
Building Skills in Object-Oriented Design

Release 2.1.1-Python

Steven F. Lott

December 30, 2009

CONTENTS

I	Front Matter	3
1	Preface	5
1.1	Why Read This Book?	5
1.2	Audience	6
1.3	Organization of This Book	7
1.4	Why This Subject?	9
1.5	Programming Style	9
1.6	Conventions Used in This Book	10
1.7	Acknowledgements	11
2	Foundations	13
2.1	Problem Statement	13
2.2	Our Simulation Application	15
2.3	Soapbox on Use Cases	15
2.4	Solution Approach	16
2.5	Methodology, Technique and Process	17
2.6	Deliverables	22
II	Roulette	25
3	Roulette Details	29
3.1	Roulette Game	29
3.2	Available Bets in Roulette	29
3.3	Some Betting Strategies	32
4	Roulette Solution Overview	35
4.1	Preliminary Survey of Classes	35
4.2	Preliminary Roulette Class Structure	36
4.3	A Walkthrough of Roulette	37
4.4	Roulette Solution Questions and Answers	38
5	Outcome Class	41
5.1	Outcome Overview	41
5.2	Design Decision – Object Identity	41
5.3	Outcome Design	44
5.4	Outcome Deliverables	45
5.5	Message Formatting	45
6	Bin Class	47

6.1	Bin Overview	47
6.2	Design Decision – Choosing A Collection	47
6.3	Bin Questions and Answers	48
6.4	Bin Design	49
6.5	Bin Deliverables	50
7	Wheel Class	51
7.1	Wheel Overview	51
7.2	Wheel Design	52
7.3	Non-Random Design	53
7.4	Wheel Deliverables	54
8	Bin Builder Class	55
8.1	Bin Builder Overview	55
8.2	Bin Builder Algorithms	56
8.3	BinBuilder Design	59
8.4	Bin Builder Deliverables	59
8.5	Internationalization and Localization	60
9	Roulette Bet Class	61
9.1	Roulette Bet Overview	61
9.2	Design Decision – Create or Locate an Outcome	61
9.3	Design Decision – Where to Keep the Outcome Map	62
9.4	Roulette Bet Questions and Answers	63
9.5	Roulette Bet Design	64
9.6	Roulette Bet Deliverables	65
10	Roulette Table Class	67
10.1	Roulette Table Overview	67
10.2	InvalidBet Exception Design	68
10.3	Roulette Table Design	69
10.4	Roulette Table Deliverables	69
11	Roulette Game Class	71
11.1	Roulette Game Overview	71
11.2	Passenger57 Design	72
11.3	Roulette Game Design	73
11.4	Roulette Game Questions and Answers	74
11.5	Roulette Game Deliverables	74
11.6	Additional Roulette Design Considerations	75
12	Review of Testability	77
12.1	Testability Overview	77
12.2	Test-Driven Design	78
12.3	Capturing Pseudo-Random Data	79
12.4	Testability Questions and Answers	79
12.5	Testable Random Events Design	80
12.6	Testability Deliverables	81
13	Player Class	83
13.1	Roulette Player Overview	83
13.2	Player Design	85
13.3	Martingale Player Design	86
13.4	Player Deliverables	87

14 Overall Simulation Control	89
14.1 Simulation Control Overview	89
14.2 Simulation Terms	89
14.3 Simulator Design	91
14.4 Player Rework	93
14.5 Simulation Control Deliverables	93
15 SevenReds Player Class	95
15.1 SevenReds Player Overview	95
15.2 SevenReds Design	96
15.3 Player Rework	97
15.4 Game Rework	97
15.5 SevenReds Player Deliverables	97
16 Statistical Measures	99
16.1 Statistics Overview	99
16.2 Some Foundations	100
16.3 Statistical Algorithms	101
16.4 IntegerStatistics Design	102
16.5 Statistics Deliverables	103
17 Random Player Class	105
17.1 Random Player Overview	105
17.2 Random Player Design	106
17.3 Random Player Deliverables	106
18 Player 1-3-2-6 Class	107
18.1 Player 1-3-2-6 Overview	107
18.2 On Polymorphism	108
18.3 Player 1-3-2-6 Questions and Answers	109
18.4 Player1326 State Design	110
18.5 Player1326 No Wins Design	111
18.6 Player1326 One Win Design	111
18.7 Player1326 Two Wins Design	111
18.8 Player1326 Three Wins	112
18.9 Player1326 Design	112
18.10 Player 1-3-2-6 Deliverables	113
18.11 Advanced Exercise – Refactoring	113
18.12 Advanced Exercise – Less Object Creation	113
19 Cancellation Player Class	115
19.1 Cancellation Player Overview	115
19.2 PlayerCancellation Design	115
19.3 Cancellation Player Deliverables	116
20 Fibonacci Player Class	117
20.1 Fibonacci Player Overview	117
20.2 PlayerFibonacci Design	118
20.3 Fibonacci Player Deliverables	118
21 Conclusion	121

III Craps	123
22 Craps Details	127
22.1 Craps Game	127
22.2 Creating A Dice Frequency Distribution	128
22.3 Available Bets	128
22.4 Some Betting Strategies	131
22.5 Wrong Betting	131
23 Craps Solution Overview	133
23.1 Preliminary Survey of Classes	133
23.2 Preliminary Class Structure	134
23.3 A Walkthrough of Craps	134
23.4 Craps Solution Questions and Answers	135
24 Outcome Class	137
24.1 Outcome Overview	137
24.2 Overloaded Methods	139
24.3 Outcome Rework	139
24.4 Outcome Deliverables	140
24.5 Advanced Exercise	140
25 Throw Class	141
25.1 Throw Overview	141
25.2 Throw Design	142
25.3 Natural Throw Design	143
25.4 Craps Throw Design	144
25.5 Eleven Throw Design	144
25.6 Point Throw Design	145
25.7 Craps Game Design	146
25.8 Throw Deliverables	147
26 Dice Class	149
26.1 Dice Overview	149
26.2 Throw Rework	150
26.3 NumberPair Design	150
26.4 Dice Design	151
26.5 Dice Deliverables	152
26.6 Dice Optimization	152
27 Throw Builder Class	155
27.1 Throw Builder Overview	155
27.2 Outcomes with Variable Odds	156
27.3 Refactoring The Outcome Hierarchy	157
27.4 Soapbox on Subclasses	158
27.5 Soapbox on Architecture	159
27.6 Throw Builder Questions and Answers	161
27.7 Soapbox on Justification	161
27.8 Design Light	163
27.9 Design Heavy	163
27.10 Common Design	164
27.11 Throw-Builder Deliverables	166
28 Bet Class	167
28.1 Bet Overview	167

28.2	Bet Rework	167
28.3	CommissionBet Design	168
28.4	Bet Deliverables	168
29	Craps Table Class	169
29.1	Craps Table Overview	169
29.2	Design Decision – Table vs. Game Responsibility	170
29.3	Design Decision – Allowable Outcomes	170
29.4	CrapsGame Stub	171
29.5	CrapsTable Design	171
29.6	Craps Table Deliverables	172
30	CrapsGame Class	173
30.1	Game State	173
30.2	Game State Class Hierarchy	175
30.3	Resolving Bets	176
30.4	Moveable Bets	177
30.5	Design Decision – Win, Lose, Wait	178
30.6	Additional Craps Design	179
30.7	Craps Game Implementation Overview	180
30.8	Throw Rework	180
30.9	ThrowBuilder Rework	182
30.10	Bet Rework	183
30.11	CrapsPlayer Class Stub	183
30.12	CrapsGameState Class	184
30.13	CrapsGamePointOff Class	185
30.14	CrapsGamePointOn Class	186
30.15	CrapsGame Class	188
30.16	Craps Game Deliverables	190
30.17	Optional Working Bets	190
31	CrapsPlayer Class	191
31.1	Craps Player Overview	191
31.2	CrapsPlayer Superclass Design	192
31.3	CrapsPlayerPass Subclass	193
31.4	Craps Martingale Subclass	193
31.5	Craps Player Deliverables	194
32	Design Cleanup and Refactoring	195
32.1	Design Review	195
32.2	RandomEventFactory Design	199
32.3	Wheel Class Design	200
32.4	Table Class	201
32.5	Game Class	203
32.6	RouletteGame Class	204
32.7	CrapsGame Class	204
32.8	Refactoring Deliverables	205
33	Simple Craps Players	207
33.1	Simple Craps Players Overview	207
33.2	CrapsPlayer Design	208
33.3	CrapsSimplePlayer superclass	208
33.4	Craps Martingale Player	209
33.5	Player1326 State	210
33.6	Craps1326 Player	211

33.7	CrapsCancellation Player	212
33.8	Simple Craps Players Deliverables	212
34	Roll-Counting Player Class	215
34.1	Roll-Counting Overview	215
34.2	BettingStrategy Design	217
34.3	NoChangeBetting Class	217
34.4	MartingaleBetting Class	218
34.5	Bet1326Betting Class	219
34.6	CrapsOneBetPlayer class	220
34.7	CrapsTwoBetPlayer class	221
34.8	CrapsSevenCountPlayer class	222
34.9	Roll-Counting Deliverables	223
35	Conclusion	225
IV	Blackjack	227
36	Blackjack Details	231
36.1	Blackjack Game	231
36.2	Available Bets and Choices	233
36.3	Betting Strategies	233
37	Blackjack Solution Overview	235
37.1	Preliminary Survey of Classes	235
37.2	Preliminary Class Structure	236
37.3	A Walkthrough	236
37.4	Blackjack Solution Questions and Answers	238
38	Card, Deck and Shoe Classes	239
38.1	Card, Deck and Shoe Overview	239
38.2	Card-Deck-Shoe Questions and Answers	240
38.3	Card Superclass	241
38.4	FaceCard Class	243
38.5	AceCard Class	243
38.6	Deck class	243
38.7	Shoe class	244
38.8	Card-Deck-Shoe Deliverables	245
39	Hand and Outcome Classes	247
39.1	Hand Overview	247
39.2	Hand Total Class Design	250
39.3	Hand Hard Total Class Design	250
39.4	Hand Soft Total Class Design	250
39.5	Card Class Updates	251
39.6	Hand Class Design	251
39.7	Hand Deliverables	253
40	Blackjack Table Class	255
40.1	Blackjack Table Overview	255
40.2	BlackjackTable Class	255
40.3	Hand Rework	256
40.4	Blackjack Table Deliverables	256

41 Blackjack Game Class	259
41.1 Blackjack Game Overview	259
41.2 Blackjack Collaboration	260
41.3 Dealer Rules	262
41.4 BlackjackPlayer Class	262
41.5 Card Rework	264
41.6 Hand Rework	264
41.7 BlackjackGame Class	264
41.8 Blackjack Game Deliverables	266
42 Simple Blackjack Player Class	269
42.1 Blackjack Player Overview	269
42.2 SimpleBlackjackPlayer Design	270
42.3 Blackjack Player Deliverables	271
43 Variant Game Rules	273
43.1 Variant Game Overview	273
43.2 BlackjackGame Rework	274
43.3 OneDeckGame Class	274
43.4 Variant Game Deliverables	274
44 Conclusion	275
 V Fit and Finish	 277
45 Python unittest Testing	281
45.1 Dependencies	282
45.2 Example	282
46 Python doctest Testing	285
46.1 Example	286
46.2 Add the Test Framework	288
46.3 Mixed unittest and doctest	288
47 Python Documentation	291
47.1 Basic RST Markup	292
47.2 RST Field Markup	294
47.3 Class Example	296
 VI Back Matter	 299
48 Bibliography	301
48.1 Use Cases	301
48.2 Computer Science	301
48.3 Design Patterns	301
48.4 Statistics	301
48.5 Python	301
48.6 Java	301
48.7 Casino Games	301
49 Toolset	303
50 Indices and tables	305

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

PREFACE

Pensés, The Provincial Letters: Provincial letter 16, p. 571.

The present letter is a very long one, simply because I had no leisure to make it shorter.

– BLAISE PASCAL

1.1 Why Read This Book?

The coffee-shop reason for reading this book is to provide the beginning designer with a sequence of interesting and moderately complex exercises in OO design.

If that's all you needed to know, skip to the next chapter. It's okay. We don't mind.

The Problem. Some software developers find themselves stalled when trying to do object-oriented (OO) design. As programmers, they've understood the syntax of a programming language, and pieced together small examples. However, it is often difficult to take the next step to becoming a designer. The transition from guided learning of language features to self-directed design work is often ignored. Programmers are left to struggle through their first design projects without appropriate skills or support.

This may be you. You've learned the language, but you can't take the next step.

While it is critically important to read examples of good design, a finished product doesn't reveal the author's decision-making process that created the design. There's little support that helps a programmer come to understand the design process that leads to a final product.

The most notable consequence of this skills gap is some n00b programmers will create of software that is far more complex than necessary to effectively solve a given problem. This, in turn, leads to software with high maintenance costs stemming from the low quality. It also leads to an unfair indictment of OO technology; this is usually voiced as "we tried OO programming and it failed."

Unrealistic Expectations. As programming team leaders, educators and consultants, we find that software development training is focused on the programming tools, but does not expose the process of creating a design. We all start out building software designed by someone else. What's involved in design?

In the building trades, we would neither expect nor allow apprentice plumbers to design the sanitary sewage system for an urban office building. Yet, in too many *Information Technology* (IT) departments, software developers are expected to leap from basic training in their tools to application design.

To continue this rant, we also find that some managers are entrusted with significant projects, but are uncomfortable with OO design on modern high-performance hardware. They tend to focus their design energies on the kinds of software architectures that were appropriate when the enterprise owned a single computer, when 64 megabytes of memory was all the enterprise would ever need, and centralized disk storage was charged back to end user departments at a rate of pennies per track per month. In some organizations,

there are a few enduring symptoms of this mind set in some of the ways that “end-user computing” is separated from “enterprise computing”; we relegate everything non-mainframe to second class status.

Management discomfort with OO technology surfaces in many ways. One shocking comment was that “no application needs more than six classes.” A consequence of this management attitude is an unrealistic expectation for schedule and the evolution of the deliverables.

Closing the Skills Gap. The deeper answer on the intent of this book is to help you, the beginning designer, by giving you a sequence of interesting and moderately complex exercises in OO design. The exercises are not focused on a language, but on a design process. The exercises are not hypothetical, but must lead directly to working programs.

The long answer is that this book will make you work.

This book can also help managers develop a level of comfort with the process of OO software development. The applications we will build are a step above trivial, and will require some careful thought and 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.

We hope to prevent managers from saying the following: “We had a good design, but were forced to compromise it to meet our schedule.” As consultants, we find this to be a sad statement of management’s emphasis of one near-term goal over long-term value. In one case, this was the result of a series of poorly-informed management decisions compounded on weak design skills. One of the root causes was the inability of the designers and managers to agree to a suitable course of action when a new kind of requirement made devastating changes to an existing design. We believe that more informed managers would have made a decision that created better long-term value.

1.2 Audience

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

You need to have an exposure to the language, but you need more time to understand objects and object-orientation. We will provide exercises that have four key features: just complex enough to require careful design work, just fun enough to be engaging, easy enough that results are available immediately, and can be built in simple stages.

In our effort to support you, 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.

OO Misdirection. Have you found your first exposure to objects to be more distasteful than empowering? Why does this happen?

Some instructors preface a course on a language (like Python or Java) with an extensive presentations on object orientation. It’s not easy to grasp OO design concepts before getting to the language fundamentals. In some cases, this leaves students lost as to how they will accomplish the noble and lofty goals of OO.

Other instructors leave OO for last, exposing the procedural side of the language first, and treating objects as a kind of add-on. This leaves students feeling that objects are optional. Additionally, some very skilled instructors are not skilled developers, and will often show examples that don’t reflect currently accepted best practices.

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 an OO programming language.

We provide specific examples in Python (at least version 2.5).

Student skills we expect include the following. If you can't do these things, this book is too advanced. Close it now.

- 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. We will avoid Python-unique features like multiple inheritance and callable objects, and focus on that subset of Python features that map directly to Java. For the Python equivalent of overloaded methods, we will assume that Python programmers can make use of default parameter values and named parameters.
- Some exposure to the various collections frameworks. For Java programmers, this means the classes in the `java.util` package. For Python programmers, this means the built-in sequence and mapping types.
- 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` or Java's `JUnit`.
- Optionally, some experience writing formal documentation. For Java programmers, this means `javadoc` comments. For Python programmers, this often means **Epydoc** or a similar documentation package. See the appendices for supplemental exercises if you aren't familiar with formal, deliverable documentation.

Language Details. There are two editions of this book, one emphasizing Python syntax and the other emphasizing Java syntax. Both books come from a common source, so that the OO Design issues are central and the language nuances are segregated.

1.3 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 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 Blackhjack, 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.

1.4 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.

1.5 Programming Style

We have to adopt a *style* for each of the languages we’re presenting. We won’t present a complete set of *coding standards*; we will omit a number of issues that should be standardized. Some IT shops have documents they call “coding standards”, but are little more than descriptive style guides. What follows is not this kind of style guide; instead, it is some justification of the style we use for the examples in this book.

Just to continue this rant, we find that source code examples speak louder than any gratuitously detailed “specification” of the desired style. We find that some IT organizations waste time trying to write definitions of the preferred style. A good example trumps the description of the example. In particular, as consultants, we are often asked to provide standards to an inexperienced team of programmers. While the programmers only look at the examples (often cutting and pasting them), some managers prefer to spend money on empty verbiage peripheral to the useful example.

Terminology. We tend to prefer Java-centric terminology: “field” and “method”. Occasionally, we will emphasize the differences between Java and Python by using the Python terms “attribute”, “instance variable” or “method function”.

Naming Style. We avoid using complex prefixes for variable names. In particular, we find prefixes to be little more than visual clutter. For example, an integer parameter with the amount of a bet might be called `pi_amount` where the prefix indicates the scope (*p* for a parameter) and type (*i* for an integer).

This style of name is only appropriate for primitive types, and doesn't address complex data structures well at all. How does one name a parameter that is a LinkedList of Sets of Outcomes? In Java programs, the variables are formally declared, therefore, we find that we don't need additional cues for their data type.

In some cases, prefixes are used to denote the scope of an instance variables. Variable names might include a cryptic one-letter prefix like "f" to denote an instance variable; sometimes programmers will use "my" or "the" as an English-like prefix. We prefer to reduce clutter.

In Python, instance variables are always qualified, typically by `self.`, making the scope very clear.

Generally, method functions (and classes) should be short enough and simple enough that complex navigational aids are not necessary.

1.6 Conventions Used in This Book

Here is a Python example.

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 "%d %.2f%%" % ( 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.

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

```
2 0.03%
3 0.06%
4 0.08%
5 0.11%
6 0.14%
7 0.17%
8 0.14%
9 0.11%
10 0.08%
11 0.06%
12 0.03%
```

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

Most of the design specifications will provide Java-style method and variable descriptions. Python doesn't use type specifications, and Python programmers will have to translate the Java specifications into Python by removing the type names.

Sidebars

When we do have a significant digression, it will appear in a sidebar, like this.

Tip: Tip

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.

1.7 Acknowledgements

We would like to thank Chuck Pyrak for putting us up to this. His idea of a One Room Schoolhouse to teach Java to an audience at multiple skill levels was a great idea. Additionally, our colleagues who collaborated through BLOKI brought infinite wisdom and insight to a complex and difficult project.

Thanks to Dion Dock and Robert Lucente for comments and corrections.

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. The use case is almost too trivial to bother defining. However, we have seen many projects run aground because they lacked even the most rudimentary description of the actor, the system and how the system helps the actor create value.

We will 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 will also describe the technical foundations. In this case, they are not terribly complex, but this is an important part of describing any software solution, no matter how simple.

We will dance around the methodology issue. Our intent is not to sell a particular methodology, but to provide some perspective on how we broke the work into manageable pieces.

Finally, we'll present some important parts of getting started on the solution. These are more specific, technical considerations that define common aspects of our approach.

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.

Here's the rule: **The first person to cut code loses.**

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, 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 have chosen two OO languages that work identically on both platform families: Java and Python.

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 individual 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 than 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.

Use Case. The high-level 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 to 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 informatin, 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. You 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 badly 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.

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 a use case written from the point of view of the 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, you will 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 make. 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, you will 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. The overall control component creates the various objects to start the simulation.

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

```
python -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. Avoiding additional libraries assures a “lowest-common denominator” multi-platform application. For Java, this standard is the J2SE set of libraries; we won’t use any J2EE extensions. For Python, it is the base installation.

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

Among the topics this book deals with in a casual – possibly misleading – manner are probability and statistics. 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 hesitate to endorse a specific methodology; doing so inevitably alienates readers who embrace a different methodology. To continue this rant, we find that almost everyone has an existing notion of the proper way to organize software development work. This leads to the common practice of customizing methodologies, in most cases without a deep background in the methodology or the changes being made.

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.

- **Embedded Documentation.** We provide appendices on how to use Epydoc or javadoc to create usable API documents.

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

First, we find that beginning designers work best with immediate feedback on their design decisions. 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.

Second, for iterative or agile methodologies, this presentation parallels the way software is developed. A project manager may use larger collections of deliverables. However, the actual creation of functional source eventually decomposes into classes, fields and methods. For project managers, this exposition will help them see where and how rework can occur; giving them a chance to plan for the kind of learning that occur in most projects.

Third, for project teams using 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.

This section addresses a number of methodology or process topics:

- Quality, in general
- Rework
- Technical Decision-Making
- Reuse
- Design Patterns

2.5.1 On Quality

Our approach to overall quality assurance is relatively simple. We feel that a focus on unit testing and documentation 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 implementation 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 and Java to avoid having interoperability and portability issues; these platforms 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, Java and Python are 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.

2.5.2 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

We consider design rework to be so important, we will summarize the idea here.

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 occurrence 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 consequence 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. 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, or the solution. We find that hands-on use of preliminary versions of software can help more than endless conversations about what could be built.

In the programming arena, we find that Java and Python (and their associated libraries) are so powerful that detailed design is done at the level of individual language statements. This leads us to write the program either in English prose or UML diagrams (sometimes both) before writing the program in the final programming language. We often develop a strong commitment to the initial design, and subsequent revisions are merely transliterations with no time permitted for substantial revisions. While we have written the program two or three times over, the additional quality isn't worth doubling or tripling the workload. We feel that the original workload should be managed as planned cycles of work and rework.

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.

2.5.3 On Decision-Making

Many of the chapters will include some lengthy design decisions that appear to be little more than hand-wringing over nuances. While this is true to an extent, 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.

Which means not saying “work smarter not harder” unless we also provide the time and budget to actually get smarter. The learning process, as with all things, must be planned and managed. Our lesson learned from Blaise Pascal is that a little more time spent on design can result in considerable simplification, which will reduce 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.

2.5.4 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 reuse 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.

2.5.5 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.

2.6 Deliverables

Each chapter defines the classes to be built and the unit testing that is expected. A third deliverable 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, you will be running this application from within your Integrated Development Environment (IDE). You may want to create a stand-alone program.

In the case of Java, we might also deliver the collection of class files. Additionally, we might bundle the class files into an executable JAR file. The source is the principle deliverable; anyone should be able to produce class and JAR files using readily available tools.

In the case of Python, it's 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 individual 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. In Java, this is often done with `JUnit`. In Python, the `unittest` module 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. In the case of Java and Python, the internal documentation is generally built from specially formatted blocks of comments within the source itself. Java programmers can use the `javadoc` tool to create documentation from the program source. Python programmers can use `Epydoc` (or `sphinx`) to create similar documentation.

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.

ROULETTE DETAILS

In the first section, *Roulette Game*, we will present a summary of the game of Roulette as played in most American casinos.

We will follow this with a review the various bets available on the Roulette table in *Available Bets in Roulette*. The definition of the various 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 are interesting and moderately complex algorithms for changing the amount that is used for each bet in an attempt to recoup losses.

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.

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.

		00	0	
Low 1-18	First 12	1	2	3
		4	5	6
		7	8	9
Even	Second 12	10	11	12
		13	14	15
		16	17	18
Red	Third 12	19	20	21
		22	23	24
		25	26	27
Black	High 19-36	28	29	30
		31	32	33
		34	35	36
		col 1	col 2	col 3

Figure 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. Any of two bins can make a split bet 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. A single bin selects one street bet; 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 72 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. This 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 occurrences, the house offers a higher payout on those bets. Since the 1-2 split bet wins 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 discrepancy 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% discrepancy in the odds.

The smallest discrepancy 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.

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 survey 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 about this preliminary design information. 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, each `Outcome` contains both the name and the payout odds. That way each `Outcome` 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 a few cases, we have looked 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.

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.

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 collaborators 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 **Outcomes** that win. This isolates the use of a random number generator to select **Outcomes**; and it encapsulates the set of winning **Outcomes** that are associated with each individual number on the wheel. Example: the "1" bin has the following winning **Outcomes**: "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 **Outcomes** into bins; used by the overall **Game** to get a next set of winning **Outcomes**.

Table Responsibilities.

A collection of bets placed on **Outcomes** 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 `Outcomes`; 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 `Outcomes`, updates the stake with amounts won and lost.

Collaborators.

Uses `Table` to place bets on `Outcomes`; 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-Responsibility-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 :class: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 :class:'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 `Outcomes`.

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 `Bets`? 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 `Outcomes`. From there, the `Game` can get all of the `Bets` from the `Table` and see which are based on winning `Outcomes` and which are based on losing `Outcomes`. 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”.

Deeper 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. We have seen highly skilled programmers who are able to limit the amount of work done in the main program, even in procedural languages. In OO languages, it becomes possible for even moderately skilled programmers to reduce the main program to a short list of object constructors, with the real work delegated to the objects. We find that “main program” classes are relatively hard to reuse, and prefer to keep them as short as possible.

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 privileged 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?

In a limited way, yes. A class can be thought of as a glorified *subroutine library* that captures and isolates data elements along with their associated functions. For some new designers, this is a helpful summary of the basic principle of encapsulation. Inheritance and subclasses, however, make a class more powerful than a simple subroutine library with private data. Inheritance is a way to create a family of closely-related subroutine libraries in a simple way that is validated by the compiler.

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. In a way, 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.

OUTCOME CLASS

In addition to defining the fundamental `Outcome` on which all gambling is based, this chapter provides a sidebar discussion on the notion of object identity and object equality. This is important because we will be dealing with a number of individual `Outcome` objects, and we need to be sure we can test for *equality* of two different objects. This is different from the test for *identity*.

5.1 Outcome Overview

The first class we will tackle is a small class that encapsulates each outcome. This class will contain the name of the outcome as a String, and the odds that are paid as an integer. We will use these objects when placing a bet and also when defining the Roulette wheel.

There will be several hundred instances of this class on a given Roulette table. The bins on the wheel, similarly, collect various `Outcomes` together. The minimum set of `Outcome` instances we will need are the 38 numbers, Red and Black. The other instances 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 foresee reworking this class to handle more complex odds, but we don’t need to handle the other cases yet.

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 `Outcomes`; the table will be holding bets. The wheel will select the winning `Outcomes`. 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.

Outcomes have Names. Since each instance of `Outcome` has a distinct `Outcome.name` attribute, it seems simple enough to compare names. This is one sense of “equal” that seems to be appropriate.

To do object comparison correctly in Python, we must provide the `__eq__()` and `__ne__()` method which compares the `Outcome.name` attribute.

However, simply comparing outcome names isn't – in the long run – going to work out well.

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.

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

The Hash Code Problem. Every object has a hash code. The hash code is simply a unique integer, perhaps the address of the object, perhaps some summary of the bits that make up the object. The default rule is this:

Distinct objects will have distinct hash codes

The default rule makes it possible for us to accidentally create objects that appear to be the same thing, but – internally – have different hash codes.

In Python, if we do nothing special, the `__eq__()` test will simply compare the 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:

Object Identity

```
>>> oc1= Outcome( "Any Craps", 8 )
>>> oc2= Outcome( "Any Craps", 8 )
>>> oc1 == oc2
False
>>> id(oc1)
6291728
>>> id(oc2)
6291760
```

Each individual `Outcome` object has a distinct distinct hashcode. 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.

This example shows that we can have two objects that appear equal, but don't compare as equal.

Layers of Meaning. The real issue is that we have three distinct layers of meaning for comparing objects to see if they are “equal”.

- Compare as Equal. We can call this “attribute equality”.

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.

- Have the same hash code. We can call this “hash equality”. This has to include attribute equality.

The `__hash__(self)()` method for several objects that represent the same `Outcome` have the same hash code.

- 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.

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

If we pursue this as an implementation, we have to be sure we don't casually create instances of `Outcome`. Rather than create them all over our application, we need to have a single source for the "official" instances of `Outcome`.

Problem. Which should we implement, **hash equality** or **object identity**?

Forces. There are two broad implementation choices for making sure that our objects both provide appropriate hash values and pass equality tests.

1. **Object Identity.** We can make sure that all of the parts of our application share the same `Outcome` objects. Two references to the same object will pass any identity test and will also have the same hash code.

This requires that we have a pool of outcomes and each `Outcome` is – effectively – a **Singleton** instance. We have to be sure that our application has only one instance of `Outcome("1", 35)`, and that object is shared by the Wheel, the Player and any Bets that are made.

We have to be sure to never create additional `Outcome` objects; there would be confusion between two discrete instances that happened to have the same name and odds.

We have to avoid casually creating new instances of the `Outcome` class.

Assuring that we work with single instances of each `Outcome` fits with the *Don't Repeat Yourself* (DRY) principle: we only create each `Outcome` once.

2. **Hash Equality.** We can make sure that all instances of an `Outcome` *act like* they are the same object. That is, they have the same hash value and they test true for equality. They may not be two references to the same underlying object, but they pass the hash and equality tests, making them equal for all practical purposes.

This allows us to create additional `Outcome` objects, confident they will behave appropriately.

This, however, also requires that we include the odds when creating an outcome. This gets wordy and unpleasant because we have the various outcomes and odds stated in more than one place in our application. This would break the *Don't Repeat Yourself* (DRY) principle.

The DRY principle is so important that this alternative is really unacceptable.

Solution. We have to assure that we're sharing single instances of `Outcome` objects. We have several choices for assuring that we have exactly one instance of each `Outcome`.

- **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.

- **Outcome Factory Object.** We can create a class which is a **Factory** for individual `Outcome` objects. When some part of the application asked for an Outcome which didn'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**.

- **Singleton Outcome Class.** A Singleton class creates and maintains a single instance of itself. This requires that the class have a static `getInstance()` 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 its 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.

All of these are acceptable techniques for implementing object identity. We don't have to choose one at the current time.

Consequences. The most important consequence of this design decision is that we're going to eventually create some global definitions of `Outcomes`, or a **Factory** object that centralizes `Outcome` creation.

We may also need to add the necessary static variable and static `getInstance()` method.

5.3 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 `Outcomes` with bets that will be paid off.

For example, the "1" bin has the following winning `Outcomes`: "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".

5.3.1 Fields

name

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

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.

5.3.2 Constructors

`__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.

5.3.3 Methods

For now, we'll assume that we're going to have global instances of each `Outcome`.

`winAmount(self, amount)`

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

Parameter *amount* (number) – amount being bet

`__eq__(self, other)`

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

Parameter *other* (Outcome) – Another Outcome to compare against.

Returns True if this name matches the other name.

Return type bool

`--ne__(self, other)`

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

Parameter *other* (Outcome) – Another Outcome to compare against.

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

Return type bool

`--str__(self)`

Easy-to-read representation of this outcome.

Python formatting is most easily done with the % operator. For some tips on how to do this, see [Message Formatting](#).

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

Return type str

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.

5.4 Outcome Deliverables

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

- The Outcome class.
- A class which performs a unit test of the Outcome 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.

5.5 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 "%s (%d:1)" % ( self.name, self.odds )
```


BIN CLASS

This chapter will present the design for the `Bin` class. In addition to that, we'll examine the collection classes available in Java and Python. We'll also present some questions and answers on this particular class and the process of creating a design.

6.1 Bin Overview

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 `Bins` of the wheel and each of the winning `Outcomes` (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 `Bins` have relatively few `Outcomes`. 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 `Bins` 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 `Outcomes`.

Some `Outcomes`, like "red" or "black", occur in as many as 18 individual `Bins`. Other `Outcomes`, 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 `Bins`.

Since a `Bin` is just a collection of individual `Outcome` objects, we have to select a collection class to contain the objects. In Java we have five basic choices, with some variations based on performance needs. In Python, we have two basic choices.

6.2 Design Decision – Choosing A Collection

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

- **Immutable Sequence tuple.** A `tuple` is immutable sequence of objects. This is a good candidate for the kind of collection we need, since the elements of a `Bin` don't change. However, a `tuple` allows duplicates and retains things in a specific order; we can't tolerate duplicates, and order doesn't matter.
- **Mutable Sequence list.** A `list` is a sequence of objects that can be changed. 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.

- **Mapping dict.** A `dict` associates a key with a value. This key-value pair is called an *item* in the map. We don't need this feature at all. A map does more than we need for representing a `Bin`.
- **Mutable Set set.** A `set` is a mutable collection of objects. Duplicates aren't allowed, and there's no inherent ordering. This looks close to what we need.
- **Immutable Set frozenset.** A `frozenset` is an immutable collection of objects. Duplicates aren't allowed, and there's no inherent ordering. This looks close to what we need.

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

6.3 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 `Outcomes` to win. Without thinking too deeply, we opted to define the `Bin` class to hold a collection of `Outcomes`. 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 we 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 that are 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 are part of a solution, and hinted at by the definition of the problem. Generally, these solution-oriented classes are part 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.

6.4 Bin Design

`class Bin()`

`Bin` contains a collection of `Outcomes` 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 `Outcomes`: “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` .

6.4.1 Fields

`outcomes`

A `frozenset` that holds the collection of individual `Outcomes`.

6.4.2 Constructors

Python programmers should provide an initializer that uses the `*` modifier so that all of the individual arguments appear as a single `list` within the initializer.

`__init__(self, * outcomes)`

Parameter `outcomes` – any number of outcomes used to populate the `Bin` initially.

This constructor can be used 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`.

6.4.3 Methods

`add(self, outcome)`

Adds an `Outcome` to this `Bin`. This can be used by a builder to construct all of the bets in this `Bin`. Since this class is really just a façade over the underlying collection object, this method could simply delegate the real work to the underlying collection.

Note that a `frozenset` is immutable; unlike a `list` it does not have an `append()` method. Instead, a new `frozenset` can be constructed by using the `|` operator which creates a new `frozenset` from a `frozenset` and a `frozenset` or `set`

```
>>> zero= Outcome("0",35)
>>> zerozero= Outcome("00",35)
>>> b = frozenset( [zero] )
>>> b
frozenset([<roulette.Outcome object at 0x6002d0>])
```

```
>>> map(str,b)
['0 (35:1)']
>>> b |= set( [zerozero] )
>>> map(str,b)
['0 (35:1)', '00 (35:1)']
>>> b
frozenset([<roulette.Outcome object at 0x6002d0>, <roulette.Outcome object at 0x600330>])
```

Parameter *outcome* (Outcome) – An outcome to add to this Bin

__str__(*self*)

An easy-to-read representation of the list of [Outcomes](#) in this [Bin](#).

A handy technique for displaying collections is the following. This maps the `str()` function to each element of `Bin.outcomes`, then joins the resulting string with `" "` separators

```
', '.join( map(str,self.outcomes) )
```

Returns String of the form `'[outcome, outcome, ...]'`.

Return type `str`

6.5 Bin Deliverables

There are two deliverables for this exercise. Both will have Javadoc comments or 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 [Outcomes](#).

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 [Bins](#). We will make increasing use of this in later sections.

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. We'll introduce some basic subclass constructor techniques, also.

7.1 Wheel Overview

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

7.1.1 The Container Responsibility

Since the `Wheel` is 38 `Bins`, 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.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 `Bins`. Each `Bin` contains a variety of individual `Outcomes`. However, each single number `Outcome` will be unique to each `Bin`.

The index values of 1 to 36 become the index to their respective `Bins`. The `Bin` at index 1 is the only bin which contains `Outcome("1", 35)`.

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 in the `Vector`. This gives us a unique place for all 38 `Bins`.

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 as an index.

Python programmers can use the `random` module. This 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 `Bins`.

Testability. Note that testing a class using random numbers isn't really the best policy. To do testing properly, we'll need to create a non-random random number generator that we can use in place of the published library random number generator.

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

1. Design our `Wheel` to accept a Random Number Generator that is inserted by the main program. For testing a non-random generator is used. For real simulation, a proper random generator is used.
2. Create a subclass of the system-supplied random number generator. One that overrides a few methods required for testing.

7.1.3 Constructing a Wheel

Each instance of `Bin` has a list of `Outcomes`. The 0 and 00 `Bins` only have two `Outcome`s. The other numbers have anywhere from twelve to fourteen `Outcomes`.

Constructing the entire collection of `Bins` would be a tedious undertaking. We'll apply a common OO principle of *deferred binding* and work on that algorithm later, after we have the basic design for the `Wheel` finished.

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

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))'`

rng

The random number generator to use to select a `Bin` from the `bins` collection.

This is not always simply `'random.Random()'`. For testing, we need to inject a non-random random number generator in place of the system random number generator.

7.2.2 Constructors

```
__init__(self, rng)
```

Parameter *rng* (`random.Random`) – A “random” number generator. For testing, this may be a non-random number generator.

Creates a new wheel with 38 empty `Bins`. It will also create a new random number generator instance.

At the present time, this does not do the full initialization of the `Bins`. We’ll rework this in a future exercise.

7.2.3 Methods

`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

`next()`

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`

`get(bin)`

Returns the given `Bin` from the internal collection.

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

Returns The requested Bin.

Return type `Bin`

7.3 Non-Random Design

We need a controlled kind of random number generation for testing purposes.

We’ll define a class `NonRandom` that extends `random.Random`, but provides a more testable sequence of values. One approach is to simply return a given integer value.

7.3.1 Fields

`value`

The non-random value to return.

7.3.2 Constructors

`__init__(self)`

Creates a non-random “random” number generator.

7.3.3 Methods

setSeed(*self*, *value*)

Saves this value as the next value to return.

Parameter *value* (long) – the value to return next.

choice(*self*, *sequence*)

Use the given seed value as an index and return the requested item from the sequence.

Parameter *sequence* – the sequence of values from which a random value is selected.

7.4 Wheel Deliverables

There are three deliverables for this exercise. The new class and the unit test will have Javadoc comments or 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 `Bins` can be added to the `Wheel`.
- A Non-Random Random Number Generator class, to be used for testing.
- A class which tests the `Wheel` and `NonRandom` class by selecting values from a `Wheel` object.

BIN BUILDER CLASS

This is a rather long chapter, which describes a number of design considerations surrounding `Outcome`, `Bin` and `Wheel`. The second section presents a number of algorithms to group `Outcomes` based on the geometry of the table. These algorithms are presented at a high level, leaving much of the detailed design to the student.

8.1 Bin Builder Overview

It is clear that enumerating each `Outcome` in the 38 `Bins` is a tedious undertaking. Most `Bins` contain about fourteen individual `Outcomes`.

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 `Bins` 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 `Bins` are only tangential to the operation of the `Wheel` object. While not essential to the design of the `Wheel` class, we find it is often helpful to segregate these 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 `Bins`, building the individual `Outcome` instances. We can then assign these `Outcome` objects to the `Bins` of a `Wheel` instance. We will use a number of steps to create the various types of `Outcomes`, and depend on the `Wheel` to assign each `Outcome` object to the correct `Bin`.

The Roulette Outcomes. Looking at the *Available Bets in Roulette* gives us a number of geometric rules for determining the various `Outcomes` 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. The center column of numbers (5, 8, 11, ..., 32) each participate in four split bets. 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, 34, 35 and 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. At the ends, 1, 3, 34, and 36, are members of just one corner. All of the remaining numbers are along an edge and are members of two corners.

- **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 has three of the six possible even money `Outcome`s. An easy way to handle these is to create the 6 individual `Outcome` instances. For each number, n , individual if-statements can be used to determine which of the `Outcome` objects are associated with the given `Bin`.

The `Bins` for zero and double zero can easily be enumerated. Each bin 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. 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)`. These are specific to a game, not general features of all `Outcomes`. We haven't designed the game yet, but we can provide a stub class with these these outcome odds definitions.

8.2 Bin Builder Algorithms

This section provides the algorithms for nine kinds of bets.

8.2.1 Generating Straight Bets

Straight bet `Outcomes` are the easiest to generate.

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

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 `Bin 0`.

Double Zero. Create an `Outcome` from the "00" with odds of 35:1. Assign this to `Bin 37`.

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 **Bins**: 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 **Bins**: 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 **Bins**: 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 , $0 \leq r < 12$:

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 **Bins**: $n, n + 1, n + 2$.

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 four corner bets: “ $n, n + 1, n + 3, n + 4$ ” corner bet. This **Outcome** object belongs to four **Bins**.

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 , $0 \leq r < 11$:

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 **Bins**: $n, n + 1, n + 3, n + 4$.

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 **Bins**: $n, n + 1, n + 3, n + 4$.

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 **Bins**.

For All Lines Between Rows. For each row, r , $0 \leq r < 10$:

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** with odds of 5:1.

Assign to Bins. Associate this object to six **Bins**: 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 , $0 \leq d < 3$:

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

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

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 , $0 \leq c < 3$:

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

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

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 , $1 \leq n < 37$:

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 $n \bmod 2 = 0$, associate the even `Outcome` with `Bin n`.

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

Red? If n is one of 1, 3, 5, 7, 9, 12, 14, 16, 18, 19, 21, 23, 25, 27, 30, 32, 34, or 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 `Outcomes` for all of the 38 individual `Bin` on a Roulette wheel.

8.3.1 Constructors

```
__init__(self)
    Initializes the BinBuilder.
```

8.3.2 Methods

```
buildBins(self, wheel)
    Creates the Outcome instances and uses the addOutcome() method to place each Outcome in the
    appropriate Bin of wheel.
```

Parameter `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 will have Javadoc comments or 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 `Bins` for the correct `Outcomes`.

8.5 Internationalization and Localization

In an advanced topic, we need to avoid *hard coding* the names of the bets. Both Java and Python provide extensive tools for localization (l10n) of programs. Since both applications already use Unicode strings, they support non-Latin characters, handling much of the internationalization (i18n) part of the process.

Python localization depends on the `locale` and `gettext` modules. The `locale` module provides a number of functions and constants to help format dates and times for the user's selected locale. The `gettext` module provides a mechanism for getting translated text from a message catalog with messages in a variety of languages.

We won't dwell on the l10n issue. However, the right preparation for l10n is to isolate all Strings used for messages, and format all dates and numbers with formatters found in the `locale` module. In this case, our `Outcome` names may be displayed, and should be isolated into a separate object. Even something as simple as `'blackBetName = _("Black")'` will set the stage for fetching strings from a `gettext` message catalog.

In order to define the `_(string)` function, the following fragment should be used. While not ubiquitous in Python programs, doing this now establishes a standard practice that will permit easy localization of an application program.

```
import gettext
gettext.NullTranslations().install()
print _("String To Be Translated")
```

1. This constructs an instance of `NullTranslations`. This is then installed as a function named `_()`, which finds a translation for the default C-locale string. This class does nothing; but it can be replaced with the `GNUTranslations` class, which uses a message catalog to replace the default strings with localized strings.
2. This shows how to use the `_()` function to translate a default C-locale string to a localized string.

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*.

9.1 Roulette Bet Overview

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

This is the classic *record declaration*: a passive association among data elements. The only methods we can see are three pairs of *getters and setters* to get and set each of the three attributes.

The general scenario is to have a `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 they are winners or losers.

A winning `Bet` has an `Outcome` that matches one in the winning `Bin`; winners 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`.

Locating Outcomes. 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.

Problem. Do we get an create an `Outcome` object with a `Bet`? Or do we locate an `Outcome` object in some collection or as a variable?

Forces. One option is to create an `Outcome` object as part of constructing a `Bet` object.

Here's what it might look like in Python.

```
my_bet= Bet( Outcome("red", 2), 25 )
```

One unfortunate feature of this is that we have to repeat the odds when creating an `Outcome`. This violates the DRY principle.

The other option is to locate an existing `Outcome` object as part of creating a `Bet` instance.

Here's what it might look like in Python.

```
my_bet= Bet( someSource["red"], 25 )
```

Solution. We want to get a complete `Outcome` from just the name of the outcome. This will prevent repeating the odds information.

Consequence. There are several consequences to this decision.

- 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`.
- Some object needs to build the collection of distinct `Outcomes`.
- Some object needs to maintain the collection of `Outcomes` 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.

9.3 Design Decision – Where to Keep the Outcome Map

Building a `Bet` involves two parts: an `Outcome` and an amount. Since we can't create an outcome, we need to get an outcome from some other object. There are several objects that could contain this `Outcome` map.

Problem. What class contains the mapping from name to `Outcome` so we can reuse references to those objects?

Forces. In *Outcome Class* we looked at three ways to achieve a single instance of an `Outcome`. This assured that we could have multiple references to a single instance of an `Outcome`. We'll modify that list slightly and reuse it here.

- Global Outcome Mapping. We can declare some global mapping object to keep references to out `Outcomes`. This object is more-or-less immutable, once the initial set of `Outcomes` have been built.

It's not a global variable, so it is a usable technique.

- Outcome Factory Object. We can define a class which produces references to `Outcomes` as needed. By using this object, we can be sure we're getting a reference to the one-and-only instance of the `Outcome`.
- The Outcome Class Itself. We can define the `Outcome` class to retain a static mapping to each `Outcome` that gets created. By using the static `Outcome.getInstance()` method, we can be sure we're getting a reference to the one-and-only instance of the `Outcome`.

This, however, limits our ability to create subclasses of `Outcome`.

After creating the `BinBuilder`, we can see that we – in effect – chose the **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. But is this the right place to do the lookup? What about the `Wheel`?

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 `Outcomes` 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` objects 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 `Outcomes` 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.

In Python, for example, we might add 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 `Bins` 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.

9.4 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.

9.5 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**.

9.5.1 Fields

amountBet

The amount of the bet.

outcome

The **Outcome** on which the bet is placed.

9.5.2 Constructors

```
__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.

9.5.3 Methods

```
winAmount(self)
```

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.

```
loseAmount(self)
```

Returns amount lost

Return type int

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

`__str__(self)`

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`.

9.6 Roulette Bet Deliverables

There are four deliverables for this exercise. The new classes will have Javadoc comments or Python doc-strings.

- 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.

ROULETTE TABLE CLASS

This section provides the design for the `Table` to hold the bets. It also introduces the concepts behind exception handling, and the proper role of exceptions.

10.1 Roulette Table Overview

The `Table` has the responsibility to keep the `Bets` 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 evaluate 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 `Bets` 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`'s odds and the `Bet`'s 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.

Winning vs. Losing. Our second 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 be 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 is 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.

Container Implementation. A **Table** is a collection of **Bets**. 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 do.

Table Limits. Table limits can be checked by providing a public method **isValid()** that compares the total of a new prospective amount plus all existing **Bet** s to the table limit. This can be used by the **Player** to evaluate each potential bet prior to creating it.

In the unlikely event of the **Player** object creating an illegal **Bet**, we can also throw (or 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.

Additionally, the game can check the overall state of a **Player**’s **Bets** to be sure that the table minimum is met. We’ll need to provide a public method **isValid()** that is used by the game. In the event of the minimum not being met, there are serious design issues, and an exception should be thrown. Generally, this situation arises because of a bug where the **Player** should have declined to bet rather than placing incorrect bets that don’t meet the table minimum.

Bet Resolution. An important consideration is the collaboration between **Table** and some potential game class for resolving bets.

The **Table** has the collection of **Bets**, each of which has a specific **Outcome**.

The **Wheel** selects a **Bin**, which has a collection of **Outcomes**. All bets with a winning outcome will be resolved as a winner.

So we’ll have multiple **Outcomes** in the winning **Bin** to be matched against multiple **Bets** active on the **Table**.

Because some games are stateful, and the winning and losing bets depend on game state, we will defer the details of this collaboration design until we get to the **Game** class. For now, we’ll simply collect the **Bets**.

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. 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

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

```
limit
```

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

```
bets
```

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

10.3.2 Constructors

```
Table()
```

Creates an empty list of bets.

Methods

```
isValid(self, bet)
```

Parameter *bet* (`Bet`) – A `Bet` instance to be validated.

Validates this bet. If the sum of all bets is less than or equal to the table limit, then the bet is valid, return `true`. Otherwise, return `false`.

```
placeBet(self, bet)
```

Parameter *bet* (`Bet`) – A `Bet` instance to be validated.

Raises `InvalidBet`

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 thrown (Java) or raised (Python). This is a rare circumstance, and indicates a bug in the `Player` more than anything else.

```
__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

```
__str__()
```

Return String representation of all current bets.

10.4 Roulette Table Deliverables

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

- An `InvalidBet` exception class. This is a simple subclass of `Exception`.
Since there's no unique programming here, there's no compelling need to construct a unit test.
- 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.

ROULETTE GAME CLASS

Between `Player` and `Game`, we have a *chicken-and-egg design problem*. 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, `Pssenger57` (they always bet on black), which will be the basis for further design in later chapters.

11.1 Roulette Game Overview

The `RouletteGame`'s responsibility is to cycle through the various steps of the game procedure, getting bets from the player, spinning the wheel and resolving the bets. 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 sequence of operations in 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; the 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.
3. **Resolve All Bets.**

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

- (a) **Winner?** If `Bet` b 's `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 's `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.

Matching Algorithm. This `Game` class will have the responsibility for matching the collection of `Outcomes` in the `Bin` of the `Wheel` with the collection of `Outcomes` of the `Bets` on the `Table`. We have two collections that must be matched: `Bin` and `Table`. We'll need to structure a loop or nested loops 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`, we would then visit each of the `Bets` contained by the `Table`. A `Bet`'s `Outcome` that matches the `Bin`'s `Outcome` is a winner and is paid off. The other bets are losers. This involves two nested loops: one to visit the winning `Outcomes` of a `Bin` and one to visit the `Bets` of a `Table`.

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

Since the winning `Bin` has a `Set` of `Outcomes`, we can exploit set membership methods to test for presence or absence of an `Bet`'s `Outcome` in the winning `Bin`'s `Set`. If the `Bin` contains the `Outcome`, the `Bet` is a winner; otherwise the `Bet` is a loser. This only requires a single loop to visit the `Bets` of a `Table`.

Player Interface. The `Game` collaborates with `Player`. We have a “chicken and egg” problem in decomposing the relationship between these classes. 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. In several future exercises, we'll revisit this design to make more sophisticated players.

For some additional design considerations, see *Additional Roulette Design Considerations*. 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

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.

table

The `Table` that is used to place individual `Bets`.

11.2.2 Constructors

```
__init__(self, table)
```

Parameter `table` (`Table`) – A `Table` instance on which bets are placed.

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

`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.

`win(self, bet)`

Parameter *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`.

`lose(self, bet)`

Parameter *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 `Bets` actually present on the `Table`.

11.3.1 Fields

`wheel`

The `Wheel` that returns a randomly selected `Bin` of `Outcomes`.

`table`

The `Table` which contains the `Bets` placed by the `Player`.

`player`

The `Player` which creates `Bets` 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.

`__init__(self, wheel, table)`

Parameters

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

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

11.3.3 Methods

`cycle(self, player)`

Parameter *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 call `thePlayer`'s `placeBets()` method to get bets. It will call `theWheel`'s `next()` method to get the next winning `Bin`.

It will then call `theTable`'s iterator to get an `Iterator` over the `Bets`. 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 Javadoc or 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 Additional Roulette Design Considerations

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 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`'s stake. We'll call this subclass 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`'s 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 `loss()` method for an ordinary `Outcome` also does nothing, but the `loss()` 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.

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.

Additionally, we'll address some issues in construction of class instances and an idealized structure for the main procedure.

12.1 Testability Overview

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

Back in *Wheel Class* we skipped over 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 controlled testing. There are three approaches to replacing the random behavior with something more controlled.

- One approach is to create a subclass of `Wheel` to create a more testable version without the random number generator. This `MockWheel` can be used to test classes that require a specific sequence of outcomes.
- A second approach is to record the sequence of random numbers actually generated from a particular seed value and use this to define the expected test results.
- Instead of changing `Wheel` or analyzing the results of a random number generator, we can define a subclass of `java.util.Random` (or Python's `random.Random`) that isn't actually random. Without much analysis, we suggest this in *Wheel Class*.

This is called the *Mock Objects* approach. We created a Mock Random Number Generator.

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 because pseudo-random numbers are generated in a fixed sequence from a given *seed value*. Computing the next value in the sequence involves a simple 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. One popular choice for the seed is the system clock. In some operating systems a special device is available for producing random seed values. In Linux this is typically `/dev/random`.

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.

12.2 Test-Driven Design

Good testability is achieved when there are no changes to the target software. For this reason, we don't want to have two versions of `Wheel`.

Instead of having two versions of `Wheel`, it's slightly better to have a random number generator that creates a known sequence with which we can test. To get this known sequence, we have a choice between creating a non-random subclass of `java.util.Random` or controlling the seed for the random number generator used by `Wheel`. Both will produce a known sequence of non-random values.

An important consequence of these design decisions is that we have to make the random number generator in `Wheel` visible. This creates 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 its implementation. It's unclear if the random number generator is an implementation detail or an explicit part of the wheel.

Generally, for most of the normal use cases, the `Wheel`'s random number generator is an invisible implementation detail. However, for testing purposes, the random number generator needs to be a configurable feature of the `Wheel` instance.

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

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

Note: Python Subclass Design

Because of the late binding in Python, and the lack of compile-time type checking, we don't have to use strict subclass inheritance to replace the Python random number generator with our own non-random number generator. We only have to provide a class with the minimal interface that our application requires.

While legal, we prefer not to bypass the inheritance tree in Python. We find that applications are much easier to maintain if reasonable care is taken to assure that the classes and interfaces are matched up as precisely as possible. In this case, we'll want to make our non-random generator a strict subclass of the `Random` class.

Note that the Mock Objects approach follows the principle of *deferred binding* or *dependency injection*. A class doesn't choose which implementation it will use; the decision is made somewhere else in the application. Ideally, by the user.

We've allowed the following options.

- `Wheel` can be built with an instance of `Random` initialized with a given seed.
- `Wheel` can be build with instance of a new subclass, `NonRandom`.
- `Wheel` can create it's own random number generator if no other choice is given.

12.3 Capturing Pseudo-Radom Data

We'll need to build and execute a program that reveals the fixed sequence of spins that are created by the non-random number generator.

Create an instance of `NonRandom` or `Random`. Set the seed of the this random number generator to a fixed value, for example, 3.

Create an instance of `Wheel` using this non-random number generator.

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 3.

This discovery procedure will give you the 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 extremes. 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 make a non-random subclass?

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 both Python and Java random number generators are 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 subclass than to control the existing class.

From another point of view, creating a subclass of a built-in class is a necessary skill. The random number generators are easy classes to extend. Extending the abstract collection classes is also a very useful skill, but somewhat more complex than extending the random number generator.

12.5 Testable Random Events Design

We'll present the design modification for `Wheel` first. This will be followed by design information for `NonRandom` in both Java and Python.

12.5.1 Wheel Rework

`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.

Note that we will be rewriting some methods to change their implementation. The external definitions of the methods don't change, but the implementations can (and do) change. This is the real strength of OO design. Once we have a well-defined interface (defined by the public methods and attributes), we are free to change the implementation as necessary.

This kind of rework – making a fundamental change to the implementation without touching the interface – is essential to successful OO programming.

Constructors

`__init__(rng=None)`

Create a wheel with the given random number generator. This will define the various bins and outcomes using an instance of `BinBuilder`.

Parameter `rng` (`random.Random`) – Random number generator

Note: Common n00b Mistake

Note that you cannot provide an instance of `random.Random` as the default argument. A `random.Random` is a mutable object, and using a mutable object as a default value of a function will lead to weird problems because that single object instance will be reused for every `Wheel` built during testing and actual simulation runs.

12.6 Testability Deliverables

There are three deliverables for this exercise. All of these deliverables need appropriate Javadoc comments or Python docstrings.

- A modified design for `Wheel`. This add the second constructor that allows us to omit the random number generator when constructing `Wheel`.
- Revised unit tests for `Wheel` based on the fixed sequence of responses from the non-random number generator.
- Revised unit tests for `Game` based on this revised version of `Wheel` based on the fixed sequence of responses from the non-random number generator.

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 Martingale betting strategy.

13.1 Roulette Player Overview

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 how long they play before they run out of time or go 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 `Outcomes` 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 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 `Bets`.

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.

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.

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 two 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.'

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. This reflects a more general situation where a stateful game has bets that change. In Craps, for example, most bets cannot be placed until a point is established.
3. The `Table` throws an "illegal bet" exception when an illegal bet is placed. While permissible, this kind of use for exceptions pushes the envelope on clarity and simplicity. One viewpoint is that exceptions should be reserved for situations that are truly unexpected. In this case, we expect to run into the table limit situation fairly often using Martingale betting.

We recommend the second choice: adding a `isValid()` method to the `Table` class. This has the consequence of allocating responsibility to `Table` , and permits us to have more advanced games where some bets are not allowed during some game states. It also obligates `Player` to validate each bet with the `Table`. It also means that at some point, the player may be unable to place a legal bet.

We could also implement this by adding to the responsibilities of the existing `placeBet()` method. For example, we could return `true` if the bet was accepted, and `false` if the bet violated the limits or other game state rules. We prefer to isolate responsibility and create a second method rather than pile too much into a single method.

Leaving the Table. In enumerating the consequences of checking for legal bets, we also uncovered 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.

Creating Bets from Outcomes. Generally, a `Player` will have a few `Outcomes` on which they are betting. Many systems are similar to the Martingale system, and place bets on only one of the `Outcomes`. 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.

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

stake

The player's current stake. Initialized to the player's starting budget.

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.

table

The `Table` used to place individual `Bets`. The `Table` contains the current `Wheel` from which the player can get `Outcomes` used to build `Bets`.

13.2.2 Constructors

`__init__(self, table)`

Constructs the `Player` with a specific `Table` for placing `Bets`.

Parameter *table* (`Table`) – the table to use

13.2.3 Methods

`playing(self)`

Returns `true` while the player is still active.

`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 Overview* for more information.

`win(self, bet)`

Parameter `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()`.

`lose(self, bet)`

Parameter `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

`lossCount`

The number of losses. This is the number of times to double the bet.

`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 $2^{\text{lossCount}}$.

13.3.2 Methods

`placeBets(self)`

Updates the `Table` with a bet on “black”. The amount bet is $2^{\text{lossCount}}$, which is the value of `betMultiple`.

`win(self, bet)`

Parameter `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.

`lose(self, bet)`

Parameter `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 Javadoc comments or 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.

OVERALL SIMULATION CONTROL

This section starts to address the overall “main program” that pulls the various classes together to create a finished application. Additionally, we’ll also address some issues in how Java handles collections of primitive data types like integers.

14.1 Simulation Control Overview

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?

The `Simulator` will be an active class with 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.

At this point, we’ll need to formalize some definitions, just to be crystal clear on the responsibility allocation.

14.2 Simulation Terms

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 player 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.

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 duration. 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, initializes the *Player* and then monitors the changes to the *Player*'s stake. We have to design two interfaces for this collaboration.

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 `Players`.

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.

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 Java `LinkedList` will do everything we need. For a deeper discussion on the alternatives available in the collections framework, see *Design Decision – Choosing A Collection*.

Statistical Summary. The `Simulator` will interrogate the `Player` after each cycle and capture the current stake. We don't want the sequence of stakes for each cycle, however, 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 to determine these two 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.

Handling Lists of Values in Java

Java programmers can't easily accumulate a `List` of `int` values. This is because a primitive `int` value is not an object. The usual solution is to make an `Integer` value from the `int`, and put the `Integer` object into the list.

When getting values back out of the list, there is a two-step dance: cast the item in the list to be an `Integer`, then get the `int` value of that object. The following examples show this as two steps.

```
for( Iterator<Integer> i = someList.iterator(); i.hasNext(); ) {
    int value = i.next().intValue();
    ... processing of value ...
}
```

14.3.1 Fields

`initDuration`

The duration value to use when initializing a `Player` for a session. A default value of 250 is a good

choice here.

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.

samples

The number of game cycles to simulate. A default value of 50 makes sense.

durations

A **List** of lengths of time the **Player** remained in the game. Each session of play produces a duration metric, which are collected into this list.

maxima

A **List** of maximum stakes for each **Player**. Each session of play produces a maximum stake metric, which are collected into this list.

player

The **Player**; essentially, the betting strategy we are simulating.

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

__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

session(self)

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.

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 add two new methods: one will set the `stake` and the other will set the `roundsToGo`.

`setStake(self, stake)`

Parameter `stake` (integer) – the Player's initial stake

`setRounds(self, rounds)`

Parameter `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 Javadoc or 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.

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.

15.1 SevenReds Player Overview

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 that 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.

Game Changes. 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 `Bets` that were wins or losses. We will need to add a method to `Player` to be given the overall list of winning `Outcomes` 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 can introduce 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.

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.

Our main program should, therefore, look something like the following:

```
theRNG= random.Random()
theWheel= Wheel( theRNG )
theTable= Table()
theGame= Game( theWheel, theTable )
thePlayer= SevenReds( table )
sim= new Simulator( theGame, thePlayer )
sim.gather()
```

1. The `theRNG` variable can also be initialized to an instance of `NonRandom`. The idea is that we build up the functionality of `Wheel` by composition. We compose the wheel of the bins plus the number generator.
2. the `thePlayer` variable can also be `Martingale` or `Passenger57`. Again, we have assembled the final simulation by composition of a `Wheel` (and its generator), the `Table` and the specific `Player` algorithm.
3. This is the actual work of the application, done by the `Simulator`, using the `Game` and `Player` objects built by main.

In many instances, the construction of objects is not done directly by the main method. Instead, the main method will construct a base group of *Abstract Factory* and *Builder* objects. Given these objects and the command-line parameters, the application objects can be built. The application objects then collaborate to perform the required functions.

15.2 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.

15.2.1 Fields

`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.

15.2.2 Methods

`placeBets(self)`

If `redCount` is zero, this places a bet on black, using the bet multiplier.

`winners(self, outcomes)`

Parameter *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.

15.3 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.

`winners(self, outcomes)`

Parameter *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.

15.4 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.

15.5 SevenReds Player Deliverables

There are six deliverables from this exercise. The new classes will require complete Javadoc comments or 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.
- The `SevenReds` subclass of `Player`.
- A unit test of the `SevenReds` class. This test should synthesize a fixed list of `Outcomes`, `Bins` 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.

STATISTICAL MEASURES

This section presents two ordinary statistical algorithms: mean and standard deviation. In one sense, this section could be treated as optional. Instead of having the simulator present useful summary statistics, raw data can be put into a spreadsheet for creating the statistics.

However, the process of designing these statistical processing classes is very interesting. This chapter examines some ways to add features to the built-in collection classes.

16.1 Statistics Overview

Currently, 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.

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.

In spite of our problems, we will 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 occurred. 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.

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 `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 can encapsulate this statistical processing in two ways: we could extend an existing class or we could delegate the statistical functions to a separate class. If we extend the library `java.util.List`, we can add the needed 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. In addition

to delegation, we have two choices for implementation of this extended version of `List`, giving us three alternative designs.

1. Extend `LinkedList`. In this case, we are simply adding two new methods to the existing `LinkedList` implementation. However, our new methods would not apply to `ArrayList` or `Vector`.
2. Extend `AbstractList`. However, in doing this we would need to provide the implementation details of the list structure itself. In effect, we would reimplement `LinkedList` or `ArrayList`.
3. Create a separate class that computes statistics on a `List` given as an argument to the methods of the statistical class. In this case, our statistical methods will work with any subclass that implements the interface `List`. This can be applied to any of `LinkedList`, `ArrayList` or `Vector`.

Alternative three – a separate statistical class – has the most reuse potential. We'll create a simple class with two methods: `mean()` and `stdev()` to compute the mean and standard deviation of a `List` of `Integer` objects. 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*.

Since our processes are relative simple, stateless algorithms, we don't need any instance variables. Consequently, we'll never need to create an instance of this class. As with `Java.lang.Math`, all of these methods can be declared as static, making them features of the class itself.

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)`’ calculation computes the measure of deviation from the average.

16.3 Statistical Algorithms

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:

Computing The Mean of a List, X , with n elements.

1. **Initialize.** Set $s \leftarrow 0$.
2. **Iterate.** For each value, x_i in the list:
 Sum and Count. Set $s \leftarrow s + x_i$.
3. **Result.** Return $s \div n$.

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:

Computing Standard Deviation of a List, X , with n elements.

1. **Get the Mean.** Compute the mean, $m \leftarrow \text{mean}(x)$.
2. **Initialize.** Set $s \leftarrow 0$.
3. **Iterate.** For each value, x_i in the list:
 - Difference.** Compute the difference from the mean, $d \leftarrow x_i - m$.
 - Sums.** Compute the sum, $s \leftarrow s + d^2$.
This is typically done as ' $d*d$ ' in Java, since there is no “squared” operator. In Python, this can be ' $d**2$ '.
4. **Variance.** Set $v \leftarrow s \div (n - 1)$. This $n - 1$ value reflects the statistical notion of “degrees of freedom”, which is beyond the scope of this book.
5. **Result.** Return the square root of the variance, \sqrt{v} .

Tip: Math Functions

For programmers new to Python, the `math` module contains the `math.sqrt()`, which computes square roots.

16.4 IntegerStatistics Design

```
class IntegerStatistics()
```

`IntegerStatistics` computes several simple descriptive statistics of `Integer` values in a `List`.

In Python, we don't actually need a class definition, we can simply declare two functions outside any class. However, to be consistent with Java, it's easier to define these functions in a class.

16.4.1 Constructors

Since this class has entirely static methods, it doesn't really need a constructor.

16.4.2 Methods

```
mean(values)
```

This requires the `@staticmethod` decorator.

Parameter `values` (list) – The list of values we are summarizing.

Computes the mean of the `List` of `Integer` values.

```
stdev(values)
```

This requires the `@staticmethod` decorator.

Parameter `values` (list) – The list of values we are summarizing.

Computes the standard deviation of the `List` of `Integer` values.

16.5 Statistics Deliverables

There are three deliverables for this exercise. These classes will include the complete Javadoc comments or 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
9
8
5
9
9
4
5
8
10
7
8
8

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 90
count 12
mean 7.5
sum d*d 608.668
stdev 1.88293
```


RANDOM PLAYER CLASS

This section will introduce a simple subclass of `Player` who bets at random.

17.1 Random Player Overview

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 `Bins`. From each `Bin`, we will need an iterator we can use to return all of the `Outcomes`.

To collect a list of all possible `Outcomes`, we would use the following algorithm:

Locating all Outcomes

1. **Empty List of Outcomes.** Create an empty set of all `Outcomes`, `all_OC`.
2. **Get Bin Iterator.** Get the `Iterator` from the `Wheel` that lists all `Bins`.
3. **For each Bin.**
 - Get Outcome Iterator.** Get the `Iterator` that lists all `Outcomes`.
 - For each Outcome.**
 - Save Outcome.** Save a reference to each `Outcome` in 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 total size-1. That is, $0 \leq u \leq s-1$.
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

`rng`

A Random Number Generator which will return the next random number.

17.2.2 Constructors

```
__init__(table, rng)
```

This uses the `super()` construct to invoke the superclass constructor using the `Table`. Then it saves the random number generator. This could be an instance of either `Random` or `NonRandom`.

Parameters

- `table` (`Table`) – The `Table` which will accept the bets.
- `rng` (`random.Random`) – A `:class:random.Random` random number generator.

17.2.3 Methods

```
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 Javadoc comments or Python docstrings.

- Updates to the class `Bin` to return an iterator over available `Outcomes`. Updates to unittests for the class `Bin`, also.
- Updates to the `Wheel` to return an iterator over available `Bins`. 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 `Outcomes`.
- An update to the overall `Simulator` that uses the `PlayerRandom`.

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. This section also has an advanced exercise that shows some alternative implementations for the state objects, using the **Singleton** design pattern.

18.1 Player 1-3-2-6 Overview

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. After one win, the bet is now 3x. After a second win, the bet is reduced to 2x, and the winnings of 4x are “taken down” or removed from play. In the event of a third win, the bet is advanced to 6x. Should there be a fourth win, the player has doubled their money, and the sequence resets.

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

State	Bet	Next State On Win	On Loss
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 implemente this more sophisticated player. This pattern suggests that we design 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 `isinstance()`.

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").

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:

```
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.

This will usually lead to 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 `instanceof`. 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` lack this 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 interchangeable. 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 Java and Python remove unused objects, the old state definitions are cleaned up by the garbage collection thread. A few tests will demonstrate that the Java memory management is efficient and reliable.

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

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

`__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

`currentBet(self)`

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. This raise statement is the functional equivalent of the Java abstract declaration.

`nextWon(self)`

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. This raise statement is the functional equivalent of the Java abstract declaration.

`nextLost(self)`

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

```
currentBet(self)
```

Constructs a new `Bet` from the player's `outcome` information. The bet multiplier is 1.

```
nextWon(self)
```

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

```
currentBet(self)
```

Constructs a new `Bet` from the player's `outcome` information. The bet multiplier is 3.

```
nextWon(self)
```

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

```
currentBet(self)
```

Constructs a new `Bet` from the player's `outcome` information. The bet multiplier is 2.

```
nextWon(self)
```

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

```
currentBet(self)
```

Constructs a new `Bet` from the player's `outcome` information. The bet multiplier is 6.

```
nextWon(self)
```

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

```
outcome
```

This is the player's preferred `Outcome`. During construction, the Player must fetch this from the `Wheel`.

```
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

```
__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

```
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.

```
win(self, bet)
```

Parameter *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`

`lose(self, bet)`

Parameter *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 `Outcomes`, `Bins`, 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 initialize 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 multiplier 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 patterns 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.

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

values

This is a map from a class name to an object instance.

`__init__(self)`

Create a new mapping from the class name to object instance. There are only four objects, so this is relatively simple.

`get(self, name)`

Parameter *name* (String) – name of one of the subclasses of `Player1326State`.

Return the `Player1326State` instance for the given class name.

CANCELLATION PLAYER CLASS

This section will describe a player who has a complex internal state that can be modeled using existing library classes.

19.1 Cancellation Player Overview

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

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.

outcome

This is the player's preferred `Outcome`.

19.2.2 Constructors

`__init__(self)`

This uses the `PlayerCancellation.resetSequence()` method to initialize 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

`resetSequence(self)`

Puts the initial sequence of six `Integer` instances into the `sequence` variable. These `Integers` are built from the values 1 through 6.

`placeBets(self)`

Creates a bet from the sum of the first and last values of `sequence` and the preferred outcome.

`win(self, bet)`

Parameter *bet* (`Bet`) – The bet which won

Uses the superclass method to update the stake with an amount won. It then removes the first and last element from `sequence`.

`lose(self, bet)`

Parameter *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 last 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 `Outcomes`, `Bins`, 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`.

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.

20.1 Fibonacci Player Overview

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 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

`recent`

This is the most recent bet amount. Initially, this is 1.

`previous`

This is the bet amount previous to the most recent bet amount. Initially, this is zero.

20.2.2 Constructors

`__init__(self)`

Initialize the Fibonacci player.

20.2.3 Methods

`win(self, bet)`

Parameter *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.

`lose(self, bet)`

Parameter *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.

$$next \leftarrow recent + previous$$
$$previous \leftarrow recent$$
$$recent \leftarrow next$$

param bet The `Bet` which lost.

type bet `Bet`

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 `Outcomes`, `Bins`, 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`.

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.

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. There are two ways to do this exploration: design everything in advance or design just enough but remain tolerant of our own ignorance.

Experienced designers can often design an entire, complex application before building any software. The process of doing this design, however, is internally 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.

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.

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 continue 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.

CRAPS DETAILS

In the first section 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 will review some of the bets available on the Craps table in some depth. 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.

Finally, 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.

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 are equal, 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*, *propositions* based on only the next throw of the dice, and hardways bets. These bets will be defined more completely, below.

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.

Table 22.1: Craps Game States

State	Roll	Bet Resolution	Next State
Point Off; the Come Out Roll. Only Pass and Don't Pass bets allowed.	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, <i>p</i> .
Point On. Any additional bets may be placed.	2, 3, 12	Nothing	Point still on
	11	Nothing	Point still on
	7	“Loser”: all bets lose. The table is cleared.	Point Off
	Point, <i>p</i>	“Winner”: point is made, Pass bets win, Don't Pass bets lose.	Point Off
	Non- <i>p</i> number	Nothing; Come bets are activated	Point still on

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.

In Python, you will have to write and run a short program to produce the frequency distribution.

```
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
```

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 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 the left 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.

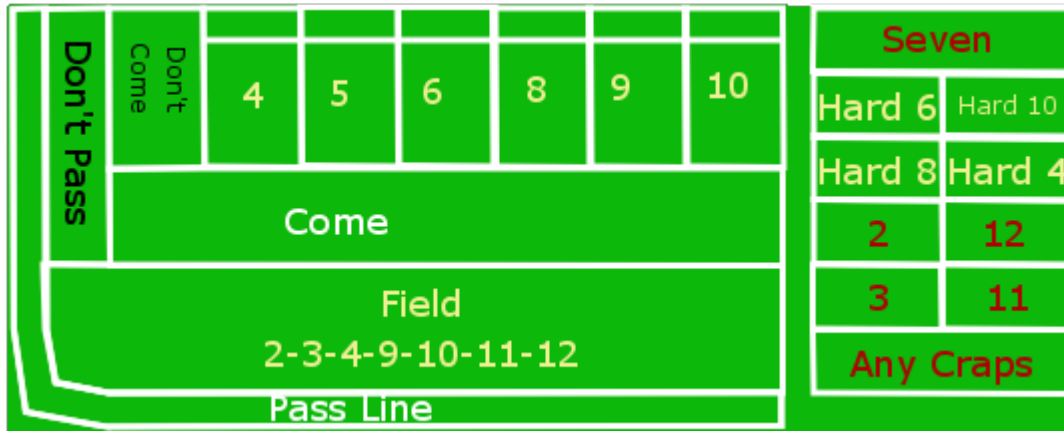


Figure 22.1: Craps Table Layout

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, there are the various proposition and hardways bets that can be placed in addition to other bets. While these propositions have odds that are incorrect, 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: there is a Pass Line bet that the game will be won, diluted by 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" – in effect it is a 1:2 bet, you will win half of what you bet if the point is missed. You 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 one of the basic rules of gambling.

CRAPS SOLUTION OVERVIEW

We will present a survey of the classes gleaned from the general problem statement in *Problem Statement* as well as the problem details in *Craps Details*. This survey is drawn from a quick overview of the key nouns in these sections.

Given this survey of the candidate classes, we will then do a walkthrough to refine the definitions, and 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 about this preliminary design information. 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 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 one `Outcome` to another `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 will give you 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. We'll revisit the question in some depth in *Soapbox on Overengineering*.

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.

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 constructors that this class will need.

24.1 Outcome Overview

For 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 `Outcomes`; a player's `Bet` referenced one of these `Outcomes`. In Craps, however, we have one-roll outcomes, hardways outcomes, plus outcomes that aren't resolved until the end of the game. Keeping this in mind, we'll defer details of dice and game problem until later, and focus on just the bet `Outcomes` first.

Varietals. There is a rich variety of bet `Outcomes`. 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 in 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 is 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 bet to one of the six numbered point boxes.

Additional Features. 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 accommodate the fractions.

- **Complex Odds.** In our simulation, we are faced with two choices for managing these more complex odds: exact fractions or approximate floating-point values. Since exact rational fractions are not part of the Java class library, we'll use simple `double` values instead of `int` values for the player's stake and the amount won or lost.

An interesting additional feature is to use rational numbers for the stakes and amounts. There are a number of rational number classes available on the Internet, any of which would be suitable for this problem. We will leave it to the interested student to replace the double values with `Rational` values for the player's stake and the amounts won and lost with each bet.

In Java, we will have to replace many instances of `int` with `double`. This is an unpleasant change, but a consequence of starting out with a built-in primitive type. The only effective escape from this kind of change is to use a class in all cases instead of a primitive type. This would allow us to use different subclasses as our design evolved. If, for example, we use `java.lang.Number`, we can switch to any subclass without a complex or sweeping change. The down-side of using a class in Python is that the built-in expression syntax only applies to primitive types. With primitive types we can write `'a + b'`. With the "language" classes like `Integer`, we have to write something like `'new Integer(a.intValue() + b.intValue())'`. With classes like `BigInteger`, we can write something like `'a.add(b)'`.

In Python, we don't have static compile-time type checking. We can easily replace an integer result with a double result with no impact throughout the application.

- **Commission Payments.** The second extension we have to consider is for the two 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 Overloaded Methods

We'll be extending the definition of `Outcome`. We don't want to disturb the simple case of Roulette when we add the more complex case of Craps. We'll do this with an overloaded constructor; a single method name that two different sets of parameters.

In Python, we use optional parameters to provide default values. We declare the default value as part of the method definition. This works well when we are initializing values to any of the immutable object classes. It doesn't, however, work well for mutable lists and mappings; these must be handled specially. The following example shows a simple required argument, followed by an optional argument.

```
class SomeClass:
    def __init__( self, reqArg, optArg=1 ):
        initialize using reqArg and optArg
```

Note that Python mutable types (lists, sets, maps) should not be provided as default values for an initializer. This is because one initialization object is created when the class is defined, and all instances will share this singleton initializer 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:
            initialize using aList
        else:
            default initialization using a fresh []
```

24.3 Outcome Rework

`Outcome` contains a single outcome on which a bet can be placed. In Roulette, each spin of the wheel has a number of `Outcomes`. For example, the "1" bin has the following winning `Outcomes`: "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" .

24.3.1 Fields

name

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

numerator

Holds the numerator portion of odds for this `Outcome`. This is the multiplier in the odds fraction. For right bets in Craps, this is larger than the denominator. For wrong bets, it is smaller than the denominator.

denominator

Holds the denominator portion of odds for this `Outcome`. For Roulette, this is always 1.

24.3.2 Constructors

`__init__(name, numerator, denominator=1)`

Parameters

- *numerator* (integer) – the payout odds numerator
- *denominator* (integer) – the payout odds denominator

Sets the name and odds from the parameters. 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.

24.3.3 Methods

`winAmount(self, amount)`

Parameter *amount* (integer) – amount of the bet

Returns the product of this `Outcome`'s odds numerator by the given amount, divided by the odds denominator.

`__str__(self)`

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)` works nicely.

24.4 Outcome Deliverables

There are three deliverables for this exercise.

- The revised `Outcome` class that handles more complex odds and returns type `double` 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`.

24.5 Advanced Exercise

For the more advanced student, locate or develop a class to work with rational numbers, represented as an integer numerator and denominator. Use this for all stakes, odds and bet payoffs, replacing any `double` variables.

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`.

Additionally, this chapter will introduce the subtle issue of over-engineering when doing design work. We'll revisit these issues a number of times in order to provide examples of good design and the issues that can lead to poor design.

25.1 Throw Overview

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 `Bins` and each one had a very simple collection of winning `Outcomes`. 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 `Outcomes` 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 `Outcomes`; 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 `Outcomes`. A `Throw` must also contain a list of losing `Outcomes`. For example, any of the two easy 8 rolls (6-2 or 5-3) would contain winning `Outcomes` 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 `Outcomes` 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 establishes the point, and changes the game state. On any other roll, this 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. See the *Soapbox on Over-Engineering* as a continuation of this rant.

Soapbox on Over-Engineering

Some of the horror stories of failed OO design projects seem to have over-engineering as the root cause. Over-engineering often includes the creation of superclasses and interfaces to embody *potentially* useful features. We suggest examining all engineering efforts by asking "does it solve a real problem?" Focus on potential value tends to dilute the creation of real value.

We note that software is developed to solve a particular problem. If a design consideration does not help solve the problem as presented, we have to classify the design as potential over-engineering. The most common response to this classification is that we will tend to focus too narrowly and build a *point solution*; the presumption is that point solutions are not as adaptable as something more broadly focused.

Our experience is that when the object design focuses on tangible real-world objects, and models those objects with a high degree of fidelity, these objects are very adaptable; they can be applied to solve a number of real-world problems. When the object design includes a number of irrelevant considerations, it takes too much effort to solve the initial problem, diluting the value of the software.

Additionally, some over-engineering is the result of conflating a number of features into a single class. We prefer to add features slowly so as to clearly separate the various concerns. As an example, it is very easy to conflate `Bet` and `Outcome` into a single class, losing site of the fact that bet amounts can change, but the definition of the outcome itself is invariant.

25.2 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.

25.2.1 Fields

outcomes

A Set of one-roll Outcomes that win with this throw.

d1

One of the two die values, from 1 to 6.

d2

The other of the two die values, from 1 to 6.

25.2.2 Constructors

`__init__(self, d1, d2, * outcomes)`

Creates this throw, and associates the given Set of Outcomes 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

25.2.3 Methods

`hard(self)`

Returns **true** if d1 is equal to d2. This helps determine if hardways bets have been won or lost.

`updateGame(self, game)`

Parameter *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.

`__str__(self)`

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.

25.3 Natural Throw Design

`class NaturalThrow()`

Natural Throw is a subclass of `Throw` for the natural number, 7.

25.3.1 Constructors

`__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 `Outcomes` for a throw of 7.

25.3.2 Methods

`hard(self)`

A natural 7 is odd, and can never be made “the hard way”. This method always returns false.

`updateGame(self, game)`

Parameter *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.

25.4 Craps Throw Design

`class CrapsThrow()`

`Craps Throw` is a subclass of `Throw` for the craps numbers 2, 3 and 12.

25.4.1 Constructors

`__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 `Outcomes` for a throw of craps.

25.4.2 Methods

`hard(self)`

The craps numbers are never part of “hardways” bets. This method always returns false.

`updateGame(self, game)`

Parameter *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.

25.5 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.

25.5.1 Constructors

`__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 [Outcomes](#) for a throw of 11.

25.5.2 Methods

`hard(self)`

Eleven is odd and never part of “hardways” bets. This method always returns false.

`updateGame(self, game)`

Parameter *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.

25.6 Point Throw Design

`class PointThrow()`

`Point Throw` is a subclass of [Throw](#) for the point numbers 4, 5, 6, 8, 9 or 10.

25.6.1 Constructors

`__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 [Outcomes](#) for a throw of craps.

25.6.2 Methods

`hard(self)`

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.

`updateGame(self, game)`

Parameter *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.

25.7 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.

25.7.1 Fields

`point`

The current point. This will be replaced by a proper *State* design pattern.

25.7.2 Constructors

`__init__(self)`

Creates this Game. This will be replaced by a constructor that uses `Dice` and `CrapsTable`.

25.7.3 Methods

`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.

`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.

`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.

`point(self, point)`

Parameter *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.

`__str__(self)`

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.

25.8 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.

DICE CLASS

Unlike Roulette, where a single `Bin` could be identified by the number in the bin, dice use a pair of numbers. In this chapter, we design `Dice`, as well as designing an inner class that is used only to make a single key out of a composite object.

26.1 Dice Overview

The dice have two responsibilities: they are a container for the `Throw`s and they pick one of the `Throws` 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. See *Soapbox on Over-Engineering* for some additional opinions on this subject.

Container. Since the `Dice` have 36 possible `Throws`, 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 `list` very appealing.

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 a new class to contain 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`.

In Python, a `tuple` does this job nicely. We don't need to define a whole new class.

We can pick one of the existing `NumberPair` objects at random from the collection of keys for the `Map` that contains the individual `Throws`.

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. $d_1 = \lfloor i \div 6 \rfloor + 1$; $d_2 = (i \bmod 6) + 1$.

Because of encapsulation, the choice of algorithm is completely hidden within the implementation of `Dice`.

While the numeric calculation seems simple, it 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.

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 the various 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 `tuples`.

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 a “getter” to return the key.

We'll add a method to `Throw` to return the `tuple` that is a key for this `Throw`.

```
getKey(self)
```

Second, 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.

26.3 NumberPair Design

```
class NumberPair()
```

In Python, it's often easiest to use a “named tuple” rather than use a simple `tuple`.

```
from collections import namedtuple
NumberPair = namedtuple('NumberPair', ['d1', 'd2'])
```

This can make the tuple slightly easier to work with.

26.3.1 Fields

`d1`

Contains the face of one die.

d2

Contains the face of the other die.

These attributes are built for us by the `collections.namedtuple` function.

26.3.2 Constructors

The constructor are built for us by the `collections.namedtuple` function.

26.4 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.

26.4.1 Fields

throws

This is a dict that associates a `NumberPair` with a `Throw`.

rng

An instance of `random.Random`

Generates the next random number, used to select a `Throw` from the `throws` collection.

26.4.2 Constructors

`__init__(self, rng=None)`

Build the dictionary of `Throw` instances.

Parameter `rng` (`random.Random`) – The random number generator to use.

At the present time, this does not do the full initialization of all of the `Throws`. We're only building the features of `Dice` related to random selection. We'll extend this class in a future exercise.

26.4.3 Methods

`addThrow(self, throw)`

Parameter `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)`

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.

```
getThrow(self, d1, d2)
```

Parameters

- *d1* – The value of one die
- *d2* – The other die

While not needed by the application, unit tests may need a method to return a specific `Throw` rather than a randomly selected `Throw`.

This method takes a particular combination of dice, locates (or creates) a `NumberPair`, and returns the appropriate `Throw`.

26.5 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 `Throws` 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 `Throws` 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 `Outcomes`.
 3. Create one instance of `Dice` that uses the two `Throws`.
 4. A number of calls to the `next()` method should return randomly selected `Throws`.

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.

26.6 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.

In this exercise, it appears that the `NumberPair` is superfluous.

We can work directly with a list of `Throws`, bypassing any mapping between `NumberPair` and `Throw`.

Removing this needless code is an exercise that should be considered, but not done at this time. In the *Design Cleanup and Refactoring* chapter, we'll clean up this design to remove several things which – in retrospect – will be poor design decisions.

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. Specifically, we didn't allow for an `Outcome` to have odds based on a throw of the dice. We'll need to add this feature carefully.

We'll present two solutions: "Design Heavy" and "Design Light". While we recommend "Design Heavy", we find that many programmers and project managers object to the apparent over-engineering of the approach. We have found that it is almost impossible justify the design heavy approach. See our *Soapbox on Justification* for more on this common blind spot. Instead, we provide the "light" alternative, which will become increasingly unwieldy throughout this part. At some point, we will eventually be forced to adopt the design heavy approach.

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 Overview

Enumerating each `Outcome` in the 36 `Throws` could be a tedious undertaking. It may be slightly simpler to hand-code the various combinations. For complete flexibility, we'll design a *Builder* to enumerate all of the `Throws` and their associated list of `Outcomes`. 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 could develop a *Builder* class that enumerates the 36 individual `Throws`, assigning the appropriate attribute values to each object. An alternative is for a *Builder* class to step through the 15 kinds of `Throws`, creating the proper number of instances of each kind. It seems slightly simpler to examine each pair of dice and determine which kind of `Throw` to build.

The proposition bets define eight one-roll `Outcomes` that need to be assigned to the various `Throw` instances we are building. We will share references to the following `Outcome` objects among the `Throws`:

- The number 2, with 30:1 odds.
- The number 3, with 15:1 odds.
- The number 7, with 4:1 odds.
- The number 11, with 15:1 odds.
- The number 12, with 30:1 odds.
- The "any craps" outcome, with 7:1 odds. This belongs to throws for 2, 3 and 12.
- The "horn" outcome, with two sets of odds. This belongs to throws for 2, 3, 11 and 12. For 2 and 12, the odds are 27:4; for 3 and 11, the odds are 3:1.

- The “field” outcome, with two sets of odds. This belongs to throws for 2, 3, 4, 9, 10, 11 and 12. For 2 and 12, this pays 2:1, all others pay 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 `Outcomes`.

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 and 10 we include a reference to the Field `Outcome`. Note that 2, 3, and 12 Field outcomes were handled above under **Craps**.

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 `Outcomes`.

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 bets also have payoffs that depend on the number on the dice. In our earlier `Outcome Class`, we missed this nuance, and did not provide for a `Dice.winAmount()` method that depends on the `Dice`. We'll present the solution to this as a fairly formal procedure that we find helps to resolve these kind of design issues.

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. Unlike Come Line bets, these bets are not moved on the table to a new `Outcome` when the point is established. 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. The `Bet` is compared with a `Set` of winning `Outcomes`. 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, 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.

- One choice is to have two `Outcomes` 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`. However, further research shows us that there are casino-specific variations on the field bet, including the possibility of three separate `Outcomes` 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 `Outcomes` into a `Bet`, we need to assign responsibility for keeping the bundle of Field `Outcomes` 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 `Outcomes`. This would allow us to keep `Bet` very simple, but we still have to construct appropriate composite `Outcome` instances for the purpose of creating `Bets`. 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 add an additional method to `Outcome` that calculates the win amount given the current `Throw`.

Consequences. There are a number of consequences of this design decision. The next design problem we have to solve is where in the `Outcome` class hierarchy do we add this additional `winAmount()` method? After that, we will need to determine the signature of the revised `winAmount()` method. 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 just the few subclasses that we use to create Horn and Field `Outcomes`.

- The Horn bet's `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 design objective is to minimize if-statements. 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 very little 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 `Bins`.

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 observe that more experienced designers develop an ability to locate this kind of commonality.

Less experienced designers tend 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 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. The minimal design effort has one unpleasant consequence: Roulette's `Outcome` instances depend on the Craps-related `Throw` class. This leads to an entanglement between Roulette and Craps around a feature that is really a special case for Craps only. This kind of entanglement may limit our ability to successfully reuse these classes. See *Soapbox on Architecture* for a discussion on this issue. The possible over-engineering prevents this entanglement. We consider this separation very desirable, even in an application as small as this exercise.

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 the subclass of `Outcome`. Since an object's membership in a class determines the available methods, there's no reason to *test* for membership. In most cases, a test for membership in a class is also 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 `Throws`.

In one `Throw` instance, we put a sticky for `Outcome` 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 `Outcomes`.

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 `Outcomes` 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.

Simulator, Statistics	
Craps Player hierarchy	Roulette Player hierarchy
Craps Game, Table, Dice, Throw	Roulette Game, Table, Wheel, Bin
Outcome, RandomEvent, Bet, Player, any other superclasses	

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 were careful to assure that our top-level classes contain minimal processing, and 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.

Didn't we notice the need for this `RandomEvent` class back in *Soapbox on Overengineering*? Why was it a bad idea then and a good idea now?

Some experienced designers immediately notice this kind of commonality between `Throw` and `Bin`, and can handle it without getting badly side-tracked. However, some beginning designers can spend too much time searching for this kind of commonality. We've seen examples of improper factoring where classes were combined in an early phase of design, only to lead to expensive rework when distinctions were uncovered later, invalidating large parts of the design. 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. Most managers and designers share a common 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 made up and who 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 idea 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 justify 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 all 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. For many people, it seems 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 “wouldn’t it be simpler” is best 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 Outcome Rework

The class `Outcome` needs a method to compute the win amount based on a `Throw`.

In Python, we use optional parameters to achieve the same degree of flexibility.

winAmount(*self*, *throw=None*)

Returns the product this `Outcome`’s odds numerator by the given amount, divided by the odds denominator.

Parameter *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.

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 depends 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()
```

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.

Fields

The most notable common feature between a `Bin` and a `Throw` is that both have a collection of `Outcomes`. This common `set` can be moved to this superclass from the subclasses.

`outcomes`

A `frozenset` that holds the collection of individual `Outcomes`.

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.

`winAmount(self, RandomEvent=None)`

Returns the product this `Outcome`'s odds numerator by the given amount, divided by the odds denominator.

Parameter *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

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

winAmount(*self*, *throw=None*)

Returns the product this [Outcome](#)'s odds numerator by the given amount, divided by the odds denominator.

Parameter *throw* (Throw) – An optional [Throw](#) that determines the actual odds to use. If not provided, this [Outcome](#)'s odds are used.

__str__(*self*)

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

winAmount(*self*, *throw=None*)

Returns the product this [Outcome](#)'s odds numerator by the given amount, divided by the odds denominator.

Parameter *throw* (Throw) – An optional [Throw](#) that determines the actual odds to use. If not provided, this [Outcome](#)'s odds are used.

__str__(*self*)

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

class ThrowBuilder()

[ThrowBuilder](#) initializes the 36 [Throws](#), each initialized with the appropriate [Outcomes](#). Subclasses can override this to reflect different casino-specific rules for odds on Field bets.

Constructors

__init__(*self*)

Initializes the [ThrowBuilder](#).

Methods

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 [Throws](#), each of which has the appropriate combination of [Outcomes](#). The [Throws](#) 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 are 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 `Throws` for the correct `Outcomes`.

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

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.

28.1 Bet Overview

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 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 it with a `Player`.

28.2.1 Methods

`setOutcome(self, outcome)`

Parameter `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`.

`price(self)`

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

`vig`

Holds the amount of the vigorish. This is almost universally 5%.

28.3.2 Methods

`price(self)`

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.

CRAPS TABLE CLASS

In Roulette, the table was a passive repository for `Bets`. 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.

29.1 Craps Table Overview

The `Table` is where the `Bets` are placed. The money placed on `Bets` on the `Table` is “at risk”; these bets either win an amount based on the odds, or lose the amount placed by the player.

- The Don’t Come and Don’t Pass bets may be returned, called a “push”.
- 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 a push is something we can allocate to `Game`, the commission price belongs to `Bet`.

Some `Bets` (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.

Game State. 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 two responsibilities that are part of handling Craps:

- changing `Bets`,
- inactivating outcomes and 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 deactivate 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 collaboration 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 `Outcomes` which change with the game state.

Problem. Which class determines the valid and invalid `Outcomes`?

Forces. We can see three places to place this responsibility.

1. We could make `CrapsTable` responsible; it could have methods to return the lists of `Outcomes` that are allowed or not allowed. `CrapsGame` can make a call to get the list of `Outcomes` and make the changes. Making each change would involve the `CrapsTable` a second time to mark the individual `Outcomes`. This information is then used by the `CrapsTable` to validate individual `Bets`.
2. We could `CrapsGame` responsible; it could invoke a method of `CrapsTable` that changes a single `Outcome`'s state to make it inactive. This information is then used by the `CrapsTable` to validate individual `Bets`.
3. An appealing 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 `Outcomes`.

The third choice seems to focus bet-handling responsibility mostly on `CrapsGame`.

Solution We need to place a valid 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.

Consequences. The game must move `Outcomes` 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.

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.

Working Bets. The rule for working and non-working bets is also relatively simple. 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. First, 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. Second, the player creates an additional odds bet placed behind this come point bet. Third, 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 non-working bets on the come-out roll.

This kind of exceptional subtlety 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.

29.4 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.

29.4.1 Methods

`isAllowed(self, outcome)`

Parameter *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.

`isWorking(self, outcome)`

Parameter *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.

29.5 CrapsTable Design

`CrapsTable` is a subclass of `Table` that has an association with a `CrapsGame` object. As a `Table`, it contains all the `Bets` 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.

29.5.1 Fields

game

The `CrapsGame` used to determine if a given bet is allowed or working in a particular game state.

29.5.2 Constructors

`__init__(self, game)`

Parameter *game* (`CrapsGame`) – The `CrapsGame` instance that controls the state of this table

Uses the superclass for initialization of the empty `LinkedList` of bets.

29.5.3 Methods

`isValid(self, bet)`

Parameter *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.

`allValid(self)`

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`.

29.6 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 `Bets` are managed by the table correctly.

For testing purposes, it is easiest to have the test method simply set 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.

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 `Outcomes` and all other `Outcomes` where losers, Craps includes winning `Outcomes`, losing `Outcomes`, and unresolved `Outcomes`. This will lead to some rework of previously created Craps classes.

We can see three necessary features to the `CrapsGame`:

- Game State,
- Resolving Bets,
- Moveable Bets when a point is established.

Also, we will discover some additional design features to add to other classes.

30.1 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 **Bets**. 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 **Outcomes** 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.

30.2 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 Overview](#)), 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 `Throws` and the two subclasses of `CrapsGameStates` 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()`. Different 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 `Throws` 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.

30.3 Resolving Bets

The `CrapsGame` class also has the responsibility for matching the `Outcomes` in the current `Throw` with the `Outcomes` of the `Bets` on the `CrapsTable`.

In addition to matching `Outcomes` in the `Throw`, we also have to match the `Outcomes` 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 `Outcomes` which are resolved as winners. All other `Outcomes` 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 `Outcomes`: 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%.

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 **Outcomes** in each **Throw**. This will oblige 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 perform the “pay a winner”, “collect a loser” or “leave unresolved” procedure based on the **Throw** or class: *CrapsGameState*.

30.4 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 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.

Moving Bets. 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 **Outcomes**.

When designing the **Bet** class, in the Craps Bet *Bet Overview*, 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**.

30.5 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.

This leaves us with a design decision on how best to handle the wide variety of bets that will be on the `CrapsTable`.

Problem. What's the best way to retain a collection of `Outcomes` that are resolved as a mixture of winning, losing, unresolved and pushes.

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 would increase the cost of maintenance and adaptation. 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 alternative 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 `Outcomes` for each state. All other `Outcomes` would be left unresolved.

This is a minor revision to `Dice` and `Throw` to properly create the two groups of `Outcomes`. Consequently `ThrowBuilder` will have to be expanded to identify losing `Outcomes` in addition to the existing winning `Outcomes`.

This will require the `CrapsGame` to make two passes through the bets. It must match all active `Bets` on the `CrapsTable` against the winning `Outcomes` in the current `CrapsGameState`; the matches are paid a winning amount and removed. It must also match match all active `Bets` on the `CrapsTable` against the losing `Outcomes` 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 `Outcomes` in addition to the existing winning `Outcomes`.

Each `GameState` would also need to respond with appropriate codes.

This will require the `CrapsGame` to make one pass through the `Bets`, 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.

- **Winning 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 `Outcomes` that are not resolved by the `Throw` or `GameState`.

This will require the `CrapsGame` to make one pass through the `Bets`, 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 **Throws** 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 **Outcomes** in addition to the collection of winning **Outcomes**, 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 **Bets**. 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 `:class:`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.

30.6 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 **Outcomes** and a Set of loser **Outcomes** 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 Overview* 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 $1 \leq d_1 < 7$:

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. The winning bets include one of the 2, 3 or 12 number **Outcome**, plus all craps, horn and field **Outcomes**. The losing bets include the other number **Outcomes**. 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 were handled above under **Craps**.

Losing Propositions. Other one-roll `Outcomes`, 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 `Outcomes` 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 `Outcomes` as winners. It will have numbers 2, 3, 7, 12 and Any Craps `Outcomes` 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*.

30.7 Craps Game Implementation Overview

Changing the `Throw` to include both winning and losing `Outcomes` is an important change. 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`.

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.

We will need to create a stub `CrapsPlayer` for testing purposes.

We will also need to create our `CrapsGameState` class hierarchy to represent the two states of the game.

Once the preliminary work is complete, we can then transform the `CrapsGame` we started in *CrapsGame Stub* into a final version of `CrapsGame`. It will collaborate with a `CrapsPlayer` and maintain a correct `CrapsGameState`. It will be able to get a random `Throw` and resolve `Bets`.

30.8 Throw Rework

`Throw` is the superclass for the various throws of the dice. A `Throw` identifies two sets of `Outcomes`: immediate winners and immediate losers. Each subclass is a different grouping of the numbers, based on the state-change rules for Craps.

30.8.1 Fields

win1Roll

A set of of one-roll **Outcomes** that win with this throw.

lose1Roll

A set of one-roll **Outcomes** that lose with this 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.

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.

d1

One of the two die values, from 1 to 6.

d2

The other of the two die values, from 1 to 6.

30.8.2 Constructors

__init__(*d1, d2, winners=None, losers=None*)

Parameters

- *d1* (int) – One die value.
- *d2* (int) – The other die value.
- *winners* (set of **Outcomes**) – All the outcomes which will be paid as winners for this Throw.
- *losers* – All the outcomes which will be collected as winners for this Throw.

Creates this throw, and associates the two given **Set** s of **Outcomes** that are winning one-roll propositions and losing one roll propositions.

30.8.3 Methods

add1Roll(*self, winners, losers*)

Parameters

- *winners* (set of **Outcomes**) – All the outcomes which will be paid as winners for this Throw.
- *losers* – All the outcomes which will be collected as winners for this Throw.

Adds outcomes to the one-roll winners and one-roll losers Sets.

addHardways(*self, winners, losers*)

Parameters

- *winners* (set of **Outcomes**) – All the outcomes which will be paid as winners for this Throw.
- *losers* – All the outcomes which will be collected as winners for this Throw.

Adds outcomes to the hardways winners and hardways losers Sets.

hard(*self*)

Returns **true** if **d1** is equal to **d2**.

This helps determine if hardways bets have been won or lost.

updateGame(*self*, *game*)

Parameter *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.

resolveOneRoll(*self*, *bet*)

Parameter *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.

resolveHardways(*self*, *bet*)

Parameter *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.

__str__(*self*)

This should return a **String** representation of the dice. A form that looks like 1,2 works nicely.

30.9 ThrowBuilder Rework

ThrowBuilder initializes the 36 **Throws**, each initialized with the appropriate **Outcomes**. Subclasses can override this to reflect different casino-specific rules for odds on Field bets.

30.9.1 Methods

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 **Throws**, each of which has the appropriate combination of **Outcomes** for one-roll and hardways. The various **Throws** are assigned to *dice*.

30.10 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**.

30.10.1 Constructors

```
__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.

30.11 CrapsPlayer Class Stub

CrapsPlayer constructs a **Bet** based on the **Outcome** named "Pass Line". This is a very persistent player.

30.11.1 Fields

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.

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.

table

That **Table** which collects all bets.

30.11.2 Constructors

```
__init__(self, table)
```

Parameter *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 **Bets**.

30.11.3 Methods

If `workingBet` is null, create a new Pass Line `Bet`, and use `Table placeBet()` to place that bet.

If `workingBet` is not null, the bet is still working. Do not place any more bets.

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

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

30.12 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.

30.12.1 Fields

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 `Bets`.

30.12.2 Constructors

`__init__(self, game)`

Parameter `game` (`Game`) – The game to which this state applies

Saves the overall `CrapsGame` object to which this state applies.

30.12.3 Methods

`isValid(self, outcome)`

Parameter `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.

`isWorking(self, outcome)`

Parameter `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.

`craps(self, throw)`

Parameter `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.

`natural(self, throw)`

Parameter *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.

`eleven(self, throw)`

Parameter *throw* (Throw) – The throw that is associated with 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.

`point(self, throw)`

Parameter *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.

`pointOutcome(self)`

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.

`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.

`__str__(self)`

In the superclass, this doesn't do anything. Each subclass, however, should display something useful.

30.13 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 `Bets`. It can then match each `Bet` against the various `Outcomes` which win and lose, and resolve the bets.

30.13.1 Constructors

`__init__(self, game)`

Parameter *game* (`CrapsGame`) – The game to which this state applies.

Uses the superclass constructor to save the overall `CrapsGame` object.

30.13.2 Methods

`isValid(self, outcome)`

Parameter *outcome* (Outcome) – The outcome to be tested for validity

There are two valid **Outcomes**: Pass Line, Don't Pass Line. All other **Outcomes** are invalid.

`isWorking(self, outcome)`

Parameter *outcome* (Outcome) – The outcome to be tested to see if it's working

There are six non-working **Outcomes**: "Come Odds 4", "Come Odds 5", "Come Odds 6", "Come Odds 8", "Come Odds 9" and "Come Odds 10". All other **Outcomes** are working.

`craps(self, throw)`

Parameter *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**.

`natural(self, throw)`

Parameter *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**.

`eleven(self, throw)`

Parameter *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**.

`point(self, throw)`

Parameter *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 **CrapsGamePointOn** 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.

`pointOutcome(self)`

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**.

`__str__(self)`

The point-off state should simply report that the point is off, or that this is the come out roll.

30.14 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.

30.14.1 Fields

point

The point value.

30.14.2 Constructors

__init__(self, point, game)

Saves the given point value. Uses the superclass constructor to save the overall `CrapsGame` object.

30.14.3 Methods

isValid(self, outcome)

Parameter *outcome* (Outcome) – The outcome to be tested for validity

It is invalid to Buy or Lay the `Outcomes` that match the point. If the point is 6, for example, it is invalid to buy the “Come Point 6” `Outcome`. All other `Outcomes` are valid.

isWorking(self, outcome)

Parameter *outcome* (Outcome) – The outcome to be tested to see if it’s working

All `Outcomes` are working.

craps(self, throw)

Parameter *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`.

natural(self, outcome)

Parameter *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 `Outcomes` lose, as do the pass-line odds `Outcomes` based on the point. Don’t Pass Line `Outcomes` 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 `Outcomes` and Come Point Number odds `Outcome` are all losers. All Don’t Come Point number `Outcomes` and Don’t Come Point odds `Outcomes` 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.

eleven(self, throw)

Parameter *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`.

point(self, throw)

Parameter *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 `Outcomes`. 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 `Outcomes`. The method should return `this`.

When the point is on and the value of `throw` matches `point`, the game is a winner. Pass Line `Outcomes` are all winners, as are the behind the line odds `Outcomes`. Don't Pass line `Outcomes` are all losers, as are the Don't Pass Odds `Outcomes`. Come Line bets are moved to the Come Point number `Outcomes`. Don't Come Line bets are moved to be Don't Come number `Outcomes`. The next state is a new instance of the `CrapsGamePointOff` state.

`pointOutcome(self)`

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.

`__str__(self)`

The point-off state should simply report that the point is off, or that this is the come out roll.

30.15 CrapsGame Class

`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.

30.15.1 Fields

`dice`

Contains the dice that returns a randomly selected `Throw` with winning and losing `Outcomes`. This is an instance of `Dice`.

`table`

The `CrapsTable` contains the bets placed by the player.

`player`

The `CrapsPlayer` who places bets on the `CrapsTable`.

30.15.2 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.

`__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.

30.15.3 Methods

__init__(self, player)

Parameter *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.

pointOutcome(self)

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.

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.

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.

30.16 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 `Outcomes`.
- The revised and expanded `ThrowBuilder`. This will construct `Throws` with winning as well as losing `Outcomes`.
- A unit test for the `Dice` class that confirms the new initializer that creates winning as well as losing `Outcomes`.

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.

30.17 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 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 `Bets` from the `Table`, and interrogates the `Player` for the disposition of the `Bet`.

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.

31.1 Craps Player Overview

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 `Bets`.

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 Superclass Design

`CrapsPlayer` is a subclass of `Player` and places bets in Craps. This is 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

stake

The player's current stake. Initialized to the player's starting budget.

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 a large number of hands or small number of multi-card hands.

table

The `CrapsTable` used to place individual `Bets`.

31.2.2 Constructors

`__init__(self, table)`

Parameter *table* (`CrapsTable`) – The table

Constructs the `Player` with a specific `CrapsTable` for placing `Bets`.

31.2.3 Methods

`playing(self)`

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.

`placeBets(self)`

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 Overview](#) for more information on the timing of this deduction, and the Craps Bet [Bet Overview](#) for more information on the price of a bet.

`win(self, bet)`

Parameter `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()`.

`lose(self, bet)`

Parameter `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

`placeBets(self)`

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

`lossCount`

The number of losses. This is the number of times to double the pass line odds bet.

`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

`placeBets(self)`

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.

`win(self, bet)`

Parameter `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.

`lose(self, bet)`

Parameter `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 `Outcomes`, `Throws`, 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 `Outcomes`, `Throws`, 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.

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.

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 having 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 recognize 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 `Bin` and `Throw` first. Unifying `Bin` and `Throw` is covered in *Design Heavy*. If the necessary design for `RandomEvent` is already in place, skip to *Unifying Dice and Wheel*. After that, we can refactor the other classes.

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`. This unification will make the `Vector` of results and the `next()` method identical.

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 `Outcomes`. Indeed, we don't even need a subclass, since a Roulette `Bin` can just ignore features of a Craps `Throw`.
2. Create a superclass based on `Bin`, make `Bin` a subclass and change `Throw` to add features to the new superclass.

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 a the Set of winning `Outcomes`, but the object has the 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. Since the languages we're using don't really support class definition by subtraction, we try to ignore the features, creating 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 Overview* for our initial thoughts on this, echoed in the *Soapbox on Refactoring* 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 `Bins` or `Throws`, 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 times. 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 `RandomEvents`.

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 `Bets` 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. However, in this case, both Craps and Roulette will use the `Game` 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. Indeed, 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.

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 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 (Martingale, 1326, Cancellation, Fibonacci). For Blackjack, there is also a playing system in addition to a betting system. Sometimes this multi-dimensional aspect of a class hierarchy indicates that we could be using multiple inheritance. In the case of Python, we have multiple inheritance in the language, and we can pursue this directly. Java, however, demands the **Strategy** design pattern to add a flexible 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 Propositions), 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 of `Player1326`. Since Java checks the validity of these references at compile time, we are forced to provide proper declarations for the `outcome` field in `Player1326`. In the event of change, this declaration may no longer be valid, alerting us to a dependency. In Python, however, the attributes of an object are kept in a dictionary, which is created dynamically during execution, making it difficult to assure that one class properly includes the attributes that will be required by a collaborating class. We don't discover problems in Python until the application crashes at run time because of a missing attribute.

Some Python programmers will bypass the simple access methods and check an object's internal `__dict__` for the presence of the expected `outcome` attribute. We don't condone this: the application should crash because of design changes, not work around them.

While the dynamic nature of Python attributes has some beneficial effects, we find it can be easily abused. We consider references to the `__dict__` object to raise serious quality concerns, and discourage its use by new OO designers.

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

rng

The random number generator, a subclass of `random.Random`.

Generates the next random number, used to select a `RandomEvent` from the `bins` collection.

current

The most recently returned `RandomEvent`.

32.2.2 Constructors

`__init__(self, rng)`

Saves the given Random Number Generator. Calls the `initialize()` method to create the vector of results.

32.2.3 Methods

`initialize(self)`

Create a collection of `RandomEvent` objects with the pool of possible results.

Each subclass must provide a unique implementation for this.

`next(self)`

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 `Bins` 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

`__init__(self, rng)`

Creates a new wheel. Create a sequence of the `Wheel.events` with with 38 empty `Bins`.

Use the superclass to save the given random number generator instance and invoke `initialize()`.

32.3.2 Methods

`addOutcome(self, bin, outcome)`

Adds the given `Outcome` to the `Bin` with the given number.

`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.

32.3.3 Dice Class

`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.

32.3.4 Constructors

`__init__(self, rng)`

Create an empty set of `Dice.events`. Use the superclass to save the given random number generator instance and invoke `initialize()`.

32.3.5 Methods

`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 `Outcomes`. For example, a throw of 3 includes four one-roll `Outcomes`: Field, 3, any Craps, and Horn.

`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 `Outcomes`.

32.4 Table Class

`Table` contains all the `Bets` 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.

32.4.1 Fields

`minimum`

This is the table lower limit. The sum of a `Player`'s bets must be greater than or equal to this limit.

`maximum`

This is the table upper limit. The sum of a `Player`'s bets must be less than or equal to this limit.

`bets`

This is a `LinkedList` of the `Bets` currently active. These will result in either wins or losses to the `Player`.

`game`

The `Game` used to determine if a given bet is allowed in a particular game state.

32.4.2 Constructors

`__init__(self)`

Creates an empty `LinkedList` of bets.

32.4.3 Methods

`setGame(self, game)`

Saves the given `Game` to be used to validate bets.

`isValid(self, bet)`

Validates this bet. The first test checks the `Game` to see if the bet is valid.

`allValid(self, bet)`

Validates the sum of all bets within the table limits. Returns false if the minimum is not met or the maximum is exceeded.

`placeBet(self, bet)`

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 thrown (Java) or raised (Python). This is a rare circumstance, and indicates a bug in the `Player` more than anything else.

`__iter__(self)`

Returns an `Iterator` over the list of bets. This gives us the freedom to change the representation from `LinkedList` to any other `Collection` with no impact to other parts of the application.

In Python, similarly, we can return an iterator over the available list of `Bet` instances. The more traditional Python approach is to return the list itself, rather than an iterator over the list. With the introduction of the *generators* in Python 2.3, however, it is slightly more flexible to return an iterator rather than the collection itself. The iterator could be built, for example, using the `yield` statement instead of the `iter()` function.

`__str__(self)`

Reports on all of the currently placed bets.

32.4.4 Player class

`Player` places bets in a `Game`. This is 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.

32.4.5 Fields

stake

The player's current stake. Initialized to the player's starting budget.

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.

table

The `Table` used to place individual `Bets`.

32.4.6 Constructors

`__init__(self, table)`

Constructs the `Player` with a specific `Table` for placing `Bets`.

32.4.7 Methods

`playing(self)`

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.

`placeBets(self)`

Updates the `Table` with the various `Bets`.

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 Overview* for more information.

`win(self, bet)`

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

`lose(self, bet)`

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

32.5 Game Class

`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`.

32.5.1 Fields

`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.

`table`

Contains a `CrapsTable` or `RouletteTable` which holds all the `Bets` placed by the `Player`.

`player`

Holds the `Player` who places bets on the `Table`.

32.5.2 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.

```
__init__(self, eventFactory, table)
```

Constructs a new `Game`, using a given `RandomEventFactory` and `Table`.

32.5.3 Methods

```
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 `Outcomes`. It will then call `table.bets()` to get an `Iterator` over the `Bets`. Stepping through this `Iterator` returns the individual `Bet` objects. The bets are resolved, calling the `thePlayer win()`, otherwise call the `thePlayer lose()`.

```
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.

32.6 RouletteGame Class

`RouletteGame` is a subclass of `Game` that manages the sequence of actions that defines the game of Roulette.

32.6.1 Methods

```
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 `Bets`. 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()`.

32.7 CrapsGame Class

`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.

32.7.1 Methods

```
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.

`pointOutcome(self)`

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.

`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.

`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.

32.8 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.

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.

33.1 Simple Craps Players Overview

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.

Craps 1-3-2-6. 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 Overview*) 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. We have three choices: extract the field and make it part of an interface, refactor the field up to the superclass, change the definition of `Player1326State` to make it more self-contained.

- **Common Interface.** The relationship between a subclass of `Player` and `Player1326State` can be formalized through an interface. We could define something like `Bet1326_Able`, and use this for `Roulette1326` and `Craps1326`. 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.** Refactoring the single instance variable up to the superclass makes a relatively minor change. However, it places a feature in the 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.** 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 `Outcomes`, 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.

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` is declared.

In Java, however, we must be very careful to sort out the relationship between `Craps1326`, and `Roulette1326`.

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.

Craps Cancellation. When can examine the Roulette Cancellation *Cancellation Player Overview* and 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.

Craps Fibonacci. When can examine the Roulette Fibonacci *Fibonacci Player Overview* and see that this player will need to compute new betting amounts based on wins and losses.

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

`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.

`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

`__init__(self, table, line, odds)`

Parameters

- *table* (CrapsTable) – The table on which bets are placed
- *line* (Outcome) – The line bet outcome
- *odds* (Outcome) – The odds bet outcome

Constructs the `CrapsSimplePlayer` with a specific **Table** for placing **Bets**. 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

`placeBets(self)`

Updates the **Table** with the various **Bets**. 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

lossCount

The number of losses. This is the number of times to double the pass line odds bet.

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

placeBets(*self*)

Extension to the superclass **placeBets()** method. This version sets the amount based on the value of **CrapsMartingale.betMultiple**.

win(*self*, *bet*)

Parameter *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.

lose(*self*, *bet*)

Parameter *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

outcome

The **Outcome** on which a **Player** will bet.

33.5.2 Constructors

__init__(*self*, *outcome*)

Parameter *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.

nextLost(*self*)

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

`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

`__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 `Bets`, 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

`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.

`win(self, bet)`

Parameter `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`

`lose(self, bet)`

Parameter `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 winnings are cancelled.

33.7.1 Fields

`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

`__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.

`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 s 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 `Outcomes`, `Throws`, 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 `Outcomes`, `Throws`, and calls a `CrapsCancellation` instance with various sequences of craps, naturals and points to assure that the bet changes appropriately.

ROLL-COUNTING PLAYER CLASS

Craps allows a large number of opportunities to bet. The bets have different resolutions, leading to quite a complex game.

In Roulette, each bet's odds are easier to calculate because all bets are resolved at once. Placing multiple bets in Roulette is a simple modification to the payout odds and the probability of a win.

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 bet. 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 $1/6$.

34.1 Roll-Counting Overview

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.

We can compute the following sequence of odds of losing for larger numbers of rolls of the dice after the point is established: 17%, 28%, 35%, 40%, 43%, 46%, 47%, 48%, 49%.

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 throws in the game, and effectively cancel their bet by betting against themselves on the Seven proposition. 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 states. They have Pass Line bet, they can use a Martingale strategy for their Pass

Line Odds bet, they are counting throws, and they are using a Martingale strategy for a Seven proposition starting with the seventh throw of the game.

We also note that this analysis doesn't work for wrong bettors 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. One simulation for this is to place Don't Pass Odds bets after several throws, where the odds bet appears more likely to win.

Decomposition. 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 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 class, and replacing the various methods and fields with one or more `BettingStrategy` objects. In the case of Roulette players, this is relatively simple. In the case of Craps players, we note that we have two bets, one with a trivial-case betting strategy where the bet never changes. If we add this `NoChange` strategy, we can 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 responsibilities. 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 `BettingStrategies`: 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.

Implementing SevenCounter. Once we have this design in place, our `SevenCounter` player can then be composed of a Pass Line bet, 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 line bet uses the `NoChange` strategy. The other two bets can use any of the strategies we have built: Martingale, 1-3-2-6 or Cancellation.

Currently, there is no notification to 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 conditions to count the throws in during a game, effectively counting unresolved bets.

34.2 BettingStrategy Design

BettingStrategy is the 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

outcome

This is the **Outcome** that will be watched for wins and losses, as well as used to create new **Bets**.

34.2.2 Constructors

__init__(*self*, *outcome*)

Parameter *outcome* (**Outcome**) – The outcome on which this strategy will create bets
Initializes this betting strategy with the given **Outcome**.

34.2.3 Methods

createBet(*self*)

Returns a new **Bet** using the **outcome Outcome** and any other internal state of this object.

win(*self*, *bet*)

Parameter *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.

lose(*self*, *bet*)

Parameter *bet* (**Bet**) – The bet which was a loser
Notification from the **Player** that the **Bet** was a loser.

__str__(*self*)

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

betAmount

This is the amount that will be bet each time. A useful default value is 1.

34.3.2 Constructors

`__init__(self, outcome)`

Parameter *outcome* (Outcome) – The outcome on which this strategy will create bets

Uses the superclass initializer with the given `Outcome`.

34.3.3 Methods

`createBet(self)`

Returns a new `Bet` using the outcome `Outcome` and `betAmount`.

`win(self, bet)`

Parameter *bet* (Bet) – The bet which was a winner

Since the bet doesn't change, this does nothing.

`lose(self, bet)`

Parameter *bet* (Bet) – The bet which was a loser

Since the bet doesn't change, this does nothing.

`__str__(self)`

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

`lossCount`

The number of losses. This is the number of times to double the pass line odds bet.

`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

`__init__(self, outcome)`

Parameter *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

createBet(*self*)

Returns a new **Bet** using the outcome **Outcome** and the **betMultiple**.

win(*self*, *bet*)

Parameter *bet* (**Bet**) – The bet which was a winner

Resets **lossCount** to zero, and resets **betMultiple** to 1.

lose(*self*, *bet*)

Parameter *bet* (**Bet**) – The bet which was a loser

Increments **lossCount** by 1 and doubles **betMultiple**.

__str__(*self*)

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

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

__init__(*self*, *outcome*)

Parameter *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

createBet(*self*)

Returns a new **Bet** using the **currentBet()** method from the **state** object.

win(*self*, *bet*)

Parameter *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**.

lose(*self*, *bet*)

Parameter *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`.

`__str__(self)`

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

`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

`__init__(self, table, lineStrategy)`

Constructs the `CrapsOneBetPlayer` with a specific `Table` for placing `Bet` s. This will save the given `BettingStrategy` in `lineStrategy`.

Python 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

`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.

`win(self, bet)`

Parameter `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 the `Bet` `winAmount()`. If the bet's `Outcome` matches the `lineStrategy`'s `Outcome`, notify the strategy, by calling the `lineStrategy`'s `win()` method.

`lose(self, bet)`

Parameter `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

`oddsStrategy`

An instance of `BettingStrategy` that applies to the line bet.

34.7.2 Constructors

34.7.3 Methods

`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`.

`win(self, bet)`

Parameter `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.

`lose(self, bet)`

Parameter *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

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 themselves 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.

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

`__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

`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.

`win(self, bet)`

Parameter *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.

`lose(self, bet)`

Parameter *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 classes 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, each with an associated unit test 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.

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. A book with a description of an already-completed structure does not help new designers learn the *process* of design, and does not help people to identify design problems and correct them. Because of that philosophy, the complete refactoring of the design in the *Design Cleanup and Refactoring* chapter was an important activity because it is the kind of thing that distinguishes a good design from a haphazard design.

Our observation is that this kind of design rework happens late in the life of a project, and project managers are 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 lots of internal operators and operands when compared with the number of operands in the interface. Halstead measures simple, transparent classes as less complex. 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.

BLACKJACK DETAILS

In the first section we will present elements of the game of Blackjack. Blackjack uses cards and has fairly complex rules for counting the number of points in a hand 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 the second section.

Finally, we will describe some common betting and playing strategies that we will simulate. 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 scenario, 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 player 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 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 casinos 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 opportunity to take another card when their total is less than 21. If they decline the hit, they are standing pat. In casinos that allow a double down on any number of cards, the player has three choices: hit, double down or stand.

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, we ignored the only offer made to a player. See *Some Betting Strategies* for details on this rule. 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 at that time.

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*.

BLACKJACK SOLUTION OVERVIEW

We will 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.

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 `Hands` for each `Player`; collected into a `Hand` for the dealer; added to by `Game`.

Deck Responsibilities

A complete set of 52 standard `Cards`.

Collaborators

Used by the `Game` to contain `Cards`.

Hand Responsibilities

A collection of `Cards` 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 `Hands`; a `Hand` has 2 or more `Cards`. The `Game` adds `Cards` 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 `Bets`, 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 `Hands`. 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 `Hands`, updates the state of the game, collects losing bets, pays winning bets. Splits `Hands`. 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 performed.
 - (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 it 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 on a 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 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`.

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. The next chapter will introduce the hand, and the problems of scoring the value of the hand.

38.1 Card, Deck and Shoe Overview

Card. The standard playing card has two attributes: a rank (Ace, 2 through 10, Jack, Queen, or King) and a suit (Clubs, Diamonds, Hearts or Spades). The set of all 52 combinations comprises a full deck. Additionally, in Blackjack, 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. However, the basic set of responsibilities of the `Card` class include keeping the rank, suit and point value of a single standard playing card.

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.

We'd like to provide symbolic variable names for the suits. In Java we'll use static final variables to define the suits. In Python we'll use class variables. This will give us four named constants for each of the suits, something that tends to make the software easier to read. Similarly, we'll provide named constants for the three face cards and the ace.

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 cards. We do note that 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. Considering this, we can provide two point-value functions for each rank, a hard value and a soft value. For all but aces, the two values are the same. For aces, the two values are different. This would allow a hand to compute a hard total and a soft

total for the hand as a whole. 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.

Deck. A deck of cards has responsibility for shuffling and dealing the various cards. Additionally, it should construct 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](#). The only collection we can disregard is [HashSet](#) as that will use more storage than necessary.

Shoe. The dealer's shoe is where [Cards](#) 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 [Cards](#) 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 thoroughly, 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 [Cards](#) in the [Shoe](#)'s 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 [ArrayList](#) or [Vector](#) could 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.

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 riffing. 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 of 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 or assemble a `SortedSet`. We note that sorting is $O(n \log n)$, where our algorithm is $O(n)$, 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	Name	Symbol
U+02660	BLACK SPADE SUIT	
U+02663	BLACK CLUB SUIT	
U+02665	BLACK HEART SUIT	
U+02666	BLACK DIAMOND SUIT	

There's one tiny problem with the Unicode characters: the suits don't sort into the order preferred by Bridge players. Since it's irrelevant for this class of problems, we'll ignore this.

For the truly fussy, we would need to assure that the suits have a symbol character *and* a preferred sort order.

38.3.1 Fields

In Python, symbolic names are often declared within the class, not within the initialization method function.

```
class Card( object ):
    Clubs, Diamonds, Hearts, Spades = u'\u2663', u'\u2666', u'\u2665', u'\u2660'
    Jack, Queen, King = 11, 12, 13
    ...
```

```
oneEyedJack= Card( Card.Jack, Card.Hearts )
```

rank

The rank of the card. This is a number from 1 (Ace) to 13 (King).

suit

The suit of the card. This is a character code for Clubs, Diamonds, Hearts or Spades.

38.3.2 Constructors

```
__init__(self, rank, suit)
```

Parameters

- *rank* (integer) – rank of this card.
- *suit* (character) – Character code for this card.

Initializes the attributes of this `Card`.

38.3.3 Methods

getRank(*self*)

Returns the rank of this card.

softValue(*self*)

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.

hardValue(*self*)

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.

`--str__(self)`

Ideally, we'd like this function to return the rank and suit of this card.

However, in Python 2, there's an issue with the definition of the `str()` function. In Python 2, this function must return an ASCII-encoded string.

If we're using Unicode suit names, this function isn't helpful, and we have to resort to an explicit Unicode formatting string.

`unicode(self)`

This will return a proper unicode string. We have to use `'print someCard.unicode()'` instead of `'print someCard'`.

38.4 FaceCard Class

`FaceCard` is a `Card` with a point value of 10. This defines jack, queens and kings.

38.4.1 Methods

`softValue(self)`

Returns the soft value of this card, 10.

`hardValue(self)`

Returns the hard value of this card, 10.

`--str__(self)`

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 defines Aces.

38.5.1 Methods

`softValue(self)`

Returns the soft value of this card, 11.

`hardValue(self)`

Returns the hard value of this card, 1.

`--str__(self)`

Returns a short String displaying the rank and suit of this card. The rank is always 'A'.

38.6 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.

38.6.1 Fields

`cards`

The collection of individual cards. The specific type of collection could be any of the Set or List implementation classes.

38.6.2 Constructors

`__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`

38.6.3 Methods

`getCards(self)`

Returns the collection of cards in `cards`.

38.7 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.

38.7.1 Fields

`deal`

An `Iterator` that is used to pick the next card from the shoe.

`stopDeal`

The approximate number of decks to be left undealt in the shoe.

38.7.2 Constructors

`__init__(self, decks, stopDeal, rng=None)`

If `rng` is `None`, create a generator from `random.Random`.

Parameters

- `decks` (integer) – A number of decks to create
- `stopDeal` (integer) – An approximate number of decks left undealt in the shoe
- `rng` (`random.Random`) – A random number generator to use.

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.

This also saves the random number generator used to shuffle.

To facilitate testing, this initializes the dealing iterator to the unshuffled List of cards. This will produce cards in a fixed order.

38.7.3 Methods

`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 $-6 \leq v \leq 6$.
2. Step through the deal iterator $stop * 52 + v$ times, removing approximately $stop$ cards from being dealt as part of ordinary play.

`__iter__(self)`

Return the `Shoe.deal` iterator so that the game can get cards from the Shoe.

38.8 Card-Deck-Shoe Deliverables

There are ten 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.

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.

39.1 Hand Overview

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. A player may have multiple hands, and 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 any kind of `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 `Outcomes`. In Roulette, there were a profusion of `Outcomes`, each relatively simple with fixed odds. In Craps, there were fewer `Outcomes`, 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 `Bets` were simply associated with `Outcomes` and the Wheel, Dice or Game gave us sets of winning (and losing) `Outcomes`.

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 `Outcomes`.

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” as a push, paying 0:1. 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.

In Craps, we identified the Field and Horn **Outcomes** which had winning amounts that depended on the specific **Throw**. See *Throw Builder Overview*, for more information. Additionally, we compared **Bets** against the **Outcomes** as part of the transition from one **GameState** to another. See *Resolving Bets*, for more information. In order for this to work correctly, some bets were changed from a generic **Outcome** (a “Line Bet”) to an **Outcome** with different odds.

What Class is Hand? 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 are 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 **RandomEvents** we saw in Roulette and Craps.

Choices. 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 **Cards**.

One could rationalize calling a **Hand** an “event” by claiming that the **Shoe** is the random event factor. The act of shuffling is when the event 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 the other subclasses of **RandomEvent**: **Bin** and **Dice**.

Hand Features. Our first design decision, then is to define **Hand** as a kind of **RandomEvent**. We’ll need to create several **Outcomes**: 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 least 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 change 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 inherited 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 subclass 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, effectively 0:1 odds.

It doesn't seem like a new class of `Outcomes` 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 assumed 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 “Even 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 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

`__init__(self, hand)`

Parameter *hand* (`Hand`) – The hand for which to compute a total

Creates a new `HandTotal` object associated with a given hand.

39.2.2 Methods

`total(self, card=None)`

Computes a total of all the cards in the associated hand. If *card* is not `None`, omit the indicated card from the total.

This method is abstract, it should return *NotImplemented*. Each subclass will provide an implementation.

Parameter *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.

`total(self, card=None)`

Computes the hard total of all the cards in the associated hand. If *card* is not `None`, omit the indicated card from the total.

Parameter *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.

```
total(self, card=None)
```

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.

Parameter *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.

```
setAltTotal(hand)
```

Parameter *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 `Cards`, and determines an appropriate total point value for the hand.

39.6.1 Fields

cards

Holds the collection of individual `Cards` of this 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.

soft

A instance of `HandSoftTotal`. This means that `'hand.soft.total()'` will produce the soft total of the cards in this hand.

altTotal

This is a reference to either `Hand.hardTotal` or `Hand.softTotal`. This will produce a soft total when one is 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

`__init__(self, card=None)`

Parameter `card` (`Card`) – A card to add

Creates an empty hand. The `Hand.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

`add(self, card)`

Parameter `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.

`value(self)`

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.

`size(self)`

Returns the number of cards in the hand, the size of the `List`.

`blackjack(self)`

Returns true if this hand has a size of two and a value of 21.

`busted(self)`

Returns true if this hand a value over 21.

`__iter__(self)`

Returns an iterator over the cards of the `List`.

`__str__(self)`

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.

BLACKJACK TABLE CLASS

The bets in Blackjack are associated with a hand. 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.

40.1 Blackjack Table Overview

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 `Hands`. Each individual `Hand` has a separate ante `Bet` and separate 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 `Hands`.
This would make the `Hand` a potential key into a Map that associated a `Hand` with a `Bet`. However, `Hands` 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 `Bets` 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, `Bets` 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 `Bets` and specific `Hands` in Blackjack.

40.2.1 Constructors

`__init__(self)`

Uses the superclass constructor to create an empty `Table`.

40.2.2 Methods

`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.

`__str__(self)`

40.3 Hand Rework

`Hand` contains a collection of individual `Cards`, 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

`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

`setBet(self, ante)`

Parameter *ante* (`Bet`) – The initial bet required to play

Sets the ante `Bet` that will be resolved when this hand is finished.

`getBet(self)`

Returns the ante `Bet` for this hand.

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 `Hands`, each with unique `Bets`.

BLACKJACK GAME CLASS

After reviewing the procedure provided in *A Walkthrough*, we'll define the basic game class for Blackjack. This will require a stub class for a Blackjack Player. 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.

41.1 Blackjack Game Overview

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 *Hands*, there are some minor sub-states and state changes. Finally, when all of the player's *Hands* 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 *Hands*.

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 involves one or more **Hand**s. The collaboration is so intensive, we have created a kind of swimlane table, showing 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	Blackjack-Table
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	returns a list iterator	
iterate through all hands; if the hand is less than 21, do the fill-hand procedure 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*.

Insurance. 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

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-jack-Table
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. if over 21, the hand is a bust	return point value; pass offers to player return point value	to double, increase the bet for this hand and return true; to hit, return true	update bet resolve the ante as a loss

There is some variation in this procedure for filling `Hands`. The most common variation only allows a double-down when the `Hand` has two cards.

Hand-specific Decisions. Some of the offers are directly to the `BlackjackPlayer`, while others require informing the `BlackjackPlayer` which of the player's `Hands` 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

hand

Some kind of `List` which contains the initial `Hand` and any split hands that may be created.

41.4.2 Constructors

```
__init__(self, table)
```

Parameter *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` for the hands.

41.4.3 Methods

```
newGame(self)
```


Creates a new, empty list in which to keep **Hands**.

placeBets(*self*)

Creates an empty **Hand** and adds it to the List of **Hands**.

Creates a new ante Bet. Updates the **Table** with this **Bet** on the initial **Hand**.

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**.

__iter__(*self*)

Returns an iterator over the List of **Hands** this player is currently holding.

evenMoney(*self*, *hand*)

Parameter *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.

insurance(*self*, *hand*)

Parameter *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.

split(*self*, *hand*)

Parameter *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 or 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.

doubleDown(*self*, *hand*)

Parameter *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.

hit(*self*, *hand*)

Parameter *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.

__str__(*self*)

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

`player`

Holds a reference to the `Player` who owns this hand. Each of the various offers from the `Game` are delegated to the `Player`.

`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

`splittable(self)`

Returns `true` if this hand has a size of two and both `Cards` have the same rank. Also sets `Hand.splitDeclined` to `true` if the hand is not splittable.

`getUpCard(self)`

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

shoe

This is the dealer's **Shoe** with the available pool of cards.

dealer

This is the dealer's **Hand**.

41.7.2 Constructors

`__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 **Cards** and a **BlackjackTable** for recording **Bets** that are associated with specific **Hands**.

41.7.3 Methods

`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 of the subclass **AceCard** will return **true** for `offerInsurance()`. All other **Card** classes will return **false**.

4. Iterate through all **Hands**, assuring that no hand is 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 **Hands** 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.

`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.

`fillHand(self, hand)`

Parameter *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.

`__str__(self)`

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.

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.

42.1 Blackjack Player Overview

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. Specifically, complex if-statements are often a stand-in for proper allocation of responsibility. In this class, however, the complex if-statements implement a kind of index or lookup scheme. We have, for this exercise, 8 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 by primitive types or object instances. In this case, we are indexing by conditions; we would have to map each condition to either a numeric code or a distinct object in order to eliminate the if-statements.

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.

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.

42.2 SimpleBlackjackPlayer Design

`class SimpleBlackjackPlayer()`

`SimpleBlackjackPlayer` is a subclass of `BlackjackPlayer` that responds to the various queries and interactions with the game of Blackjack.

This player implements a very simple strategy, shown above in the Blackjack Player Strategy table.

42.2.1 Methods

`evenMoney(self, hand)`

Parameter *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.

`insurance(self, hand)`

Parameter *hand* (Hand) – the hand which is being offered insurance

Returns `true` if this Player accepts the insurance offer. This player always rejects this offer.

`split(self, hand)`

Parameter *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.

`doubleDown(self, hand)`

Parameter *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.

`hit(self, hand)`

Parameter *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 [Hands](#) and confirm which offers are accepted and rejected.

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.

43.1 Variant Game Overview

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 to 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 responsibility 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

`adjustOdds(self, hand)`

Parameter *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.

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 for two reasons.

- **Intellectual Property.** We didn't create any strategies. Rather than get permission to quote existing blackjack strategies, 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.
- **Irrelevant Complexity.** We find that all of the detailed programming required to implement a particular strategy doesn't have too much to do with object-oriented design. Rather, it is an exercise in construction of detailed if-statements. It appears to be the wrong focus for learning OO design.

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.

This application is a command-line application, which makes it easy to build and test.

Another area for building skills in design is the implementation of programs that make extensive use of a database.

A graphical user interface (GUI) application will add the technology for graphical display, and responding to user input events.

A web application will add the technology for HTTP web transactions, HTML, CSS, and – potentially – a database.

- **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 endeavors 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.

PYTHON UNITTEST TESTING

Python **unittest** 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.

Workflow. 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 target class. One 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 target class, exercise those instances, and make assertions about the state of those instances.
3. Run the test, knowing that will fail. Sometimes our target class is so simple that we get it right the first time. Other times, the target class is more complex and there are parts we didn't finish when we wrote the skeleton version.
4. While tests are still 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 target class 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.

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, we'll organize our project directory to have a **test** directory and a **src** directory.

Here is an example of running a test from the command line.

```
python test/testDeck.py
```

```
..
```

```
-----  
Ran 2 tests in 0.001s
```

```
OK
```

45.1 Dependencies

Let's assume we've built two classes in some chapter; pretend that 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`.

45.2 Example

testCard.py

```
import unittest
import card
class TestCard( unittest.TestCase ):
    def setUp( self ):
        self.aceClubs= card.Card(1,card.Clubs)
        self.twoClubs= card.Card(2,card.Clubs)
        self.tenClubs= card.Card(10,card.Clubs)
        self.kingClubs= card.Card(13,card.Clubs)
        self.aceDiamonds= card.Card(1,card.Diamonds)
    def testString( self ):
        self.assertEqual( " AC", str(self.aceClubs) )
        self.assertEqual( " 2C", str(self.twoClubs) )
        self.assertEqual( "10C", str(self.tenClubs) )
    def testOrder( self ):
        self.assertTrue( self.tenClubs < self.kingClubs )
        self.assertFalse( self.tenClubs >= self.kingClubs )
        self.assertTrue( self.kingClubs < self.aceClubs )
        self.assertTrue( self.aceClubs == self.aceDiamonds )

if __name__ == "__main__":
    unittest.main()
```

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` module.
 - (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. We can add these later. In some cases, where we are working in large teams, we may need to produce tests which exhaustively enumerate all possible alternatives. For the purposes of learning OO design, we only need to sketch out our class by defining the tests it must pass.

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 the `__cmp__()` method 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 only need to execute the various paths within the `__cmp__()` method. When we initially create the test class, we may not have written the `Card` class yet. What we have to do is develop enough tests to get started, and add tests as we start writing our `Card` class.

4. This is the standard main program for unittest modules.

Our initial `Card` class needs to have just enough of an API to allow the tests to run. Here's our skeleton `Card` class.

`card.py`

```
Diamonds= "D"
Clubs= "C"
Spades= "S"
Hearts= "H"
class Card( object ):
    def __init__( self, rank, suit ):
        pass
    def __str__( self ):
        return ""
    def __cmp__( self ):
        return 0
```

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.

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.

```
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 of the module's interface.

```
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 its internal doctests.

```
python roulette.py
```

Test-Driven Development. Note that the strict test-driven development (TDD) 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, because the test cases come first, driving the rest of the work.

The test-driven approach doesn't fit well with `doctest`. It's very difficult to develop the test output without having the class available to exercise interactively. Also, we don't want to put comprehensive test scripts into our class docstring; they can easily get too long to be useful.

Also, it's difficult to "fake" the interactive log when designing the class. It's usually simpler to have a working class and then add test cases.

Finally, we need to pick essential features for demonstration in the `doctest` comments. We can't easily write comprehensive tests that cover all features or all line of code in `doctest` comments.

Limitations. Because of a few small gaps in Python 2's support for Unicode, it's difficult to use `doctest` for testing the `Card` class.

Specifically, Python 2 source is assumed to be ASCII. The Unicode results of formatting a card can't easily be put into the Python source. Even if the explicit source encoding comment is used, the `doctest` module can't cope well with tests that involve Unicode.

It can be challenging to do stand-alone unit tests with Mock classes to replace real classes. This can make for a rather long `doctest` string.

46.1 Example

Let's assume we've built two classes in some chapter; pretend that we're building `Card` and `Deck`. One class defines a standard playing card and the other class deals individual card instances. We'll define some minimal `doctests`.

The first step is to develop our baseline class. Here's a module that seems like it works.

This version uses ASCII suit names instead of Unicode suit names.

`card.py` - Initial

```
Clubs="C"
Diamonds="D"
Hearts="H"
Spades="S"

class Card( object ):
    """The card superclass, appropriate for number cards.

    .. todo:: Doctest goes here
    """
    def __init__( self, rank, suit ):
        self.rank= rank
        self.suit= suit
        self.points= rank

    def hardValue( self ):
        return self.points

    def softValue( self ):
        return self.points

    def __eq__( self, other ):
        return self.rank == other.rank
```



```

def __lt__( self, other ):
    return self.rank < other.rank

def __le__( self, other ):
    return self.rank <= other.rank

def __gt__( self, other ):
    return self.rank > other.rank

def __ge__( self, other ):
    return self.rank >= other.rank

def __str__( self ):
    return self.unicode()

def unicode (self ):
    return u"%d%s" % (self.rank, self.suit)

class AceCard( Card ):
    """The Ace class.

    .. todo:: Doctest goes here
    """
    def __init__( self, suit ):
        self.rank= 14
        self.suit= suit

    def hardValue( self ):
        return 1

    def softValue( self ):
        return 11

    def unicode (self ):
        return u"A%s" % (self.suit,)

```

Exercise the Class. Once we have the class, we need to exercise it using interactive Python. Here's what we saw.

```

MacBook-5:python slott$ python
Python 2.6.3 (r263:75184, Oct  2 2009, 07:56:03)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from card import *
>>> c2d = Card( 2, Diamonds )
>>> str(c2d)
'2D'
>>> cas = AceCard( Spades )
>>> str(cas)
'AS'
>>> cas.softValue()
11

```

During our session, small parts of the script show the preferred use cases for our class. We copy these from the interactive session and paste them into our class docstrings.

Update the Docstrings. After we have some output that shows correct behavior of our class, we can put

that output into the class docstrings. Here's our updated `card.py` module with doctest comments.

card.py - Revised

```
Clubs="C"
Diamonds="D"
Hearts="H"
Spades="S"
class Card( object ):
    """Create a new Ordinary (non-Ace, non-Face) Card.

    >>> c2d = Card( 2, Diamonds )
    >>> str(c2d)
    '2D'
    """
    etc.

class AceCard( Card ):
    """The Ace class.

    >>> cas = AceCard( Spades )
    >>> str(cas)
    'AS'
    >>> cas.softValue()
    11
    """
```

46.2 Add the Test Framework

There are two `doctest` frameworks. Doctest can be used for “stand-alone” testing. It can also be incorporated into `unittest` testing.

The general approach to stand-alone testing is to add the following to a module. When you run the module, it executes all `doctest` strings found throughout the module. Tests will pass silently. Any output is a failure of some kind.

```
def _test():
    import doctest
    doctest.testmod()
if __name__ == "__main__":
    _test()
```

46.3 Mixed unittest and doctest

A more complete approach is to use `unittest` for most testing and `doctest` to emphasize specific use cases in the docstrings. When doing this, you'll have a test module that looks something like the following.

testCard.py

```
import unittest
import doctest
import cards

suite = unittest.TestSuite()
suite.addTest(doctest.DocTestSuite(cards))
runner = unittest.TextTestRunner()
runner.run(suite)
```


PYTHON DOCUMENTATION

Application Program Interface (API) documentation is absolutely essential. The easiest and most reliable way to produce this documentation is by using a tool that examines the source itself and develops the document directly from the programs. By using cleverly formatted Python docstrings, we can augment that analysis with easy-to-understand descriptions.

In the case of Python, there are several tools for extracting documentation from the source. Two popular ones are

- **Epydoc** <http://epydoc.sourceforge.net/>.
- **Sphinx** <http://sphinx.pocoo.org/>.

Workflow. Generally, the workflow has the following outline.

1. Develop the skeleton class.
2. Develop unit tests for most features.
3. Rework the class until it passes the unit tests. This may involve adding or modifying tests as we understand the class better. This also involves writing docstrings for modules, classes and methods.
4. Revisit the modules, classes and methods, finishing each description using the ReStructured Text (RST) markup language. We won't cover all of this language, just enough to provide a sense of how the tool is used to make clean, professional API documentation.
5. Run the documentation tool (epydoc or sphinx) to create the documentation. Fix any errors in your docstring markup. Also, rework the documentation as necessary to be sure that you've captured important information about the class, the methods, the design decisions.

Using the Epydoc. When you install Epydoc, it creates a script that you can use. To run Epydoc, you'll use a command similar to the following.

```
epydoc -v --docformat restructuredtext --output apidoc card.py
```

The `-v` option provides detailed debugging information for incorrectly formatted docstrings. The `--docformat` option specifies that the various docstrings use RST markup. The `--output` option defines the directory to which the documentation is written. The arguments (in this case, `card.py`) is a list of package directories or module files to be documented.

Note that RST is not the default markup language for Epydoc. You must either specify this on the command line or in the module source. To identify the formatting in each module you must provide the format as the value of a `__docformat__` module global variable.

```
#!/usr/bin/env python
"""
```

```
Some docstring for a module.
"""
__docformat__ = "restructuredtext en"
```

Using Sphinx. When you install Sphinx, it creates a script that you can use.

Generally, you'll run the `$ sphinx-quickstart` script to create an empty documentation directory.

```
$ sphinx-quickstart
```

This interactive script will ask a number of questions, allowing you to pick sensible directories and file names.

You'll need to include two extensions to enable Sphinx to create your API documentation.

- **autodoc** will create module documents based on the RST markup in the docstrings.
- **autosummary** will create summaries of packages and modules.

You'll then add content to your `index.rst` file. This file is the root for the documentation. It can be very simple, or it can do include other files to break a large document into manageable pieces.

The most important commands are the following.

To run Sphinx, you'll use a command similar to the following.

```
$ sphinx-build -b html sourcedir builddir
```

ReStructured Text. Basic RST rules allow you to have a nice-looking document, with numbered and bulleted lists. You can include code examples and doctest results, also. With the “inline” markup, you can specify italic, bold or code samples.

Additionally, RST offers “field” markup to tie your documentation to specific pieces of Python syntax. This includes method parameters, return values, class instance variables, module variables, etc.

47.1 Basic RST Markup

We write our module, class and method docstrings using the RST markup language. RST is one of many *structured text* markup languages, where simple indentation and formatting rules are used to interpret your documentation. These formatting rules help the tools deduces the structure of your document (sections, list, etc.) and the presentation (bold, italic, or font changes.) These rules are – generally – pretty simple rules that plain text documents have been using since documents were first stored on computers.

Fundamentally, the RST formatting rules are implemented by a module named **docutils**. For additional documentation on RST, see the Sphinx web site (<http://sphinx.pocoo.org>) and the Docutils web site (<http://docutils.sourceforge.net/>).

In addition to document structure, deduced from the layout, RST also uses explicit directives and interpreted text (or “inline”) markup.

Document Structure. Generally, you must write in paragraphs. A blank line is the end of one paragraph and the beginning of another. Also, different indentation will signal paragraph changes. The indent rule is handy for making bullet lists or numbered lists.

A line that is indented and begins with `-` is a bulleted list.

A line that starts with digits and periods is a numbered list. Lists can nest within each other.

If you need to have sections and subsections (you'll almost always do this in the docstring for a module or package), you “underline” the title with lines of `===`, `---` or `~~~`.

To include code code samples and doctest, you'll use either the simple `::` marker. Or the explicit `.. code-block::` directive.

Generally, the first paragraph must be a pithy summary of the material to follow. This paragraph will often be used in overviews and index pages.

Example testCard Module. For example, our module document might look like this.

```
#!/usr/bin/env python
"""The testCard module includes a number of unit tests for the Card
and Deck classes in the card module.

Overview
=====

This module tests the following classes:
- Card
- Deck

Card Tests
-----

.. autoclass:: TestCards

Deck Tests
-----

.. autoclass:: TestDeck

Usage
=====

This module uses the standard text runner, so it can be executed
from the command line as follows::

    python testcard.py
"""
```

1. We started with an overview paragraph. It summarizes the module in a single sentence.
2. We have an underlined section heading. In this document, the highest level of the outline is underlined with `=`'s. The second level is underlined with `-`.
RST is very flexible with the underlining rules.
3. We have a bullet list. Simply indent, and begin the paragraph with `-`. Lists can be nested, the RST processor will work this out based on the indentation of the various sublists.
4. We used the `.. autoclass::` directive to gather docstring documentation from the class docstring and include it in the overall module docstring.
5. We included a literal block of code after a paragraph which ends with `::`. The following material is indented to show what should be formatted specially.

Interpreted Text, or Inline Markup. In addition to basic document structure rules, RST includes a way to show that a span of characters should be interpreted specially. This is sometimes called Inline Markup because it is within the structural markup.

- `*words*` will render the words in *italic*. This can also be done with `:emphasis:`words``.
- `**words**` will render the words in **bold**. This can also be done with `:strong:`words``.

- ```words``` will render the words in a ‘fixed’ font to indicate that it is a code sample. Literal programming text can also use `:literal:``words```.

Code samples can also be done with `:samp:‘word {replaceable}’`. In this case, the material enclosed in `{}` is displayed in italics: `‘word replaceable’`.

Additionally, there are several cross-reference inline markup formats available for code elements.

Here’s an example of a paragraph using some inline markup to make some characters italic, other bold, and still others a fixed-width font that makes them look like code.

```
This method computes the square root of a number, *n*.
It returns a value, *r*, such that :samp:`r**2 == n`.
```

```
**Note**. Square root of a negative number will raise a ``TypeError`` exception.
```

47.2 RST Field Markup

RST Field markup is primarily used to tie your documentation directly to Python language structures. Field markup is a separate paragraph of the form `:tag arg: body`. We won’t cover all of the tags, just enough to produce basic documentation.

Field markup also provides several standard kinds of standard “formal paragraphs” within your document. We’ll look at these separately.

Functions and Method Functions. The following tags are used to define detailed features of a method function.

- `:param p: description` A description of the parameter, *p*, for a function or method.
- `:return: description` A description of the return value from the function or method.
- `:keyword k: description` A description of the keyword parameter, *k*, for a function or method.

Here’s an example of a method function docstring that uses `@param` tags.

```
def __init__( self, rank, suit ):
    """Build a card with a given rank and suit.

    :param rank: numeric rank, ace is 1, 2-13 for 2 through King.
    :param suit: String suit, use one of the module constants.
    """
    self.rank= 14 if rank == 1 else rank
    self.suit= suit
```

Classes and Modules. The following tags are used to define specific variables in a module or class.

- `:ivar v: description` A description of the an instance variable of a class. Generally, this will be in the class level docstring, and will refer to one of the class’s `self.` variables.
- `:var v: description` A global variable of a module. Generally, this will be in the module docstring, and will refer to one of the global variables created by the module.

Here’s an example of a class definition docstring that uses `:ivar:` tags.

```
class Card( object ):
    """A single playing card, suitable for Blackjack or
    Poker. While a suit is retained, it doesn't figure into
    the ordering of cards, as it would in Bridge.
```


*****Reminder**.** The internal rank of an Ace is 14. The constructor, however, expects a value of 1.*

*:ivar rank: the numeric rank of the card. 2-13, with ace as 14.
:ivar suit: the string suit of the card.
"""*

Directives. RST has a number of useful directives available for structuring the documentation. Sphinx uses all of the basic RST directives and adds a large number of additional directives.

A directive is an RST markup command starts with a special `..` line, and may include a number of lines indented within the directive. One of the common ones is the `.. code-block::` directive. The `::` is an essential part of the directive syntax.

It looks like this.

Example.

```
.. code-block:: python

def fact(n):
    if n == 0: return 1
    return n*fact(n-1)
```

More Text.

The directive starts with the `..` line and continues through the indented section. The directive content ends with the end of the indentation.

Epydoc uses the `:tag:` syntax; it doesn't define new RST directives. Because of this, there are two styles for some of the more advanced markup: an Epydoc tag style and a Sphinx directive style.

Standard Paragraphs. There are several kinds of standard paragraphs that are part of any well-written document set. These include things like

- Related Topics.

Epydoc uses the `:seealso:` tag to generate a "Related Topics" paragraph with references to other documents.

Sphinx uses the `.. seealso::` directive line to generate additional references.

- Admonitions (Notes, Warnings, etc).

Epydoc uses tags like `:note:`, `:attention:`, `:bug:`, and `:warning:` tags to generate standard types of admonition paragraphs.

Sphinx uses directives like `.. note::`, `.. attention::`, and `.. warning::` directives to generate standard types of admonition paragraphs. Other admonitions include "caution", "danger", "error", "hint", "important", "tip".

- Status. You can track the development and deployment status of your programs. Tags like `:version:`, `:todo:`, `:deprecated:`, `:since:`, `:status:`, and `:change:` are used by both Epydoc and Sphinx.
- Bibliographic Information. You can provide standard publication information. Tags like `:author:`, `:organization:`, `:copyright:`, `:license:` and `:contact:` can be used.

47.3 Class Example

Here's an example of a module with detailed class documentation.

```
#!/usr/bin/env python
"""The cards module defines the Card class and constants
for the four suits.

:author: S. Lott
:license: http://creativecommons.org/licenses/by-nc-nd/3.0/us/

.. data:: Clubs
    Use this to build Cards in the clubs suit

.. data:: Diamonds
    Use this to build Cards in the diamonds suit

.. data:: Hearts
    Use this to build Cards in the hearts suit

... data:: Spades:
    Use this to build Cards in the spades suit
"""
Clubs="C"
Diamonds="D"
Hearts="H"
Spades="S"

class Card( object ):
    """A single playing card, suitable for Blackjack or
    Poker. While a suit is retained, it doesn't figure into
    the ordering of cards, as it would in Bridge.

    .. note:: Aces and Facecards.
        Ace and Facecards are separate subclasses.

    .. attribute:: rank
        The numeric rank of the card. 2-13, ace has an effective
        rank of 14 when used in Poker.

    .. attribute:: suit
        The string suit of the card. This should be from the
        named constants (Clubs, Diamonds, Hearts, Spades).
    """
    def __init__( self, rank, suit ):
        """Build a card with a given rank and suit.

        :param rank: numeric rank, 2-10. Aces and FaceCards are separate.
        :type rank: integer in the range 2 to 10 inclusive.

        :param suit: String suit, use one of the module constants.
        """
        self.rank= rank
        self.suit= suit
        self.points= rank

    def hardValue( self ):
```

```
        """For blackjack, the hard value of this card.
        :returns: int
        """
        return self.points

def softValue( self ):
    """For blackjack, the soft value of this card.
    :returns: int
    """
    return self.points

def __eq__( self, other ):
    """Compare cards, ignoring suit.

    >>> Card( 2, Diamonds ) == Card( 2, Spades )
    True
    >>> Card( 2, Diamonds ) == Card( 10, Spades )
    False
    """
    return self.rank == other.rank

def __lt__( self, other ):
    """Compare cards, ignoring suit.

    >>> Card( 2, Diamonds ) < Card( 3, Spades )
    True
    >>> Card( 10, Diamonds ) < Card( 10, Spades )
    False
    """
    return self.rank < other.rank

def __le__( self, other ):
    return self.rank <= other.rank

def __gt__( self, other ):
    return self.rank > other.rank

def __ge__( self, other ):
    return self.rank >= other.rank

def __str__( self ):
    return "%d%s" % (self.rank, self.suit)
```


Part VI

Back Matter

BIBLIOGRAPHY

48.1 Use Cases

48.2 Computer Science

48.3 Design Patterns

48.4 Statistics

48.5 Python

48.6 Java

48.7 Casino Games

TOOLSET

The following toolset was used for production of this book.

- Python 2.6.3.
- Sphinx 0.63.
- Docutils 0.5.
- Komodo Edit 5.2.2.
- pyPDF 1.12.
- MacTeX-2008.

INDICES AND TABLES

- *Index*
- *Search Page*

BIBLIOGRAPHY

- [Jacobson92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard. *Object-Oriented Software Engineering. A Use Case Driven Approach*. 1992. Addison-Wesley. 0201544350.
- [Jacobson95] Ivar Jacobson, Maria Ericsson, Agenta Jacobson. *The Object Advantage. Business Process Reengineering with Object Technology*. 1995. Addison-Wesley. 0201422891.
- [Parnas72] Parnas D. *On the Criteria to Be Used in Decomposing Systems into Modules*. 1053-1058. 1972. Communications of the ACM.
- [Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns. Elements of Object-Oriented Software*. 1995. Addison-Wesley Professional. 0201633612.
- [Larman98] Craig Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design*. 1998. Prentice-Hall. 0137488807.
- [Neter73] John Neter, William Wasserman, G. A. Whitmore. *Fundamental Statistics for Business and Economics*. 4. 1973. Allyn and Bacon, Inc.. 020503853.
- [vanRossum04] Guido van Rossum, Fred L. Drake. *Python Documentation*. 2004. Python Labs.
- [Silberstang05] Edwin Silberstang. *The Winner's Guide to Casino Gambling*. 4th. 2005. Owl Books. 0805077650.
- [Skiena01] Steven Skiena. *Calculated Bets. Computers, Gambling, and Mathematical Modeling to Win*. 2001. Cambridge University Press. 0521009626.
- [Shackleford04] Michael Shackleford. *The Wizard Of Odds*. 2004.