cards played, which may be large number of hands or small number of multi-card hands.

`CrapsPlayer.`**`table`**
: The `CrapsTable` used to place individual *Bet*s.

31.2.2 Constructors

`CrapsPlayer.`**`__init__`**`(`*self*, *table*`)`

Parameters **`table`** (*CrapsTable*) – The table

Constructs the *Player* with a specific `CrapsTable` for placing *Bet*s.

31.2.3 Methods

`CrapsPlayer.`**`playing`**`(`*self*`)` → boolean
: Returns `true` while the player is still active. A player with a stake of zero will be inactive. Because of the indefinite duration of a craps game, a player will only become inactive after their *roundsToGo* is zero and they

have no more active bets. This method, then, must check the `CrapsTable` to see when all the bets are fully resolved. Additionally, the player's betting rules should stop placing new bets when the *roundsToGo* is zero.

`CrapsPlayer.placeBets(self) → boolean`
Updates the `CrapsTable` with the various *Bet* s.

When designing the `CrapsTable`, we decided that we needed to deduct the price of the bet from the stake when the bet is created. See the Roulette Table *Roulette Table Analysis* for more information on the timing of this deduction, and the Craps Bet *Bet Analysis* for more information on the price of a bet.

`CrapsPlayer.win(self, bet)`

Parameters **bet** (Bet) – that was a winner

Notification from the *CrapsGame* that the *Bet* was a winner. The amount of money won is available via `theBet winAmount()`.

`CrapsPlayer.lose(self, bet)`

Parameters **bet** (Bet) – that was a loser

Notification from the *CrapsGame* that the *Bet* was a loser.

31.3 CrapsPlayerPass Subclass

`CrapsPlayerPass` is a `CrapsPlayer` who places a Pass Line bet in Craps.

31.3.1 Methods

`CrapsPlayer.placeBets(self) → boolean`
If no Pass Line bet is present, this will update the *Table* with a bet on the Pass Line at the base bet amount.

Otherwise, this method does not place an additional bet.

31.4 Craps Martingale Subclass

`CrapsMartingale` is a `CrapsPlayer` who places bets in Craps. This player doubles their Pass Line Odds bet on every loss and resets their Pass Line Odds bet to a base amount on each win.

31.4.1 Fields

`CrapsPlayer.lossCount`
The number of losses. This is the number of times to double the pass line odds bet.

`CrapsPlayer.betMultiple`
The the bet multiplier, based on the number of losses. This starts at 1, and is reset to 1 on each win. It is doubled in each loss. This is always set so that $betMultiple = 2^{lossCount}$.

31.4.2 Methods

`CrapsPlayer.placeBets(self) → boolean`
If no Pass Line bet is present, this will update the *Table* with a bet on the Pass Line at the base bet amount.

If no Pass Line Odds bet is present, this will update the *Table* with an Pass Line Odds bet. The amount is the base amount times the *betMultiple*.

Otherwise, this method does not place an additional bet.

`CrapsPlayer.win(self, bet)`

Parameters **bet** (Bet) – that was a winner

Uses the superclass *win()* method to update the stake with an amount won. This method then resets *lossCount* to zero, and resets *betMultiple* to 1.

`CrapsPlayer.lose(self, bet)`

Parameters **bet** (Bet) – that was a loser

Increments *lossCount* by 1 and doubles *betMultiple*.

31.5 Craps Player Deliverables

There are six deliverables for this exercise.

- The CrapsPlayer abstract superclass. Since this class doesn't have a body for the placeBets() method, it can't be unit tested directly.
- A CrapsPlayerPass class that is a proper subclass of CrapsPlayer, but simply places bets on Pass Line until the stake is exhausted.
- A unit test class for CrapsPlayerPass. This test should synthesize a fixed list of *Outcome*s, *Throw* s, and calls a CrapsPlayerPass instance with various sequences of craps, naturals and points to assure that the pass line bet is made appropriately.
- The CrapsMartingale subclass of CrapsPlayer.
- A unit test class for CrapsMartingale. This test should synthesize a fixed list of *Outcome*s, *Throw* s, and calls a CrapsMartingale instance with various sequences of craps, naturals and points to assure that the bet doubles appropriately on each loss, and is reset on each win.
- The unit test class for the *CrapsGame* class should still work with the new CrapsPlayerPass. Using a non-random generator for *Dice*, this should be able to confirm correct operation of the *CrapsGame* for a number of bets.

CHAPTER

THIRTYTWO

DESIGN CLEANUP AND REFACTORING

We have taken an intentionally casual approach to the names chosen for our various classes and the relationships among those classes. At this point, we have a considerable amount of functionality, but it doesn't reflect our overall purpose, instead it reflects the history of its evolution. This chapter will review the design from Craps and more cleanly separate it from the design for Roulette.

We expect two benefits from the rework in this chapter. First, we expect the design to become "simpler" in the sense that Craps is separated from Roulette, and this will give us room to insert Blackjack into the structure with less disruption in the future. Second, and more important, the class names will more precisely reflect the purpose of the class, making it easier to understand the system, which means it will be easier to debug, maintain and adapt.

We'll start with a review of the current design in *Design Review*. This will include a number of concerns:

- *Unifying Bin and Throw*,
- *Unifying Dice and Wheel*,
- *Refactoring Table and CrapsTable*,
- *Refactoring Player and CrapsPlayer*, and
- *Refactoring Game and CrapsGame*.

Based on this, we'll need to rework some existing class defintions. This will involve making small changes to a large number of classes. The work is organized as follows:

- *RandomEventFactory Design*,
- *Wheel Class Design*,
- *Dice Class Design*,
- *Table Class Design*,
- *Player Class Design*,
- *Game Class Design*,
- *RouletteGame Class Design*, and
- *CrapsGame Class Design*.

In *Refactoring Deliverables* we'll detail all of the deliverables for this chapter.

32.1 Design Review

We can now use our application to generate some more usable results. We would like the `Simulator` class to be able to use our Craps game, dice, table and players in the same way that we use our Roulette game, wheel, table and players. The idea would be to give the `Simulator`'s constructor Craps-related objects instead of Roulette-related objects and have everything else work normally. Since we have generally made Craps a subclass of Roulette, we are reasonably confident that this should work.

Our *Simulator*'s constructor requires a *Game* and a *Player*. Since *CrapsGame* is a subclass of *Game* and CrapsPlayer is a subclass of *Player*, we can construct an instance of *Simulator*.

Looking at this, however, we find a serious problem with the names of our classes and their relationships. When we designed Roulette, we used generic names like *Table*, *Game* and *Player* unwisely. Further, there's no reason for Craps to be dependent on Roulette. We would like them to be siblings, and both children of some abstract game simulation.

> **Soapbox On Refactoring**
>
> We feel very strongly that our *design by refactoring* helps beginning designers produce a more functional design more quickly than the alternative approach, which is to attempt to define the game abstraction and player abstraction first and then specialize the various games. When defining an abstract superclass, some designers will build a quick and dirty design for some of the subclasses, and use this to establish the features that belong in the superclass. We find that a more successful superclass design comes from have more than one working subclasses and a clear understanding of the kinds of extensions that are likely to emerge from the problem domain.
>
> While our approach of refactoring working code seems expensive, the total effort is often smaller. The largest impediment we've seen seems to stem from the project-management mythology that once something passes unit tests it is done for ever and can be checked off as completed. We feel that it is very important to recognized that passing a unit test is only one of many milestones. Passing an integration test, and passing the *sanity test* are better indicators of done-ness.
>
> The sanity test is the designer's ability to explain the class structure to someone new to the project. We feel that class and package names must make obvious sense in the current project context. We find that any explanation of a class name that involves the words "historically" or "originally" means that there are more serious design deficiencies that need to be repaired.

We now know enough to factor out the common features of *Game* and *CrapsGame* to create three new classes from these two. However, to find the common features of these two classes, we'll see that we have to unify *Dice* and *Wheel*, as well as *Table* and CrapsTable and *Player* and CrapsPlayer.

Looking into *Dice* and *Wheel*, we see that we'll have to tackle first. Unifying *Bin* and *Throw* is covered in *Design Heavy*.

We have several sections on refactoring these various class hierarchies:

- *Bin* and *Throw* in *Unifying Bin and Throw*.
- *Wheel* *Dice* and in *Unifying Dice and Wheel*.
- *Table* and CrapsTable in *Refactoring Table and CrapsTable*.
- *Player* and CrapsPlayer in *Refactoring Player and CrapsPlayer*.
- *Game* and *CrapsGame* in *Refactoring Game and CrapsGame*.

This will give us two properly parallel structures with names that reflect the overall intent.

32.1.1 Unifying Bin and Throw

We need to create a common superclass for *Bin* and *Throw*, so that we can then create some commonality between *Dice* and *Wheel*.

The first step, then, is to identify what are the common features of *Bin* and *Throw*. The relatively simple *Bin* and the more complex *Throw* can be unified in one of two ways.

1. Use *Throw* as the superclass. A Roulette *Bin* doesn't need a specific list of losing *Outcome*s. Indeed, we don't even need a subclass, since a Roulette *Bin* can just ignore features of a Craps *Throw*.
2. Create a new superclass based on *Bin*. We can then make *Bin* a subclass that adds no new features. We can change *Throw* to add features to the new superclass. This makes *Bin* and *Throw* peers with a common parent.

The first design approach is something we call the **Swiss Army Knife** design pattern: create a structure that has every possible feature, and then ignore the features you don't need. This creates a distasteful disconnect between the use of *Bin* and the declaration of *Bin*: we only use the Set of winning *Outcome*s, but the object also has a losing Set that isn't used by anything else in the Roulette game.

We also note that a key feature of OO languages is inheritance, which *adds* features to a superclass. The **Swiss Army Knife** design approach, however, works by subtracting features. This creates a distance between the OO language and our design intent.

Our first decision, then, is to refactor *Throw* and *Bin* to make them instances of a common superclass, which we'll call *RandomEvent*. See the Craps Throw *Throw Analysis* for our initial thoughts on this, echoed in the *Soapbox on Refectoring* sidebar.

The responsibilities for *RandomEvent* are essentially the same as *Bin*. We can then make *Bin* a subclass that doesn't add any new features, and *Throw* a subclass that adds a number of features, including the value of the two dice and the Set of losing *Outcome*s. Note that we have made *Throw* and *Bin* siblings of a common superclass. See *Soapbox on Architecture* for more information on our preference for this kind of design.

32.1.2 Unifying Dice and Wheel

When we take a step back from *Dice* and *Wheel*, we see that they are nearly identical. They differ in the construction of the *Bin*s or *Throw*s, but little else. Looking forward, the deck of cards used for Blackjack is completely different. Dice and a Roulette wheel use *:phrase:'selection with replacement'* : an event is picked at random from a pool, and is eligible to be picked again any number of timers. Cards, on the other hand, are *selection without replacement*: the cards form a sequence of events of a defined length that is randomized by a shuffle. If we have a 5-deck shoe, we can only see five kings of spades during the game, and we only have 260 cards. However, we can roll an indefinite number of 7's on the dice.

We note that there is a superficial similarity between the rather complex *BinBuilder* methods and the simpler method in *ThrowBuilder*. However, there is no compelling reason for polymorphism between these two classes. We don't have to factor these into a common class hierarchy.

Our second design decision, then, is to create a `RandomEventFactory` out of *Dice* and *Wheel* . This refactoring will make the `Vector` of results and the *next()* method part of the superclass. Each subclass provides the initialization method that constructs the `Vector` of *RandomEvent*s.

When we move on to tackle cards, we'll have to create a subclass that uses a different definition of *next()* and adds `shuffle()`. This will allow a deck of cards to do selection without replacement.

32.1.3 Refactoring Table and CrapsTable

We see few differences between *Table* and `CrapsTable`. When we designed `CrapsTable` we had to add a relationship between `CrapsTable` and *CrapsGame* so that a table could ask the game to validate individual *Bet*s based on the state of the game.

If we elevate the `CrapsTable` to be the superclass, we eliminate a need to have separate classes for Craps and Roulette. We are dangerously close to embracing a **Swiss Army Knife** design. The distinction may appear to be merely a matter of degree: one or two features can be pushed up to the superclass and then ignored in a subclass.

In this case, both Craps and Roulette will use the *Game* as well as the *Table* to validate bets: the feature will not be ignored. It happens that the Roulette Game will permit all bets, but we have made that the Game's responsibility, not the Table's.

Viewed this way, the Roulette version of Table implicitly took a responsibility away from the Roulette Game because the Table failed to collaborate with the Game for any game-state specific rules. At the time, we overlooked this nuance because we knew that Roulette was stateless and we were comfortable making that assumption part of the design.

We actually have to sets of rules that must be imposed on bets. The table rules that impose an upper (and lower) limit on the bets. The game rules that specify which outcomes are legal in a given game state.

The *Game* rules are effectively a set of *Outcome*s. The *Table* rules is a method that checks the sum of the amount attributes.

Our third design decision is to merge *Table* and `CrapsTable` into a new *Table* class and use this for both games. This will simplify the various Game classes by using a single class of *Table* for both games.

32.1.4 Refactoring Player and CrapsPlayer

Before we can finally refactor *Game*, we need to be sure that we have sorted out a proper relationship between our various players. In this case, we have a large hierarchy, which will we hope to make far larger as we explore different betting alternatives. Indeed, the central feature of this simulation is to expand the hierarchy of players as needed to explore betting strategies. Therefore, time spent organizing the *Player* hierarchy is time well spent.

We'd like to have the following hierarchy.

- Player.
  - RoulettePlayer.
    - RouletteMartingale.
    - RouletteRandom.
    - RouletteSevenReds.
    - Roulette1326.
    - RouletteCancellation.
    - RouletteFibonacci.
  - CrapsPlayer.
    - CrapsMartingale.

Looking forward to Blackjack, see see that there is much richer player interaction, because there are player decisions that are not related to betting. This class hierarchy seems to enable that kind of expansion.

We note that there are two "dimensions" to this class hierarchy. One dimension is the game (Craps or Roulette), the other dimension is a betting system (Matingale, 1326, Cancellation, Fibonacci). For Blackjack, there is also a playing system in addition to a betting system. Sometimes this multi-dimensionsal aspect of a class hierarchy indicates that we should be using multiple inheritance to define our classes.

In the case of Python, we have multiple inheritance in the language, and we can pursue this directly. We can also follow the **Strategy** design pattern to add a betting strategy object to the basic interface for playing the game.

In Roulette, where we are placing a single bet, there is almost no distinction between game interface and the betting system. However, in Craps, we made a distinction in our Martingale player by separating their Pass Line bet (where the payout doesn't match the actual odds very well) from their Pass Line Odds bet (where the payout does match the odds). This means that our Martingale Craps player really has two betting strategies objects: a flat bet strategy for Pass Line and a Martingale Bet strategy for the Pass Line Odds.

If we separate the player and the betting system, we could easily mix and match betting systems, playing systems and game rules. In the case of Craps, where we can have many working bets (Pass Line, Come Point Bets, Hardways Bets, plus Propostions), each player would have a mixture of betting strategies used for their unique mixture of working bets. This leads to an interesting issue in the composition of such a complex object. For the current exercise, however, we won't formally separate the player from the various betting strategies.

Rather than fully separate the player's game interface and betting system interface, we can simply adjust the class hierarchy and the class names to those shown above. We need to make the superclass, *Player* independent of any game. We can do this by extracting anything Roulette-specific from the original *Player* class and renaming our

Roulette-focused *Passenger57* to be RoulettePlayer, and fix all the Roulette player subclasses to inherit from RoulettePlayer.

We will encounter one design difficulty when doing this. That is the dependency from the various *Player1326State* classes on a field of *Player1326*. Currently, we will simply be renaming *Player1326* to Roulette1326. However, as we go forward, we will see how this small issue will become a larger problem. In Python, we can easily overlook this, as described in *Python and Interface Design*.

Python and Interface Design

Because of the run-time binding done in Python, there is no potential problem in having the *Player1326State* classes depend on a field defined in *Player1326*.
Other languages – like Java – involve compile-time binding. A small change can lead to recompiling the world. In some respects this can be helpful for identifying a problem.
In Python, however, the attributes of an object are created dynamically, making it difficult to assure in advance that one class properly includes the attributes that will be required by a collaborating class. We don't discover problems in Python until the unit tests raise exceptions because of a missing attribute.

32.1.5 Refactoring Game and CrapsGame

Once we have a common RandomEventFactory, a common *Table*, and a common *Player*, we can separate *Game* from RouletteGame and *CrapsGame* to create three new classes.

- The abstract superclass, *Game*. This will contain a RandomEventFactory, a *Table* and have the necessary interface to reset the game and execute one cycle of play. This class is based on the existing *Game*, with the Roulette-specific *cycle()* replaced with an abstract method definition.
- The concrete subclass, RouletteGame. This has the *cycle()* method appropriate to Roulette that was extracted from the original *Game* class.
- The concrete subclass, *CrapsGame*. This has the *cycle()* method appropriate to Craps. This is a small change to the parent of the *CrapsGame* class.

While this appears to be a tremendous amount of rework, it reflects lessons learned incrementally through the previous chapters of exercises. This refactoring is based on considerations that would have been challenging, perhaps impossible, to explain from the outset. Since we have working unit tests for each class, this refactoring is easily validated by rerunning the existing suite of tests.

32.2 RandomEventFactory Design

RandomEventFactory is a superclass for Dice, Wheel, Cards, and other casino random-event generators.

32.2.1 Fields

RandomEventFactory.**rng**

The random number generator, a subclass of random.Random.

Generates the next random number, used to select a *RandomEvent* from the bins collection.

RandomEventFactory.**current**

The most recently returned *RandomEvent*.

32.2.2 Constructors

`RandomEventFactory.__init__(self, rng)`
Saves the given Random Number Generator. Calls the `initialize()` method to create the vector of results.

32.2.3 Methods

`RandomEventFactory.initialize(self)`
Create a collection of `RandomEvent` objects with the pool of possible results.

Each subclass must provide a unique implementation for this.

`RandomEventFactory.next(self)` → RandomEvent
Return the next `RandomEvent`.

Each subclass must provide a unique implementation for this.

32.3 Wheel Class Design

`Wheel` is a subclass of `RandomEventFactory` that contains the 38 individual `Bin`s on a Roulette wheel. As a `RandomEventFactory`, it contains a random number generator and can select a `Bin` at random, simulating a spin of the Roulette wheel.

32.3.1 Constructors

`Wheel.__init__(self, rng)`
Creates a new wheel. Create a sequence of the `Wheel.events` with with 38 empty `Bin`s.

Use the superclass to save the given random number generator instance and invoke `initialize()`.

32.3.2 Methods

`Wheel.addOutcome(self, bin, outcome)`
Adds the given `Outcome` to the `Bin` with the given number.

`Wheel.initialize(self)`
Creates the `events` vector with the pool of possible events. This will create an instance of `BinBuilder`, `bb`, and delegate the construction to the `buildBins()` method of the `bb` object.

32.4 Dice Class Design

`Dice` is a subclass of `RandomEventFactory` that contains the 36 individual throws of two dice. As a `RandomEventFactory`, it contains a random number generator and can select a `Throw` at random, simulating a throw of the Craps dice.

32.4.1 Constructors

`Dice.__init__(self, rng)`
Create an empty set of `Dice.events`. Use the superclass to save the given random number generator instance and invoke `initialize()`.

32.4.2 Methods

Wheel.**addOutcome**(*self*, *faces*, *outcome*)
Adds the given *Outcome* to the *Throw* with the given NumberPair. This allows us to create a collection of several one-roll *Outcome*s. For example, a throw of 3 includes four one-roll *Outcome*s: Field, 3, any Craps, and Horn.

Wheel.**initialize**(*self*)
Creates the 8 one-roll *Outcome* instances (2, 3, 7, 11, 12, Field, Horn, Any Craps). It then creates the 36 *Throw* s, each of which has the appropriate combination of *Outcome*s.

32.5 Table Class Design

Table contains all the *Bet*s created by the *Player*. A table has an association with a *Game*, which is responsible for validating individual bets. A table also has betting limits, and the sum of all of a player's bets must be within this limits.

32.5.1 Fields

Table.**minimum**
This is the table lower limit. The sum of a *Player*'s bets must be greater than or equal to this limit.

Table.**maximum**
This is the table upper limit. The sum of a *Player*'s bets must be less than or equal to this limit.

Table.**bets**
This is a LinkedList of the *Bet*s currently active. These will result in either wins or losses to the *Player*. :noindex:

Table.**game**
The *Game* used to determine if a given bet is allowed in a particular game state.

32.5.2 Constructors

Table.**__init__**(*self*)
Creates an empty list of bets.

32.5.3 Methods

Table.**setGame**(*self*, *game*)
Saves the given *Game* to be used to validate bets.

Table.**isValid**(*self*, *bet*) → boolean
Validates this bet. The first test checks the *Game* to see if the bet is valid.

Table.**allValid**(*self*, *bet*) → boolean
Validates the sum of all bets within the table limits. Returns false if the minimum is not met or the maximum is exceeded.

Table.**placeBet**(*self*, *bet*) → boolean
Adds this bet to the list of working bets. If the sum of all bets is greater than the table limit, then an exception should be raised. This is a rare circumstance, and indicates a bug in the *Player* more than anything else.

`Table.__iter__(self)` → iter
: Returns an `Iterator` over the list of bets. This gives us the freedom to change the representation from `list` to any other `Collection` with no impact to other parts of the application.

  We could simply return the list object itself. This may, in the long run, prove to be a limitation. It's handy to be able to simply iterate over a table and example all of the bets.

`Table.__str__(self)` → str
: Reports on all of the currently placed bets.

32.6 Player Class Design

Player places bets in a *Game*. This an abstract class, with no actual body for the `placeBets()` method. However, this class does implement the basic `win()` and `lose()` methods used by all subclasses.

Roulette Player Hierarchy. The classes in the Roulette Player hierarchy need to have their superclass adjusted to conform to the newly-defined superclass. The former *Passenger57* is renamed to `RoulettePlayer`. All of the various Roulette players become subclasses of `RoulettePlayer`.

In addition to renaming *Player1326* to `Roulette1326`, we will also have to change the references in the various classes of the *Player1326State* class hierarchy. We suggest leaving the class names alone, but merely changing the references within those five classes from *Player1326* to `Roulette1326`.

Craps Player Hierarchy. The classes in the Craps Player hierarchy need to have their superclass adjusted to conform to the newly-defined superclass. We can rename `CrapsPlayerMartigale` to `CrapsMartigale`, and make it a subclass of `CrapsPlayer`. Other than names, there should be no changes to these classes.

32.6.1 Fields

`Player.stake`
: The player's current stake. Initialized to the player's starting budget.

`Player.roundsToGo`
: The number of rounds left to play. Initialized by the overall simulation control to the maximum number of rounds to play. In Roulette, this is spins. In Craps, this is the number of throws of the dice, which may be a large number of quick games or a small number of long-running games. In Craps, this is the number of cards played, which may be large number of hands or small number of multi-card hands.

`Player.table`
: The *Table* used to place individual *Bet*s.

32.6.2 Constructors

`Player.__init__(self, table)`
: Constructs the *Player* with a specific *Table* for placing *Bet*s.

32.6.3 Methods

`Player.playing(self)` → boolean
: Returns `true` while the player is still active. There are two reasons why a player may be active. Generally, the player has a *stake* greater than the table minimum and has a *roundsToGo* greater than zero. Alternatively, the player has bets on the table; this will happen in craps when the game continues past the number of rounds budgeted.

`Player.placeBets(self)`
: Updates the `Table` with the various `Bet`s.

  When designing the `Table`, we decided that we needed to deduct the amount of a bet from the stake when the bet is created. See the Table *Roulette Table Analysis* for more information.

`Player.win(self, bet)`
: Notification from the `Game` that the `Bet` was a winner. The amount of money won is available via `bet.winAmount()`.

`Player.lose(self, bet)`
: Notification from the `Game` that the `Bet` was a loser.

32.7 Game Class Design

`Game` manages the sequence of actions that defines casino games, including Roulette, Craps and Blackjack. Individual subclasses implement the detailed playing cycles of the games. This superclass has methods for notifying the `Player` to place bets, getting a `RandomEvent` and resolving the `Bet` s actually present on the `Table`.

32.7.1 Fields

`Game.eventFactory`
: Contains a `Wheel` or `Dice` or other subclass of `RandomEventFactory` that returns a randomly selected `RandomEvent` with specific `Outcome` s that win or lose.

`Game.table`
: Contains a `CrapsTable` or `RouletteTable` which holds all the `Bet`s placed by the `Player`.

`Game.player`
: Holds the `Player` who places bets on the `Table`.

32.7.2 Constructors

We based this constructor on an design that allows any of these objects to be replaced. This is the **Strategy** (or **Dependency Injection**) design pattern. Each of these objects is a replaceable strategy, and can be changed by the client that uses this game.

Additionally, we specifically do not include the `Player` instance in the constructor. The `Game` exists independently of any particular `Player`, and we defer binding the `Player` and `Game` until we are gathering statistical samples.

`Game.__init__(self, eventFactory, table)`
: Constructs a new `Game`, using a given `RandomEventFactory` and `Table`.

32.7.3 Methods

`Game.cycle(self, player)`
: This will execute a single cycle of play with a given `Player`. For Roulette is is a single spin of the wheel. For Craps, it is a single throw of the dice, which is only one part of a complete game. This method will call `player.placeBets()` to get bets. It will call `eventFactory.next()` to get the next Set of `Outcome`s. It will then call `table.bets()` to get an `Iterator` over the `Bet`s. Stepping through this `Iterator` returns the individual `Bet` objects. The bets are resolved, calling the `thePlayer win()`, otherwise call the `thePlayer lose()`.

`Game.reset(self)`
As a useful default for all games, this will tell the table to clear all bets. A subclass can override this to reset the game state, also.

32.8 RouletteGame Class Design

`RouletteGame` is a subclass of *Game* that manages the sequence of actions that defines the game of Roulette.

32.8.1 Methods

`RouletteGame.cycle(self, player)`
This will execute a single cycle of the Roulette with a given *Player*. It will call `player.placeBets()` to get bets. It will call `wheel.next()` to get the next winning *Bin*. It will then call `table.bets()` to get an `Iterator` over the *Bet*s. Stepping through this `Iterator` returns the individual *Bet* objects. If the winning *Bin* contains the *Outcome*, call the `thePlayer win()`, otherwise call the `thePlayer lose()`.

32.9 CrapsGame Class Design

CrapsGame is a subclass of *Game* that manages the sequence of actions that defines the game of Craps.

Note that a single cycle of play is one throw of the dice, not a complete craps game. The state of the game may or may not change.

32.9.1 Methods

`RouletteGame.cycle(self, player)`
This will execute a single cycle of play with a given *Player*.

1. It will call `player.placeBets()` to get bets. It will validate the bets, both individually, based on the game state, and collectively to see that the table limits are met.
2. It will call `dice.next()` to get the next winning *Throw*.
3. It will use the `throw.updateGame()` to advance the game state.
4. It will then call `table.bets()` to get an `Iterator`; stepping through this `Iterator` returns the individual *Bet* objects.
    - It will use the *Throw*'s `resolveOneRoll()` method to check one-roll propositions. If the method returns true, the *Bet* is resolved and should be deleted.
    - It will use the *Throw*'s `resolveHardways()` method to check the hardways bets. If the method returns true, the *Bet* is resolved and should be deleted.

`CrapsGame.pointOutcome(self)` → Outcome
Returns the *Outcome* based on the current point. This is used to create Pass Line Odds or Don't Pass Odds bets. This delegates the real work to the current `CrapsGameState` object.

`CrapsGame.moveToThrow(self, bet, throw)`
Moves a Come Line or Don't Come Line bet to a new *Outcome* based on the current throw. This delegates the move to the current `CrapsGameState` object.

This method should – just as a precaution – assert that the value of `theThrow` is 4, 5, 6, 8, 9 or 10. These point values indicate that a Line bet can be moved. For other values of `theThrow`, this method should raise an exception, since there's no reason for attempting to move a line bet on anything but a point throw.

`CrapsGame.reset(self)`

This will reset the game by setting the state to a new instance of `GamePointOff`. It will also tell the table to clear all bets.

32.10 Refactoring Deliverables

There are six deliverables for this exercise.

- If necessary, create `RandomEvent`, and revisions to `Throw` and `Bin`. See *Design Heavy* .
- Create `RandomEventFactory`, and associated changes to `Wheel` and `Dice`. The existing unit tests will confirm that this change has no adverse effect.
- Refactor `Table` and `CrapsTable` to make a single class of these two. The unit tests for the original `CrapsTable` should be merged with the unit tests for the original `Table`.
- Refactor `Player` and `CrapsPlayer` to create a better class hierarchy with `CrapsPlayer` and `RoulettePlayer` both sibling subclasses of `Player` . The unit tests should confirm that this change has no adverse effect.
- Refactor `Game` and `CrapsGame` to create three classes: `Game`, `RouletteGame` and `CrapsGame`. The unit tests should confirm that this change has no adverse effect.
- Create a new main program class that uses the existing `Simulator` with the `CrapsGame` and `CrapsPlayer` classes.

CHAPTER

THIRTYTHREE

SIMPLE CRAPS PLAYERS

This chapter defines a variety of player strategies. Most of this is based on strategies already defined in *Roulette*, making the explanations considerably simpler. Rather than cover each individual design in separate chapters, we'll rely on the experience gained so far, and cover four variant Craps players in this chapter. We'll mention a fifth, but leave that as a more advanced exercise.

In *Simple Craps Players Analysis* we'll expand on existing definition of craps players. We'll add a number of betting strategies. In *Craps 1-3-2-6 Player*, *Craps Cancellation Player*, and *Craps Fibonacci Player* we'll look at different betting strategies and how those can be implemented for a craps player.

In *CrapsPlayer Design* we'll look at the general design for these simple players. We'll look at the superclass in *CrapsSimplePlayer superclass*. We'll look at each of the strategies in *Craps Martingale Player*, *Player1326 State*, *Craps1326 Player* and *CrapsCancellation Player*.

In *Simple Craps Players Deliverables* we'll detail the deliverables for this chapter.

33.1 Simple Craps Players Analysis

When we looked at our `Player` hierarchy, we noted that we could easily apply a number of strategies to the Pass Line Odds bet. We use the Martingale strategy for our `CrapsMartingale` player. We could also use the 1-3-2-6 system, the Cancellation system, or the Fibonacci system for those odds bets. In each of these cases, we are applying the betting strategy to one of the two bets the player will use.

An additional design note for this section is the choice of the two basic bets: the Pass Line and the Pass Line Odds bet. It is interesting to compare the results of these bets with the results of the Don't Pass Line and the Don't Pass Odds Bet. In particular, the Don't Pass Odds Bets involve betting large sums of money for small returns; this will be compounded by any of these betting systems which accelerate the amount of the bet to cover losses. This change should be a simple matter of changing the two base bets used by all these variant players.

All of these players have a base bet (either Pass Line or Don't Pass) and an odds bet (either Pass Line Odds or Don't Pass Odds). If we create a superclass, called `SimpleCraps` , we can assure that all these simple betting variations will work for Pass Line as well as Don't Pass Line bets. The responsibility of this superclass is to define a consistent set of fields and constructor for all of the subclasses.

33.1.1 Craps 1-3-2-6 Player

This player uses the 1-3-2-6 system for managing their odds bet. From the Roulette 1-3-2-6 player (see *Player 1-3-2-6 Analysis*) we can see that this player will need to use the `Player1326State` class hierarchy. The craps player will use one of these objects track the state changes of their odds. The base bet will not change.

However, the current definitions of the `Player1326State` class hierarchy specifically reference `Roulette1326`. Presenting us with an interesting design problem. How do we repair our design so that `Player1326State` can work with `Roulette1326` and `Craps1326`?

The only dependency is the field `outcome`, which both `Roulette1326` and `Craps1326` must provide to *Player1326State* objects.

Problem. Where do store the *Outcome* object used by both the player and the state?

Forces. We have three choices:

- **Create a Common Interface Class**. We extract the field and make it part of an interface.
- **Create a Common Superclass**. In this case, we refactor the field up to the superclass.
- **Delegate to The State Object**. This changes the defintion of *Player1326State* to make it more self-contained.

Common Interface Class.

The relationship between a subclass of *Player* and *Player1326State* can be formalized through an interface class definition. We define a class `Bet1326_Able`, which contains the *Outcome* and use this for `Roulette1326` and `Craps1326`.

```
class Bet1326_Able:
    def __init__(self):
        self.outcome= None

class CrapsPlayer(Player, Bet1326_Able):
    def __init__(self):
        super().__init__()
```

In this case, this appears to be an example of the **Very Large Hammer** design pattern. The problem seems too small for this language feature.

Common Superclass.

We can refactor the single instance variable up to the superclass. This is a relatively minor change. However, it places a feature in a superclass which all but a few subclasses must ignore. This is another example of **Swiss Army Knife** design, where we will be subtracting a feature from a superclass.

Delegate to the State Class.

If we change the *Player1326State* class to keep its own copy of the desired *Outcome* we cleanly remove any dependence on *Player*. The *Player* is still responsible for keeping track of the *Outcome*s, and has subcontracted or delegated this responsibility to an instance of *Player1326State*.

The down side of this is that we must provide this *Outcome* to each state constructor as the state changes.

Solution. The solution we embrace is changing the definition of *Player1326State* to include the *Outcome*. This delegates responsibility to the state, where it seems to belong. This will change all of the constructors, and all of the state change methods, but will cleanly separate the *Player1326State* class hierarchy from the *Player* class hierarchy.

Python Duck-Typing

In Python, the relationship between a *Player1326State* object and the `Craps1326` is completely casual. We don't have to sweat the details of where – precisely – the *Outcome* object is kept.
Python's flexibility is called *duck typing*:

"if it walks like a duck and quacks like a duck, it *is* a duck."

In this case, **any** class with an `outcome` attribute is a candidate owner for a *Player1326State* object.

33.1.2 Craps Cancellation Player

When we examine the Roulette Cancellation *Cancellation Player Analysis* we see that this player will need to use a List of individual betting amounts. Each win for an odds bet will cancel from this List, and each loss of an odds bet will append to this List.

As with the Craps Martingale player, we will be managing a base Pass Line bet, as well as an odds bet that uses the Cancellation strategy. The Cancellation algorithm can be easily transplanted from the original Roulette version to this new Craps version.

33.1.3 Craps Fibonacci Player

We can examine the Roulette Fibonacci *Fibonacci Player Analysis* and see that this player will need to compute new betting amounts based on wins and losses. THis will parallel the way the cancellation player works.

We'll can have a Fibanacci series for some bets (like pass line bets) but a flat bet for the behind the line odds bet.

33.2 CrapsPlayer Design

We'll extend `CrapsPlayer` to create a `CrapsSimplePlayer` that can place both Pass Line and Pass Line Odds bets, as well as Don't Pass Line and Don't Pass Odds bets. This will allow us to drop the `CrapsPlayerPass` class, and revise the existing `CrapsMartingale`.

We have to rework the original Roulette-focused *`Player1326State`* hierarchy, and the `Roulette1326` class to use the new version of the state objects.

Once this rework is complete, we can add our `Craps1326` and `CrapsCancellation` players.

For additional exposure, the more advanced student can rework the Roulette Fibonacci player to create a `CrapsFibonacci` player.

33.3 CrapsSimplePlayer superclass

`CrapsSimplePlayer` is a subclass of `CrapsPlayer` and places two bets in Craps. The simple player has a base bet and an odds bet. The base bet is one of the bets available on the come out roll (either Pass Line or Don't Pass Line), the odds bet is the corresponding odds bet (Pass Line Odds or Don't Pass Odds). This class implements the basic procedure for placing the line bet and the behind the line odds bet. However, the exact amount of the behind the line odds bet is left as an abstract method. This allows subclasses to use any of a variety of betting strategies, including Martingale, 1-3-2-6, Cancellation and Fibonacci.

33.3.1 Fields

`CrapsSimplePlayer.`**`lineOutcome`**

`Outcome` for either Pass Line or Don't Pass Line. A right bettor will use a Pass Line bet; a wrong bettor will use the Don't Pass Line.

`CrapsSimplePlayer.`**`oddsOutcome`**

`Outcome` for the matching odds bet. This is either the Pass Line Odds or Don't Pass Line Odds bet.

A right bettor will use a Pass Line Odds bet; a wrong bettor will use the Don't Pass Line Odds.

33.3.2 Constructors

`CrapsSimplePlayer.__init__(self, table, line, odds)`

Parameters

- **table** (*CrapsTable*) – The table on which bets are palced
- **line** (Outcome) – The line bet outcome
- **odds** (Outcome) – The odds bet outcome

Constructs the `CrapsSimplePlayer` with a specific *Table* for placing *Bet*s. Additionally a line bet (Pass Line or Don't Pass Line) and odds bet (Pass Line Odds or Don't Pass Odds) are provided to this constructor. This allows us to make either Pass Line or Don't Pass Line players.

33.3.3 Methods

`CrapsSimplePlayer.placeBets(self)`

Updates the *Table* with the various *Bet*s. There are two basic betting rules.

1. If there is no line bet, create the line *Bet* from the `line` *Outcome*.
2. If there is no odds bet, create the behind the line odds *Bet* from the `odds` *Outcome*.

Be sure to check the price of the *Bet* before placing it. Particularly, Don't Pass Odds bets may have a price that exceeds the player's stake. This means that the *Bet* object must be constructed, then the price must be tested against the `stake` to see if the player can even afford it. If the `stake` is greater than or equal to the price, subtract the price and place the bet. Otherwise, simply ignore it.

33.4 Craps Martingale Player

`CrapsMartingale` is a subclass of `CrapsSimplePlayer` who places bets in Craps. This player doubles their Pass Line Odds bet on every loss and resets their Pass Line Odds bet to a base amount on each win.

33.4.1 Fields

`CrapsMartingale.lossCount`

The number of losses. This is the number of times to double the pass line odds bet.

`CrapsMartingale.betMultiple`

The the bet multiplier, based on the number of losses. This starts at 1, and is reset to 1 on each win. It is doubled in each loss. This is always $betMultiple = 2^{lossCount}$.

33.4.2 Methods

`CrapsMartingale.placeBets(self)`

Extension to the superclass *placeBets()* method. This version sets the amount based on the value of *CrapsMartingale.betMultiple*.

`CrapsMartingale.win(self, bet)`

Parameters **bet** (Bet) – The bet that was a winner

Uses the superclass *win()* method to update the stake with an amount won. This method then resets *lossCount* to zero, and resets *betMultiple* to 1.

`CrapsMartingale.lose(self, bet)`

Parameters **bet** (Bet) – The bet that was a loser

Increments *lossCount* by 1 and doubles *betMultiple*.

33.5 Player1326 State

Player1326State is the superclass for all of the states in the 1-3-2-6 betting system.

33.5.1 Fields

Player1326State.**outcome**

The *Outcome* on which a *Player* will bet.

33.5.2 Constructors

Player1326State.**__init__**(*self*, *outcome*)

Parameters **outcome** (Outcome) – The outcome on which to bet

The constructor for this class saves *Outcome* on which a *Player* will bet.

33.5.3 Methods

Much of the original design for this state hierarchy should remain in place. See *Player 1-3-2-6 Class* for more information on the original design.

Player1326State.**nextLost**(*self*) → Player1326State

Constructs the new *Player1326State* instance to be used when the bet was a loser. This method is the same for each subclass: it creates a new instance of *Player1326NoWins*.

This method is defined in the superclass to assure that it is available for each subclass. This will use the *outcome* to be sure the new state has the *Outcome* on which the owning Player will be betting.

33.6 Craps1326 Player

Craps1326 is a subclass of CrapsSimplePlayer who places bets in Craps. This player changes their Pass Line Odds bet on every loss and resets their Pass Line Odds bet to a base amount on each win. The sequence of bet multipliers is given by the current *Player1326State* object.

33.6.1 Fields

Player1326State.**state**

This is the current state of the 1-3-2-6 betting system. It will be an instance of one of the four states: No Wins, One Win, Two Wins or Three Wins.

33.6.2 Constructors

`Player1326.__init__`(*self*, *table*, *line*, *odds*)

Parameters

- **table** (*CrapsTable*) – The table on which bets are palced
- **line** (Outcome) – The line bet outcome
- **odds** (Outcome) – The odds bet outcome

Uses the superclass to initialize the `Craps1326` instance with a specific *Table* for placing *Bet*s, and set the line bet (Pass Line or Don't Pass Line) and odds bet (Pass Line Odds or Don't Pass Odds).

Then the initial state of *Player1326NoWins* is constructed using the odds bet.

33.6.3 Methods

`Player1326.placeBets`(*self*)

Updates the *Table* with a bet created by the current state. This method delegates the bet creation to *state* object's `currentBet()` method.

`Player1326.win`(*self*, *bet*)

Parameters **bet** (Bet) – The bet that was a winner

Uses the superclass method to update the stake with an amount won. Uses the current state to determine what the next state will be by calling *state*'s objects `nextWon()` method and saving the new state in *state*

`Player1326.lose`(*self*, *bet*)

Parameters **bet** (Bet) – The bet that was a loser

Uses the current state to determine what the next state will be. This method delegates the next state decision to *state* object's `nextLost()` method, saving the result in *state*.

33.7 CrapsCancellation Player

`CrapsCancellation` is a subclass of `CrapsSimplePlayer` who places bets in Craps. This player changes their Pass Line Odds bet on every win and loss using a budget to which losses are appended and winings are cancelled.

33.7.1 Fields

`CrapsCancellation.sequence`

This `List` keeps the bet amounts; wins are removed from this list and losses are appended to this list. THe current bet is the first value plus the last value.

33.7.2 Constructors

`CrapsCancellation.__init__`(*self*, *table*, *line odds*)

Parameters

- **table** (*CrapsTable*) – The table on which bets are palced
- **line** (Outcome) – The line bet outcome
- **odds** (Outcome) – The odds bet outcome

Invokes the superclass constructor to initialize this instance of `CrapsCancellation`. Then calls `resetSequence()` to create the betting budget.

33.7.3 Methods

There are few real changes to the original implementation of CancellationPlayer.

See *Cancellation Player Class* for more information.

`CrapsCancellation.placeBets(self)`

Creates a bet from the sum of the first and last values of *`sequence`* and the preferred outcome.

This uses the essential line bet and odds bet algorithm defined above. If no line bet, this is created.

If there's a line bet and no odds bet, then the odds bet is created.

If both bets are created, there is no more betting to do.

33.8 Simple Craps Players Deliverables

There are eight deliverables for this exercise.

- The `CrapsSimplePlayer` abstract superclass. Since this class doesn't have a body for the `oddsBet()` method, it can't be unit tested directly.
- A revised `CrapsMartingale` class, that is a proper subclass of `CrapsSimplePlayer`. The existing unit test for `CrapsMartingale` should continue to work correctly after these changes.
- A revised *`Player1326State`* class hierarchy. Each subclass will use the `outcome` field instead of getting this information from a *`Player1326`* instance. The unit tests will have to be revised slightly to reflect the changed constructors for this class.
- A revised `Roulette1326` class, which reflects the changed constructors for this *`Player1326State`*. The unit tests should indicate that this change has no adverse effect.
- The `Craps1326` subclass of `CrapsSimplePlayer`. This will use the revised *`Player1326State`*.
- A unit test class for `Craps1326`. This test should synthesize a fixed list of *`Outcome`*s, *`Throw`*s, and calls a `Craps1326` instance with various sequences of craps, naturals and points to assure that the bet changes appropriately.
- The `CrapsCancellation` subclass of `CrapsSimplePlayer`.
- A unit test class for `CrapsCancellation`. This test should synthesize a fixed list of *`Outcome`*s, *`Throw`*s, and calls a `CrapsCancellation` instance with various sequences of craps, naturals and points to assure that the bet changes appropriately.

CHAPTER

THIRTYFOUR

ROLL-COUNTING PLAYER CLASS

A common Craps strategy is to add bets as a kind of "insurance" against losing the line bet. This means that we'll have numerous working bets: the mandatory line bet, the behind the line odds bet, plus an additional bets. For example, buying the 6 or 8 is a bet that has a payout that matches the actual odds.

We'll tackle a particularly complex betting strategy. In this case, a player that judges that a game has gone "too long" without a successful resolution. This is a common fallacy in probability theory. A seven is not "due". The odds of throwing a seven are always 1/6.

In order to handle this, we'll need to have a larger number of independent bets, with independent betting strategies. The previous design will have to be expanded to allow for this.

In *Roll-Counting Analysis* we'll examine the essential betting strategy. This will have large implications. We'll look at them in *Decomposing the Player* and *Implementing SevenCounter*.

This will lead to a round of redesigning a number of classes. In *BettingStrategy Design* we'll disentangle the game-based betting from the various betting strategies that dictate amounts.

We can then implement each betting strategy in a way that's separate from each player. We'll look at the details in:

- *NoChangeBetting Class*,
- *MartingaleBetting Class*, and
- *Bet1326Betting Class*.

Once we've separated betting strategies from game playing strategies, we can then create a number of more advanced players. These include

- *CrapsOneBetPlayer class*,
- *CrapsTwoBetPlayer class*, and
- *CrapsSevenCountPlayer class*.

We'll enumerate the deliverables in *Roll-Counting Deliverables*.

34.1 Roll-Counting Analysis

There is a distinction between one-roll odds and cumulative odds. The one roll odds of rolling a 7 are $1/6$. This means that a Pass Line bet will win one time in six on the come out roll. The cumulative odds of rolling a 7 on a number of rolls depends on not rolling a seven (a $5/6$ chance) for some number of rolls, followed by rolling a 7. The odds are given in the following table.

Throws	Rule	Odds of 7
1	$1/6$	17%
2	$5/6 \times 1/6$	31%
3	$(5/6)^2 \times 1/6$	42%
4	$(5/6)^3 \times 1/6$	52%
5	$(5/6)^4 \times 1/6$	60%
6	$(5/6)^5 \times 1/6$	67%

This cumulative chance of rolling a 7 means that the odds of the game ending with a loss because of throwing a 7 grow as the game progresses.

The idea is that the longer a game runs, the more likely you are to lose your initial Pass Line bet. Consequently, some players count the thows in the game, and effectively cancel their bet by betting against themselves on the Seven proposition.

Important: Bad Odds

Note that the Seven proposition is a 1/6 probability that pays "5 for 1", (effectively 4:1).

While the basic probability analysis of this bet is not encouraging, it does have an interesting design problem: the player now has multiple conccurrently changing states:

- They have Pass Line bet,
- they can use a Martingale strategy for their Pass Line Odds bet,
- they are counting throws, and using a Martingale strategy for a Seven proposition starting with the seventh throw of the game.

Either the class will become quite complex. Or we'll have to decompose this class into a collection of simpler objects.

Wrong Bettors

The simple counting doesn't work for wrong bettors – those using the Don't Pass Line bet. Their concern is the opposite: a short game may cause them to lose their Don't Pass bet, but a long game makes it more likely that they would win.

34.1.1 Decomposing the Player

This leads us to consider the `Player` as a composite object with a number of states and strategies. It also leads us to design a separate class just to handle Martingale betting.

Note that when we were looking at the design for the various players in *Design Cleanup and Refactoring*, we glanced at the possibility of separating the individual betting strategies from the players, and opted not to. However, we did force each strategy to depend on a narrowly-defined interface of `oddsBet()`, `won()` and `lost()`. We can exploit this narrow interface in teasing apart the various strategies and rebuilding each variation of `Player` with a distinct betting strategy object.

The separation of `Player` from `BettingStrategy` involves taking the betting-specific information out of each `Player` subclass, and replacing the various methods and fields with one or more `BettingStrategy` objects.

In the case of Roulette players, this is relatively simple. We generally use just one bet with a variety of strategies.

In the case of Craps players, we often have two bets, one with a trivial-case betting strategy where the bet never changes. We'll need a special `NoChange` strategy for the Pass Line. We'll need a Martigale (or 1-3-2-6, Cancellation, or Fibonacci) for the Odds. We can then redefine all Craps player's bets using `BettingStrategy` objects.

The responsibilities of a `BettingStrategy` are to

- maintain a preferred `Outcome`,

- maintain a bet amount, and change that amount in response to wins and losses.

The existing `win()` and `lose()` methods are a significant portion of these reponsibilities. The `oddsBet()` method of the various `CrapsSimplePlayer` embodies other parts of this, however, the name is inappropriate and it has a poorly thought-out dependency on *`Player`*.

The responsibilities of a *`Player`* are to

- keep one or more betting strategies, so as to place bets in a *`Game`*.

All of the Roulette players will construct a single `BettingStrategy` object with their preferred *`Outcome`*. The various `CrapsSimplePlayer` classes will have two `BettingStrategys`: one for the line bet and one for the odds bet.

The only difference among the simple strategies is the actual `BettingStrategy` object, simplifying the *`Player`* class hierarchy to a single Roulette player and two kinds of Craps players: the stub player who makes only one bet and the other players who make more than one bet and use a betting strategy for their odds bet.

34.1.2 Implementing SevenCounter

Once we have this design in place, our `SevenCounter` player can then be composed of three betting strategies:

- a Pass Line bet that uses the `NoChange` strategy,
- a Pass Line Odds bet, and
- a Seven proposition bet that will only be used after seven rolls have passed in a single game.

The other two bets can use any of the strategies we have built: Martingale, 1-3-2-6, Cancellation, or Fibonacci.

Currently, there is no method to formally notify the `CrapsPlayer` of unresolved bets. The player is only told of winners and losers.

The opportunity to place bets indicates that the dice are being rolled. Additionally, the ability to place a line bet indicates that a game is beginning. We can use these two methods to count the throws in during a game, and reset the counter at the start of a game, effectively counting unresolved bets.

34.2 BettingStrategy Design

`BettingStrategy` is an abstract superclass for all betting strategies. It contains a single *`Outcome`*, tracks wins and losses of *`Bet`*s built on this *`Outcome`*, and computes a bet amount based on a specific betting strategy.

34.2.1 Fields

`BettingStrategy.`**`outcome`**

This is the *`Outcome`* that will be watched for wins and losses, as well as used to create new *`Bet`*s.

34.2.2 Constructors

`BettingStrategy.`**`__init__`**`(`*self*, *outcome*`)`

Parameters **`outcome`** (Outcome) – The outcome on which this strategy will create bets

Initializes this betting strategy with the given *`Outcome`*.

34.2.3 Methods

`BettingStrategy.createBet(self)` → Bet
Returns a new *Bet* using the *outcome Outcome* and any other internal state of this object.

`BettingStrategy.win(self, bet)`

Parameters bet (Bet) – The bet which was a winner

Notification from the *Player* that the *Bet* was a winner. The *Player* has responsibility for handling money, this class has responsibility for tracking bet changes.

`BettingStrategy.lose(self, bet)`

Parameters bet (Bet) – The bet which was a loser

Notification from the *Player* that the *Bet* was a loser.

`BettingStrategy.__str__(self)` → str
Returns a string with the name of the class and appropriate current state information. For the superclass, it simply returns the name of the class. Subclasses will override this to provide subclass-specific information.

34.3 NoChangeBetting Class

`NoChangeBetting` is a subclass of `BettingStrategy` that uses a single, fixed amount for the bet. This is useful for unit testing, for modeling simple-minded players, and for line bets in Craps.

34.3.1 Fields

`BettingStrategy.betAmount`
This is the amount that will be bet each time. A useful default value is 1.

34.3.2 Constructors

`NoChangeBetting.__init__(self, outcome)`

Parameters outcome (Outcome) – The outcome on which this strategy will create bets

Uses the superclass initializer with the given *Outcome*.

34.3.3 Methods

`NoChangeBetting.createBet(self)` → Bet
Returns a new *Bet* using the `outcome` *Outcome* and `betAmount`.

`NoChangeBetting.win(self, bet)`

Parameters bet (Bet) – The bet which was a winner

Since the bet doesn't change, this does nothing.

`NoChangeBetting.lose(self, bet)`

Parameters bet (Bet) – The bet which was a loser

Since the bet doesn't change, this does nothing.

`NoChangeBetting.__str__(self)` → str
Returns a string with the name of the class, `outcome` and `betAmount`.

34.4 MartingaleBetting Class

`MartingaleBetting` is a subclass of `BettingStrategy` that doubles the bet on each loss, hoping to recover the entire loss on a single win.

34.4.1 Fields

`MartingaleBetting.`**`lossCount`**

The number of losses. This is the number of times to double the pass line odds bet.

`MartingaleBetting.`**`betMultiple`**

The the bet multiplier, based on the number of losses. This starts at 1, and is reset to 1 on each win. It is doubled in each loss. This is always $betMultiple = 2^{lossCount}$.

34.4.2 Constructors

`MartingaleBetting.`**`__init__`**`(`*self*, *outcome*`)`

Parameters **`outcome`** (Outcome) – The outcome on which this strategy will create bets

Uses the superclass initializer with the given *`Outcome`*. Sets the initial lossCount and betMultiplier.

34.4.3 Methods

`MartingaleBetting.`**`createBet`**`(`*self*`)` → Bet

Returns a new *`Bet`* using the `outcome` *`Outcome`* and the *`betMultiple`*.

`MartingaleBetting.`**`win`**`(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – The bet which was a winner

Resets *`lossCount`* to zero, and resets *`betMultiple`* to `1`.

`MartingaleBetting.`**`lose`**`(`*self*, *bet*`)`

Parameters **`bet`** (Bet) – The bet which was a loser

Increments *`lossCount`* by `1` and doubles *`betMultiple`*.

`NoChangeBetting.`**`__str__`**`(`*self*`)` → str

Returns a string with the name of the class, `outcome`, the current `betAmount` and `betMultiple`.

34.5 Bet1326Betting Class

`Bet1326Betting` is a subclass of `BettingStrategy` that advances the bet amount through a sequence of multipliers on each win, and resets the sequence on each loss. The hope is to magnify the gain on a sequence of wins.

34.5.1 Fields

`Bet1326Betting.`**`state`**

This is the current state of the 1-3-2-6 betting system. It will be an instance of one of the four subclasses of *`Player1326State`*: No Wins, One Win, Two Wins or Three Wins.

34.5.2 Constructors

`Bet1326Betting.__init__`(*self*, *outcome*)

Parameters **outcome** (Outcome) – The outcome on which this strategy will create bets

Initializes this betting strategy with the given *Outcome*. Creates an initial instance of *Player1326NoWins* using `outcome`.

34.5.3 Methods

`Bet1326Betting.createBet`(*self*) → Bet
Returns a new *Bet* using the `currentBet()` method from the *state* object.

`Bet1326Betting.win`(*self*, *bet*)

Parameters **bet** (Bet) – The bet which was a winner

Determines the next state when the bet is a winner. Uses *state*'s `nextWon()` method and saves the new state in *state*.

`Bet1326Betting.lose`(*self*, *bet*)

Parameters **bet** (Bet) – The bet which was a loser

Determines the next state when the bet is a loser. Uses *state*'s `nextLost()`, method saving the result in `myState`.

`Bet1326Betting.__str__`(*self*) → str
Returns a string with the name of the class, `outcome` and *state*.

34.6 CrapsOneBetPlayer class

`CrapsOneBetPlayer` is a subclass of `CrapsPlayer` and places one bet in Craps. The single bet is one of the bets available on the come out roll (either Pass Line or Don't Pass Line). This class implements the basic procedure for placing the line bet, using an instance of `BettingStrategy` to adjust that bet based on wins and losses.

34.6.1 Fields

`CrapsOneBetPlayer.lineStrategy`
An instance of `BettingStrategy` that applies to the line bet.

Generally, this is an instance of `NoChangeBetting` because we want to make the minimum line bet and the maximum odds bet behind the line.

34.6.2 Constructors

`CrapsOneBetPlayer.__init__`(*self*, *table*, *lineStrategy*)
Constructs the `CrapsOneBetPlayer` with a specific *Table* for placing *Bet* s. This will save the given `BettingStrategy` in *lineStrategy*.

Creation of A Player

```
passLine= table.dice.get( "Pass Line" )
bet= new MartingaleBetting( passLine )
passLineMartin= new CrapsOneBetPlayer( bet );
```

1. Get the basic Pass Line `Outcome` from the Dice.
2. Creates a Martingale betting strategy focused on the basic Pass Line outcome.
3. Creates a one-bet player, who will employ the Martingale betting strategy focused on the basic Pass Line outcome.

34.6.3 Methods

`CrapsOneBetPlayer.placeBets(self)`

Updates the `Table` with the various `Bet`s. There is one basic betting rule.

1. If there is no line bet, create the line `Bet` from the `lineStrategy`.

Be sure to check the price of the `Bet` before placing it. Particularly, Don't Pass Odds bets may have a price that exceeds the player's stake. This means that the `Bet` object must be constructed, then the price must be tested against the `stake` to see if the player can even afford it. If the `stake` is greater than or equal to the price, subtract the price and place the bet. Otherwise, simply ignore it.

`CrapsOneBetPlayer.win(self, bet)`

Parameters **bet** (Bet) – The bet which was a winner

Notification from the `Game` that the `Bet` was a winner. The amount of money won is available via `theBet winAmount()`. If the bet's `Outcome` matches the `lineStrategy` 's `Outcome`, notify the strategy, by calling the `lineStrategy` 's `win()` method.

`CrapsOneBetPlayer.lose(self, bet)`

Parameters **bet** (Bet) – The bet which was a loser

Notification from the `Game` that the `Bet` was a loser. If the bet's `Outcome` matches the `lineStrategy` 's `Outcome`, notify the strategy, by calling the `lineStrategy` 's `lose()` method.

34.7 CrapsTwoBetPlayer class

`CrapsTwoBetPlayer` is a subclass of `CrapsOneBetPlayer` and places one or two bets in Craps. The base bet is one of the bets available on the come out roll (either Pass Line or Don't Pass Line). In addition to that, an odds bet (either Pass Line Odds or Don't Pass Odds) can also be placed. This class implements the basic procedure for placing the line and odds bets, using two instances of `BettingStrategy` to adjust the bets based on wins and losses.

Typically, the line bet uses an instance of `NoChangeBetting`.

The odds bets, however, are where we want to put more money in play.

34.7.1 Fields

`CrapsTwoBetPlayer.oddsStrategy`

An instance of `BettingStrategy` that applies to the line bet.

34.7.2 Constructors

34.7.3 Methods

`CrapsTwoBetPlayer.placeBets(self)`

Updates the *Table* with the various *Bet* s. There are two basic betting rules.

1. If there is no line bet, create the line *Bet* from the `lineStrategy`.
2. If there is no odds bet, create the odds *Bet* from the *oddsStrategy*.

`CrapsTwoBetPlayer.win(self, bet)`

Parameters bet (Bet) – The bet which was a winner

Notification from the *Game* that the *Bet* was a winner. The superclass handles the money won and the line bet notification. This subclass adds a comparison between the bet's *Outcome* and the *oddsStrategy*'s *Outcome*; if they match, it will notify the strategy, by calling the *oddsStrategy*'s *win()* method.

`CrapsTwoBetPlayer.lose(self, bet)`

Parameters bet (Bet) – The bet which was a loser

Notification from the *Game* that the *Bet* was a loser. The superclass handles the line bet notification. If the bet's *Outcome* matches the *oddsStrategy*'s *Outcome*, notify the strategy, by calling the *oddsStrategy*'s *lose()* method.

34.8 CrapsSevenCountPlayer class

`CrapsSevenCountPlayer` is a subclass of `CrapsTwoBetPlayer` and places up to three bets in Craps. The base bet is a Pass Line bet. In addition to that, a Pass Line Odds bet can also be placed. If the game runs to more than seven throws, then the "7" proposition bet (at 4:1) is placed, using the Martingale strategy.

The Pass Line bet uses an instance of `NoChangeBetting`. The Pass Line Odds bet uses an instance of `Bet1326Betting`.

34.8.1 Fields

`CrapsSevenCountPlayer.sevenStrategy`

The `BettingStrategy` for the seven bet. Some argue that this should be a no-change strategy. The bet is rare, and – if effect – the player bets against themself with this. One could also argue that it should be a Martingale because each throw after the seventh are less and less likely to win.

`CrapsSevenCountPlayer.throwCount`

The number of throws in this game. This is set to zero when we place a line bet, and incremented each time we are allowed to place bets.

34.8.2 Constructors

`CrapsSevenCountPlayer.__init__(self, table)`

This will create a `NoChangeBetting` strategy based on the Pass Line *Outcome*. It will also create a `MartingaleBetting` strategy based on the Pass Line Odds *Outcome*. These will be given to the superclass constructor to save the game, the line bet and the odds bet. Then this constructor creates a `Bet1326Betting` strategy for the Seven Proposition *Outcome*.

34.8.3 Methods

`CrapsSevenCountPlayer.placeBets(self)`

Updates the *Table* with the various *Bet*s. There are three basic betting rules.

1. If there is no line bet, create the line *Bet* from the `lineStrategy`. Set the *throwCount* to zero.
2. If there is no odds bet, create the odds *Bet* from the `oddsStrategy`.
3. If the game is over seven throws and there is no seven proposition bet, create the proposition *Bet* from the *sevenStrategy*.

Each opportunity to place bets will also increment the *throwCount* by one.

`CrapsSevenCountPlayer.win(self, bet)`

Parameters **bet** (Bet) – The bet which was a winner

Notification from the *Game* that the *Bet* was a winner. The superclass handles the money won and the line and odds bet notification.

`CrapsSevenCountPlayer.lose(self, bet)`

Parameters **bet** (Bet) – The bet which was a loser

Notification from the *Game* that the *Bet* was a loser. The superclass handles the line and odds bet notification.

34.9 Roll-Counting Deliverables

There are two groups of deliverables for this exercise. The first batch of deliverables are the new Betting Strategy class hierarchy and unit tests. The second batch of deliverables are the two revised Craps Player classes, the final Roll Counter Player, and the respective unit tests.

Also, note that these new classes make the previous `CrapsSimplePlayer`, `CrapsMartingale`, `Craps1326` and `CrapsCancellation` classes obsolete. There are two choices for how to deal with this change: remove and reimplement. The old calsses can be removed, and the Simulator reworked to use the new versions. The alternative is to reimplement the original classes as *Facade* over the new classes.

Betting Strategy class hierarchy. There are four classes, with associated unit tests in this group of deliverables.

- The `BettingStrategy` superclass. This class is abstract; there is no unit test.
- The `NoChangeBetting` class.
- A unit test for the `NoChangeBetting` class. This will simply confirm that the `win()` and `lose()` methods do not change the bet amount.
- The `MartingaleBetting` class.
- A unit test for the `MartingaleBetting` class. This will confirm that the `win()` method resets the bet amount and `lose()` method doubles the bet amount.
- The `Bet1326Betting` class.
- A unit test for the `Bet1326Betting` class. This will confirm that the `win()` method steps through the various states, and the `lose()` method resets the state.

CrapsPlayer class hierarchy. There are three classes, each with an associated unit test in this group of deliverables.

- The `CrapsOneBetPlayer` class.
- A unit test for the `CrapsOneBetPlayer` class. One test can provide a No Change strategy for a Pass Line bet to verify that the player correctly places Line bets. Another test can provide a Martingale strategy for a Pass Line bet to verify that the player correctly changes bets on wins and losses.
- The `CrapsTwoBetPlayer` class.

- A unit test for the `CrapsTwoBetPlayer` class. One test can provide a No Change strategy for a Pass Line bet and a Martingale strategy for a Pass Line Odds bet to verify that the player correctly places Line bets and correctly changes bets on wins and losses.
- The `CrapsSevenCountPlayer` class.
- A unit test for the `CrapsSevenCountPlayer` class. This will require a lengthy test procedure to assure that the player correctly places a Seven proposition bet when the game is over seven throws long.

CHAPTER

THIRTYFIVE

CONCLUSION

The game of Craps has given us an opportunity to extend and modify an application with a considerable number of classes and objects. It is large, but not overly complex, and produces interesting results. Further, as a maintenance and enhancement exercise, it gave us an opportunity to work from a base of software, extending and refining the quality of the design.

We omitted exercises which would integrate this package with the `Simulator` and collect statistics. This step, while necessary, doesn't include many interesting design decisions. The final deliverable should be a working application that parses command-line parameters, creates the required objects, and creates an instance of `Simulator` to collect data.

Refactoring. We note that many design decisions required us to explore and refactor a great deal of the application's design. In writing this part, we found it very difficult to stick to our purpose of building up the design using realistic steps and realistic problem solving.

Our goal is explicitly **not** to be a book with a description of an already-completed structure: this does not help new designers learn the *process* of design, and does not help people to identify design problems and correct them. This leads to the complete refactoring of the design in the *Design Cleanup and Refactoring* chapter as an important activity. It it is the kind of thing that distinguishes a good design from a haphazard design.

We observe this kind of design rework happening late in the life of a project. Project managers are often uncomfortable evaluating the cost and benefit of the change. Further, programmers are unable to express the cost of accumulating *technical debt* by making a series of less-than-optimal decisions.

Simpler is Better. Perhaps, the most important lesson that we have is the constant search for something we call *The Big Simple*. We see the history of science as a search for simpler explanations of natural phenomena. The canonical example of this is the distinction between the geocentric model and the heliocentric model of the solar system. Both can be made to work: astronomers carefully built extremely elaborate models of the geocentric heavens and produced accurate predictions of planetary positions. However, the model of planetary motion around the sun describes real phenomena more accurately and has the added benefit of being much simpler than competing models.

To continue this rant, we find that software designers and their managers do not feel the compulsion and do not budget the time to identify the grand simplification that is possible. In some cases, the result of simplifying the design on one axis will create more classes. Designers lack a handy metric for "understandability"; managers are able to count individual classes, no matter how transparently simple. Designers often face difficulties in defending a design with many simple classes; some people feel that a few complex classes is "simpler" because it has fewer classes.

As our trump card, we reference the metrics for complexity:

- McCabe's Cyclomatic Complexity penalizes if-statements. By reducing the number of if-statements to just those required to create an object of the proper class, we reduce the complexity.
- The Halstead metrics penalize programs with many internal operators and operands when compared with few interface operands. Halstead emphasizes simple, transparent classes.

Neither measure penalizes overall size in lines of code, but rather they penalize decision-making and hidden, internal state. A badly designed, complex class has hidden internal states, often buried in nested if-statements. We emphasize small, simple classes with no hidden features.

Our concrete examples of this simplification process are contained in three large design exercises. In *Throw Builder Class*, we showed a kind of rework necessary to both generalize and isolate the special processing for Craps. In *Design Cleanup and Refactoring*, we reworked classes to create an easy-to-explain architecture with layers and partitions of responsibility. Finally, in *Roll-Counting Player Class*, we uncovered a clean separation between game rules and betting strategies.

Part IV

Blackjack

This part describes the more complex game of Blackjack. Both the player and the dealer are trying to build a hand that totals 21 points without going over. The hand closest to 21 wins.

This game has a number of states and a number of complex state-change rules. It has very few different kinds of bets, but moderately complex rules for game play. However, it does have the most sophisticated playing strategy, since the player has a number of choices to make.

The chapters of this part presents the details on the game, an overview of the solution, and a series of six relatively complex exercises to build a complete simulation of the game. In the case of Blackjack, we have to create a design that allows for considerable variation in the rules of the game as well as variation in the player's betting strategies.

CHAPTER

THIRTYSIX

BLACKJACK DETAILS

In *Blackjack Game*, we'll present elements of the game of Blackjack. Blackjack uses cards and has fairly complex rules for counting the number of points in a hard of cards.

Blackjack offers relatively few bets, most of which are available based on the state of the game. We'll cover these bets and the conditions under which they are allowed in *Available Bets and Choices*.

Finally, we will describe some common betting and playing strategies that we will simulate. in *Betting Strategies*. In this case, we have playing strategies that are unique to Blackjack, combined with betting strategies initially defined in *Roulette* and reworked in *Craps*.

36.1 Blackjack Game

Blackjack centers around *hands* composed of *cards* drawn from one or more standard 52-card *decks*. The standard deck has 13 *ranks* in 4 *suits*; the suit information has no bearing on game play. The player and the house are both dealt hands, starting with two cards. The house has one card exposed (the *up card*) and one card concealed (the *hole card*), leaving the player with incomplete information about the state of the game. The player's objective is to make a hand that has more points than the dealer, but less than or equal to 21 points. The player is responsible for placing bets when they are offered, and taking additional cards to complete their hand. The dealer will draw additional cards according to a simple rule: when the dealer's hand is 16 or less, they will draw cards (or *hit*), when it is 17 or more, they will not draw additional cards (or *stand pat*).

An interesting complication is the point values of the cards. The number cards (2-10) have the expected point values. The face cards (Jack, Queen and King) all have a value of 10 points. The Ace can count as one point or eleven points. Because of this, an Ace and a 10 or face card totals 21. This two-card winner is called "blackjack". Also, when the points include an ace counting as 11, the total is called *soft* ; when the ace counts as 1, the total is called *hard*. For example, A-5 is called a soft 16 because it could be considered a hard 6. A-10-5 is a hard 16.

The betting surface is marked with two places for bets: a single bet, placed before any cards are dealt, and an insurance bet, offered only when the dealer's up card is an ace. There are a few additional bets, and a few player choices. We'll step through some variations on the sequence of play to see the interactions a player has during a game. Note that a casino table seats a number of players; like Craps and Roulette, the player opposes the house, and the presence or absence of other players has no bearing on the game.

Note: Rule Variations

There are seemingly endless variations in the exact playing rules used by different casinos. We'll focus on a relatively common version of the rules. With this as a basis, a number of variations can be explored.

Typical Scenario. The player places an initial bet. Since the bet is "blind", it is like an ante in poker. The player and dealer are each dealt a pair of cards. Both of the player's are face up, the dealer has one card up and one card down. If the dealer's card is an ace, the player is offered insurance. The details will be described in a separate sceanario, below.

Initially, the player has a number of choices.

- If the two cards are the same rank, the player can elect to split into two hands. This is a separate scenario, below.
- The player can double their bet and take just one more card. In some casinos this opportunity may be limited to certain totals on the cards, for instance, only 10 and 11; or it may be limited to a certain number of cards, for instance, two cards.
- The more typical scenario is for the player to take additional cards (a *hit*) until either their hand totals more than 21 (they *bust*), or their hand totals exactly 21, or they elect to *stand*.

If the player's hand is over 21, their bet is resolved immediately as a loss. Resolving these bets early is an important part of the house's edge in Blackjack. If the player's hand is 21 or less, however, it will be compared to the dealer's hand for resolution.

The dealer then reveals the hole card and begins taking cards according to their fixed rule. When their total is 16 or less, they take an additional card; if their total is 17 or more, they stand pat. This rule is summarized as "hit on 16, stand on 17". In some casinos a dealer will hit a soft 17 (A-6), which improves the house's edge slightly.

If the dealer busts, the player wins. If the dealer did not bust, then the hands are compared: if the player's total is more than the dealer, the player wins; if the totals are equal, the bet is a push; otherwise the dealer's total is more than the play and the player loses.

If the player's hand is an ace and a 10-point card (10, Jack, Queen or King), the hand is blackjack and the the ante is paid off at 3:2. Otherwise, winning hands that are not blackjack are paid off at 1:1.

Dealer Shows An Ace. If the dealer's up card is an ace, the player is offered an insurance bet. This is an additional proposition that pays 2:1 if the dealer's hand is exactly 21 (a 4/13 probability). The amount of the bet is half the original ante. If this insurance bet wins, it will, in effect, cancel the loss of the ante. After offering insurance to the player, the dealer will check their hole card and resolve the insurance bets. If the hole card is 10-point card, the dealer has blackjack, the card is revealed, and insurance bets are paid. If the hole card is not a 10-point card, the insurance bets are lost, but the card is not revealed.

In the unusual case that the dealer shows an ace and the player shows blackjack (21 in two cards), the player will be offered "even money" instead of the insurance bet. If the player accepts the even money offer, their hand is resolved at 1:1 immediately, without examining the dealer's hole card or playing out the hand. If the player declines even money, they can still bet or decline insurance. Checking the odds carefully, there is a 4/13 (30.7%) chance of the dealer having 21, but insurance is paid as if the odds were 1/3 (33.3%). Since the player knows they have 21, there is a 4/13 probability of a push plus winning the insurance bet (both player and dealer have 21) and a 9/13 probability of winning at 3:2, but losing the insurance bet (effectively a push).

Split Hands. When dealt two cards of the same rank, the player can split the cards to create two hands. This requires an additional bet on the new hand. The dealer will deal an additional card to each new hand, and the hands are played independently. Generally, the typical scenario described above applies to each of these hands. The general rule of thumb is to always split aces and eights.

The ideal situation is to split aces, and get dealt a 10-point card on each ace. Both hands pay 3:2. A more common situation is to have a low card (from 2 to 7) paired up with the ace, leading to soft 13 through soft 18. Depending on the dealer's up card, these are opportunities to double down, possibly increasing the bet to 4 times the original amount.

Some casinos restrict doubling down on the split hands. In rare cases, one or more of the new cards will match the original pair, possibly allowing further splits. Some casinos restrict this, only allowing a single split. Other casinos prevent resplitting only in the case of aces.

Note that the player's election to split hands is given after any offer and resolution of insurance bets.

36.2 Available Bets and Choices

Unlike Roulette and Craps, Blackjack has only a few available bets. Generally, the following choices all involve accepting an offer by creating an additional bet.

- **Ante**. This bet is mandatory to play. It must be within the table limits.

- **Insurance**. This bet is offered only when the the dealer shows an ace. The amount must be half the ante. Note that the even money offer is an option for resolution of the ante instead of an insurance bet. It is sometimes described as a separate kind of bet, but this doesn't seem accurate.
- **Split**. This can be thought of as a bet that is offered only when the the player's hand has two cards are of equal rank. Or this can be thought of as a playing option, akin to hitting or standing. The amount of the bet must match the original ante.
- **Double**. This can be thought of as a bet that is offered instead of a taking an ordinary hit. Some casions only offer this when the the player's hand has two cards. Or this can be thought of as a playing option, akin to hitting or standing. The amount of the bet must match the original ante.

Blackjack also offers the player some choices that don't involve creating additional bets. In the casino these are shown through gestures that can be seen clearly by dealers and other casino staff.

- **Even Money**. This resolution of insurance is offered only when the the dealer shows an ace and the player shows 21 in two cards. It is offered instead of an insurance bet. If accepted, the hand is resolved. If declined, the insurance offer can then be accepted or declined.
- **Hit**. The player is offered the opporutunity to take another card when their total is less than 21. In this case, they elect to take that card. The hand may reach 21, or go over 21 and bust, or remain below 21.
- **Stand**. The player is offered the opporutunity to take another card when their total is less than 21. If they decline the hit, they are standing pat. Their hand remains under 21. The game transitions to the dealer taking cards and resolving the bets.

Play begins with a sequence of offers which can be accepted or declined: insurance, even money resolution of insurance, and splitting a hand. After these offers, the player must select between the three remaining choices (hit, double or stand) for each of their hands with a total less than 21.

In Roulette, there are no additional offers for the player to accept or decline.

In Craps, players may be offered opportunities to activate or deactive point odds bets; we ignored this. Adding this interaction to Craps would require defining an additional method for `CrapsPlayer` to accept or decline an offer. We would also have the *`CrapsGame`* interrogate the *`Table`* for the presence of come point odds bets, make the offer to the player, and then activate or deactivate the bet for the next throw only. This level of interaction was a nuance we elected to ignore.

36.3 Betting Strategies

Because of the complexity of Blackjack, the strategies for play focus on the cards themselves, not on the bets. Some players use a single fixed bet amount. Some players will attempt to count cards, in an effort to determine the approximate distribution of cards in the deck, and vary their play or bets accordingly. The casinos actively discourage counting in a number of ways. The most common way is to shuffle 5 decks of cards together, and only deal the first 156 or so of the available 260 cards. Additionally, they will ask people to leave who are obviously counting.

The player's responses to the various offers are what defines the playing strategy in Blackjack. The information available to the player is their hand (or hands), and the dealer's up card. Therefore, all of the strategies for play decompose to a matrix showing the player's total vs. dealer's up card and a recommendation for which offers to accept or decline.

Most players will decline the insurance offer, except when they hold a 21. In that rare case the even money offer should be declined, since the expected value analysis of the result shows a slightly better payout by competing against the dealer.

The decision matrix has two parts: accepting or rejecting the split offer, and choosing among hit, stand or double down. You can buy cards in casino gift-shops that summarize a playing strategy in a single, colorful matrix with a letter code for split, hit, double and stand. Note that each decision to hit results in a new card, changing the situation used for decision-making. This makes the strategy an interesting, stateful algorithm.

A player could easily add the betting strategies we've already defined to their Blackjack play strategies. A player could, for example, use the Martingale system to double their bets on each hand which is a loss, and reset their betting total on each hand which is a win. Indeed, our current design permits this, since we disentangled the betting strategies from the individual games in *Roll-Counting Player Class*.

CHAPTER

THIRTYSEVEN

BLACKJACK SOLUTION OVERVIEW

In *Preliminary Survey of Classes* we;ll present a survey of the new classes gleaned from the general problem statement in *Problem Statement* as well as the problem details in *Blackjack Details*. This survey is drawn from a quick overview of the key nouns in these sections. We will not review those nouns already examined for Craps and Roulette.

From the nouns we can start to define some classes. We'll present this in *Preliminary Class Structure*.

We'll confirm our notion with a walkthrough of parts of a scenario. We'll show this in *A Walkthrough*.

In *Blackjack Solution Questions and Answers* we'll provide some additional ideas on the overall solution.

37.1 Preliminary Survey of Classes

In reading the background information and the problem statement, we noticed a number of nouns that seemed to be new to the game of Blackjack.

- Card
- Deck
- Point Value
- Hand
- Number Card
- Face Card
- Offer
- Insurance
- Split
- Double
- Hit
- Stand
- Player
- Game

The following table summarizes some of the new classes and responsibilities that we can identify from the problem statement. This is not the complete list of classes we need to build. As we work through the exercises, we'll discover additional classes and rework some of these classes more than once.

We also have a legacy of classes available from the Roulette and Craps solutions. We would like to build on this infrastructure as much as possible.

37.2 Preliminary Class Structure

Card Responsibilities

Three apparent subclasses: `NumberCard`, `FaceCard` and `Ace``

A standard playing card with a rank and a suit. Also has a *point value* from 1 to 11. Aces have point values that depend on the *Hand*.

Collaborators

Collected in a `Deck`; collected into *Hand*s for each *Player*; collected into a *Hand* for the dealer; added to by *Game*.

Deck Responsibilities

A complete set of 52 standard *Card*s.

Collaborators

Used by the *Game* to contain *Card*s.

Hand Responsibilities

A collection of *Card*s with one or two point values: a hard value (an ace counts as 1) and a soft value (an ace counts as 11). The house will reveal one *Card* to the player.

Collaborators

A *Player* may have 1 or more *Hand*s; a *Hand* has 2 or more *Card*s. The *Game* adds *Card*s to the *Hand*. The *Game* checks the number of cards, the point totals and the ranks of the cards to offer different bets. The *Game* compares the point totals to resolve bets.

Player Responsibilities

Places the initial ante *Bet*s, updates the stake with amounts won and lost. Accepts or declines offered additional bets, including insurance, and split. Accepts or declines offered resolution, including even money. Chooses among hit, double and stand options.

Collaborators

Uses *Table*, and one or more *Hand*s. Examines the dealer's *Hand*. Used by game to respond to betting offers. Used by *Game* to record wins and losses.

Game Responsibilities

Runs the game: offers bets to *Player*, deals the `Cards` from the `Deck` to *Hand*s, updates the state of the game, collects losing bets, pays winning bets. Splits *Hand*s. Responds to player choices of hit, double and stand. This encapsulates the basic sequence of play into a single class.

Collaborators

Uses `Deck`, *Table*, *Outcome*, *Player*.

37.3 A Walkthrough

The unique, new feature of Blackjack is the more sophisticated collaboration between the game and the player. This interaction involves a number of offers for various bets, and bet resolution. Additionally, it includes offers to double, hit or stand. We'll examine parts of a typical sequence of play to assure ourselves that we have all of the necessary collaborations and responsibilities.

A good way to structure this task is to do a CRC walkthrough. For more information on this technique see *A Walkthrough of Roulette*. We'll present the overall sequence of play, and leave it to the student to manage the CRC walkthrough.

Typical Blackjack Game

1. **Place Bets**. The Game will ask the player to place a bet. If the player doesn't place a bet, the session is over.
2. **Create Hands**. The Game will deal two cards to the Player's initial Hand.

   The Game will create an initial hand of two cards for the dealer. One of the cards is the up card, and is visible to the player.
3. **Insurance?** The Game gets the Dealer's Hand's up card. If it is an Ace, then insurance processing is perforemed.
   (a) **Offer Even Money**. The Game examines the Player's hand for two cards totalling a soft 21, blackjack. If so, the Game offers the Even Money resolution to the Player. If the player accepts, the entire game is resolved at this point. The ante is paid at even money; there is no insurance bet.

   (b) **Offer Insurance**. The Game offers insurance to the Player, who can accept by creating a bet. For players with blackjack, this is a second offer after even money is declined. If the player declines, there are no further insurance considerations.

   (c) **Examine Hole Card**. The Game examines the Dealer's Hand's hole card. If is is a 10-point value, the insurance bet is resolved as a winner, the ante is resolved as a loser, and for this player, the game is over. Otherwise the insurance is resolved as a loser, the hole card is not revealed, and play will continue. Note that in a casino with multiple players, it is possible for a player declining insurance to continue to play with the dealer's hole card revealed. For casinos that offer "early surrender" this is the time to surrender.
4. **Split?** The Game examines the Player's Hand to see if the two cards are of equal rank. If so, it offers a split. The player accepts by creating an additional Bet. The original hand is removed; The Game splits the two original Cards then deals two additional Cards to create two new Hands.

   Some casinos prevent further splitting, others allow continued splitting of the resulting hands.
5. **Play Out Player Hands**. The following are done to play out each of the Player's Hands.
   (a) **Bust? Double? Hit? Stand?** While the given Hand is under 21 points, the Game must extend three kinds of offers to the Player. If the Player accepts a Hit, the hand gets another card and this process repeats.

   If the Player accepts Double Down, the player must create an additional bet, and the hand gets one more card and play is done. If the Player Stands Pat, the play is done. If the hand is 21 points or over, play is done.

   (b) **Resolve Bust**. The Game must examine each Hand; if it is over 21, the Hand is resolved as a loser.
6. **Play Out Dealer Hand**. The Game then examines the Dealer Hand and deals Cards on a point value of 16 or less, and stops dealing Cards cards on point value of 17 or more.
   (a) **Dealer Bust?** The Game then examines the Dealer Hand to see if it is over 21. If so, the player's bets are resolved as winners. Player Hands with two cards totalling 21 ("blackjack") are paid 3:2, all other hands are paid 1:1.
7. **Compare Hands**. For each hand still valid, the Game compares the Player's Hand point value against the Dealer's Hand point value. Higher point value wins. In the case of a tie, it is a push and the bet is returned.

   When the Player wins, a winning hand with two cards totalling 21 ("blackjack") is paid 3:2, any other winning hand is paid 1:1.

37.4 Blackjack Solution Questions and Answers

Will we really need both `Deck` and the multiple deck `Shoe`? Wouldn't it be simpler to combine this functionality into a single class?

> There are two separate responsibilities here. The deck owns the basic responsibility to build the 52 cards. The shoe, on the other hand, owns the responsibility to deal cards to hands.

We want to be able to simulate games with 1 to 8 decks. A single deck game can simply deal directly from the deck. In a multi-deck game, all of the decks are shuffled together and loaded into a *shoe* for dealing. The difference between one deck and a five-deck shoe is that the shoe can produce 20 kings in a row. While rare, our simulation does need to cover situations like this.

Also, we may want to build a slightly different shoe that simulates the continuous shuffling machine that some casinos use. In this case, each hand is reshuffled back into the shoe, preventing any attempt at card counting. We don't want to disturb the basic, common deck when introducing this additional feature.

Won't all those player interactions break our design?

That's unlikely. All of that player interaction is in addition to the `placeBets()` interface. Since we've separated the core features of all players from the game-specific features, we can add a subclass to player that will be handle the Blackjack interaction. This new player subclass will have a number of additional methods to handle insurance, even money, split and the regular play questions of hit, double and stand.

In parallel, we've separated the core features of all games from the unique features for a specific game. We can now add a subclass for Blackjack which adds a number of methods to offer insurance, even money, split and the regular play questions of hit, double and stand to the Blackjack player.

I can't find an Outcome in Blackjack. Is it the Ante? If so, the odds vary based on the player's Hand, but that doesn't seem to be a RandomEvent.

Good point. We'll examine this in detail in the exercises. Clearly, the bets are placed on the Ante and Insurance as the two core *`Outcome`* s in Blackjack. The Insurance outcome (really a "dealer has blackjack" outcome) is fixed at 2:1. The ante payoff depends on a complex condition of the hand: for a soft 21, or blackjack, it pays 3:2; otherwise it pays 1:1. This will lead to a new subclass of *`Outcome`* that collaborates with the hand to determine the payout to use.

The "even money" is offered before ordinary insurance to a player with blackjack. It, however, pays even money on the ante, and doesn't create a new bet; in this respect it could be thought of as a change in the outcome on which the ante bet is created. Accepting the even money offer is a little bit like moving the ante to a "even money for dealer blackjack" outcome, which has 1:1 odds instead of 3:2 odds. Further, this special outcome is resolved before the dealer peeks at their hole card. Perhaps this is a special best resolution procedure, not a proper instance of *`Outcome`*.

CHAPTER

THIRTYEIGHT

CARD, DECK AND SHOE CLASSES

This chapter introduces a number of simple classes. When exploring Roulette, we introduced classes very slowly. In this chapter, we are introducing the class hierarchy for cards, the deck of cards and the shoe, used by the dealer to create hands.

In *Card, Deck and Shoe Analysis* we'll look at a number of issues related to cards. This includes *points*, *ordering*, *suits*, and *identity*. We'll also look at *Card Subclass Factory*, *Unicode Images*, *Deck*, and *Shoe*.

We'll follow this with *Card-Deck-Shoe Questions and Answers* to address any lingering doubts or minor issues.

In *Card Superclass* we'll present the overall design for `Card`. This is followed by face cards in *FaceCard Class*, and the special cases for aces in *AceCard Class*.

We'll look at the need for a factory function in *Card Factory Function*.

We'll also design containers for cards including *Deck class* and *Shoe class*.

We'll enumerate the deliverables in *Card-Deck-Shoe Deliverables*.

38.1 Card, Deck and Shoe Analysis

The standard playing card has two attributes: a rank (Ace, 2 through 10, Jack, Queen, or King) and a suit (□ , $\diamondsuit, \heartsuit, or \square).Thesetofall52combinationscomprisesafulldeck.$

The basic set of responsibilities of the `Card` class include keeping the rank and suit of a single standard playing card. It turns out, however, that these objects are fairly complex.

We'll look at several topics:

- *Points*. A card has a point value, which is a number from 1 to 11, and is based on the rank. In the case of an Ace, it is also based on the hand in which the card is evaluated. This collaboration with a hand complicates the responsibilities for a single card.
- *Ordering*. An Ace has a complex life. What order do we put the cards in?
- *Suits*. Suit doesn't matter, right? What about the name of the game, "black jack"?
- *Identity*. How do we compare cards?
- *Card Subclass Factory*. How do we create card objects that are in their correct subclass?
- *Unicode Images*. Since Unicode offers us images of cards, we can use these.
- *Deck*. Cards come in a standard deck of 52 cards.
- *Shoe*. Blackjack dealers often use a shoe with up to eight decks.

We'll start by looking at points.

38.1.1 Points

We have three different rules for establishing the point value of a card. This is a big hint that we have three subclasses.

- One subclass includes the Aces, which are either 1 or 11 points.
- Another subclass includes the number cards from 2 to 10, where the rank is the point value.
- Finally, the third subclass includes the face cards, where the point value is fixed at 10.

These three subclasses implement slightly different versions of a method to return the point value of the card. For more discussion on this, see *Card-Deck-Shoe Questions and Answers* FAQ for more discussion.

The problem of determining the value of aces is something we will have to defer until after we create the `Hand` class. First, we'll implement the basic `Card`, then we'll develop the `Hand`, and decide how the `Hand` computes its number of points from the `Card`s.

The terminology used gives us some hint as to how to structure our design. Players refer to a *soft* total and a *hard* total. An A-6 hand is called a soft 17, and a hard 7. A soft hand has some flexibility in how it is played. If you hit a soft 17, and get a face card (for example, a Jack), you now have an A-J-6, totalling hard 17.

For now, we'll consider two methods: `softValue()` and `hardValue()` to return the various values of each card. For ordinary cards, the values are both the rank. For face cards, the values are both 10. For aces, however, the hard value is 1 and the soft value is 11.

While this design does have some potential problems in dealing with multiple aces in a single hand, we'll let it stand until we have more design in place.

38.1.2 Ordering

Another exasperating detail of standard playing cards is the ordering of the various ranks.

In many games the ordering of cards is 2-10, J, Q, K, A. The Ace (rank of 1) is placed after the King.

In Blackjack, Ace is both 1 and 11, leading to a variable ranking, depending on other cards in the hand.

For games like Poker and Bridge, we need to distinguish the ordering and the rank because the ordering of the cards doesn't depend on the simplistic rank value.

38.1.3 Suits

The issue of suit requires some care. In the game of Blackjack, suits don't matter. Indeed, for the purposes of simulation, we could simply discard the notion of suit. However, for the purposes of making a reasonably complete model of real-world objects, it makes sense to implement the suit of a card, even if we do nothing with it.

Also, suits have two colors: red and black. Even though the name of the game includes a color and rank ("black", "jack") these are both irrelevant to the modern game. Historically, an ace of spades and either of the black jacks had a special 10:1 payout. Since this is no longer the case, we find the color and suit no longer matter.

We'd like to provide symbolic variable names for the suits. In Python we have a number of choices.

- We can create an `enum` class for the suits.
- We can use class-level variables to define four named constants.

We would also like to provide named constants for the face cards: Jack, Queen, and King, as well as the Ace. This does not appear to be a perfectly sensible use of the `enum` definitions, since these labels are comingled with integer ranks.

For this reason, we'll suggest class-level variables for all of these symbolic constants. We can then say `Card.Spades` to reference the unicode character `u'\N{BLACK SPADE SUIT}'`.

There's a compelling case to be made for defining an `enum.Enum` for the suits. While the examples will assume class level variables, this isn't necessary, and the student may want to pursue the `enum` solution.

38.1.4 Identity

Cards need to be compared. We have a rich set of comparisons for cards.

- Hash Codes. We must implement a proper hash function, `__hash__()` that includes both rank and suit.

  The definition for `__hash__()` in section 3.3.1 of the *Language Reference Manual* tells us to do the calculation using a modulus. The modulus is based on `sys.hash_info.width`, which is the number of bits. The actual value we want to use is `sys.hash_info.modulus`.

  $$h(c) = (h(c_r) + h(c_s)) \mod 2^{61} - 1$$

  The hash for a card, $h(c)$, is equal to the sum of the hash for the card's rank, $h(c_r)$ plus the hash for the card's suit $h(c_s)$. The sum is taken modulus `sys.hash_info.modulus`. The value shown, $2^{61} - 1$ is typical, but not the only value used. This is based on `sys.hash_info.width`.

- Equality. We must implement `__eq__()` and `__ne__()` so that we can compare cards to each other. This is a subtle issue because – in Blackjack – suits don't matter. A 10 of Clubs is essentially equal to a 10 of Spades.

- Rank. We might want to implement `__gt__()`, `__lt__()`, `__ge__()`, `__le__()` so that cards can be properly ordered.

38.1.5 Card Subclass Factory

We need to be able to create a Card without being deeply concerned about which specific subclass it belongs to. This is where a **Factory** or **Builder** is essential.

Ideally, we'll have a *`card_factory()`* function that emits an object of the appropriate subclass of *`Card`*.

We can then do things like this:

Creating Cards

```
for suit in Card.Spades, Card:Hearts, Card.Diamonds, Card.Clubs:
    for rank in range(1,14):
        c = card_factory(rank, card)
```

This kind of nested loop should properly create all 52 cards.

38.1.6 Unicode Images

The Unicode character set has individual card images available as characters. There are a number of images available for each suit.

The mapping works like this:

spades 0x1F0A0 to 0x1F0AE

hearts 0x1F0B0 to 0x1F0BE

diamonds 0x1F0C0 to 0x1F0CE

clubs 0x1F0D0 to 0x1F0DE

There's an interesting wrinkle in this mapping. Unicode includes a "knight" which we need to skip to map rank and suit to a Unicode character.

Ranks 1 to 11 (Ace to Jack) are simple additions to the base value for the suit.

> 5 Diamonds = 0x1F0C0 + 5 = □

Ranks 12 and 13 (Queen and King) are incremented by one: we add 13 and 14 (0xD and 0xE)

> Queen Spades = 0x1F0A0 + 13 = □

The images are only legible if we use a fairly large font.

38.1.7 Deck

A deck of cards has responsibility for shuffling and dealing the various cards. Additionally, it should constuct the complete set of 52 cards. We note that shuffling uses a random number generator, but a deck isn't the same kind of random event factory that a Wheel or pair of Dice is. In the case of Wheel and Dice, the random events were based on random selection with replacement: an individual event can be regenerated any number of times. In the case of a deck, however, once a card has been dealt, it will not show up until the deck is shuffled.

In Roulette and Craps, the odds depended on a *`RandomEvent`*: either the *`Bin`* or *`Throw`*. In Blackjack, the ante's win amount depends on the player's entire hand. This means that being dealt an individual card isn't the same kind of thing that a throw of the dice is; rather it suggests that the dealer's shoe is the random event factory, and the entire hand is the random event created by dealing cards. Continuing this line of thought, an *`Outcome`*'s win amount could depend on a *`RandomEvent`*, if we consider the entire hand to be the random event.

Considering an entire hand to be a single random event is skating on pretty thin ice, so we won't force-fit a `Deck` into the random event factory part of our framework. Instead, we will let `Deck` stand alone. We'll design a simple initialization for a deck that constructs the 52 cards. Beyond that, the notions of shuffling and dealing can be assigned to the shoe.

Since a `Deck` is a container, we have to examine the available collection classes to determine which concrete class we need. Interestingly, we only need two features of `Collection`: the `add()` method and the `iterator()`. These methods are implemented for all of the variations of `Set` and `List`.

38.1.8 Shoe

The dealer's shoe is where *`Card`*s are shuffled and dealt to create individual hands. A shoe will be built from some number of `Decks`. More properly, the shoe will contain the *`Card`*s from those `Decks`. The `Deck` objects aren't really used for much more than constructing batches of individual *`Card`* objects. The `Shoe` responsibilities include proper initialization using a given number of decks, periodic shuffling and proper dealing.

In a casino, the shuffling involves a ritual of spreading the cards on the table and stirring them around thouroughly, then stacking them back into the shoe. The top-most card is extracted, shown to the players, discarded, and a marker card is cut into the shoe at least two decks from the end, leaving about 100 cards unplayable after the marker. While most of the ritual does not require careful modeling, the presence of undealt cards at the end of the shoe is important as a way to defeat card-counting strategies.

Since a `Shoe` is a container, we have to examine the available collection classes to determine which concrete class we need. Interestingly, we only need two features of `Collection`: the `addAll()` method to put another deck into the shoe and the `iterator()`. These methods are implemented for all of the variations of `List`. We will have to disregard the various `Set` implementations because they impose their own unique orders on the elements, different from our shuffled order.

The simplest shuffling algorithm iterates through all of the *`Card`*s in the `Shoes` collection, and exchanges that *`Card`* with the card in a randomly-selection position. In order to move *`Card`*s around freely within the structure, an `list` must be used. See *Card-Deck-Shoe Questions and Answers* for more discussion on shuffling.

38.2 Card-Deck-Shoe Questions and Answers

Why are there three subclasses of Card? Isn't it simpler to have one class and use an if-statement to sort out the point values?

> Primarily, there are three classes because they have different behaviors. Merging them into a single class and sorting out the behaviors with an if-statement is often a problem.
>
> First, and most important, if-statements add complexity. The question "wouldn't it be simpler to use an if-statement" is a kind of oxymoron.
>
> Second, and almost as important, if-statements dilute responsibility assignments. Combining all three subclasses into one puts three slightly different responsibilities into one place, making it more difficult to debug problems. Further, we could wind up repeating or other reusing the if-statement in inappropriate ways. If we create separate subclasses, the clear separation of responsibility becomes a matter of definition, not a matter of following a complex thread of programming logic.
>
> Third, if-statements limit growth, adaptation and change. If we have a modification to the rules, for example, making 1-eyed Jacks wild, we would prefer to simply introduce another subclass. We find that chasing down one or more related if-statements to assure ourselves that we are correctly handling the new subtlety rapidly gets out of hand.

What about getters for rank and suit of a `Card`?

> Below, we'll excplicitly avoid writing getter methods for these attributes.
>
> Generally, it's atypical Python programming. We avoid getters for attributes. In the event of a design change, we can replace an attribute with a property to preserve the syntax but add features.
>
> The notion of Encapsulation does not require all attributes be wrapped with method functions. This is an implementation choice used in Java to permit ready introspection of classes.
>
> Our approach of defining responsibilities narrowly is all that's required to have a properly encapsulated design. Collaborators are allowed to examine attributes. In Python, we know that attributes are part of the public interface of a class.

Is that the best shuffling algorithm? Won't it sometimes move a card twice? Won't it sometimes put a card back into the original spot?

> Yes, it may move some cards twice and it may leave a card in position. This is part of random behavior. This algorithm touches every card, swapping it with a randomly selected card. We are assured that every card was put into a random position. Sometimes a card will have been moved more than once, but the minimum criteria is that every card has been moved.
>
> While a shuffling algorithm that models the real world is tempting, this adds complexity for no actual improvement in the randomization. A popular technique in the real world is to cut the deck in half and then riffle the cards into a single pile. If done with the kind of perfection that software provides (cutting the deck exactly in half and exactly alternating the cards) this shuffle leads to a perfectly predictable cycle of orders. What makes this shuffle work in the real world is the random inaccuracies in cutting and riffling. We don't see any value in modeling these physical phenomenon.
>
> A similar analysis holds for the kind of shuffle done in the casino. In essence, they do a shallow copy if the original `List` object, and then rebuild the shoe's `List` by picking cards at random from the copy. This produces a result that is statistically indistinguishable from our algorithm, which uses an element-by-element swap.
>
> One fruitless side-track is using the seemingly-random hash code values of the `Card` objects. This only puts the cards into a single fixed, but arbitrary order. An apparently interesting alternative is to generate a random index for each `Card` and then sort by this index. We note that sorting is $\mathbf{O}(n \log n)$, where our algorithm is $\mathbf{O}(n)$, running much faster than any sort.

38.3 Card Superclass

class Card

Card defines a basic playing card. It has a rank, a suit, a hard point value and a soft point value. The point value methods are defined for the number cards from 2 to 10. Two subclasses handle face cards, where the point values are both 10, and aces, where the soft point value is 1, and the hard point value is 11.

This class also defines symbolic names for the suits (Clubs, Diamonds, Hearts and Spades) and face cards (Jack, Queen and King).

Here are Unicode characters for the suits.

Unicode	Number	Name	Symbol
U+02660	`u'\u2660'`	`u'\N{BLACK SPADE SUIT}'`	□
U+02661	`u'\u2661'`	`u'\N{WHITE HEART SUIT}'`	♡
U+02662	`u'\u2662'`	`u'\N{WHITE DIAMOND SUIT}'`	♢
U+02663	`u'\u2663'`	`u'\N{BLACK CLUB SUIT}'`	□

Note that Unicode officially includes "RED HEART SUIT" and "RED DIAMOND SUIT". Python, however, may not recognize these. It seems necessary to use Unicode names shown above.

38.3.1 Fields

In Python, symbolic names are often declared within the class, not within the initialization method function.

Class-Level Names

```
class Card( object ):
    Clubs, Diamonds, Hearts, Spades = u'\u2663', u'\u2662', u'\u2661', u'\u2660'
    Ace, Jack, Queen, King = 1, 11, 12, 13
    ...

oneEyedJack= Card(Card.Jack, Card.Hearts)
```

`Card.`**rank**

The rank of the card. This is a number from 1 (Ace) to 13 (King).

`Card.`**suit**

The suit of the card. This is a character code for Clubs, Diamonds, Hearts or Spades.

38.3.2 Constructors

`Card.`**__init__**(*self*, *rank*, *suit*)

Parameters

- **rank** (*integer*) – rank of this card.
- **suit** (Character from the set `{u'\u2663', u'\u2662', u'\u2661', u'\u2660'}` Ideally based on a class-level variable.) – Character code for this card.

Initializes the attributes of this *Card*.

38.3.3 Methods

Note that "getters" are something we don't bother with in Python. In other languages, e.g., Java, this class would need to have `getRank()` and `getSuit()` methods. This is atypical for Python.

`Card.`**`softValue`**`(`*self*`)` → int
: Returns the soft value of this card. The superclass simply returns the rank. Subclasses can override this. Face cards will return 10, Aces will return 11.

`Card.`**`hardValue`**`(`*self*`)` → int
: Returns the hard value of this card. The superclass simply returns the rank. Subclasses can override this. Face cards will return 10, Aces will return 1.

`Card.`**`__str__`**`(`*self*`)` → str
: Return the rank and suit of this card.

  Note that Python 2 won't work well with Unicode suit names. If you're struggling with proper printing of the Unicode characters, upgrade to Python 3.

38.4 FaceCard Class

`FaceCard` is a *`Card`* with a point value of 10. This defines jack, queens and kings.

38.4.1 Methods

`Card.`**`softValue`**`(`*self*`)` → int
: Returns the soft value of this card, 10.

`Card.`**`hardValue`**`(`*self*`)` → int
: Returns the hard value of this card, 10.

`Card.`**`__str__`**`(`*self*`)` → str
: Returns a short String displaying the rank and suit of this card. The ranks should be translated to single letters: 11 to `'J'`, 12 to `'Q'` and 13 to `'K'`.

38.5 AceCard Class

`AceCard` is a *`Card`* with a soft point value of 11 and a hard point value of 1. This defineces Aces.

38.5.1 Methods

`Card.`**`softValue`**`(`*self*`)` → int
: Returns the soft value of this card, 11.

`Card.`**`hardValue`**`(`*self*`)` → int
: Returns the hard value of this card, 1.

`Card.`**`__str__`**`(`*self*`)` → str
: Returns a short String displaying the rank and suit of this card. The rank is always `'A'`.

38.6 Card Factory Function

card_factory()

Parameters

- **rank** (*int*) – Numeric rank
- **suit** (Character from the set `{u'\u2663', u'\u2662', u'\u2661', u'\u2660'}` Ideally based on a class-level variable.) – Symbolic suit

This function creates the proper subclass of card based on the rank.

- 1 maps to `AceCard`.
- 2 to 10 maps to `Card`.
- 11, 12, and 13 map to `FaceCard`.

38.7 Deck class

`Deck` defines the standard deck of 52 cards. It both constructs the deck and acts as a container for one instance of a deck.

38.7.1 Fields

cards

The collection of individual cards. The specific type of collection could be any of the Set or List implementation classes.

38.7.2 Constructors

`Deck.__init__(self)`

Creates the Collection, `cards`, and then creates the 52 cards. A simple nest pair of loops to iterate through the suits and ranks will work nicely for this.

Creating A Full Deck

For all four suits:

Create the `AceCard` of this suit and a rank of 1; add to the `cards`

For ranks 2 to 10:

Create a `Card` of this suit and rank; add to the `cards`

For ranks 11 to 13:

Create a `FaceCard` of this suit and rank; add to the `cards`

38.7.3 Methods

`Deck.getCards(self)` → collection

Returns the collection of cards in `cards`.

38.8 Shoe class

`Shoe` defines the dealer's shoe, which contains from 1 to 8 decks of cards. For one deck shoes, one card is reserved as undealable. For multiple deck shoes from 1 to 3 decks can be left undealt. The exact number is selected at random within 6 cards of the expected number of decks.

38.8.1 Fields

`Shoe.`**`deal`**
An `Interator` that is used to pick the next card from the shoe.

`Shoe.`**`stopDeal`**
The approximate number of decks to be left undealt in the shoe.

38.8.2 Constructors

`Shoe.`**`__init__`**(*self*, *decks*, *stopDeal*)

Parameters

- **`decks`** (*integer*) – A number of decks to create
- **`stopDeal`** (*integer*) – An approximate number of decks left undealt in the shoe

This creates a random number generator from `random.Random`.

It initializes the Shoe by creating the required number of decks and building the *`cards`* `List`. This saves the *`stopDeal`* value, which is the number of decks left in the shoe. Typically, this is two, and approximately 104 cards are left in the shoe.

To facilitate testing, this initializes a dealing iterator to the unshuffled `List` of cards. This will produce cards in a fixed order.

Most unit tests will mock the shoe with a fixed sequence of cards used to exercise player and dealer processing.

38.8.3 Methods

`Shoe.`**`shuffle`**(*self*)
Shuffles the shoe by swapping every element in the `Shoe.cards` `List` with a random element.

Creates an `Iterator`, *`Shoe.deal`* that can be used to deal cards.

If the *`stopDeal`* is non-zero, do the following to exclude several decks from the deal.

1. Create a random number, v, such that $-6 \leq v < 6$.
2. Step through the deal iterator $stop * 52 + v$ times, removing approximately *stop* cards from being dealt as part of ordinary play.

`Shoe.`**`__iter__`**(*self*) → iter
Return the *`Shoe.deal`* iterator so that the game can get cards from the Shoe.

38.9 Card-Deck-Shoe Deliverables

There are many deliverables for this exercise.

- The three classes of the *`Card`* class hierarchy, including `FaceCard` and `AceCard`.

- A class which performs a unit tests of the *Card* class hierarchy. The unit test should create several instances of *Card*, FaceCard and AceCard.
- The Deck class.
- A class which performs a unit test of the Deck class. This simply creates a Deck object and confirms the total number of cards. A thorough test would also check some individual *Card* objects in the cards collection.
- The Shoe class.
- A class which performs a unit test of the Shoe class. This simply creates a Shoe object and confirms that it deals cards. In order to test the shuffle() method, you will need to construct the Shoe with a random number generator that has a fixed seed and produces cards in a known sequence.

CHAPTER THIRTYNINE

HAND AND OUTCOME CLASSES

This chapter introduces the hand, and the problems of scoring the value of the hand. It also introduces a subclass of *Outcome* that can handle the more complex evaluation rules for Blackjack.

In *Hand Analysis* we'll look at a number of issues related to Blackjack hands. In *Payout Odds* we'll look at how the odds depend on the cards and the way the hands are resolved. In *Hard and Soft Totals* we'll examine how an Ace leads to a hand having distinct hard and soft totals. This will cause us to rexamine the nature of outcomes; this is the subject of *Blackjack Outcomes*.

In *Hand Total Class Design* we'll address the issue of hard totals and soft totals for a hand. We'll detail two subclasses in *Hand Hard Total Class Design* and *Hand Soft Total Class Design*.

This will lead to some small changes in the card class hierarchy. We'll look at this in *Card Class Updates*. This will include design details in *Card Class* and *AceCard Class*.

We can then design the overall *Hand* class. This is described in *Hand Class Design*. We'll look at all of the deliverables for this chapter in *Hand Deliverables*.

39.1 Hand Analysis

The hand of cards is both a container for cards, but is also one dimension of the current state of the player's playing strategy. In addition to the player's hand, the other dimension to the strategy is the dealer's up card.

A player may have multiple hands. The hands are resolved independently.

For each hand, the responsibilities include collecting the cards, producing a hard and soft point total and determining the appropriate payout odds.

Collecting cards is trivial. Each hand is a simple *bag* or *multiset* of cards. A hand can't be a `Set` because there can easily be duplicate cards in a multiple deck game. In an 8-deck game, there are 8 separate instances of A□ .

Determining the payout odds for a Hand is somewhat more complex.

39.1.1 Payout Odds

To be compatible with other games, we'll be creating *Bet* objects which are associated with *Outcome*s. In Roulette, there were a profusion of *Outcome*s, each relatively simple with fixed odds. In Craps, there were fewer *Outcome*s, some of which had odds that depended on a throw of the dice.

The player's *Hand* must be associated with an *Outcome* so that we can match the *Outcome* of a *Bet* and the *Outcome* of a *Hand* to determine the payout.

In the case of Roulette, each *Bin* was simply a set of winning `Outcomes`. In the case of Craps, both the dice and the game had a mixtures of winning, losing and unresolved `Outcomes`. The *Bet*s were simply associated with *Outcome*s and the Wheel, Dice or Game gave us sets of winning (and losing) *Outcome*s.

In Blackjack, there are relatively few outcomes. And it's not clear how each *Outcome* associates with a *Hand*.

Survey of Outcomes. To figure out how to associate *Hand* and *Outcome*, we'll start by enumerating all the individual *Outcome*s.

1. **Insurance** at 2:1. This is a winner when the dealer's hand is blackjack; and the "Ante" bet will be a loser. It is only offered when the up card is an Ace. This is a loser when the dealer's hand is not blackjack, and the Ante bet is unresolved.
2. **Even Money** at 1:1. This is offered in the rare case of the player's hand is blackjack and the dealer's up card being an Ace. If accepted, it can be looked at as a switch of the Ante bet to an "Even Money" outcome, which is then resolved as a winner.
3. **Ante** paying 1:1. This payout occurs when the player's hand is less than or equal to 21 and the dealer's hand goes over 21. This payout also occurs when the player's hand is less than or equal to 21 and greater than the dealer's hand.

   This outcome is a loser as soon as the player's hand goes over 21. It is also a loser when the player's hand is less than or equal to 21 and less than the dealer's hand.
4. **Ante** paying 3:2. This payout occurs when the player's hand is blackjack. The odds depend on the player's hand.
5. **Ante** resolved as a push, paying 1:0. This payout occurs when the player's hand is less than or equal to 21 and equal to the dealer's hand. The odds depend on both player and dealer's hand.

Problem. What kind of class is *Hand*? Is it a collection of *Outcome*s? Or is it something different?

It appears that the *Hand*, as a whole, is not simply associated with an *Outcome*. It appears that a *Hand* must produce an *Outcome* based on its content, the dealer's content, and possibly the state of the game.

This is a small change from the way *Dice* and *Bin* work. Those classes were associated with an *Outcome*. A Blackjack *Hand*, however, must do a bit of processing to determine which *Outcome* it represents.

- A two-card hand totalling soft 21 produces a blackjack *Outcome* that pays 3:2.
- All other hands produce an *Outcome* that pays 1:1 and could be resolved as a win, a loss, or a push.

Also, changes to the state of the game depend on the values of both hands, as well as the visible up card in the dealer's hand. This makes the state of the hand part of the evolving state of the game, unlike the simple *RandomEvent*s we saw in Roulette and Craps.

Forces. We have a few ways we can *deal* with *Hand*.

- We can make *Hand* a subclass of *RandomEvent*, even though it's clearly more complex than other events.
- We can make *Hand* a unique kind of class, unrelated to other games.

Hand is an "Event"? While a *Hand* appears to be a subclass of *RandomEvent*, it jars our sensibilities. A *Hand* is built up from a number of *Card*s. The card seems more event-like.

One could rationalize calling a *Hand* an "event" by claiming that the `Shoe` is the random event generator. The act of shuffling is when the events is created. The complex event is then revealed one card at a time.

It seems that we need to define a *Hand* class that shares a common interface with a *RandomEvent*, but extends the basic concept because a hand has an evolving state.

Hand is different. While we can object to calling a *Hand* a single "event", it's difficult to locate a compelling reason for making a *Hand* into something radically different from *Bin* or *Dice*.

Hand Features. Our first design decision, then is to define *Hand* as a kind of *RandomEvent*. We'll need to create several *Outcome*s: Insurance, Even Money, Ante and Blackjack.

The *Hand* will produce an appropriate *Outcome* based on the hand's structure, the game state, and the dealer's hand. Generally, each *Hand* will produce a simple Ante outcome as a winner or loser. Sometimes a *Hand* will produce a Blackjack outcome.

Sometimes the Player and Blackjack Game will collaborate to add an Insurance or Even Money outcome to the *Hand*.

39.1.2 Hard and Soft Totals

Our second design problem is to calculate the point value of the hand. Because of aces, hands can have two different point totals. If there are no aces, the total is *hard*. When there is an Ace, there are two totals. The hard total uses Aces as 1. The *soft* total uses Aces as 11.

We note that only one ace will participate in this hard total vs. soft total decision-making. If two aces contribute soft values, the hand is at least 22 points. Therefore, we need to note the presence of at most one ace to use the soft value of 11, all other cards will contribute their hard values to the hand's total value.

A *Hand* has a two states.

- Hard. When the hand has no aces, there is one total.

  Also, when a hand has one or more aces, but the soft total is over 21, then the hand only has a hard total.
- Soft. When a hand has at least one Ace and the soft total is 21 or less, there is a hard and a soft total.

A hand with no cards could be said to be in the hard total state. As each card is added, the state may change. It may change to soft when an Ace is added and the soft total will be 21 or less. It may changes to hard when the soft total is over 21.

State Change Rules. We can see that our *Hand* state change is split among the various subclass of *Card*.

The *Card* (which will be inhereted by `FaceCard`) we need to check the *Hand* hard total. If the hard total is over 21, the state change method needs to switch the *Hand* to the hard total state. Otherwise, it leaves the state alone.

The `AceCard`, however, must use a slightly different algorithm. If the hard total is over 21, the state change method needs to switch the *Hand* to the hard total state. Otherwise, the hard total is 21 or under, and the state needs to switch the *Hand* to the soft total state.

Since there are only two distinct total algorithms, we can actually put both of them right into each *Hand* object and use simple assignment statements to choose which algorithm is in force at any given state of the *Hand*.

39.1.3 Blackjack Outcomes

As a final design decision, we need to consider creating any subclasses of *Outcome* to handle the variable odds for the "Ante" bet. We don't need a subclass for the "Insurance" or "Even Money", because the base *Outcome* does everything we need.

The notable complication here is that there are three different odds. If the player's hand beats the dealer's hand and is blackjack, the odds are 3:2. If the player's hand beats the dealer's hand, but is not blackjack, the odds are 1:1. If the player's hand equals the dealer's hand, the result is a push; something like 1:0 odds where you get your money back.

It doesn't seem like new subclasses of *Outcome* are necessary. We simply have some alternative outcomes that can be produced by a *Hand*.

The player can only place one of four basic bets.

- The "Ante" bet that starts play. This is assmed to be 1:1 unless game conditions change this to 3:2 or a push. This is the essential *Outcome* for the player's primary *Bet*.
- The "Insurance" and "Event Money" bets. One of these may be active after the first cards are dealt.

  For the insurance *Outcome* to be active, the dealer must be showing an Ace and the player's hand is not 21.

  For even money *Outcome* to be active, the dealer must be showing an Ace and the player's hand is 21.

  These are resolved immediately: if the dealer does not have 21, these bets are lost. If the dealer has 21, these bets win, but the Ante is a loss.
- The "Double Down" bet. This is generally offered at any time. It can be looked at as an an additional amount added to the "ante" bet and a modification to game play.

Objects. It seems simplest to create a few `Outcome` instances: Ante, Insurance, Even Money and Double Down. The Table will have to merge the double-down bet amount into the Ante bet amount.

39.2 Hand Total Class Design

`HandTotal` computes a total of the cards in a hand. There are distinct subclasses for the two algorithms.

39.2.1 Constructor

`HandTotal.__init__`(*self*, *hand*)

Parameters **hand** (Hand) – The hand for which to compute a total

Creates a new HandTotal object associated with a given hand.

39.2.2 Methods

`HandTotal.total`(*self*, *card=None*) → int

Computes a total of all the cards in the associated hand. If card is not None, omit the the indicated card from the total.

This method is abstract, it should return *NotImplemented*. Each subclass will provide an implementation.

Parameters **card** (Card) – A card to exclude from the total

39.3 Hand Hard Total Class Design

class **HandHardTotal**

Computes a hard total of the cards in a hand.

`HandHardTotal.total`(*self*, *card=None*) → int

Computes the hard total of all the cards in the associated hand. If card is not None, omit the the indicated card from the total.

Parameters **card** (Card) – A card to exclude from the total

39.4 Hand Soft Total Class Design

class **HandSoftTotal**

Computes a soft total of the cards in a hand.

`HandSoftTotal.total`(*self*, *card=None*) → int

Computes the soft total of all the cards in the associated hand. If card is not None, omit the the indicated card from the total.

Parameters **card** (Card) – A card to exclude from the total

39.5 Card Class Updates

Each subclass of Card needs to provide a method that sets the hand's total algorithm.

`Card.setAltTotal(hand)`

Parameters **hand** (Hand) – The hand for which to set a total algorithm

There are two different implementations for this method in *Card*, and `AceCard`.

Note that each implementation differs only in what is done with the soft total is 21 or less. With a little care, this single difference can be factored out into the `AceCard` subclass.

39.5.1 Card Class

To determine what alternative total object is appropriate for this hand, do the following.

1. Get the soft total of all cards except this one.
2. Add the soft points of this card. This happens to be the same as the hard points, but a good program doesn't repeat this piece of information; it uses the *Card.softValue()* method.
3. If this is over 21, then set the *Hand.altTotal* to to *Hand.hard*.

   If the soft total is 21 or less, then the *Hand.altTotal* is left untouched.

39.5.2 AceCard Class

To determine what alternative total object is appropriate for this hand, do the following.

1. Get the soft total of all cards except this one.
2. Add the soft points of this card. This happens to be 11 for an Ace, but a good program doesn't repeat this piece of information; it uses the *Card.softValue()* method.
3. If this is over 21, then set the *Hand.altTotal* to to *Hand.hard*.

   Otherwise, the soft total is 21 or less, then set the *Hand.altTotal* to *Hand.soft*.

39.6 Hand Class Design

class Hand

Hand contains a collection of individual *Card*s, and determines an appropriate total point value for the hand.

39.6.1 Fields

Hand.cards

Holds the collection of individiual *Card*s of this hand.

Hand.hard

A instance of *HandHardTotal*. This means that `hand.hard.total()` will produce the hard total of the cards in this hand.

This method is used as one of the point totals for the hand.

Hand.soft

A instance of *HandSoftTotal*. This means that `hand.soft.total()` will produce the soft total of the cards in this hand.

`Hand.altTotal`
This is a reference to either `Hand.hardTotal` or `Hand.softTotal`. This will produce a soft total when one is a appropriate; when a soft total isn't appropriate, it will produce a hard total.

This is set by each Card.

This method is used as one of the point totals for the hand.

39.6.2 Constructors

`Hand.__init__(self, card=None)`

Parameters **card** (Card) – A card to add

Creates an empty hand. The `Hard.cards` variable is initialized to an empty sequene.

The `Hand.hard` and `Hand.soft` objects are initialized to `HandHardTotal` and `HandSoftTotal` objects.

Also, `Hand.altTotal` is set to `Hand.hard`.

If `card` is provided, then use the `add()` method to add this card to the hand..

39.6.3 Methods

`Hand.add(self, card)`

Parameters **card** (Card) – A card to hadd

Add this card to the `Hand.cards` list.

Evaluate `Card.setAltTotal()` to update the hard vs. soft total calculation for this hand.

`Hand.value(self)` → int
Computes the alternate total of this hand using the `Hand.altTotal` object.

If there are any aces, and the soft total is 21 or less, this will be the soft total. If there are no aces, or the soft total is over 21, this will be the hard total.

`Hand.size(self)` → int
Returns the number of cards in the hand, the size of the `List`.

`Hand.blackjack(self)` → bool
Returns true if this hand has a size of two and a value of 21.

`Hand.busted(self)` → bool
Returns true if this hand a value over 21.

`Hand.__iter__(self)` → iter
Returns an iterator over the cards of the `List`.

`Hand.__str__(self)` → str
Displays the content of the hand as a String with all of the card names.

39.7 Hand Deliverables

There are six deliverables for this exercise.

- The `HandTotal` class hierarchy: `HandTotal`, `HandHardTotal`, `HandSoftTotal`.
- A unit test for each of these classes.
- The `Hand` class.

- A class which performs a unit tests of the *Hand* class. The unit test should create several instances of *Card*, FaceCard and AceCard, and add these to instances of *Hand*, to create various point totals.
- The *Card* and AceCard modifications required to set the appropriate values in a *Hand*
- A set of unit tests for assembling a hand and changing the total object in use to correctly compute hard or soft totals for the hand.

CHAPTER

FORTY

BLACKJACK TABLE CLASS

The bets in Blackjack are associated with a hand. A player may have more than one hand in play. This will lead us to create a subclass of table to handle this complexity. In order to manage the relationships between hand and bet, we'll rework hand, also.

We'll look at the table in *Blackjack Table Analysis*.

Based on the classes defined so far, we can look at the design for the table in *BlackjackTable Class*.

We'll look at some minor rework to `Hand` in *Hand Rework*.

In *Blackjack Table Deliverables* we'll enumerate the deliverables for this chapter.

40.1 Blackjack Table Analysis

When we look at the game of Blackjack, we note that a player's `Hand` can be split. In some casinos, resplits are allowed, leading to the possibility of 3 or more `Hand`s. Each individual `Hand` has a separate ante `Bet` and seperate resolution. This is different from the way bets are currently managed for Roulette and Craps.

We have several alternatives.

- Assign responsibility to the `Table` to keep track of bets by `Player` tied to the various `Hand`s.

  This would make the `Hand` a potential key into a Map that associated a `Hand` with a `Bet`. However, `Hand`s change state, making them poor choices for keys to a Map.

- We could put a reference to a `Hand` into the `Bet`. In this way, as each `Bet` is resolved, the relevant `Hand` can be evaluated.
- We could put a reference to the Ante `Bet` in the `Hand`. In this way, as each `Hand` is resolved, the relevant `Bet` can be paid or lost.

We'll design `Hand` to contain the associated ante `Bet`. This is least disruptive to the `Bet` which is a simple thing used widely in other games.

Additional Bets. While most `Bet`s are associated with a specific `Hand`, the insurance `Bet` is always resolved before an additional hand can be created. There doesn't seem to be an essential association between the initial `Hand` and the insurance `Bet`. We can treat insurance as a `Bet` that follows the model established for Craps and Roulette.

Currently, `Bet`s are placed on the `Table`. If we create a subclass named `BlackjackTable` that uses a `Hand` when creating a `Bet`, we can have this method do both tasks: it can attach the `Bet` to the `Hand`, and it can save the `Bet` on the `Table`.

40.2 BlackjackTable Class

class **`BlackjackTable`**

BlackjackTable is a *Table* that handles the additional association between *Bet*s and specific *Hand*s in Blackjack.

40.2.1 Constructors

BlackjackTable.**__init__**(*self*)
Uses the superclass constructor to create an empty *Table*.

40.2.2 Methods

BlackjackTable.**placeBet**(*self*, *bet*, *hand*)

Parameters

- **bet** (Bet) – A bet for this hand; an ante bet is required. An insurance bet, even money bet or double down bet is optional.
- **hand** (Hand) – A hand on which the player is creating a Bet.

Updates the given hand to reference the given bet. Then uses the superclass *placeBet()* to add this bet to the list of working bets.

BlackjackTable.**__str__**(*self*) → str
Provides a nice string display of the state of the table.

40.3 Hand Rework

Hand contains a collection of individual *Card*s, and determines an appropriate total point value for the hand.

We need to add a field and some appropriate methods for associating a Bet with a Hand.

40.3.1 Fields

Hand.**ante**
Holds a reference to the ante *Bet* for this hand. When this hand is resolved, the associated bet is paid and removed from the table.

40.3.2 Methods

We have two implementation choices here. We'll show these as setters and getters. However, it's common to make these both properties of a hand.

setBet(*self*, *ante*)

Parameters **ante** (Bet) – The initial bet required to play

Sets the ante *Bet* that will be resolved when this hand is finished.

getBet(*self*) → Bet
Returns the ante *Bet* for this hand.

Here's the alternative implementation. We can use properties for this feature.

Properties for getter and setter

```
class Hand:

    @property
    def bet( self ):
        return self.ante

    @bet.setter
    def bet(self, bet ):
        self.ante = bet
```

In this example, we've created a property, `bet`, so that can write code like this: `h.bet` to fetch the bet associated with the hand.

By itself, this isn't too useful. The setter property, however, allows us to write code like this `h.bet = Bet("Ante",1)`. We can then implement any additional processing in the hand that needs to be done when the bet is changed.

40.4 Blackjack Table Deliverables

There are four deliverables for this exercise.

- The revised *Hand* class.
- A class which performs a unit tests of the *Hand* class. The unit test should create several instances of *Card*, `FaceCard` and `AceCard`, and add these to instances of *Hand*, to create various point totals. Additionally, the unit test should create a *Bet* and associate it with the *Hand*.
- The *BlackjackTable* class.
- A class which performs a unit tests of the *BlackjackTable* class. The unit test should create several instances of *Hand* and *Bet* to create multiple *Hand*s, each with unique *Bet*s.

CHAPTER

FORTYONE

BLACKJACK GAME CLASS

In *Blackjack Game Analysis* we'll look at the game and how the state of play is going to be managed.

The game offers a large number of decisions to a player. This is different from Roulette, where the player's choices are limited to a list of bets to place.

In *Blackjack Collaboration* we'll look at the various player interactions. In *Insurance Collaboration* we'll look at the insurance bet. In *Filling the Hands* we'll look at the hit vs. stand decision. There are some additional decisions – like splitting – which require more interaction. We'll look at this in *Hand-Specific Decisions*.

We'd like to segregate dealer rules from the rest of the game. This allows us to alter how a dealer fills their hand without breaking anything else. We'll look at this in *Dealer Rules*.

This will require a stub class for a Blackjack Player. We'll look at this design in *BlackjackPlayer Class*.

We'll tweak the design for `Card` in order to determine if insurance should be offered. This covered in *Card Rework*.

We'll also revisit the fundamental relationship between Game, Hand and Player. We'll invert our viewpoint from the Player containing a number of Hands to the Hands sharing a common Player. This is the subject of *Hand Rework*.

We'll provide the design for the game in *BlackjackGame Class*. In *Blackjack Game Deliverables* we'll enumerate all of the deliverables.

41.1 Blackjack Game Analysis

The sequence of operations in the game of Blackjack is quite complex. We can describe the game in either of two modes: as a sequential procedure or as a series of state changes.

- A sequential description means that the state is identified by the step that is next in the sequence.
- The state change description is what we used for Craps, see *Craps Game*. Each state definition included a set of methods that represented conditions that could change state; and each `Throw` object invoked one of those methods, in effect, announcing a condition that caused a state change.

Additionally, we need to look at the various collaborations of the Game. We also need to address the question of handling the dealer's rules.

Maintaining State. The sequential description of state, where the current state is defined by the step that is next, is the default description of state in most programming languages. While it seems obvious beyond repeating, it is important to note that each statement in a method changes the state of the application; either by changing state of the object that contains the method, or invoking methods of other, collaborating objects. In the case of an *active* class, this description of state as next statement is adequate. In the case of a *passive* class, this description of state doesn't work out well because passive classes have their state changed by collaborating objects. For passive objects, instance variables and state objects are a useful way to track state changes.

In the case of Roulette, the cycle of betting and the game procedure were a simple sequence of actions. In the case of Craps, however, the game was only loosely tied to each the cycle of betting and throwing the dice, making the game

state a passive part of the cycle of play. In the case of Blackjack, the cycle of betting and game procedure are more like Roulette.

Most of a game of Blackjack is simply sequential in nature: the initial offers of even money, insurance and splitting the hands are optional steps that happen in a defined order. When filling the player's `Hand`s, there are some minor sub-states and state changes. Finally, when all of the player's `Hand`s are bust or standing pat, the dealer fills their hand. Finally, the hands are resolved with no more player intervention.

Most of the game appears to be a sequence of offers from the `Game` to the `BlackjackPlayer`; these are offers to place bets, or accept cards, or a combination of the two, for each of the player's `Hand`s.

41.2 Blackjack Collaboration

In Craps and Roulette, the `Player` was the primary collaborator with the Game. In Blackjack, however, focus shifts from the `Player` to the `Hand`. This changes the responsibilities of a `BlackjackPlayer`: the `Hand` can delegate certain offers to the `BlackjackPlayer` for a response. The `BlackjackPlayer` can become a plug-in strategy to the `Hand`, providing responses to offers of insurance, even money, splitting, doubling-down, hitting and standing pat. The `BlackjackPlayer`'s response will change the state of the `Hand`. Some state changes involve getting a card, and others involve placing a bet, and some involve a combination of the two.

We'll use the procedure definition in *A Walkthrough*. Following this procedure, we see the following methods that a `Hand` and a `BlackjackPlayer` will need to respond to the various offers from the `BlackjackGame`. The first portion of the game involves the `BlackjackPlayer`, the second portion invovles one or more `Hand`s.

The collaboration is so intensive, we have created a kind of swimlane table. This table shows the operations each object must perform. This will allow us to expand `Hand` and `BlackjackTable` as well as define the interface for `BlackjackPlayer`.

Table 41.1: Blackjack Overall Collaboration

BlackjackGame	Hand	BlackjackPlayer	BlackjackTable
calls player's placeBet		creates empty Hand, creates initial Ante Bet	accepts the ante bet, associate with the hand
gets the initial hand; deal 2 cards to hand	add cards	returns the initial hand	
deal 2 cards to dealer's hand	add cards		
gets up card from dealer's hand; if this card requires insurance, do the insurance procedure	return up card		
iterate through all hands; is the given hand splittable?	return true if two cards of the same rank	returns a list iterator	
if the hand is splittable, offer a split bet	get the player's response; return it to the game	to split: create a split bet, and an empty hand; return the new hand	accept the split bet for the new hand
if splitting, move card and deal cards; loop back, looking for split offers	take a card out; add a card		
iterate through all hands; if the hand is less than 21, do the fill-hand procedure		returns a list iterator	
while the dealer's point value is 16 or less, deal another card	return point value of the hand; add a card		
if the dealer busts, iterate through all hands resolving the ante as a winner	return point value of the hand	returns a list iterator	resolve bet as winner and remove
if the dealer does not bust, iterate through all hands comparing against the dealer's points, determining win, loss or push	return point value of the hand	returns a list iterator	resolve bet and remove

There are a few common variation in this procedure for play. We'll set them aside for now, but will visit them in *Variant Game Rules*.

41.2.1 Insurance Collaboration

The insurance procedure involves additional interaction between `Game` and the the `Player`'s initial `Hand`. The following is done only if the dealer is showing an ace.

Table 41.2: Blackjack Insurance Collaboration

BlackjackGame	Hand	Blackjack-Player	Blackjack-Table
if player's hand is blackjack: offer even money	return true if 2 cards, soft 21	to accept, return true	
if player accepted even money offer: change bet, resolve; end of game			update bet; resolve and remove bet
offer insurance		to accept, create new bet; return true	accept insurance bet
if player accepted insurance offer: check dealer's hand; if blackjack, insurance wins, ante loses, game over; otherwise insurance loses	return point value		resolve and remove bet

41.2.2 Filling the Hands

The procedure for filling each *Hand* involves additional interaction between *Game* and the the *Player*'s initial *Hand*. An `Iterator` used for perform the following procedure for each individual player *Hand*.

Table 41.3: Blackjack Fill-Hand Collaboration

BlackjackGame	Hand	BlackjackPlayer	Black-jackTable
if player's hand is blackjack: resolve ante bet	return true if 2 cards, soft 21		resolve and remove bet
while points less than 21, offer play options of double or hit; rejecting both offers is a stand.	return point value; pass offers to player	to double, increase the bet for this hand and return true; to hit, return true	update bet
if over 21, the hand is a bust	return point value		resolve the ante as a loss

There is some variation in this procedure for filling *Hand*s. The most common variation only allows a double-down when the *Hand* has two cards.

41.2.3 Hand-Specific Decisions

Some of the offers are directly to the *BlackjackPlayer*, while others require informing the *BlackjackPlayer* which of the player's *Hand*s is being referenced.

How do we identify a specific hand?

- One choice is to have the *BlackjackGame* make the offer to the *Hand*. The *Hand* can pass the offer to the *BlackjackPlayer*; the *Hand* includes a reference to itself.
- An alternative is to have the *BlackjackGame* make the offer directly to the *BlackjackPlayer*, including a reference to the relevant *Hand*.

While the difference is minor, it seems slightly more sensible for the *BlackjackGame* to make offers directly to the *BlackjackPlayer*, including a reference to the relevant *Hand*.

41.3 Dealer Rules

In a sense, the dealer is a special player. They have a fixed set of rules for hitting and standing. They are not actually offered an insurance bet, nor can they split or double down.

However, the dealer does participate in the hand-filling phase of the game, deciding to hit or stand pat.

The dealer's rules are quite simple. Should the Dealer be a special subclass of `BlackjackPlayer`; one that implements only the dealer's rules?

Or should the Dealer be a feature of the Game. In this case, the Game would maintain the dealer's Hand and execute the card-filling algorithm.

Using an subclass of `BlackjackPlayer` is an example of **Very Large Hammer** design pattern. We only want a few features of the `BlackjackPlayer` class.

Refactoring. To avoid over-engineering these, we could refactor player into two components. An object that handles hand-filling, and an object that handles betting strategies.

The dealer would only use the hand-filling component of a player.

Mutability. To avoid over-engineering this, we can look at features that are likely to change. The dealer hand-filling rules seem well-established throughout the industry.

Further, a change to the hand-filling rules of the dealer would change the nature of the game enough that we would be hard-pressed to call in Blackjack. A different hand-filling rule would constitute a new kind of game.

We're confident, then, that the dealer's hand can be a feature of the `BlackjackGame` class.

41.4 BlackjackPlayer Class

class `BlackjackPlayer`

`BlackjackPlayer` is a subclass of `Player` that responds to the various queries and interactions with the game of Blackjack.

41.4.1 Fields

`BlackjackPlayer.hand`

Some kind of `List` which contains the initial `Hand` and any split hands that may be created.

41.4.2 Constructors

`BlackjackPlayer.__init__(self, table)`

Parameters **`table`** (BlackjackTable) – The table on which bets are placed

Uses the superclass to construct a basic `Player`. Uses the `newGame()` to create an empty List fot the hands.

41.4.3 Methods

`BlackjackPlayer.newGame(self)`

Creates a new, empty list in which to keep `Hand`s.

`BlackjackPlayer.placeBets(self)`

Creates an empty `Hand` and adds it to the List of `Hand`s.

Creates a new ante Bet. Updates the `Table` with this `Bet` on the initial `Hand`.

`BlackjackPlayer.`**`getFirstHand`**`(`*self*`)`
Returns the initial *Hand*. This is used by the pre-split parts of the Blackjack game, where the player only has a single *Hand*.

`BlackjackPlayer.`**`__iter__`**`(`*self*`)` → iter
Returns an iterator over the List of *Hand*s this player is currently holding.

`BlackjackPlayer.`**`evenMoney`**`(`*self*, *hand*`)` → bool

Parameters **hand** (Hand) – the hand which is offered even money

Returns `true` if this Player accepts the even money offer. The superclass always rejects this offer.

`BlackjackPlayer.`**`insurance`**`(`*self*, *hand*`)` → bool

Parameters **hand** (Hand) – the hand which is offered insurance

Returns `true` if this Player accepts the insurance offer. In addition to returning true, the Player must also create the Insurance *Bet* and place it on the *BlackjackTable*. The superclass always rejects this offer.

`BlackjackPlayer.`**`split`**`(`*self*, *hand*`)` → Hand

Parameters **hand** (Hand) – the hand which is offered an opportunity to split

If the hand has two cards of the same rank, it can be split. Different players will have different rules for determine if the hand should be split ot not.

If the player's rules determine that it wil accepting the split offer for the given *Hand*, *hand*, then the player will

1. Create a new Ante bet for this hand.
2. Create a new one-card *Hand* from the given *hand* and return that new hand.

If the player's rules determine that it will not accept the split offer, then `None` or `null` is returned.

If the hand is split, adding cards to each of the resulting hands is the responsibility of the Game. Each hand will be played out independently.

`BlackjackPlayer.`**`doubleDown`**`(`*self*, *hand*`)` → boolean

Parameters **hand** (Hand) – the hand which is offered an opportunity to double down

Returns `true` if this Player accepts the double offer for this *Hand*. The Player must also update the *Bet* associated with this *Hand*. This superclass always rejects this offer.

`BlackjackPlayer.`**`hit`**`(`*self*, *hand*`)` → boolean

Parameters **hand** (Hand) – the hand which is offered an opportunity to hit

Returns `true` if this Player accepts the hit offer for this *Hand*. The superclass accepts this offer if the hand is 16 or less, and rejects this offer if the hand is 17 more more. This mimics the dealer's rules.

Failing to hit and failing to double down means the player is standing pat.

`BlackjackPlayer.`**`__str__`**`(`*self*`)` → str
Displays the current state of the player, and the various hands.

41.5 Card Rework

Card must provide the Game some information required to offer insurance bets.

We'll need to add an `offerInsurance()` method on the class *Card*. The *Card* superclass must respond with False. This means that the `FaceCard` subclass will also respond with False.

The `AceCard` subclass, however, must respond with True to this method.

41.6 Hand Rework

Hand should retain some additional hand-specific information. Since some game allow resplitting of split hands, it's helpful to record whether or not a player has declined or accepted the offer of a split.

41.6.1 Fields

Hand.**player**

Holds a reference to the *Player* who owns this hand. Each of the various offers from the *Game* are delegated to the *Player*.

Hand.**splitDeclined**

Set to `true` if split was declined for a splittable hand. Also set to `true` if the hand is not splittable. The split procedure will be done when all hands return `true` for split declined.

41.6.2 Methods

Hand.**splittable**(*self*) → bool

Returns `true` if this hand has a size of two and both *Card*s have the same rank. Also sets *Hand.splitDeclined* to `true` if the hand is not splittable.

Hand.**getUpCard**(*self*) → Card

Returns the first *Card* from the list of cards, the up card.

41.7 BlackjackGame Class

class **BlackjackGame**

BlackjackGame is a subclass of *Game* that manages the sequence of actions that define the game of Blackjack.

Note that a single cycle of play is one complete Blackjack game from the initial ante to the final resolution of all bets. Shuffling is implied before the first game and performed as needed.

41.7.1 Fields

BlackjackGame.**shoe**

This is the dealer's `Shoe` with the available pool of cards.

BlackjackGame.**dealer**

This is the dealer's *Hand*.

41.7.2 Constructors

BlackjackGame.**__init__**(*self*, *shoe*, *table*)

Parameters

- **shoe** (*Shoe*) – The dealer's shoe, populated with the proper number of decks
- **table** (BlackjackTable) – The table on which bets are placed

Constructs a new *BlackjackGame*, using a given `Shoe` for dealing *Card*s and a *BlackjackTable* for recording *Bet*s that are associated with specific *Hand*s.

41.7.3 Methods

`BlackjackGame.cycle(self)`

A single game of Blackjack. This steps through the following sequence of operations.

1. Call `BlackjackPlayer.newGame()` to reset the player. Call `BlackjackPlayer.getFirstHand()` to get the initial, empty `Hand`. Call `Hand.add()` to deal two cards into the player's initial hand.
2. Reset the dealer's hand and deal two cards.
3. Call `BlackjackGame.hand.getUpCard()` to get the dealer's up card. If this card returns `true` for the `Card.offerInsurance()`, then use the `insurance()` method.

   Only an instance fo the subclass `AceCard` will return true for `offerInstance()`. All other `Card` classes will return false.
4. Iterate through all `Hand`s, assuring that no hand it splittable, or split has been declined for all hands. If a hand is splittable and split has not previously been declined, call the `Hand`'s `split()` method.

   If the `split()` method returns a new hand, deal an additional Card to the original hand and the new split hand.
5. Iterate through all `Hand`s calling the `fillHand()` method to check for blackjack, deal cards and check for a bust. This loop will finish with the hand either busted or standing pat.
6. While the dealer's hand value is 16 or less, deal another card. This loop will finish with the dealer either busted or standing pat.
7. If the dealer's hand value is bust, resolve all ante bets as winners. The `OutcomeAnte` should be able to do this evaluation for a given `Hand` compared against the dealer's bust.
8. Iterate through all hands with unresolved bets, and compare the hand total against the dealer's total. The `OutcomeAnte` should be able to handle comparing the player's hand and dealer's total to determine the correct odds.

`BlackjackGame.insurance(self)`

Offers even money or insurance for a single game of blackjack. This steps through the following sequence of operations.

1. Get the player's `BlackjackPlayer.getFirstHand()`. Is it blackjack?

   If the player holds blackjack, then call `BlackjackPlayer.evenMoney()`.

   If the even money offer is accepted, then move the ante bet to even money at 1:1. Resolve the bet as a winner. The bet will be removed, and the game will be over.
2. Call `BlackjackPlayer.insurance()`. If insurance declined, this method is done.
3. If insurance was accepted by the player, then check the dealer's hand. Is it blackjack?

   If the dealer hold blackjack, the insurance bet is resolved as a winner, and the ante is a loser; the bets are removed and the game will be over.

   If the dealer does not have blackjack, the insurance bet is resolved as a loser, and the ante remains.

   If insurance was declined by the player, nothing is done.

`BlackjackGame.fillHand(self, hand)`

Parameters **hand** (Hand) – the hand which is being filled

Fills one of the player's hands in a single game of Blackjack. This steps through the following sequence of operations.

1. While points are less than 21, call `BlackjackPlayer.doubleDown()` to offer doubling down. If accepted, deal one card, filling is done.

If double down is declined, call `BlackjackPlayer.hit()` to offer a hit. If accepted, deal one card. If both double down and hit are declined, filling is done, the player is standing pat.

2. If the points are over 21, the hand is bust, and is immediately resolved as a loser. The game is over.

`BlackjackGame.__str__(self)` → str
Displays the current state of the game, including the player, and the various hands.

41.8 Blackjack Game Deliverables

There are eight deliverables for this exercise.

- The stub `BlackjackPlayer` class.
- A class which performs a unit test of the `BlackjackPlayer` class. Since this player will mimic the dealer, hitting a 16 and standing on a 17, the unit test can provide a variety of `Hand` s and confirm which offers are accepted and rejected.
- The revised `Hand` class.
- A class which performs a unit tests of the `Hand` class. The unit test should create several instances of `Card`, `FaceCard` and `AceCard`, and add these to instances of `Hand`, to create various point totals. Since this version of `Hand` interacts with a `BlackjackPlayer`, additional offers of split, double, and hit can be made to the `Hand`.
- The revised `Card` class.
- Revised unit tests to exercise the `Card.offerInsurance()` method.
- The revised `BlackjackGame` class.
- A class which performs a unit tests of the `BlackjackGame` class. The unit test will have to create a `Shoe` that produces cards in a known sequence, as well as `BlackjackPlayer`. The `cycle()` method, as described in the design, is too complex for unit testing, and needs to be decomposed into a number of simpler procedures.

CHAPTER

FORTYTWO

SIMPLE BLACKJACK PLAYER CLASS

Our objective is to provide variant player strategies. This chapter will upgrade our stub player class to give it a complete, working strategy. This simple player can serve as the superclass for more sophisticated strategies.

We'll look at a simple strategy for play in *Blackjack Player Analysis*.

In *Decision By Lookup* we'll look at an alternative design. We can use a mapping instead of if statements.

In *SimpleBlackjackPlayer Design* we'll design the player. This will rework some of the draft designs we've used earlier. We'll list all of the deliverables in *Blackjack Player Deliverables*.

42.1 Blackjack Player Analysis

In addition to the player's own hand, the player also has the dealer's up card available for determining their response to the various offers. The player has two slightly different goals: not bust and have a point total larger than the dealer's. While there is some overlap between these goals, these lead to two strategies based on the dealer's up card. When the dealer has a relatively low card (2 through 6), the dealer has an increased probability of going bust, so the player's strategy is to avoid going bust. When the dealer has a relatively high card (7 through 10), the dealer will probably have a good hand, and the player has to risk going bust when looking for a hand better than the dealer's.

A Simple Strategy. We'll provide a few rules for a simple player strategy. This strategy is not particularly good. Any book on Blackjack, and a number of web sites, will have a superior strategy. A better strategy will also be considerably more complex. We'll implement this one first, and leave it to the student to research more sophisticated strategies.

1. Reject insurance and even money offers.
2. Accept split for aces and eights. Reject split on other pairs.
3. Hit any hand with 9 or less. The remaining rules are presented in the following table.

Table 42.1: Blackjack Player Strategy

Player Shows	2-6	7-10, Ace
10 or 11	hit	double down
hard 12 to 16	stand	hit
soft 12 to 16	hit	hit
17 to 21	stand	stand

These rules will boil down to short sequences of if-statements in the `split()`, `hit()` and `doubleDown()` methods.

In some contexts, complex if-statements are deplorable. Often, a sequence of complex if-statements is a stand-in for proper allocation of responsibility. In this class, however, the complex if-statements implement a kind of index or lookup scheme.

If we want to eliminate the if-statements, what can we do?

42.1.1 Decision By Lookup

We have identified eight alternatives which depend on a two-dimensional index.

- One dimension contains four conditions that describe the player's hand.
- The other dimension involves two conditions that describe the dealer's hand.

When we look at the various collections, we see that we can index into a mapping using any kind of immutable object instance. In this case, we can index by objects that represent the various conditions. We would have to map each condition to a distinct object.

When we look at the conditions that describe the player's hand, these are clearly state-like objects. Each card can be examined and a state transition can be made based on the the current state and the card. After accepting a card, we would check the total and locate the appropriate state object. We can then use this state object to index into a collection of player choices.

When we look at the conditions that describe the dealer's hand, there are only two state-like objects. The dealer's op card can be examined, and we can locate the appropriate state object. We can use this state object to index into a collection.

The final strategy could be modeled as a collection with a two-part index. This can be nested collection objects, or a Map that uses a 2-valued tuple as an index.

Why use a mapping instead of a lot of if-statements?

We can easily change the decision by tweaking a cell in the mapping.

42.2 SimpleBlackjackPlayer Design

class **`SimpleBlackjackPlayer`**

`SimpleBlackjackPlayer` is a subclass of *`BlackjackPlayer`* that responds to the various queries and inteactions with the game of Blackjack.

This player implements a very simple strategy, shown above in the Blackjack Player Strategy table.

42.2.1 Methods

`SimpleBlackjackPlayer.`**`evenMoney`**`(`*self*, *hand*`)` → boolean

Parameters **`hand`** (Hand) – the hand which is being offered even money

Returns `true` if this Player accepts the even money offer. This player always rejects this offer.

`SimpleBlackjackPlayer.`**`insurance`**`(`*self*, *hand*`)` → boolean

Parameters **`hand`** (Hand) – the hand which is being offered insurance

Returns `true` if this Player accepts the insurance offer. This player always rejects this offer.

`SimpleBlackjackPlayer.`**`split`**`(`*self*, *hand*`)` → Hand

Parameters **`hand`** (Hand) – the hand which is being offered the opportunity to split

Returns a new, empty *`Hand`* if this Player accepts the split offer for this *`Hand`*. The Player must create a new *`Hand`*, create an Ante *`Bet`* and place the on the new *`Hand`* on the *`BlackjackTable`*.

If the offer is declined, both set *`Hand.splitDeclined`* to `true` and return `null`.

This player splits when the hand's card's ranks are aces or eights, and declines the split for all other ranks.

`SimpleBlackjackPlayer.`**`doubleDown`**`(`*self*, *hand*`)` → boolean

Parameters **`hand`** (Hand) – the hand which is being offered the opportunity to double down

Returns `true` if this Player accepts the double offer for this *Hand*. The Player must also update the *Bet* associated with this *Hand*.

This player tries to accept the offer when the hand points are 10 or 11, and the dealer's up card is 7 to 10 or ace. Otherwise the offer is rejected.

Note that some games will restrict the conditions for a double down offer. For example, some games only allow double down on the first two cards. Other games may not allow double down on hands that are the result of a split.

`SimpleBlackjackPlayer.`**`hit`**`(`*self*, *hand*`)` → boolean

Parameters **`hand`** (Hand) – the hand which is being offered the opportunity to hit

Returns `true` if this Player accepts the hit offer for this *Hand*.

If the dealer up card is from 2 to 6, there are four choices for the player. When the hand is 11 or less, hit. When the hand is a hard 12 to 16, stand. When the hand is a soft 12 to soft 16 (hard 2 to hard 6), hit. When the hand is 17 or more, stand.

If the dealer up card is from 7 to 10 or an ace, there are four choices for the player. When the hand is 11 or less, double down. When the hand is a hard 12 to 16, hit. When the hand is a soft 12 to soft 16 (hard 2 to hard 6), hit. When the hand is 17 or more, stand.

Otherwise, if the point total is 9 or less, accept the hit offer.

42.3 Blackjack Player Deliverables

There are two deliverables for this exercise.

- The *SimpleBlackjackPlayer* class.
- A class which performs a unit test of the *SimpleBlackjackPlayer* class. The unit test can provide a variety of *Hand*s and confirm which offers are accepted and rejected.

CHAPTER

FORTYTHREE

VARIANT GAME RULES

There are a number of variations in the sequence of game play offered by different casinos. In addition to having variations on the player's strategy, we also need to have variations on the casino rules.

We'll look at just a few of the variants in *Variant Game Analysis*.

In order to support some of the variants, we'll rework our game class. The details are in *BlackjackGame Rework*.

We'll define a single-deck game in *OneDeckGame Class*. We won't delve into too many other variants. We'll detail the deliverables in *Variant Game Deliverables*.

43.1 Variant Game Analysis

There are wide variations in the ways people conduct Blackjack games. We'll list a few of the variations we have heard of.

- **Additional Win Rule: Charlie**. Informal games may allow a "five-card Charlie" or "six-card Charlie" win. If the player's hand stretches to five (or six) cards, they are paid at 1:1.
- **Additional Offer: Surrender**. This variation allows the player to lose only half their bet. The offer is made after insurance and before splitting. Surrender against ace and 10 is a good strategy, if this offer is part of the game.
- **No Resplit**. This variation limits the player to a single split. In the rare event of another matching card, resplitting is not allowed.
- **No Double After Split**. This variation prevents the player from a double-down after a split. Ordinarily, splitting Aces is followed by double-downs because 4/13 cards will lead to blackjack.
- **Dealer Hits Soft 17**. This variation forces the dealer to hit soft 17 instead of standing. This tends in increase the house edge slightly.
- **Blackjack Pays 6:5**. This variation is often used in single-deck games. It limits the value of card-counting, since the house edge is stacked more heavily against the player.

The restrictions on splitting and doubling down are small changes to the offers made by a variation on `BlackjackGame`. This would require creating a subclass of `BlackjackGame` to implement the alternative rules.

The variations in the dealer's rules (hitting a soft 17) is also a small change best implemented by creating a subclass of `BlackjackGame`.

Reducing the number of decks is an easy change to our application. Since our main application constructs the `Shoe` before constructing the `Game`, it can construct a single-deck `Shoe`.

Payout Odds. Handling the variations in the payout odds is a bit more complex.

1. The Player creates each Hand, associated with a Bet. The Bet is associated with the simple Ante outcome.

2. At the end of the Game, the Hand does a comparison between itself and the Dealer's Hand to determine the odds for the Ante outcome.

   It's a 1:1 Outcome if the player does not have blackjack.

   It's a 3:2 bet if the player does have blackjack.

Allocating this responsiblity to the Hand was a bad design decision.

We should have allocated responsibility to the Game. We will need to add a method to the Game which compares a player's `Hand` with the dealer's `Hand`, sets the `Outcome` correctly, and resolves the bet.

43.2 BlackjackGame Rework

`BlackjackGame` is a subclass of `Game` that manages the sequence of actions that define the game of Blackjack.

Note that a single cycle of play is one complete Blackjack game from the initial ante to the final resolution of all bets. Shuffling is implied before the first game and performed as needed.

43.2.1 Methods

`BlackJackGame.`**`adjustOdds`**`(`*self*, *hand*`)`

> **Parameters** **`hand`** (Hand) – The hand which has the ante bet odd's corrected

> This method is used at the end of the game to resolve the player's Ante bet.
>
> If the player went bust, their hand was already resolved.
>
> If the dealer went bust, all remaining bets are resolved as winners.
>
> The remaining situation (both player and dealer have non-bust hands) requires this `BlackjackGame.adjustOdds()` method. This method will set the hand's outcome's odds to 3:2 if the player holds Blackjack.

43.3 OneDeckGame Class

`OneDeckGame` is a subclass of `BlackjackGame` that manages the sequence of actions for a one-deck game of Blackjack with a 6:5 blackjack `OutcomeAnteLowOdds`.

Typically, this is built with a one-deck instance of `Shoe`.

The `OneDeckGame.adjustOdds()` method overrides the superclass. This method will set the hand's outcome's odds to 6:5 if the player holds Blackjack.

43.4 Variant Game Deliverables

There are three deliverables for this exercise.

- The revised `BlackjackGame` class. All of the original unit tests should apply to the refactored function that sets the outcome odds.
- The `OneDeckGame` class.
- A class to perform a unit test of the `OneDeckGame` class.

CHAPTER FORTYFOUR

CONCLUSION

The game of Blackjack has given us an opportunity to further extend and modify an application with a considerable number of classes and objects. These exercises gave us an opportunity to work from a base of software, extending and refining our design.

We omitted concluding exercises which would integrate this package with the `Simulator` and collect statistics. This step, while necessary, doesn't include many interesting design decisions. The final deliverable should be a working application that parses command-line parameters, creates the required objects, and creates an instance of `Simulator` to collect data.

We have specifically omitted delving into the glorious details of specific player strategies. We avoided these details because there are so many We leave it to the interested student to either buy any of the available books on Blackjack or download a strategy description from the Internet.

Indeed, one of the more interesting things this simulation can be used for is to create a machine learning environment in which something like Simulated Annealing is used to try different strategies looking for one that's optimal.

Rather than test existing strategies, create one via simulation.

Next Steps. There are a number of logical next steps for the programmers looking to build skills in object-oriented design. We'll split these along several broad fronts.

- **Additional Technology**. There are several technology directions that can be pursued for further design experience.

  Another area for building skills in design is the implementation of programs that make extensive use of a database. All of the statistical results can be accumulated in a database instead of `.csv` files for analysis.

  A graphical user interface (GUI) can be added to show that the simulation is doing something.

  A web framework can be used to provide configuration changes and run simulations. Rather than using the command line, a RESTful API can be built to provide alternative strategies or rules.

- **Application Areas**. We selected simulation because it's part of the historical foundation for object-oriented programming. We selected casino games because they have a reasonable level of complexity.

  Clearly, numerous other application areas can be selected as the basis for problems. The number and variety of human endevors that can be automated is quite large.

  Moving beyond simulation or doing simulation on something more complex than a casino table game is a good next step.

- **Additional Depth in Design Patterns**. It's possible to use additional and different design patterns to extend and refine the application that you have built.

  Any book or web site on OO design patterns will provide numerous examples of patterns. These can be used for add flexibility to these casino game simulators.

Part V

Fit and Finish

A finished application includes more than just a working program. There are two additional questions.

- How do you know it works? That is, do you have any tests that demonstrate that it works correctly?

  We address this by creating unit tests.

- How is it designed?

  We address this by creating documentation within the program's source files. This documentation can be extracted to create a tidy, complete reference for all the classes and functions within our application.

CHAPTER FORTYFIVE

PYTHON UNIT TESTING

The Python `unittest` module encourages us to build a test module separate from the application class we're testing. The test module will have one more more `TestCase` classes, each of which has one or more test methods.

In *Using Unit Tests* we'll look at an overall development process that leverages unit tests as a way to write testable specifications.

In *Dependencies* we'll touch briefly on testing dependent classes together or creating mock objects.

We'll look at an example in *Example Unit Test*. To be complete, we'll show the class which was tested in *Example Class Definition*.

In *Python doctest Testing* we'll look at how we can execute the test cases in the document strings of a module, class, function, or method.

In *Building Doctest Examples* we'll show how we can take interactive Python output and transform it into a doctest example. This will involve copy and paste. It's not too challenging.

In *Mixed unittest and doctest* we'll show how we can combine unittest tests with doctest tests.

45.1 Using Unit Tests

One approach to unit testing is to build the tests first, then write a class which at least doesn't crash, but may not pass all the tests. Once we have this in place, we can now debug the tests until everything looks right. This is called test-driven development.

We'd start with unittest files in the `test` directory of our project. In this case, we'd have `testCard.py` and `testDeck.py`. We also need skeleton files in our `src` directory: `cards.py` and `deck.py`.

Generally, the process for creating a class with the unit tests has the following outline.

1. Write a skeleton for the class that's the Unit Under Test (UUT). This class that doesn't really do anything, but at least compiles and has the right method names.
2. Write the test class. This will create instances of the class under test, exercise those instances, and make assertions about the state of those instances. The test class may create mocks for the various collaborators of the target class.
3. Run the test, knowing that the first few attempts will fail. Sometimes our class under test is so simple that we get it right the first time. Other times, the class under test is more complex and there are parts we didn't finish when we wrote the skeleton version.
4. While at least one test is failing.
   (a) Refactor. We finish our target class. Sometimes we simply fill in the missing parts. Other times, we have more serious design work to do.
   (b) Run the test suite.

5. At this point, the class under test passes the suite of tests. However, it may still fail to meet other quality criteria. For example, it may have a convoluted structure, or it may be inefficient, or it may lack appropriate documentation. In any case, we're not really done with development.
6. While our target class fails to meet our quality standards.
   (a) Refactor. We correct the quality problems in our target class.
   (b) Run the test suite. If we have refactored properly, the tests still pass. If we have introduced a problem, tests will fail.

Using unittest. A test runner will locate all the `TestCase` instances, execute them and summarize the pass/fail status of each individual test method.

Generally, the useful modules will be in package with a name like `casino_sim`. The test modules will be in a separate package named `test`.

We might have something like this:

```
casino_sim
    __main__.py
    common.py
    roulette.py
    craps.py
    blackjack.py
test
    __main__.py
    test_common.py
    test_roulette.py
    test_craps.py
    test_blackjack.py
```

(This isn't a requirement, it's just a notion. In many cases, we'll want to subdivide each game into game and player modules because they evolve independently.)

Here is an example of running a test from the command line.

```
python3 -m unittest test.test_blackjack.TestDeck
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

45.2 Dependencies

Let's assume we've built two classes in some chapter; for example, we're building *Card* and `Deck`. One class defines a standard playing card and the other class deals individual card instances. We need unit tests for each class.

Generally, unit tests are taken to mean that a class is tested in isolation. In our case, a unit test for Card is completely isolated because it has no dependencies.

However, our Deck class depends on Card, leading us to make a choice. Either we have to create a Mock Card that can be used to test Deck in complete isolation, or our Deck test will depend on both Deck and Card. The choice depends on the relative complexity of Card, and whether or not Deck and Card will evolve independently.

Some folks demand that **all** testing be done in "complete" isolation with a lot of mock classes. Other folks are less strict on this, recognizing that Deck and Card are very tightly coupled and Card is very simple. The Mock Card is almost as complex as Card.

45.3 Example Unit Test

test_card.py

```
import unittest
from blackjack import Card, AceCard, FaceCard

class TestCard(unittest.TestCase):
    def setUp(self):
        self.aceClubs= AceCard(Card.Ace, Card.Clubs)
        self.twoClubs= Card(2, Card.Clubs)
        self.tenClubs= Card(10, Card.Clubs)
        self.kingClubs= FaceCard(Card.King, Card.Clubs)
        self.aceDiamonds= AceCard(Card.Ace, Card.Diamonds)
    def testString(self):
        self.assertEqual( " A□ ", str(self.aceClubs) )
        self.assertEqual( " 2□ ", str(self.twoClubs) )
        self.assertEqual( "10□ ", str(self.tenClubs) )
        self.assertEqual( " K□ ", str(self.kingClubs) )
```

1. Generally, we create a number of object instances in the setup method. In this case, we created five distinct *Card* instances. These object constructors imply several things in our *Card* class.

   (a) There will be a set of manifest constants for the suits: `Clubs`, `Diamonds`, `Hearts` and `Spades`.

   (b) The constructor (`Card.__init__()`) will accept a rank and a suit constant.

   Note that we didn't write tests to create all suits or all ranks. In same cases, we may need to produce tests which exhaustively enumerate all possibe alternatives. We may also want to include edge and corner cases like trying to create an `AceCard` with a rank of 2.

   For the purposes of learning OO design, we only need to sketch out our class by defining the tests it must pass. We sometimes call these happy path test cases.

2. In `testString()`, we exercise the `__str__()` method of the *Card* class to be sure that it formats cards correctly. These tests tell us what the formatting algorithm will look like.

3. In `testOrder()`, we exercise some comparison methods of the *Card* class to be sure that it compares card ranks correctly. Note that we have explicitly claimed that the equality test only checks the rank and ignores the suit; this is typical for Blackjack, but won't work well for Bridge or Solitaire.

   Note that we didn't exhaustively test all possible comparisons among the four cards we defined. We tested enough to be sure we had an implementation that was good enough to get started.

4. This is the standard main program for a `unittest` module. We often include this in each test module. We'll also include a `__main__` module that includes all of the tests into a master test suite.

45.4 Example Class Definition

Our initial *Card* class needs to have just enough of an API to allow the tests to run. Here's our skeleton *Card* class.

blackjack.py

```
import sys

class Card:
```

```
    Clubs = u'\N{BLACK CLUB SUIT}'
    Diamonds = u'\N{WHITE DIAMOND SUIT}'
    Hearts = u'\N{WHITE HEART SUIT}'
    Spades = u'\N{BLACK SPADE SUIT}'
    Jack = 11
    Queen = 12
    King = 13
    Ace = 1

    def __init__(self, rank, suit):
        assert suit in (Card.Clubs, Card.Diamonds, Card.Hearts, Card.Spades)
        assert 1 <= rank < 14
        self.rank= rank
        self.suit= suit
        self.order= rank

    @property
    def hardValue(self):
        return self.rank

    @property
    def softValue(self):
        return self.rank

    def __repr__(self):
        return "{class_}({rank!r},{suit!r})".format(
            class_=type(self).__name__, **vars(self)
        )

    def __str__(self):
        return "{rank:2d}{suit}".format_map(vars(self))

    @property
    def image(self):
        s = {Card.Spades:0x1F0A0, Card.Hearts:0x1F0B0, Card.Diamonds:0x1F0C0, Card.Clubs:0x1F0D0}[se
        r = self.rank if self.rank < 12 else self.rank+1
        return chr(s+r)

    def __le__(self, other):
        return self.order <= other.order

    def __lt__(self, other):
        return self.order < other.order

    def __ge__(self, other):
        return self.order >= other.order

    def __gt__(self, other):
        return self.order > other.order

    def __eq__(self, other):
        return self.order == other.order

    def __ne__(self, other):
        return self.order != other.order

    def __hash__(self):
        return (hash(self.rank)+hash(self.suit))%sys.hash_info.width
```

This class will – minimally – participate in the testing. It won't pass many tests, but it serves as a basis for developing the class implementation.

Note that there's almost no documentation in this class. We'll address that gap in *Python Documentation*.

CHAPTER

FORTYSIX

PYTHON DOCTEST TESTING

Python **doctest** module requires us to put our test cases and expected results into the docstring comments on a class, method or function. The test case information becomes a formal part of the API documentation. When a docstring includes doctest comments, the string serves dual duty as formal test and a working example.

Workflow. To use **doctest** is to build the class, exercise it in the Python interpreter, then put snippets of the interactive log into our docstrings.

Generally, we follow this outline.

1. Write and debug the class, including docstring comments.
2. Exercise the class in an interactive Python interpreter. You can use **IDLE** or any other command line.
3. Copy the snippets out of the interactive log. Paste them into the docstring comments.
4. Run doctest to be sure that you've copied and pasted correctly.

Using Doctest. There are two ways to use doctest.

Start with Python 2.6, you can run doctest from the command line like this.

Running Doctest from the Command Line

```
python -m doctest -v roulette.py
```

You can also add a test function to a module which runs doctest on the module. This test function should have a name which begins with _ to make it a name that's private to the module and not part fo the module's interface.

Running Doctest Within a Module

```
def _test():
    import doctest
    doctest.testmod()

if __name__ == "__main__":
    _test()
```

This `_test()` function is the main function of the module, so that when you run the module, it performs it's internal doctests.

Running A Module with Doctest

```
python roulette.py
```