

---

# **C++ and Object Design: the least you need to know**

---

<b>INTRODUCTION</b>	<b>1</b>
1.1 Why this book now? .....	1
1.2 How is it different? .....	2
1.3 What not to expect? .....	3
1.4 Standing on the Shoulders of Giants .....	3
<b>2 OBLIGATORY HELLO WORLD</b>	<b>5</b>
2.1 Setting up your environment.....	5
2.1.1 The Tools	5
2.1.2 Installing the Java Developers Kit (JDK)	6
2.1.3 Installing Eclipse CDT	6
2.1.4 Starting Eclipse CDT	6
2.1.5 Installing the Wascana Plugin	8
2.1.6 Downloading CppUTest	8
2.1.7 Building CppUTest	8
2.2 Mechanics of CppUTest.....	9
2.2.1 Creating a Project	10
2.2.2 Include Path, Library Path and Included Libraries	10
2.2.3 A Few Convenient Settings	11
2.2.4 Adding a CppUTest main() function	12
2.2.5 Adding a first test	13
2.2.6 Experiments in Failure	15
2.3 Recap.....	15
2.3.1 Terminology	15
2.3.2 Highlights	18
<b>3 THE DICE GAME</b>	<b>20</b>
3.1 What's Ahead? .....	20
3.2 The Design .....	20
3.2.1 Notes on UML	20
3.2.2 The Rules of the Game	21
3.2.3 One Development Strategy	21
3.3 The First Test.....	21
3.3.1 Create the test file	23
3.3.2 Die Header	23
3.3.3 Die Source	25
3.4 Exercise.....	25
3.4.1 Results	25
3.4.2 Exercises in Failure	26
3.4.3 Concept Review	27
3.4.4 Final Observation	29

---

3.5	Making Improvements .....	29
3.5.1	Updated Test	29
3.5.2	When to Clean Up	30
3.5.3	Updated Die Header File	30
3.5.4	Updated Die Source File	30
3.5.5	Exercise: Update Die	31
3.6	DRY Violation .....	31
3.6.1	Removing Duplication	31
3.7	A Message on Test Granularity .....	33
3.7.1	Recap	34
3.8	Fixing an anemic roll method.....	35
3.8.1	Validating Roll Distribution	35
3.8.2	Updated Header File	41
3.8.3	Updated Source File	42
3.8.4	Recap	47
3.9	C++ Idioms .....	51
3.9.1	Updated Header File	51
3.9.2	Updated Source	52
3.9.3	Recap	53
3.10	What's coming up? .....	54
3.11	Review Game Rules .....	54
3.11.1	Test Control	55
3.11.2	Dependency Injection	56
3.11.3	Polymorphism Moving Parts	57
3.12	Testing Into It: LoadedDieTest.....	58
3.12.1	Options: Interface/Concrete Inheritance	58
3.12.2	LoadedDie Implementation	59
3.12.3	Get your test passing	59
3.12.4	Experiment in Failure	59
3.12.5	Fixing It	60
3.12.6	Overloading faceValue versus roll	60
3.12.7	Fixing Die: Command Query Separation	62
3.12.8	Review	62
3.13	What's on Deck?.....	62
3.14	Test-Driven Walkthrough.....	63
3.14.1	What's required to make this work?	63
3.14.2	DiceGame Header	64
3.14.3	DiceGame Source	64
3.14.4	Get it Compiling	65
3.14.5	Handle the memory leak, fix the test	66
3.14.6	Always losing is no fun	68
3.14.7	Experiments in Failure	70
3.15	Recap .....	75
3.16	What's coming up? .....	79

3.16.1	Remember Cup class?	79
3.16.2	Refactoring: Definition	80
3.16.3	Updated Cup Header	80
3.16.4	Updated Cup Source	81
3.16.5	Getting to Compiling	81
3.16.6	Updating DiceGame	82
3.16.7	What of the idiom?	86
3.16.8	A Logical Fix to Cup	87
3.17	What is going on with const? .....	88
3.18	Taking Small Steps, Recap .....	91
3.19	Memory Allocation .....	92
3.19.1	std::shared_ptr	92
3.19.2	Fixing DiceGame	94
3.19.3	Fixing Cup	96
3.20	Warning: Circular References .....	97
3.20.1	The Problem: A concrete example	97
3.20.2	Options	100
3.21	Recap .....	102
3.22	What's Coming Up? .....	102
3.23	A Few Built-In Algorithms .....	103
3.23.1	Updated roll()	103
3.23.2	Updated total()	106
3.23.3	Recap	108
3.24	Improved Test Writing? .....	109
3.24.1	Pass a factory into DiceGame	109
3.24.2	Oh Wait, testability	110
3.25	The 4-contact points of software development.....	110
3.25.1	Why?	111
3.26	Create a concrete Factory.....	112
3.26.1	First Test against the Factory	112
3.26.2	Define the class: LoadedDieFactory	113
3.26.3	Define the methods: LoadedDieFactory	113
3.27	Update the cup .....	113
3.27.1	The Test	113
3.27.2	A new constructor	114
3.28	Dice Game Instantiation .....	114
3.28.1	First a test	114
3.28.2	Notice a pattern? New Constructor	115
3.28.3	Update the second test	115
3.28.4	Oops, not there yet	116
3.29	Extending Loaded Die Factory .....	116
3.29.1	Here's a test	116

3.29.2	The Updated Class	116
3.29.3	Return to green	117
3.29.4	Back to that final test	117
3.29.5	Why modify this final test at all?	118
3.30	Final Cleanup.....	118
3.30.1	DiceGame	118
3.30.2	Cup	118
3.30.3	Common typedef	118
3.31	Is this better? .....	119
3.31.1	Can we even play a real game	119
3.31.2	Problem with Test Doubles	119
3.32	Refactor: Extract Interface.....	119
3.32.1	The Class Definition	119
3.32.2	Implementing the pure virtual destructor	120
3.32.3	Update LoadedDieFactory	120
3.33	Now DieFactory .....	120
3.33.1	First the test	121
3.33.2	The Implementation	121
3.33.3	Get to Green	122
3.34	A Smoke Test .....	122
3.34.1	Make the required updates	122
3.34.2	Back to green	123
3.34.3	Where does this test belong?	123
3.35	Wrap-up.....	123
3.36	Final Recommendations.....	124
3.36.1	Books	124
3.36.2	Katas	124
3.36.3	Practice	124
3.37	What's coming up? .....	125
<b>4</b>	<b>RPN CALCULATOR</b>	<b>126</b>
4.1	Project Description.....	127
4.2	What's Coming Up? .....	127
4.3	Biting off just enough.....	127
4.4	Develop Examples .....	128
4.5	Project Setup .....	129
4.6	The first unit check.....	129
4.6.1	Was all of this necessary?	132
4.6.2	Add: The First Test	132

---

4.7	Subtract: The Second Test.....	134
4.8	What about Actual Values? .....	135
4.9	Drop .....	136
4.9.1	First introduce the stack .....	137
4.10	Getting Factorial Working.....	139
4.11	Revisit Add .....	140
4.11.1	Feature Envy .....	141
4.12	Resolving Feature Envy: Writing Our Own Stack.....	141
4.12.1	First test: top works on an empty stack .....	142
4.12.2	Update RpnCalculator .....	144
4.13	Finish subtract .....	144
4.14	Dreaded Duplication or DRY .....	146
4.14.1	Extract Classes .....	147
4.14.2	Keeping it the same .....	149
4.14.3	Updating Subtract .....	149
4.14.4	Drop .....	150
4.14.5	Factorial .....	151
4.15	Removing Duplication .....	151
4.15.1	It consumes two values .....	152
4.15.2	It calls an extension point with the correct parameters .....	156
4.15.3	It stores the result .....	157
4.15.4	Updating Add and Subtract .....	157
4.15.5	Not updating drop .....	158
4.16	All those methods .....	158
4.16.1	An example based migration .....	158
4.16.2	Migrating the subtract() method .....	159
4.16.3	Finish the transformation .....	160
4.17	Type un-safe.....	161
4.18	Long Method.....	162
4.19	A Concrete Factory .....	164
4.19.1	Actually using the factory .....	165
4.20	Retargeting Automated Checks.....	166
4.20.1	Add .....	168
4.20.2	Drop .....	169
4.20.3	Factorial .....	170
4.20.4	MathOperation .....	171
4.20.5	MathOperationFactory .....	171
4.20.6	Subtract .....	171
4.21	Adding Multiplication .....	172
4.22	Adding Division .....	173

4.23	MathOperationFactory refactoring: Storing Math Operations.....	175
4.24	MathOperationFactory refactoring: Automatic Math Operation Registration .....	177
4.24.1	An object for registration	177
4.24.2	Automatically Register Multiplication	181
4.24.3	Split registrant	182
4.25	Add Missing Examples .....	183
4.26	Sprint Summary .....	184

## **5 RPN CALCULATOR – SPRINT 2 – GROWING FEATURES 186**

5.1	Adding Sum .....	186
5.2	Less Than.....	188
5.3	Equal To and Greater than.....	189
5.4	Swap XY .....	189
5.5	Dup.....	191
5.6	N Dup .....	191
5.7	Prime Factors .....	193
5.7.1	Of 2 ...	194
5.7.2	Of 3...	195
5.7.3	Of 4 ... multiple values	195
5.7.4	Of 5 ...	195
5.7.5	Of 6 ... two values, but they are different	195
5.7.6	As will 7 ...	196
5.7.7	But 8 is different, 3 values, instead of just 2.	196
5.7.8	Is 9 different?	196
5.7.9	Register It	197
5.8	Examples as Rejection Checks .....	197

## **6 RPN CALCULATOR – SPRINT 3 – MACROS 200**

6.1	Happy Path.....	200
6.1.1	A Macro	201
6.1.2	Adding to factory	203
6.1.3	Adding it to RpnCalculator	204
6.2	Empty macros not allowed .....	205
6.2.1	Must call start first()	206
6.2.2	Unknown operation cannot be added to a macro	206
6.2.3	Cannot save under existing name	207
6.2.4	Adding missing check on the factory	209
6.2.5	Macros can refer to other macros	209
6.3	Cleaning up the calculator.....	209
6.3.1	Calculation Mode	212

6.3.2	Executes Operations Directly	214
6.3.3	Throw exception when told to save	214
6.3.4	Change to Programming Mode When Told To Start	215
6.3.5	Programming Mode	217
6.3.6	Record Steps for Execution	218
6.3.7	Adding macro to factory	219
6.3.8	Saving causes state change	219
6.3.9	Other checking	220
6.3.10	What about the start method?	222
6.3.11	Ready to finish what we've started...	222
6.4	Updating RpnCalculator to use state...	222
6.4.1	Final Cleanup	223
6.4.2	Summary	223
<b>7</b>	<b>RPN CALCULATOR – SPRINT 4 – MORE COMPLEX BLOCKS</b>	<b>224</b>
7.1	Output Operations.....	225
7.1.1	The "." operator	225
7.1.2	It Should Be Registered...	228
7.2	Emit and a problem with growing interfaces.....	228
7.2.1	Emit should be registered	229
7.3	Finally, cr .....	230
7.4	Migrating to new Perform Interface.....	231
7.4.1	BinaryMathOperation	231
7.4.2	Update the calculator	233
7.4.3	The magic of checks	234
7.4.4	The Newest Math Operations	235
7.4.5	Dup	235
7.4.6	What Remains	236
7.5	Numeric Constants as Operations.....	236
7.6	If ... then ... else .....	237
<b>8</b>	<b>RPN CALCULATOR – SPRINT 5 – FITNESS &amp; CSLIM</b>	<b>238</b>
8.1.1	A spec-driven example	238
8.1.2	A sequence diagram showing flow	238
8.2	Adding a basic text ui .....	238
8.3	Adding several more operators.....	238
8.3.1	ifelse	238
8.3.2	ntimesdo	238
8.3.3	ConditionWhileDo	238
8.4	Programming the Calculator with a string .....	238
8.4.1	Example forth program	238
8.4.2	Breaking it into parts	238
8.4.3	Building a Basic Sequence	238
8.4.4	Building a conditional sequence	238



8.4.5	Building a complex sequence	238
8.4.6	Adding the behavior into the calculator	238
8.4.7	Exercising the new behavior from the text ui	238
8.4.8	Saving your extensions	238
<b>9</b>	<b>WHERE TO GO NEXT?</b>	<b>239</b>
9.1	TDD Is Not Enough	239
9.1.1	GRASP	239
9.1.2	SOLID + D	239
9.1.3	Code Smells	239
9.1.4	WELC	239
9.1.5	Test Doubles	239
9.1.6	Coding Katas	239
9.1.7	The 4 Actions (should be sooner)	239
<b>10</b>	<b>APPENDIX A: REVEALING THE MAGICIAN</b>	<b>240</b>
10.1	Arrays versus pointers	240
10.1.1	Koenig's <code>i[3] == 3[i]</code> trick	240
10.2	Methods versus functions	240
10.3	Operator Overloading	240
10.4	Overloading <code>&lt;&lt;</code>	240
10.5	Overloading <code>++i</code> versus <code>i++</code>	240
10.6	Virtual Functions	240
10.7	<code>new</code> & <code>delete</code> versus <code>malloc</code> & <code>free</code>	240
<b>11</b>	<b>APPENDIX B: MORE COMPLEX COMPOSITION WITH BIND</b>	<b>241</b>
<b>12</b>	<b>APPENDIX C: FITNESS, A QUICK INTRODUCTION</b>	<b>242</b>
<b>13</b>	<b>TO BE DELETED</b>	<b>243</b>
13.1	My Story Until 2010	243
<b>14</b>	<b>INDEX</b>	<b>247</b>



## Introduction

I started using C++ and Smalltalk in 1989; C++ as a research assistant, and Smalltalk in an Object Oriented Languages Seminar, both at the University of Iowa. In the spring semester of 1990, while taking an advanced Operating Systems course, I made a decision to use C++ (C Front 1.1), not allowing myself to fall back to “plan old C” until I understood C++. That was the first and luckily only time I didn’t finish a project. It was the first project of the course and it turns out I was trying to create objects with virtual member functions in shared memory, which was a no-no. In any case, I continued using C++ and Smalltalk for some years. My chances to use Smalltalk diminished faster than C++.

C++’s library, while meager, grew. The boost library came into existence and then became quite a substantial collection of classes. Many of those classes are useful, some esoteric and several make the language behave more like some people would like it to behave.

In 1997, I finally switched to Java on my 3<sup>rd</sup> attempt. I had tried prior to Java 1 with some success. Then about the time Java 1 was released I tried again. A few months after Java 1.02 was released, I finally decided to jump the C++ ship. So I stopped using C++ professionally in 1997. For several years I managed to not use it. I’d occasionally take a look but for the most part, I did not use it between 1997 and around 2007.

In 2007 I joined Object Mentor and several of the jobs involved working with large legacy C++ code bases. So I managed to start picking it up again. However, unlike my last time using C++, this time I had a few more years of design experience, large system development and support, experience with test driven development and myriad other experiences that made my approach substantially different.

### 1.1 Why this book now?

If that were the end of the story, then I would not have written this book. Even though I noticed that the approach to development with C++ was missing out on several modern ideas, I didn’t see a need. Then in mid-2010 my good friend David Nunn asked if I’d like to teach a C++ and Object Oriented Design class at NASA. Of course I said yes.

I did some looking around to try and find some material that included good Object Oriented Design principles, C++ idioms, modern design practices such as the use of Test Doubles for test isolation, etc. I did not find anything. I looked at the course that Object Mentor offered and it wasn’t quite what I was looking for either. I considered a few other avenues but in the end, I did not find anything that fitted the particular situation, so I built my own C++ class (for the 4<sup>th</sup> time actually).

I designed a course with the intention of introducing C++, the basics of Test Driven Development, basic design principles, deeper design principles and design patterns. All of this to be introduced through exercises. In my original course design I planned for 2 projects. I added a third “simple” problem at the front, which became the first problem as it blossomed into something at once simple and rich enough to cover much of the language my students needed to know to start becoming effective C++ programmers.

While teaching this class, I finally came to the realization that there really is not yet a book that covers learning just the part of C++ you need to know for Object Oriented Programming. I looked but I did not find anything that quite fit the bill. There are excellent books to be sure. Nothing like the focus I wanted. Additionally, the books available did not take the approach I've taken in this book; embedded exercises and deep dives into the language to provide additional context to what was happening in the problem.

I had a similar experience like this back in 1988 when, faced with having my students shell out over \$100 USD for books on micro-computers (a term probably before most reader's time), none of which the students would need after the class, I decided to write a book for a computer literacy course I taught at a local community college.

While the book was initially published in 1988, it grew. It went from about 110 pages to over 300 and was used for roughly 9 years before being ousted for something more appropriate. I hope that's what happens with this book as well. I'd like to think that people learning C++ will stop learning all of the language and focus on learning how to effectively use the language. To me, that means not using much of the language's power. At least as a first step towards learning C++, the subset I'll cover is a good start. Learn how to do things somewhat cleanly, and then learn the guts of the language if you plan to write libraries or you just want to be a language nerd. C++ offers many opportunities to do so, but you don't need to start there.

## 1.2 How is it different?

As the title suggests, I do not intend to cover all of the language. In fact, I'll cover a comparatively small subset of the language and the standard libraries. Even so, that is not the primary difference. This book is meant to be an example of a journey through two problems.

The approach for each problem is similar; start with simple goals, write a solution. Review that and fix it as necessary. As I work through the solution, I'll be articulating forces driving my decisions. I'll be bringing into the mix things from C++, Test Driven Development, Refactoring, C, Test Isolation, Agile Software Development, etc.

Also, there will be two reasons why I cover something in the language. First and primarily, it will come up as a response to something in the current problem. Second, there will be background and context I might cover to fill out details a little bit. There will be times when I pick one solution over another to delve a bit more into the language or library. However, when I make that kind of decision, you'll know because I'll tell you.

Another key difference with this book is that I intend for you to write code as you work through this book. That, in and of itself, is not unique. What is, however, is that I'm going to provide you tutorials so that you'll be able to write working code. Something that has always frustrated me is seeing code snippets with just enough context to get you interested but leaving "some assembly required" for the reader.

To address this, I have done three things:

- First, I have picked a tool set. It's free and should be available on Windows, OS X and most UNIX varieties.

- Second, I have uniquely identified all of the tutorial sections with GUID's (globally unique identifiers – that's a geeky way of saying something obvious)
- Finally, I have included the same detailed instructions for other platforms online (<<site URL here>>)

To make this a bit easier, each tutorial section will cover mechanics at the beginning and then get into coding. As a result, there will be times where I have you do mechanics work ahead of time to reduce the number of unique times where you have to do that.

So even if you do not know C++, you should be able to get working examples. I hope that the text and code examples will enable you to pick up quite a bit of C++ along the way. I'll be rewarding the ability to copy the provided code, however, with passing unit tests.

As a final difference, you'll be running everything under the umbrella of a unit test. We will cover 2 projects. This means you'll create two main programs. After that, all execution will be from tests written using a unit testing framework. I'll provide both working code and failing code so you can see the errors. In some cases I'll deliberately have bad designs; in some cases I'll ask you to perform some experiments in your code. The experiments are a controlled way for you cause a failure, know why the failure happened, and know how to fix it. So rather than avoiding all errors, I'll try to give you a chance to experience them in a safe manner.

### 1.3 What not to expect?

I'm assuming you have some basic background in C for reading this book. That's really my target audience. I think you can use this book if you're learning C++ from other languages, but I'm not going to make any wild guesses as to whether that will or will not work. If you are using C++ or learning C++, then you mostly likely have been using C in some capacity. I have not recently come across anybody learning C++ as their first language, so that is not what I am targeted. Having said that, I will be delving into C stuff every so often to explain why C++ is the way it is. You will be able to pick up this book and start programming in C++

Bottom line, you are not going to learn all of C++. In fact I'd guess you'll learn maybe 30% of the language (I'm guessing at the number because a precise number without context isn't of much value). However, of that 30% you do use, it'll be what you need 90% of the time. The other 10% comes with practice, experience and the many references I'll suggest to you later in the book.

This is the start of a journey. I hope it will set you on a good course and heavily influence your use and future learning of C++.

### 1.4 Standing on the Shoulders of Giants

The only thing I'm offering in this book is my particular combination of choices; there is nothing new in this material.

First and foremost, I need to give appreciation to several early influences in my software development career. My early interest in computers was fed by my friend John Navitsky. I was given a summer job by Marjorie Scriver working in a second-hand shop, which allowed me to earn enough money to buy my first computer. One week before I turned

18, Dr. Gretchen L. Moine hired me and some of my friends to work in a computer lab at Kirkwood Community College. When two of her professors left, she gave us the opportunity to teach computer literacy courses, and then like a mother hen, she defended us as the main campus tried to get rid of us. If I have any ability as a teacher, it is directly related to early, eager and consistent stewardship.

I had several good professors at The University of Iowa and one in particular got me started learning about OO programming, Dr. Mahesh Dodani. He offered an “Advanced Seminar in OO Programming” in the Fall semester of 1989. I used Smalltalk, but more importantly, because I took that class I was eligible to get a job as a research assistant developing a large C++ application. My friend Jeff Francis was instrumental in helping me with the C part of C++, and some of the basics, like using CVS. Unlike course work, we had to demo our work to Ford Motor Company and TACOM, so it was an amazing learning experience. I would not have started using C++ if not for Mahesh.

When I started using C++, Google did not exist and there were no books on C++. So when books came out, I bought them and devoured them. C++ is the invention of Bjarne Stroustrup. The first book I read on C++ was the original printing of The Annotated C++ Reference. The second book I read was by James Coplien, Advanced C++ Programming Styles and Idioms. Many books and articles followed. Anybody who’s read the work of Andrew Koenig will probably recognize his influence on my mental model of the language. Much of how I think about the language was certainly influenced by his book C Traps and Pitfalls and his many excellent articles in the C++ Report, many of which are available in Ruminations on C++.

James introduced me to book reviews. I purchased the first edition of his book Advanced C++ Programming Styles and Idioms. I read it and began an email conversation with him. As a result of giving him feedback, he connected me with his publisher and I ended up reviewing books. One of those books was Robert Martin’s first book Designing Object Oriented C++ Applications using the Booch Method.

There are many more, too many to remember. So I’m only bringing my perspective to material that already existed in other forms written by other people.

<fill in? shorten?>

## 2 Obligatory Hello World

Traditional language books often cover a so-called hello world “application.” Just for completeness, here you go:

```
#include <iostream>

int main(int argc, char *argv[]) {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

However, that’s the last example of code that doesn’t include some form of self-validation. Throughout this book I’ll be using a unit testing framework in lieu of a `main()` program. You won’t see much explicit output either. You are welcome to add output yourself, but I’m going to rely on programmatic execution and validation.

This section covers the “hello world” meme, unit test style. That’s what the rest of this section covers.

### 2.1 Setting up your environment

#### 2.1.1 The Tools

For this book, I will be using the following tool set:

- Eclipse CDT<sup>1</sup>
- Wascana Eclipse Plugin
- CppUTest

The first tool is our IDE<sup>2</sup> and it requires that you have a Java VM<sup>3</sup> installed. The second is a plugin to Eclipse that gives you a complete C++ tool set. The final item is a library that gives you support for writing automated tests. It is called a unit test framework, but its ability to assist in writing automated tests is, for my purposes, its most important feature.

I have chosen these tools for several reasons:

- They are all free and open source
- They work cross platform
- They give a consistent way to work across those platforms – other than default shortcut keys
- In the case of CppUTest, it does basic memory leak detection out of the box. Since you are reading this book to learn C++, something helping with memory leak detection is an important feature.

What follows are detailed steps for installation of these tools.

---

<sup>1</sup> C Development Toolkit.

<sup>2</sup> Integrated Development Environment.

<sup>3</sup> Virtual Machine.

### 2.1.2 Installing the Java Developers Kit (JDK)

You'll need the JDK (or its equivalent) to run Eclipse. The easiest way to install the JDK is to run an installer from <http://www.java.com/> for Windows, Linux and Solaris platforms. In the case of OS X, you already have a JDK installed.

### 2.1.3 Installing Eclipse CDT

Download the Eclipse CDT from <http://www.eclipse.org/cdt/>. This book was written using 7.0 of the Eclipse CDT, which is based on Eclipse Helios. Older versions may work; newer versions almost certainly will work.

You will be downloading a zip file. You can safely unzip it anywhere. You will only need to know where you downloaded it for two reasons:

- Primarily, you'll be running it, so you'll need to find the top-level executable
- One time only, you may be using programs installed underneath the eclipse directory, added by an Eclipse plugin.

To keep everything self-contained, I'll be storing everything under a directory called *learncpp*. This avoids spaces in paths and it will make it easier to find everything you do. If you choose to use a different directory, make sure to replace *learncpp* with the directory you used. Here are two example full path names for an idea of just where I'm putting things:

- Under windows: *C:\learncpp*
- Under OS X: *~/learncpp* → which actual becomes */Users/Schuchert/learncpp*

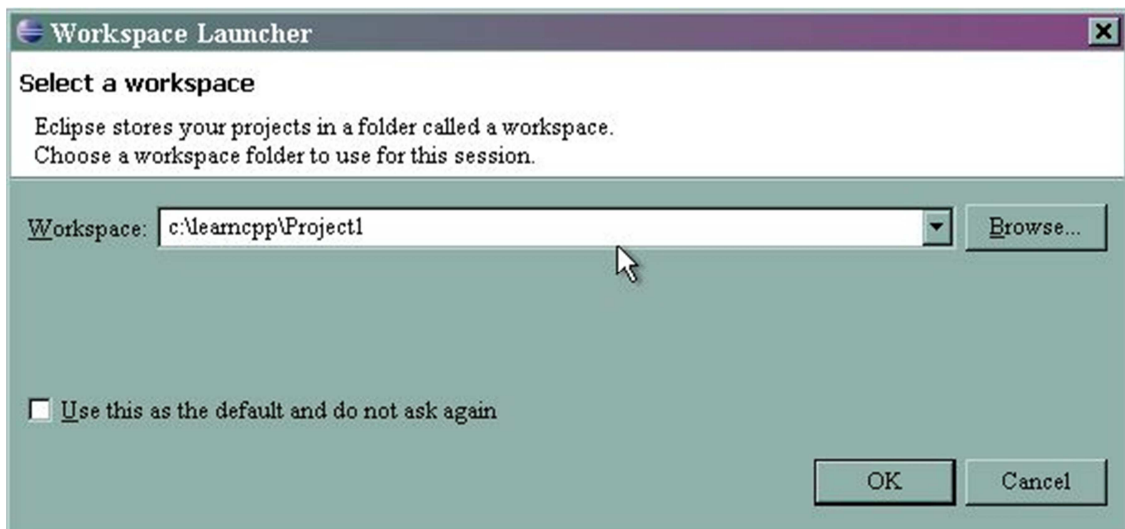
If you extract the Eclipse CDT zip file to *learncpp*, it will create:

- Under windows: *C:\learncpp\eclipse*
- Under OS X: *~/learncpp/eclipse*

### 2.1.4 Starting Eclipse CDT

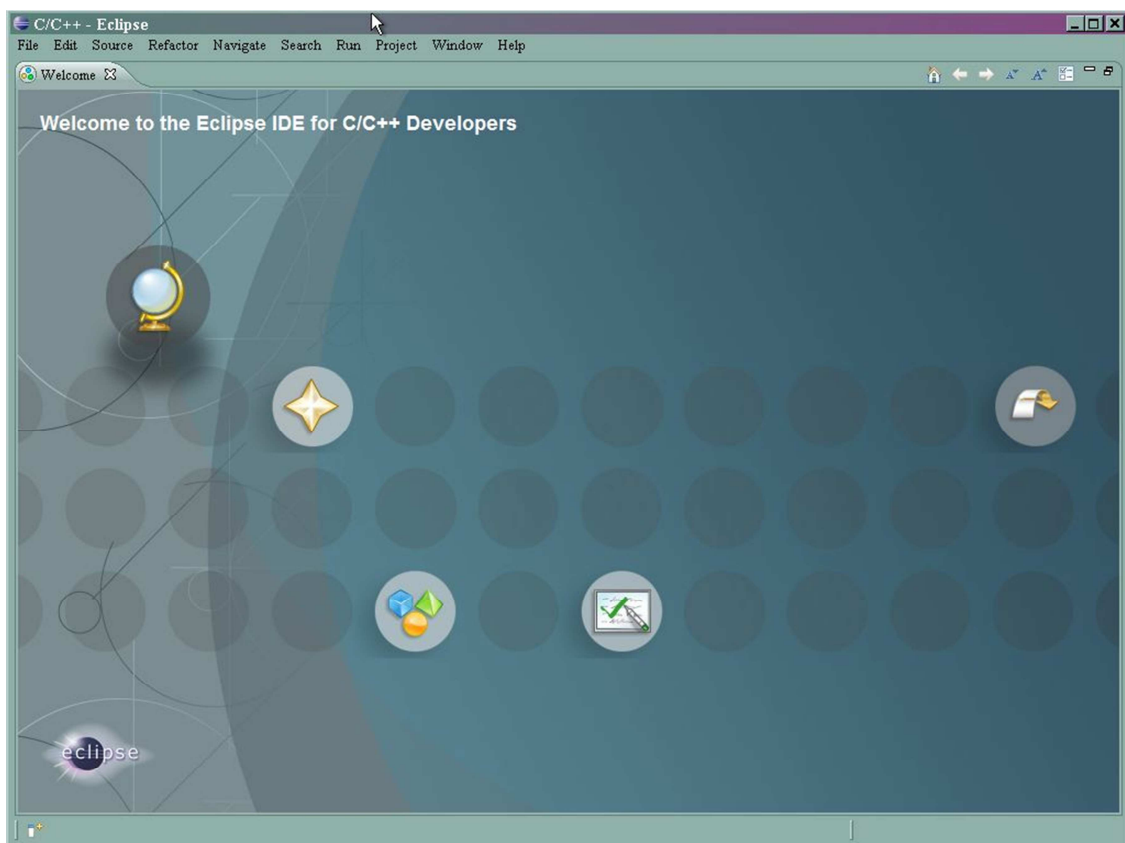
Now that you've installed Eclipse, you can start it. Double-click on the eclipse application under the eclipse directory. When you do, you will be asked to provide a location for a workspace:





Enter the location for a project in the Workspace text box. For this first project, I'm using *c:\learncpp\Project1*.

After a few moments, Eclipse will respond with the welcome screen:



Close the welcome screen by clicking on "x" the right side of the *Welcome tab*.

Before going any further, you now need to install the *Wascana* plug-in if you do not already have the Gnu C++ compiler 4.4 or later installed (OS X Users, you have G++

installed, but it's probably earlier than 4.4, so you'll need to use a port package to install it – give yourself several hours for this).

### 2.1.5 Installing the Wascana Plugin

This step is primarily for Windows Users. However, I have online instructions for OS X users at <cite site>.

The *Wascana* plugin gives you a minimal Unix environment, gcc 4.4, gdb and other basic tools to make the Eclipse CDT work. It is installed as an Eclipse plugin.

The basic instructions are here: <http://code.google.com/a/eclipselabs.org/p/wascana/>. But quickly:

- Pull down the *Help* menu in Eclipse and select *Install New Software*.
- In the *Work with* text box, enter:  
<http://svn.codespot.com/a/eclipselabs.org/wascana/repo>
- Then click *Add*
- Enter a name like *Wascana* in the pop-up dialog
- Select the *Wascana C/C++ Developer for Windows*
- Click on *Next* twice until you get the *Review Licenses* screen
- Assuming you accept the terms of the license, select the *Accept* radio button and click *Finish*.

Wait for the download to complete and the restart Eclipse. When asked for the workspace, verify that C:\learncpp\Project1 is the workspace and click *OK*.

### 2.1.6 Downloading CppUTest

CppUTest is a unit testing framework we'll be using throughout this book. You'll need to download and build it one time. Unfortunately, to do this, you will need to update your path, but that's in the next section.

For now, download CppUTest from: <http://sourceforge.net/projects/cpputest/>. As of this writing, I'm using CppUTest 2.2d.

Once you've downloaded that zip file, unzip its contents to *c:\learncpp\CppUTest*. Note that the zip file does not appear to have a top-level directory, so you'll need to create it.

### 2.1.7 Building CppUTest

When you installed the Wascana plug-in, it added two directories under your eclipse directory: mingw and msys. Both of these directories have bin directories that need to be added to your path.

Begin by updating your path to include the *bin* directories under mingw and msys. If you used the recommend directories, their full paths are:

- C:\learncpp\eclipse\mingw\bin
- C:\learncpp\eclipse\msys\bin

Since these paths are also used by Eclipse, you'll need to set your environment's path. To do this, you'll need to update the PATH environment used by your command shell.

- Under the control panel, select system preferences (or under the category System and Security, select System).

- Select Advanced Features.
- Add the two directories to the PATH environment variable under system preferences by appending: `;C:\learncpp\eclipse\mingw\bin;C:\learncpp\eclipse\msys\bin`
- Note that there is one ; (semi-colon) between directory entries.
- If you cannot edit the system preferences, instead add a new entry under User variables. It's value should be: `%PATH%;C:\learncpp\eclipse\mingw\bin;C:\learncpp\eclipse\msys\bin`

Now switch to the CppUTest directory. If you used the recommend directories, the full path is: `C:\learncpp\CppUTest`.

By adding the bin directories from mingw and msys, you have some basic UNIX tools such as gcc, g++, make, pwd, etc. Now it's a simple case of issuing the *make* command:

```
C:\learncpp\CppUTest>make
compiling AllTests.cpp
compiling CommandLineArgumentsTest.cpp
compiling CommandLineTestRunnerTest.cpp
compiling FailureTest.cpp
<snip>
compiling TestRegistry.cpp
compiling TestResult.cpp
compiling Utest.cpp
compiling UtestPlatform.cpp
Building archive lib/libCppUTest.a
c:\Program Files\eclipse\mingw\bin\ar.exe: creating lib/libCppUTest.a
a - src/CppUTest/CommandLineArguments.o
a - src/CppUTest/CommandLineTestRunner.o
<snip>
a - src/Platforms/Gcc/UtestPlatform.o
Linking CppUTest_tests
Running CppUTest_tests
.....!.....
.....!.....
.....
...!!.....
OK (171 tests, 167 ran, 599 checks, 4 ignored, 0 filtered out, 16 ms)
```

```
C:\learncpp\CppUTest>
```

Note: Depending on when you try these steps, the build might fail because it cannot find “cc”. To fix this, update the Makefile and include the following line after the comment line with Inputs in it (#--- Inputs):

```
CC = gcc
```

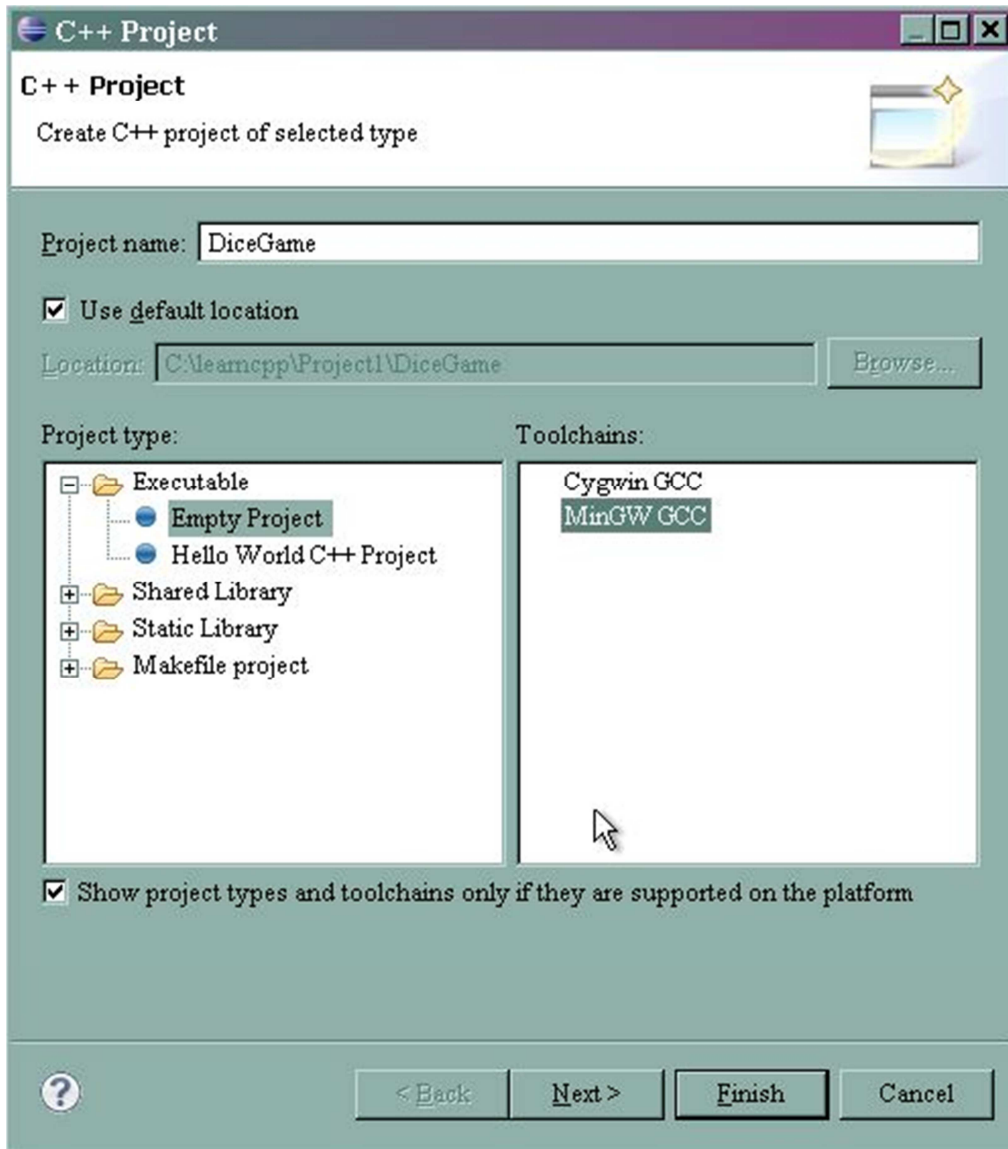
You can look at the *lib* directory to confirm that you have a file called libCppUTest.a. However, given running unit tests, it really isn't necessary.

## 2.2 Mechanics of CppUTest

Now that you have all of the moving parts installed, you need to configure an Eclipse project to use CppUTest and then get your first test running.

## 2.2.1 Creating a Project

Now it's time to go back into Eclipse and create a project. Begin by selecting *File:New* then select *C/C++ Project*. You'll see the following dialog:



Enter `DiceGame` for the Project Name, and make sure you've selected *MinGW GCC* as the Toolchain. Once you've done these two things, click *Finish*.

## 2.2.2 Include Path, Library Path and Included Libraries

Now that you have a project listed under the *Project Explorer*, you can configure it. Select that project, right-click and select *Properties* at the bottom of the popup menu.

### **Include Path**

Open up *C/C++ Build*, select *Settings*. You'll see several sub-options. Look for *GCC C++ Compiler* and under it, select *Includes*. Click the add button, which looks like a page with a green plus. In the dialog, enter: `C:\learncpp\CppUTest\include`

Before moving on, click *Apply* to make sure these changes stick.

### **Library Path & Included Libraries**

Again, under *C/C++ Build, Settings* look now for *MingGW C++ Linker* and select *Libraries*. There are now two sections, Libraries (-I) and Library search path (-L). In Libraries, add *CppUTest*.

Under the Library search path, add: `C:\learncpp\CppUTest\lib`.

Note that the full name of the library is libCppUTest.a. However, when specifying a library with this tool set, you exclude the leading "lib" and the file extension.

Before moving on, click *Apply* to make sure these changes stick.

### **Modernizing C++**

There are several modern features supported by GCC but not available until you tell the compiler to allow for them.

- Find *GCC C++ Compiler:Settings*
- Select the *Miscellaneous* option, notice that its current value is probably:  
`-c -fmessage-length=0`
- Add (append) the following additional setting: `-std=c++0x`

That is C++ zero x, not letter o x.

Before moving on, click *Apply* to make sure these changes stick.

### **That's All the Project Settings**

Finally click *OK* to close the dialog.

## 2.2.3 A Few Convenient Settings

There are some top-level (not project-specific) settings that will improve your overall experience. To begin making changes, pull down the *window* menu and select *Preferences*.

### **Auto Save and Refresh**

Under the preferences you'll see a box that contains "enter filter text", type *save* in that box to shorten the list of options.

Select *Workspace* under the *General* option and enable the following settings (the first should already be set):

- Build automatically
- Refresh automatically
- Save automatically before build

Before moving on, click *Apply* to make sure these changes stick.

**Build automatically** only affects when you perform a “clean” on a project. This will remove all the built files. Build automatically will then turn around and rebuild the system by default. It will not rebuild after you save a single file.

**Refresh automatically** is important because building causes an executable to be created. Sometimes Eclipse doesn’t notice that this executable was created and when you try to run your program, Eclipse will complain about no executable existing. Setting this option fixes that problem.

**Save automatically before build** will cause Eclipse to ask you to save before each build. If you do not, you can change a file, not notice it is unsaved and then get compilation errors because the file is not saved and not because of anything you typed.

### **One Last Thing Run**

Eclipse attempts to run something relative to the file or think currently selected when you use the shortcut key ctrl-F11. This can be confusing behavior.

**Under Run/Debug**, find **Launching**. Under the **Launching** section near the bottom, select the radio-button with the text “Always launch the previously launched application.”

Before moving on, click **Apply** to make sure these changes stick.

### **That’s All the Workspace Settings**

Finally click **OK** to close the dialog.

## 2.2.4 Adding a CppUTest main() function

Now you are ready to create a main function. As with C, main() is the entry point into any application in C++. Main can have one of two forms:

```
int main()
```

And the more traditional:

```
int main(int argc, char *argv[])
```

With that in mind, create the following main program:

```
#include <CppUTest/CommandLineTestRunner.h>

int main(int argc, char *argv[]) {
    return CommandLineTestRunner::RunAllTests(argc, argv);
}
```

### **Quick File Description**

The first line includes a class that runs unit tests. The top-level include directory for CppUTest has a directory under called CppUTest. This is important because it makes a namespace of sorts for header files. By starting the name with CppUTest, there is a much smaller chance of name collision in include file names.

This is a standard main() function that uses a class called `CommandLineTestRunner`. That class has a method on it called `RunAllTests`. You pass in `argc` and `argv` and it determines what to do based on command line arguments. We are not passing in anything, so we’ll get the default behavior provided by the library.

The return statement on that line makes sure that the result of running tests is reflected back in the shell that started the program. We won't be making use of this, but if you were to include running unit tests as part of a larger build process, you could use test failure as a reason to stop the build process. The return here facilitates that.

#### **Create this file**

- Select your project, right click and select *New*.
- In the list, select C++ *Source file*
- For the name, enter: RunAllTests.cpp

#### **Build your project**

Compile your project by hitting ctrl-b. Alternatively, right-click and select **Build**.

#### **Run your project**

The first time you run your project, you'll want to select it, right-click and select **Run As:Local C/C++ Application**. You should see output like this in the Console:

```
OK (0 tests, 0 ran, 0 checks, 0 ignored, 0 filtered out, 0 ms)
```

Congratulations, you have all of the major moving parts in place and working.

### 2.2.5 Adding a first test

However, don't celebrate too quickly, because now it is time to add a test. Here is a simple "smoke" test to verify that things work.

```
#include <CppUTest/TestHarness.h>

TEST_GROUP(SmokeShould) {
};

TEST(SmokeShould, NeverBeLost) {
    LONGS_EQUAL(1,1);
}
```

#### **Quick File Description**

This is mostly boilerplate code. The #include statement, among other things, makes available a set of macros to help in the definition of tests. There are three major parts:

Macro	Description
TEST_GROUP	<p>Introduce a place holder for a number of tests. Typically know as a test fixture. The name, <i>SmokeShould</i>, is somewhat arbitrary.</p> <ul style="list-style-type: none"> <li>▪ This name must be a valid C++ identifier;</li> <li>▪ It should make sense;</li> <li>▪ It must be unique across all tests in a single project (across all the library and object files linked to make a single executable).</li> </ul> <p>This macro introduces a <code>struct</code> into your solution. The name of the <code>struct</code> embeds the parameter name, <i>SmokeShould</i> in this case. A <code>struct</code> is a collection of data members in C. In C++ it can also include member functions. CppUTest does this as it forms a natural place to put common code and data used between individual</p>

Macro	Description
	automated tests.
TEST	<p>Introduce a single test. It depends on a test fixture, <i>SmokeShould</i>. We will be taking advantage of this later on. However, it is a requirement of the particular tool, so you must do it. The name of this test is <i>NeverBeLost</i>. This name must be unique within the fixture (<i>SmokeShould</i>); it should make sense; it must be a valid C++ identifier.</p> <p>This macro introduces a <code>class</code> into your solution. This <code>class</code> is a subclass of the <code>struct</code> created by the <code>TEST_GROUP</code> macro. CppUTest does this to enable automatic registration of unit tests possible. If an object module that contains a <code>TEST</code> is linked into your final executable, it will be available for execution by the <code>CommandLineTestRunner</code>.</p>
LONGS_EQUAL	This macro performs a check. A check either returns or does not. If the check returns (in this case the longs are equal so it returns), then continue with the test. If it does not return (say, the longs were not equal), then this test would terminate and be a failed test. In this unit test framework, and traditionally across most of them, the first value is the expected value; the second value is the actual value.

### Create This File

Create a new file called `SmokeTest.cpp` (ctrl-n, add source file).

### Run Your Tests

Since you've run your project once, you can now use the shortcut key ctrl-F11 (mac: Command-Shift F11) to re-run your project.

^ You did not save your file, Eclipse will confirm that you want to save before building; do so. The project will run and you will see the following output:

```
.
OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 0 ms)
```

The period represents the one test that executed. If that is too terse for your tastes, update `RunAllTests.cpp`:

```
#include <CppUTest/CommandLineTestRunner.h>

int main() {
    const char *args[] = { "", "-v" };
    return CommandLineTestRunner::RunAllTests(2, args);
}
```

### Quick File Description

Rather than passing in the command line options, hard-code them. In UNIX systems, the first parameter passed into a C or C++ program is the name of the executable. Some programs use this to do different things. CppUTest does not, so it always ignores the first



element in the array. That's what the first "" is in the args array. The second parameter, -v, tells CppUTest to output test names and test execution time rather than "." for each test. It means "verbose."

### Run Your Tests

Rerun your program (ctrl-F11/Comman-F11) and notice the additional output:

```
TEST(SmokeShould, NeverBeLost) - 0 ms
```

```
OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 0 ms)
```

## 2.2.6 Experiments in Failure

It is important to see things both work and fail. Getting something working will hopefully give you a sense of success. Seeing something fail in a controlled fashion can better prepare you to fix problems on your own when they happen.

### Failing Check

Update the check in your SmokeShould:NeverBeLost test:

```
LONGS_EQUAL(9999,1);
```

### Run Your Failing Test

Rerun your program and notice the failed test output (snipped):

```
..\SmokeTest.cpp:7: error: Failure in TEST(SmokeShould, NeverBeLost)
    expected <9999 0x270f>
    but was < 1 0x0001>
```

So here are some questions you might ask (and their answers):

- **Which test failed?** The one on line 7 of the SmokeTest.cpp file. It's called SmokeShould, NeverBeLost.
- **How did it fail?** A check on line 7 was expecting the value 9999 but it was given 1. Since 1 does not equal 9999 (even for very large values of 1), the LONGS\_EQUAL macro caused this test to stop at the failure.

Fix the test back to passing before moving on. (Do you make both values 9999, 1, or some other value?)

## 2.3 Recap

### 2.3.1 Terminology

Term	Description
Auto Test Discovery	In CppUTest, tests are automatically discovered. Simply creating a new TEST_GROUP and adding a TEST to it will cause the test to be found assuming the object module containing the TEST_GROUP and TEST are linked in to the final application.

<b>Term</b>	<b>Description</b>
CommandLineTestRunner	A class from CppUTest used to execute tests created with TEST_GROUP and TEST. It can handle a few command line arguments. In the second form of your main( ) function, you passed in -v, for verbose output.
CommandLineTestRunner.h	This is the header file you include to get the CommandLineTestRunner. However, the recommended include path for CppUTest makes you qualify the name with CppUTest/ in front of it.
Compilation Unit	When the C++ compiler compiles a single source file, it first processes all of the include files and macros. The result of that phase is called a compilation unit. It becomes the source material for the production of an object module, which is what gets linked into a final executable.
Ctrl-B (OS X: Command - B)	Build the application and produce an executable.
Ctrl-F11 (OS X: Command-Shift F11)	Run the last thing you ran. This is over simplified and depends on the Eclipse configuration. In this case, you've changed the Eclipse configuration to always run the last thing run, so it will not be relative to the current project. Also, you only have one project in your workspace, so, again, it will only run the one executable.
Include Directory	The include directory path for CppUTest in our examples is C:\learncpp\CppUTest\include. This directory contains a directory under it called CppUTest. This is by design to make sure header file names in CppUTest do not conflict with other header file names.
libCppUTest.a	This was the library file produced when building CppUTest. We need to include it to get our project to link. When we added it to our project, we followed the UNIX convention of dropping "lib" at the beginning and ".a" at the end.
Library Directory	The library directory in our case is C:\learncpp\CppUTest\lib. We added that so Eclipse could link our project.
LONGS_EQUAL	A macro in the CppUTest library. It calls an underlying method that compares the first (expected) value to the second (actual) value. If the longs are equal, then the method returns. If not, then it stops this unit test from continued execution and marks the test as failing.
Object Module	A file created by the C++ compiler as part of building a system. It contains all of the code from, typically, a single source file, along with anything introduced by header files.
RunAllTests	This is a method on the class CommandLineTestRunner. It happens to be a static (or class) method. This means it can be

Term	Description
	called without having any objects. The most important thing about this method is that it causes all of the unit tests to run.
standard library	The standard library ships with all C++ compilers. However, just what ships is somewhat variable. We've seen <code>std::cout</code> , and object that comes as part of the standard library. The <code>std::vector</code> class is also part of the standard library. We'll be seeing this class quite a bit more as we move into the <code>DiceGame</code> in earnest.
std	Classes can be part of a namespace. A namespace is simply a way to group related classes. The standard library classes are in a namespace called <code>std</code> (or <code>std::tr1</code> ).
struct	In C++ a struct and a class are the same thing. The only difference is that the default access level in a struct is <i>public</i> , whereas in classes it is <i>private</i> .
template class	The <code>std::vector</code> class is a template class. Template classes are parameterized classes with their parameters being provided in <code>&lt;&gt;</code> . In our particular example, the parameter is <code>&lt;int&gt;</code> . An <code>std::vector&lt;int&gt;</code> stores a variable number of <code>int</code> values.
TEST	A macro from <code>CppUTest</code> that introduces a new test into the system. The <code>TEST</code> macro takes two parameters; the first is the name of the test fixture introduced by the <code>TEST_GROUP</code> macro. The second is the name of the test. Both names must be valid C++ names. <code>TEST</code> names must be unique within their test fixture.  The <code>TEST</code> macro actually creates a class, which is a child class of the structure created by the <code>TEST_GROUP</code> macro. The name of the class embeds the test name parameter.
<i>TEST_GROUP</i>	A macro from <code>CppUTest</code> that introduces a new test fixture into the system. The <code>TEST_GROUP</code> macro takes one parameter, a name, which must be unique across all object modules linked into an application. The name must be a valid C++ name.  THE <code>TEST_GROUP</code> macro creates a struct. The name of the struct embeds the test fixture name.
TestHarness.h	This is a header file from <code>CppUTest</code> . It's the main header file used in test files. As with all <code>CppUTest</code> header files, it resides in a subdirectory called <code>CppUTest</code> , so qualify its name with <code>CppUTest/</code> when including it.  Remember that order of includes is important. Include this file last in any source file to avoid problematic interactions with the standard library classes.
vector	A class from the C++ standard library. We used it to

Term	Description
	demonstrate a potential issue you might come across with order of includes. A vector is variable-sized array. The class is in the namespace std, so it's full name is <code>std::vector</code> . You'll be using this class more.

## 2.3.2 Highlights

### **Tool Installation**

In this section you installed a number of tools and libraries:

- Java Developer Toolkit
- Eclipse CDT
- Wascana Eclipse Plug-in
- CppUTest

These are the basic tools of your environment and with them you will be able to work with C++ in a relatively controlled manner.

Luckily, this is a one-time process, so now that you've done this, you have your development environment for the remainder of the book.

### **Project Configuration**

Each time you start a project, you may need to configure it. In our case, we:

- Added a path to the include directories to find CppUTest header files
- Added a path to the library directories to find libCppUTest.a
- Added a library, CppUTest, to be linked into our executable.

### **Eclipse Configuration**

You also configured Eclipse to automatically save files before building and to find files created during the build process automatically.

In both the project and eclipse configuration steps, if you can remember a key word or two, you might be able to use the filter box at the top of the list of configuration options to shorten the list.

### **Complete Build Execute Cycle**

You created some source files, a `main()` and two test source files, all of which got linked into an executable. Initially no tests ran. Then you added a test, which ran. You then made that test fail and added another test that caused a compilation failure.

Did you notice how quick it was to simply perform little experiments to see "what-if"? This is a key to learning. Rather than speculate about what might be, try it out and see what happens. The one caveat is that it may not be clear if the results are speaking towards your environment or something about the language standard. This comes with a combination of research and continuous questioning. The biggest recommendation I can make here is to continuously question what you've learned to make sure your understanding of the results is not too far reaching (this result is about my current environment, and not a general principle), or it goes far enough (this is part of the language specification and if another environment behaves differently, it is incorrect).

Now it's on to the seeming trivial Dice Game.

### 3 The Dice Game

This is a simple project to get your feet wet, or so it seems. By the end of this project, you'll have used much of what you'll need to know about C++ to effectively use it.

#### 3.1 What's Ahead?

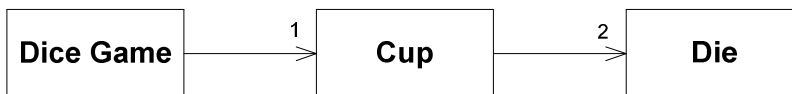
Here's a quick idea of what's coming up during this project:

- Creating Classes
- Subclassing
- Interfaces
- Guts of C++
- Unit testing
- Test driven development

The work that we'll be doing going forward will be driven by writing automated tests. Initially I'll provide the design up front; at some times we'll be "discovering" the design as we go along. Writing tests where there's already a design in place is often called Test-First Development, whereas using tests to incrementally discover the design is typically called Test-Driven Development.

#### 3.2 The Design

Here is a quick model for our top-level Dice Game:



##### 3.2.1 Notes on UML<sup>4</sup>

This book is not about UML. Even so, I need some basic way to express ideas at a level higher than code, so that's the language I'll be using. Here are the parts of this diagram so you can effectively read it.

	<p>These boxes represent classes. This particular example is called Dice Game, meaning there will be a class called <code>DiceGame</code> in our solution. In our use of C++, this means we will have a header file called <code>DiceGame.h</code> and a source file called <code>DiceGame.cpp</code>. The diagram shows <code>Cup</code> and <code>Die</code>, so there are two other classes and therefore four more files (two header files, two source files).</p> <p>Since we'll be writing tests as well, there will be test source files for each of these design-level concepts.</p>
	<p>A navigable connection. The line itself, ignoring the arrow head, suggests a connection between two classes. So there's a connection between <code>DiceGame</code> and <code>Cup</code>. The diagram also shows a connection between <code>Cup</code> and <code>Die</code>. Without the arrow head, this just means there's an association. The arrow head is called navigability, which means that</p>

<sup>4</sup> Unified Modeling Language.

	<p>the class on the side without the arrow head can navigate to the side with the arrow head. Since this is a solid line, it strongly suggests that the <code>DiceGame</code> class will have some kind of member data (attribute/field) that holds onto to a <code>Cup</code> object.</p> <p>You'll also notice numbers on those lines. That is the multiplicity of the relationship. In this particular example it shows "1", which is the default value. I won't be using default values because I think a default value of 1 was a poor choice in the design of UML and I don't want you to have to be a "language lawyer" when reading these diagrams.</p> <p>In the case of the <code>Cup</code> to <code>Die</code> relationship it shows a 2, meaning an object of the <code>Cup</code> class holds onto 2 objects of the <code>Die</code> class.</p> <p>Ultimately, when there's more than 1 or a variable number, you'll end up using some kind of collection class. We've already seen a collection class, <code>std::vector</code>, which is what we'll use to represent this relationship.</p>
--	---

### 3.2.2 The Rules of the Game

This is a simple game. A player plays the game by rolling the dice. Here are the possible results:

- The player wins 1 if the roll is  $> 7$
- The game is a push if the player rolls 7
- The player loses 1 if the roll is  $< 7$

The point of this project is not to create a `DiceGame`. The point of `DiceGame` is to serve as a vehicle to cover Object Oriented Design and C++, so the rules are mostly irrelevant. That there are some rules, which are different in some consistent way, is important. What those rules are, however, isn't really important.

### 3.2.3 One Development Strategy

There are several ways to go about writing the code for this project. Possibly the most familiar is to simply create all the classes and then see if things work. We will not be taking that approach, and instead we will be writing automated tests first. Those tests often won't compile initially, so we'll then create the production code to get the tests to compile but probably not pass. Then we'll get the tests to pass. When the tests are passing, we might clean up our work a bit in an activity called *refactoring*.

This may seem backwards to how you were taught, or what you are used to doing. Here's an observation: Nobody was born knowing how to program. So everything you do now is a learned activity. If this seems strange or backwards (and I expect it will), that's because it is different from what you've done in the past. This doesn't make it worse or better, it's just different. I hope by the time you've worked through this book you'll think it is better. However, if you don't that's fine. At the very least, I hope you'll find value in the idea of writing automated unit tests.

## 3.3 The First Test

We have a rough design so now it's on to writing a failing unit tests. Here is one such failing test:

```

01: #include "Die.h"
02:
03: #include <CppUTest/TestHarness.h>
04:
05: TEST_GROUP(Die) {
06: };
07:
08: TEST(Die, InitialValueInRange1to6) {
09:     Die d;
10:     CHECK(d.faceValue() >=1 && d.faceValue() <= 6);
11: }

```

Line	Description
01	You must <code>#include</code> the header file for any classes you'll be using. This lets the source file know about the class' definition: its member variables (attributes) and its member functions (methods).
01	Compare <code>#include</code> using <code>" "</code> versus <code>#include</code> using <code>&lt;&gt;</code> . When using <code>" "</code> , the current directory is checked as well as standard include locations and any include directories you've added (remember, you added the include directory for CppUTest). The <code>&lt;&gt;</code> does not check the current directory but otherwise it is the same list.  Traditionally, your files are <code>#included</code> using <code>" "</code> while system and library files are included using <code>&lt;&gt;</code> .
03	The order of includes, as mentioned above, is important. CppUTest overrides <code>new</code> and <code>delete</code> , which are used to manage memory allocated as the program runs (dynamically allocated memory). These two operators replace <code>malloc</code> and <code>free</code> , respectively, in C++. Note, they are operators in C++, or reserved words, where <code>malloc</code> and <code>free</code> are functions in a library. <code>new</code> and <code>delete</code> are different because they do more, which we'll get into later. Don't mix and match <code>new</code> and <code>delete</code> with <code>malloc</code> and <code>free</code> ; only use <code>new</code> and <code>delete</code> with classes (or structs that have been written as classes).
05 – 06	Define a struct for a test fixture. This creates a base struct (class) with <code>Die</code> in its name. The actual full name of the class depends on the version of CppUTest you are using. This variability suggests a particular style of writing tests, which we'll see later on.
06	The <code>TEST_GROUP</code> creates a struct. It is important to end a class definition with a <code>;"</code> . <code>TEST</code> creates a class, but unlike <code>TEST_GROUP</code> , you are adding a method to a class via the macro rather than fully defining a class. So it does not require a <code>;"</code> at the end because the macro does it for you. It's still there.  It is safe to simply always use a <code>;"</code> at the end of both of these macros.
08	Create a test. This macro creates a class which includes <code>InitialValueInRange1to6</code> in its name. The class inherits from the struct created by the <code>TEST_GROUP</code> macro. In reality, the class created has a single



Line	Description
	method added to it that holds the code between { and }, and it is that method that gets executed by the unit test framework.
10	Test assertion. This is a macro that calls a method, like LONGS_EQUAL. If the expression evaluates to true (a non-0 result) then the method returns and the test continues execution. If the expression is false (a 0 result), then the method terminates test execution (throwing an exception under the covers).

### 3.3.1 Create the test file

Add a new source file to your project. Call it `DieTest.cpp`. Enter the code above into that file. Eclipse will warn you that line 1, the `#include` of `Die.h`, names a file that does not exist. That's OK. You've written a test and the test is currently failing, but not for long.

Build your application and review the compiler errors.

### 3.3.2 Die Header

```
01: #pragma once
02: #ifndef DIE_H_
03: #define DIE_H_
04:
05: class Die {
06: public:
07:     int faceValue() const;
08: };
09:
10: #endif
```

Line	Description
01 – 03, 21	Guard against this header file getting <code>#included</code> multiple times. A header file may contain <i>declarations</i> or <i>definitions</i> . A <i>declaration</i> can be repeated, while a <i>definition</i> cannot. This header file <i>defines</i> the class <code>Die</code> . You can tell it is a <i>definition</i> because <code>class Die</code> is followed by <code>{ ... }</code> ; If the code instead was: <code>class Die;</code> then it would only be a <i>declaration</i> , which can safely be repeated.  A <i>declaration</i> states that something exists but does not provide details about size, structure or functionality. A <i>definition</i> does what a <i>declaration</i> does and additionally describes size, structure and/or functionality. This <i>class definition</i> states that there is a <code>class</code> called <code>Die</code> with a single member function (method), <code>int faceValue() const;</code>  Any <i>compilation unit</i> can include multiple <i>declarations</i> of a given thing (class, function, variable) but only one <i>definition</i> . A <i>declaration</i> after a <i>definition</i> is also OK.
01	The <code>#pragma</code> macro is a standard way to introduce compiler-specific code. By definition, if a compiler does not understand the text after <code>#pragma</code> it ignores it silently. This particular <code>#pragma</code> says to actually only include this physical file

Line	Description
	once. Most modern compilers support it and it often is enough to just include this expression. If you know your compiler and you know your compiler supports this and you only use one compiler, you don't need lines 2, 3 or 10.
02 – 03, 21	This is a traditional way to guard against multiple inclusion of a header file. Unlike <code>#pragma once</code> , multiple inclusions still happen, but only the first one is processed, the others are skipped. The compiler reads up to the matching <code>#endif</code> and simply ignores those lines. It takes time to read those lines but it does not cause the compilation to fail.
05	Begin defining a class called <code>Die</code> (the singular of dice). You can tell it is a definition in this case because it is followed by <code>{</code> versus <code>;</code> .
06	Everything after this line is accessible to any code. The default access level for classes is <code>private:</code> , which means only code in the class (and friends, which we will not cover in this book) can access that element.  Note, the checking of <code>public:</code> , <code>private:</code> (and <code>protected:</code> ) is done at compilation time.
07	<b>Declare a member function</b> (method). You can tell this is a <b>declaration</b> for a method because it is followed by <code>;</code> instead of <code>{</code> . This simple rule applies to <code>class</code> , <code>struct</code> and functions.  The return type of this method is <code>int</code> , meaning a caller can use that value for calculations.  The method takes no arguments as its formal argument list is <code>()</code> . This can also be written as <code>(void)</code> , but this distinction is only relevant for older C code.  Finally, the method is declared to be <code>const</code> . A <code>const</code> member function states an intention or a contract. It says that calling this method does not change the state of the receiving object. This makes sense at a non-quantum level, observing what was rolled does not change the value of a die.
08	Finish the <code>class definition</code> .  Don't forget <code>;</code> at the end – we'll have a failure exercise coming up to see what happens. Modern compilers can suggest that you've made such an error some times. More typically, you'll get 25 or 100 errors, enough for the compiler to just give up.  The reason you must finish a class with <code>;</code> is as follows: <ul style="list-style-type: none"> <li>▪ A class is the same as a struct, with a different default access level.</li> <li>▪ C++ is heavily backwards compatible with C.</li> <li>▪ In C, it was common practice to define a structure and immediately define a single variable of that structure.</li> <li>▪ To end a structure definition and not add an immediate variable definition, a struct is followed by a <code>;</code>.</li> <li>▪ To end a structure definition and define an immediate variable, a struct is followed by some variable name and then a <code>;</code>.</li> <li>▪ So a struct is always followed by <code>;</code></li> <li>▪ Since <code>struct == class</code> (with different default access level), therefore you must</li> </ul>

Line	Description
	end a class with ; to indicate that you've completed the class definition Method definitions do not need a trailing ;.
10	Balance the #ifndef on line 02 with a matching #endif.

### 3.3.3 Die Source

```
01: #include "Die.h"
02:
02: int Die::faceValue() const {
03:     return 1;
04: }
```

Line	Description
01	Include the header file for this class as the first thing in the source file. It is not necessary to include the header file as the first thing in the source file. However, the header file must be #included before <i>defining</i> any of the class' methods.
03	<i>Define</i> the method Die::faceValue. The method signature must match that in the header file. However, since a source file can have either methods or functions (not associated with a class), use Die:: to indicate that this is meant to <i>define a member function</i> as opposed to a non-member (or global) function. The :: operator is the scope operator. It says that the thing on its right is a member of the thing on its left. So faceValue is a member of Die. The header file included the class <i>definition</i> . You cannot <i>define</i> a member function without a class <i>definition</i> . You can tell this is a <i>definition</i> versus a <i>declaration</i> because it is followed by { instead of ;.
03	The test only requires a value between 1 and 6. So return 1. It's good enough for now. We'll use tests for force a better implementation.

## 3.4 Exercise

It's time to get your failing test compiling and passing. Create two files, Die.h and Die.cpp. If you use Eclipse to create a new class rather than creating a header file and source file, you will have a few additional methods. That's OK, we'll be getting to those.

### 3.4.1 Results

Once you've created the Die class, build and run your tests. You should see something similar to:

```
TEST(vectorShould, HaveInitialSizeOf0) - 0 ms
TEST(SmokeShould, NeverBeLost) - 0 ms
TEST(Die, InitialValueInRange1to6) - 0 ms
```

**OK (3 tests, 3 ran, 3 checks, 0 ignored, 0 filtered out, 0 ms)**

Before moving on, delete the files `SmokeTest.cpp` and `VectorTest.cpp`. They were examples to demonstrate a working system and a compilation failure.

If you simply remove the source files, Eclipse will leave the object modules around and those tests will still be there. To fix it:

- Delete the two files (if you have not already done so)
- Perform a clean build. Right-click on the project and select Clean Project.
- Re-run your tests.

### 3.4.2 Exercises in Failure

Now it is time for some controlled experiments.

#### ***Forgetting ; at the end of TEST\_GROUP***

Edit your file `DieTest.cpp`. Remove the `;` at the end of `TEST_GROUP` then build your system.

- What errors do you see?
- Do those errors seem to be related to what you just did?

After reading the error messages, replace the `;` at the end of `TEST_GROUP` and verify that your system still builds.

#### ***Forgetting ; at the end of class Die***

Try the same experiment but instead remove it from the `Die` class in `Die.hpp`.

- As before, what errors do you see?
- Are they even the same?
- Is there any clue as to the real problem?

After making those observations, fix your system by adding the `;` back on to the end of the `Die` class definition.

#### ***Getting the signature incorrect***

Try removing the word `const` from the end of the `faceValue` method declaration in `Die.h`, but not `Die.cpp`, and build.

- Do these errors seem to make sense?
- What can you do to make finding these kinds of errors easier?
- What can you do when writing your own code to make committing these kinds of errors less likely?

#### ***Why Experiment?***

One thing that most C++ compilers will do is give you errors that only make sense, occasionally, to an expert. When I type some syntax error, most of the time I look at the offending line and try to figure out it without reading the error message. If that does not work, then I try skimming the message to see if I recognize it as an error I've made in the past or if it has certain keywords that resonate with me. This is the nature of dealing with compilation errors. The only way you'll get skilled at this is practice. Each compiler is

different but learning one compiler will help. Learning the syntax will also help. There are other techniques that can help as well, for example commenting out sections of a file to see if you can guess what's going on. In all cases, these are just approaches to figuring out what's going wrong. Experience and practice with the language is the cost of entry for solving these kinds of problems.

### 3.4.3 Concept Review

Term	Description
<code>#define</code>	Defines a macro. We used it as a technique to make sure that if a header file is included more than once, it is only processed the first time. This is necessary because <i>definitions</i> cannot be repeated, while <i>declarations</i> can. A class header file typically contains a class <i>definition</i> , so it cannot be processed once in any single compilation unit.
<code>#endif</code>	This balances the <code>#ifndef</code> macro that starts the beginning of the old-style conditional guard started with <code>#ifndef</code> .
<code>#ifndef</code>	This means "if not defined". It's used to see if a macro has or has not yet been defined. If it has not been defined, continue reading and compiling up to the matching <code>#endif</code> . If the macro has been defined, read, but don't otherwise process up to the matching <code>#endif</code> . Note that for any single compilation unit, you need to make sure that the identifier (macro name) after <code>#define</code> is unique. We do this by using the name of the file, which happens to match the name of the class. This is a good basic approach.
<code>#include with ""</code>	The list of directories searched when using "" includes the current directory as well as standard directories and ones you specify. We used Eclipse to extend the list of directories searched to include the CppUTest base include directory. Eclipse simply passes <code>-I</code> (capital letter i) flag to the command-line compiler.
<code>#pragma once</code>	This is a modern way to avoid multiple inclusions of the same header file. If a compiler supports this, then really only include the file once. If the compiler does not understand this, then by definition it silently ignores it.
CHECK	This is a macro that is part of CppUTest and it is made available by including TestHarness.h. It verifies that the resulting expression evaluates to true (non-0). If it does, then nothing happens and the test continues. If it does not, then the test terminates and is recorded as a failed test.
class	A combination of member data (fields or attributes) and member functions (methods). A class is equivalent to a struct other than its default access level, which is private.
const member function	A member function that does not logically change the state of the object. You should be able to call a <code>const</code> member function over and over and get the same result.

<b>Term</b>	<b>Description</b>
CppUTest	CppUTest is an open-source automated testing library. If you use it to create focused, small tests that test only one thing, then you may be creating unit tests. If you are testing larger chunks of code, then you are creating automated tests that are not actually unit tests. This is neither a good thing nor a bad thing.
Declaration	Expressing that something exists. You can declare that a class exists, a function exists or some variable exists. A declaration only provides basic type information, not storage or implementation details. It is lighter weight than a definition.
Definition	Describes something in enough detail to know its size, its methods (if it is a class) or its code if is a method or function. Typically, classes are defined in a header file but their methods are only declared in the header file. The method bodies reside in a source file, thus they are defined in a source file.  For classes and methods, you can tell a definition from a declaration depending on what comes after the name. If there is a ; before a {, then it is a declaration. If there is a { before a ;, then it's a definition.
Member function	A function that is declared within a class definition. The only explicit example so far is <code>faceValue</code> on the <code>Die</code> class.
Member function definition	The implementation of a member function. The one example we have so far is <code>Die::faceValue</code> .
public:	Access level of member data or member functions. Public members are accessible to any code.
Scope ::	The name on the right is a part of the name on the left. Examples include <code>std::vector</code> (the <code>vector</code> class is a member of the <code>std</code> namespace) and <code>Die::faceValue</code> ( <code>faceValue</code> is a method in the <code>Die</code> class).
struct versus class	Classes and structs are equivalent in C++. The only difference is the default access; <code>private</code> for classes, <code>public</code> for structs.
TEST	A macro that defines an automated test, which is represented as a class. The class is a sub-class of the struct created by the related <code>TEST_GROUP</code> .
TEST_GROUP	A macro that defines a test fixture, which is implemented as a C++ struct.  Traditionally, you'll only have one of these per source file. You can have more, but in general one source file for one purpose produces cleaner code that compiles and ages better.
TestHarness.h	The header file that gives the macros you'll use for most of your test writing.

### 3.4.4 Final Observation

Notice that our test does not directly produce any output. This is by design, not accidental. For now consider this comment:

- Automated tests should produce no output.

We will come back to this but for now let your brain chew on that thought.

## 3.5 Making Improvements

The first test got us a class with a single, hard-coded method. That's a good start. Also, considering how much C++ you've already covered, hopefully you've taken a break before moving into the second test for this class.

In any case, you'll be adding a new test, which will force an update to the `Die` class. Though, it may not require as much as you'd like to write yourself. Here's something to begin observing:

- As you increase the number of tests, your production code should become more general.

This can be used as either a metric or a guideline. As a metric you might see if the code is in fact becoming more general as the number of tests increase. You might notice:

- Hard-coded values becoming conditions then moving into loops
- Handling more cases overall
- Gracefully dealing with edge and exceptional conditions

### 3.5.1 Updated Test

As with all the work in this book, we'll be changing code either by writing a test or with tests already in place. With that in mind, here is a second test that you can add after the last test:

```
01: #include "Die.h"
02:
03: #include <CppUTest/TestHarness.h>
04:
05: TEST_GROUP(Die) {
06: };
07:
08: TEST(Die, InitialValueInRange1to6) {
09:     Die d;
10:     CHECK(d.faceValue() >=1 && d.faceValue() <= 6);
11: }
12:
13: TEST(Die, RollesInRange1To6) {
14:     Die d;
15:     for(int i = 0; i < 10000; ++i) {
16:         d.roll();
17:         CHECK(d.faceValue() >= 1);
18:         CHECK(d.faceValue() <= 6);
19:     }
20: }
```

Line	Description
13	Introduce a new test called <code>RollsInRange1To6</code> .
14	Create an instance of a <code>Die</code> . Notice the duplication from line 09?
16	Call a new method, <code>roll</code> . If line 14 were in the body of the for loop, a new die object would be created for each time through the loop.
17 – 18	Verify that the die's <code>faceValue</code> is within the range 1 to 6. Do this in 2 checks rather than one like before. Notice the duplicate with line 10? It's even worse in a sense because it does the same thing in a slightly different way, so it could be harder to notice.

### 3.5.2 When to Clean Up

Notice the commentary about duplication? There's an underlying design principle known as Don't Repeat Yourself (DRY<sup>5</sup>). Duplication can cause several problems:

- Increased overall amount of code someone has to maintain
- When something changes, there's more to change and more opportunities to miss some of the needed places
- Ultimately, it serves as inertia resisting change.

So we will strive to remove duplication. However, to remove the duplication we will have to change another test. Rather than deal with the duplication now, we will finish what we have started and then remove the duplication, staying focused on the current task.

### 3.5.3 Updated Die Header File

```
01: #pragma once
02: #ifndef DIE_H_
03: #define DIE_H_
04:
05: class Die {
06: public:
07:     int roll();
08:     int faceValue() const;
09: };
10:
11: #endif
```

The only change is the declaration of a new method, `roll`, on line 7.

### 3.5.4 Updated Die Source File

```
01: #include "Die.h"
02: int Die::roll() {
03:     return 9999;
04: }
05:
06: int Die::faceValue() const {
```

---

<sup>5</sup> Site Pragmatic Programmer, Dave Thomas.



```
07:     return 1;
08: }
```

The only change is the definition of the `Die::roll` member function on lines 2 – 4. Notice the arbitrary return value? What is your reaction to that?

### 3.5.5 Exercise: Update Die

Make the following changes:

- Add the new test to `DieTest.cpp`
- Update `Die.h` with the roll method declaration
- Update `Die.cpp` with the roll method definition

Get back to green, meaning your code compiles and your tests all pass.

## 3.6 DRY Violation

Now that you are at a stable point, you have some options:

- Clean up tests to remove duplication
- Add new tests to force the extension of functionality
- Clean up the production code

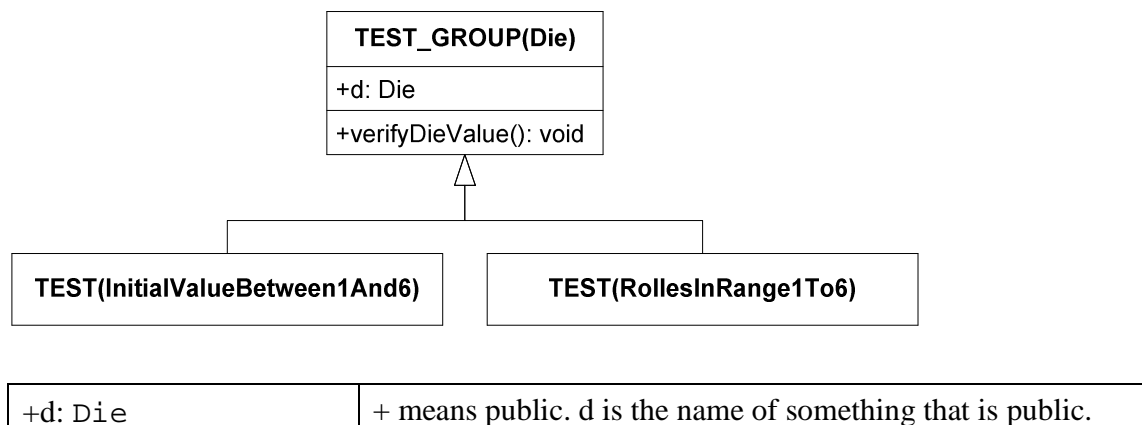
Since we observed duplication, now is a good time to remove it. We are going to change the structure of the code without changing its behavior. In this case we’re removing duplication from test code, but we could do something similar with production code.


Changing the structure of the code without changing its behavior is known as *refactoring*. Our working definition of behavior will be “passing tests.” So long as we keep the tests passing, we are refactoring.

### 3.6.1 Removing Duplication

The `TEST_GROUP` macro introduces a test fixture. It is time to take advantage of that. Any fields or methods added to the `TEST_GROUP` are available to all `TESTS` that use that `TEST_GROUP`. So simply adding a `Die` object and a supporting method will help remove the duplication.

In fact, this is an example of inheritance. Here’s a diagram to represent what we are about to do:



	:Die means it is an instance of a Die.
+verifyDieValue(): void	+ means public. verifyDieValue() is the name of a public method. :void means the method returns nothing.
	Inheritance. Each box represents a class. In this case the base type is a test fixture. We don't know the exact name, but the name comes from the TSET_GROUP macro, so I'm taking a little liberty with UML. The derived class' names come from the TEST macro. Everything in the base class is public so it will be available to the derived classes

**Step 1: Add**

Start by adding new functionality rather than changing anything in place. The diagram shows a public attribute (field/member data) and a public method (member function). Here's one way to accomplish both of those:

```

01: #include "Die.h"
02:
03: #include <CppUTest/CommandLineTestRunner.h>
04:
05: TEST_GROUP(Die) {
06:     Die d;
07:     void verifyDieValue() {
08:         CHECK(d.faceValue() >= 1);
09:         CHECK(d.faceValue() <= 6);
10:     }
11: };

```

Line	Description
06	Add a member field to the struct created using the TEST_GROUP macro. Its name is d, its type is Die. Since TEST_GROUP creates a struct, default access is public. The reason TEST_GROUP creates a struct versus a class is because it is meant to hold common data and methods to be used by tests, which inherit from the so-called test fixture.
07 – 10	Define a method called verifyDieValue(). It takes no parameters and returns no values. This is a method definition in a class definition. There's a term for that, it's called an <i>implicit inline function</i> . Inline functions are something you should eventually learn how to write, but they are not a subject explicitly covered in this book. I'm only mentioning it here so that: <ul style="list-style-type: none"> <li>▪ You will understand what this code actually does</li> <li>▪ You will know what this is called so you can look it up in a C++ Reference's index.</li> </ul>

**Run It**

Add these two changes and confirm that your code still compiles. You can also re-run your tests. When people discuss Test Driven Development and refactoring, they often discuss “getting to green fast and often”. This one trick to accomplish that goal:

- Initially add code, don't change it
- Bite off a very small bit and get it working
- Make sure it compiles and the tests still pass

#### **Update First Test**

Make sure these appear to work by only updating one test. Here's the first test updated:

```
13: TEST(Die, InitialValueInRange1To6) {  
14:     verifyDieValue();  
15: }
```

Note that the entire body is replaced by a call to the `verifyDieValue` method. The original test created a single die then performed a CHECK. This method does the same.

#### **Run It**

Update the test and confirm that you are still green. See? Keeping things clean and green isn't so bad when you understand how to break down a problem into smaller and smaller steps.

#### **Update Second Test**

Here's an updated version of the second test:

```
17: TEST(Die, RollesInRange1To6) {  
18:     for(int i = 0; i < 10000; ++i) {  
19:         d.roll();  
20:         verifyDieValue();  
21:     }  
22: }
```

This example removes the creation of a `Die` and it calls the public base method, thereby removing the duplication

#### **Run It**

Again, get to green. Change the test and verify that your code compiles and the tests still pass.

### **3.7 A Message on Test Granularity**

The first two tests are similar. They both create an object and they both check that the face value is within a particular range. However, they are testing very different things.

The first test confirms that after an instance of the `Die` class is created that it has a valid face value. That is, the object is created in a well-defined, usable state. The second test verifies that the die rolls a value in a valid range of 1 – 6. Notice that it does not confirm the distribution of the values, just that a value outside that range does not happen for a large number of uses.

What is the same is the mechanism of validation; confirming the die is valid by looking at its face value. This is an example of checking state. Looking at the value of a die and confirming that its value is in a range is verifying that its state, its member data (or attributes) are in a well-defined state. We happen to be checking all of the state in the die because there's only one value. This is incidental. It is reasonable to test only part of an

object's state in one test and another part in another test. For now, more, smaller tests are better than fewer, large tests.

For a quick answer as to why, consider how likely a test is to break relative to its size. The larger a test, typically, the more it covers. The more a test covers the more things that can change and break the test. Larger tests tend to have more moving parts, each of which might cause a test to break. As a corollary, if a test must be large, then the code it is testing is probably highly coupled and hard to understand.

Stated differently: Bad code is hard to test; if you can write clear tests, your code is probably well written.

### 3.7.1 Recap

Term	Description
As # tests increases, production code should	Become more general. More tests, testing different things, forces your production code to handle more situations. As it handles more situations, it will either become ugly or clean. The tests force change and they support experimentation to consider alternatives solutions to find one that is clean for the given set of tests.
DRY	Don't Repeat Yourself.
Keeping Tests Clean	We will strive to have clean production code and clean test code. Tests can be an asset, allowing you to make changes safely without breaking existing code. However, if you do not treat your tests with respect, or let them age badly, then they will eventually wither and die.  In fact, all code is either getting better or worse all the time. Even if you are not changing the code, your users are finding other ways to make it break, so unless you are actively caring for and feeding your code (test and production code), it's going to die.
Order of Tests	The order of test execution is, by design, unknown. While you can figure out the test order just looking at the output, do not take this information into consideration when writing tests. Tests should be written as mini-universes, with their own laws of physics (starting conditions) and their own, independent checks.
Test Fixture	The TEST_GROUP macro introduces a test fixture, implemented as a C++ struct. It holds common member data and member functions for tests. A good way to group tests is by common setup. A test fixture is a way to capture common setup.  I recommend one test fixture per source file.
Test Fixture Fields	Fields added to a TEST_GROUP are available to each individual test. This is a way to reduce duplication across tests. We used a common Die object in the TEST_GROUP, and each TESTs is able to access it. These fields are available because of inheritance.
Test Fixture Methods	Methods in a TEST_GROUP are accessible by TESTs. We added a method in the TEST_GROUP to validate that the die's roll was

Term	Description
	within the range 1 – 6 and then used that validation in each of the TESTs.

### 3.8 Fixing an anemic roll method

The roll method returns a hard-coded value. How can we, in general, improve the code<sup>6</sup>? As stated earlier, we'll be growing our code through the additional of additional tests.

#### 3.8.1 Validating Roll Distribution

Now it is time for a test that verifies all values are rolled. While we're at it, we'll also verify that the distribution is "reasonable." There were be several steps to the final form of the test, which uses the standard library and iterators.

##### *Writing it the hard way*

Here's a test that will get the job done:

```
01: TEST(Die, ShouldRollAllValuesEvenly_1) {
02:     int values[6] = { 0, 0, 0, 0, 0, 0 };
03:
04:     for(int i = 0; i < 600000; ++i) {
05:         d.roll();
06:         int faceValue = d.faceValue();
07:         int countIndex = faceValue - 1;
08:         ++values[countIndex];
09:     }
10:
11:     for(int i = 0; i < 6; ++i) {
12:         CHECK(values[i] > 95000);
13:         CHECK(values[i] < 105000);
14:     }
15: }
```

In a nutshell:

- Create an array of size 6 (6-sided dice), initialized to all 0's.
- Roll 600,000 times, incrementing the count by one for a given face value (1 is stored at index 0, 2 at 1, etc).
- Verify that all values rolled are in within 5% of balanced

This test is fine as is, but since part of the purpose of this book is to introduce classes in the standard library, read on.

##### *The std::array class*

There's a simple wrapper class for raw arrays. It gives a way to initialize values and iterate over a collection; it allows raw arrays to be used with other parts of the standard library in the same way that the collection classes work.

Here is the first update to the previous example (only the changes):

```
01: #include <array>
```

<sup>6</sup> The general answer to this question is: "Write another test." – Really.

```
02: TEST(Die, ShouldRollAllValuesEvenly_2) {
03:     std::array<int, 6> values;
04:     values.fill(0);
```

Line	Description
01	Include the std::array class. Note that in general the order of header files is important to CppUTest. The real issue is if the class uses dynamic memory allocation. This class does not, so it can be included before or after the #include of CppUTest/TestHarness.h.
03	Define an std::array that holds 6 ints. The array class wraps “raw” arrays. Its initial index is 0. Its last index is the size (6), minus 1.
04	Initialize all values in the array to 0. Note that the fill method uses the second template parameter (6) to know the range. It starts at 0 and goes to 5 (1 minus the second parameter).

If there were all the std::class did, then there’d be little reason to use it. However, before moving on, a few observations:

- The variable values is no longer a primitive as it was in the first version.
- Therefore, a constructor is called upon its definition.
- If you review the first version, specifically lines 8, 12, and 13, you’ll notice the array is accessed using []. This is an overloaded operator member function on the std::array class. The full method name is actually operator[] . You could write line 08 as:

```
++values.operator[](countIndex);
```

### Using a typedef

When I use collection classes, any many other classes in the standard library, I’ll use typedef statements to introduce synonyms for the sometimes long names. It just so happens, as you’ll see, it also removes a DRY violation.

Here’s one more minor change:

```
01: typedef std::array<int, 6> RollArray;
02: RollArray values;
03: values.fill(0);
```

Line 01 introduces a name, RollArray, which is a synonym for std::array<int , 6>. As with the previous example, not much change and it might not seem like much value. However, read on to see additional changes that will move this to an idiomatic use of classes in the standard library.

### Moving towards an iterator by using raw pointers

The standard collections and algorithms use a so-called pair of iterators to describe a range. An iterator is a logical way to process elements in a collection. Each collection declares two methods: begin() and end(). The first method, begin(), represents the logical beginning of a collection. The end() is a logical identifier representing “one past the end”.

The validation loop at the bottom of the test could be rewritten using this idea as follows:

```
01: for (int *i = &values[0]; i != &values[6]; ++i) {
```

```
02: CHECK(*i > 95000);
03: CHECK(*i < 105000);
04: }
```

A raw array stores values in contiguous bytes. The array class is a lightweight wrapper around a raw array, so its underlying storage is the same. The address of values[0] is the beginning of the array. The address of values[6] is just after the last element in the array. This loop starts at the first element and continues while i is not on the address just after the last element.

If this were a raw array, the for loop should have been:

```
for (int *i = values; i != values + 6; ++i) {
```

However, since values is an instance of a class, there's an even better way to write this:

```
for (int *i = values.begin(); i != values.end(); ++i) {
```

In all of these examples, i represents an iterator into the collection. It turns out that for classes like array, an iterator is just a raw pointer. For more complex classes, the iterator is also a more complex type. However, there's a convention used in the standard library. Each collection type has a nested typedef, iterator, which represents its iterator type.

The full name of that type in this case is:

```
std::array<int, 6>::iterator;
```

However, we have a typedef for the first part, so in our case, the name is:

```
RollArray::iterator;
```

This changes the for-loop one more time:

```
for (RollArray::iterator i = values.begin(); i != values.end(); ++i) {
```

*Introducing second typedef*

The almost-final version of the for loop is still a bit long winded. I'll introduce a second typedef to "promote" the nested typedef in the collection class up to the current scope:

```
TEST(Die, ShouldRollAllValuesEvenly_5) {
    typedef std::array<int, 6> RollArray;
    typedef RollArray::iterator iterator;
    RollArray values;
```

Now the final for-loop looks like this:

```
for (iterator i = values.begin(); i != values.end(); ++i) {
    CHECK(*i > 95000);
    CHECK(*i < 105000);
}
```

*Shortening the middle loop*

The middle loop uses several temporary variables to make its intent clearer. Here's a "tighter" version that is probably more typical:

```
for (int i = 0; i < 600000; ++i) {
    d.roll();
    ++values[d.faceValue() - 1];
```

}

This is a more c-ish way of doing things. Regardless of whether you prefer the longer version using temporary variables<sup>7</sup> or this shorter version, most compilers are going to generate the same code.

### **Putting it all together**

&lt;TO DO&gt;

The previous section and the next two code examples no longer flow... Need to integrate them. Also, the name in the previous examples is different.

&lt;/TO DO&gt;

The tests do not currently check to see if all values are rolled. This next test accomplishes that, making sure roll values are reasonably distributed:

```
01: TEST(Die, DistributionReasonable) {
02:     std::array<int, 6> values;
03:     values.fill(0);
04:     for(int i = 0; i < 600000; ++i) {
05:         d.roll();
06:         ++values[d.faceValue() - 1];
07:     }
08:
09:     for(std::array<int, 6>::iterator
10:         iter = values.begin();
11:         iter != values.end();
12:         ++iter) {
13:         CHECK(*iter > 95000);
14:         CHECK(*iter < 105000);
15:     }
16: }
```

Line	Description
01	Introduce a new test called DistributionReasonable.
02	Use the standard library array class. This class holds a raw array but gives a few convenience methods. This class looks like an <code>std::vector</code> , but it is a fixed size.
02	This class is available by <code>#including &lt;array&gt;</code> .
03	Initialize the contents of the array to 0. If you used a raw array, you'd either need to loop through the values, or initialize all of them with 0 when you created the array.
04	Loop 600,000 times.
05 – 06	Roll the die. Get the <code>faceValue</code> , subtract one. This gives a range of 0 – 5 ( <code>std::array</code> 's are 0-based, as are <code>std::vectors</code> and raw arrays). Increment the count at index <code>faceValue - 1</code> by one. So the count of the number of times 1

<sup>7</sup> These temporary variables are called “explaining variables” by Martin Fowler in Refactoring.



Line	Description
	was rolled will be in index 0.
09 – 12	Iterate over the entire array. This is a bit much to take in, however this is how modern C++ using the standard library should be written. I'll be giving a simpler version coming up using something called a <code>typedef</code> .
09	Use a nested <code>typedef</code> in the <code>std::array</code> class called <code>iterator</code> . The type <code>iterator</code> is actually just a pointer to what is in the array. Since the array contains <code>ints</code> , the type <code>std::array&lt;int,6&gt;::iterator</code> is actually <code>int*</code> .
02, 09	Notice the duplication? Both lines contain the fully-qualified name <code>std::array&lt;int, 6&gt;</code> . This violated DRY. We'll fix this before creating the test in our project.
09 – 10	Define an instance of a thing called an <code>iterator</code> . That instance is known as <code>iter</code> . It is initialized to the <code>values.begin()</code> , which is the address of the first element in the array. That is, it points to what is in <code>values[0]</code> ; it is equal to <code>&amp;values</code> .
11	So long as <code>iter</code> is not at the end of the array, continue going into the loop. Since <code>array</code> is a thin wrapper around a raw array, you can figure out what <code>values.end()</code> returns. It is the address just after the last element in the array. Since this is a <code>std::array&lt;int, 6&gt;</code> , the last element is at index 5. Therefore, <code>values.end()</code> returns <code>&amp;values + 6</code> .
12	Increment the counter. You are possibly more accustomed to seeing <code>iter++</code> (that is, post increment versus pre-increment). On a <code>for()</code> loop, they produce the same result. Modern compilers and processors can easily fix this at compile time. However, if the type of the iterator is actually a complex type as opposed to a raw type, then pre-increment is more efficient. I'll have more to say about this later. For now, you can safely make this substitution universally. Most of the time it will make no difference. Occasionally it will be more efficient because it will avoid the creation of an unnecessary object. Since it is a hard habit to change, better to start practicing it now.  Technically, post-increment returns the value before assignment, which requires the creation of a temporary value. This temporary value is called an r-value. It is called an r-value because it can appear on the right-side of assignment ( <code>a = b</code> ) but not on the left side of assignment.
13, 14	To get to the current value in the array through the <code>iterator</code> , dereference it. That's what <code>*iter</code> does, it turns the <i>pointer to an</i> <code>int</code> into an <code>int</code> .  Again, this is idiomatic use of the standard library. So this is something that you'll need to practice so that it becomes second nature.
13, 14	Make sure that the total number of rolls for a given value is approximately one sixth of the total number of rolls (with a 5% margin for error).

This test is a bit of a jump. It introduces several new things:

- The `std::array` class from the standard library.
- Iteration across all elements of a collection.

- The iterator nested type
- Dereferencing iterators
- The begin() and end() methods of collections

For now, I want you to replicate it. You'll be getting more practice with this in the future but to be able to type it, you'll need to build up the complex muscle memory required. Think of this as the natural language approach to learning C++.

### **Fixing DRY Violation with an Idiom**

Before you actually write this test, there's one more thing I want to introduce to make this code a touch easier: using `typedef`

Here's the same code using `typedefs` to remove the duplication, and I hope, make the code more readable:

```
01: #include <array>
02: typedef std::array<int, 6> RollArray;
03: typedef RollArray::iterator iterator;
04:
05: TEST(Die, DistributionReasonable) {
06:     RollArray values;
07:     values.fill(0);
08:     for (int i = 0; i < 600000; ++i) {
09:         d.roll();
10:         ++values[d.faceValue() - 1];
11:     }
12:
13:     for (iterator iter=values.begin(); iter!=values.end(); ++iter) {
14:         CHECK(*iter > 95000);
15:         CHECK(*iter < 105000);
16:     }
17: }
```

Line	Description
01	This is the header file for <code>std::array</code> . This header file can be included after <code>&lt;CppUTest/TestHarness.h&gt;</code> . It happens to work because the <code>std::array</code> class does do anything with dynamic memory allocation.
02	Use <code>typedef</code> to introduce a synonym for the full name of the array. The original version had duplication, this version does not. Now, anywhere below line 02 (in the compilation unit) using <code>RollArray</code> will actually be using <code>std::array&lt;int, 6&gt;</code> .
03	The type <code>iterator</code> is actually a nested <code>typedef</code> in the <code>std::array</code> class. This line introduces a synonym for <code>iterator</code> called the same thing. This is a way to “promote” a nested <code>typedef</code> up to the current scope.  Any line of code below line 03 (in the compilation unit) using <code>iterator</code> will actually be using <code>std::array&lt;int, 6&gt;::iterator</code> .  The name you use is arbitrary but it should make sense. Since the <code>iterator</code>

Line	Description
	meme is embedded deeply in the C++ standard library, I avoid originality and just use it.
06	Define a <code>std::array&lt;int, 6&gt;</code> by using the typedef.
13	Define a <code>std::array&lt;int, 6&gt;::iterator</code> by using the typedef.

### Create the Test

Create the second form of the test. Add it to the existing `DieTest.cpp` class. You should be able to get a compiling test that fails executed.

### 3.8.2 Updated Header File

Now that the `Die` needs to produce different values, it's time to give it a better implementation. The header file needs some updates:

```

01: #pragma once
02: #ifndef DIE_H_
03: #define DIE_H_
04:
05: class Die {
06: public:
07:     Die();
08:     int roll();
09:     int faceValue() const;
10:
11: private:
12:     int value;
13: };
14:
15: #endif

```

Line	Description
07	Declare a no-argument constructor (ctor). If you declare no constructors, C++ will declare a no-argument constructor for your class, which is typically called the default constructor. The compiler-provided no-argument constructor performs default initialization. Default initialization for primitives is to do nothing. Default initialization for non-primitives (instances of classes, or objects) is to call the object's no-argument constructor.  We've added member data on line 12 to this class, an <code>int</code> . Since <code>int</code> is a primitive type, its default initialization is to do nothing. We will use the constructor to set <code>value</code> to some reasonable initial value.
11	Everything from this line on is private: until the next <code>public:</code> , <code>private:</code> , or <code>protected:</code> .
12	Add member data (fields, instance data). Each instance of <code>die</code> will have its own value member data.

This header file declares a constructor, which is considered a special function. That it is special is not important, what is important, however, is that that constructor is implicitly used throughout your code base.

### **Experiment**

Make the change to the header file and attempt to build. All the code will compile, but it will not link. The linking error will be related to the fact that your class definition contains a declaration for a no-argument constructor, but you have not yet defined it (in the source file).

Often, you can add a member function to a class. So long as it is not used, adding just a declaration but no definition will cause no problems. Since calls to constructors are implicit (added by the compiler), they are used, even if it is not obvious. In fact, any time you define an instance, a constructor will be called. Not so for primitives (`int`, `int*`, `char`, `char*`, etc.).

### 3.8.3 Updated Source File

The header file contains a declaration of no-argument constructor, so it is time to add it. This section contains a few steps so you can observe things actually happening rather than just believe a book.

#### **First Pass**

```
01: #include "Die.h"
02:
05: Die::Die() {}
06:
07: int Die::roll() {
08:     return 9999;
13: }
14:
15: int Die::faceValue() const {
16:     return value;
17: }
```

There are missing line numbers to maintain the same line numbers from this version to the final version for this section.

Line	Description
05	Define the no-argument constructor. For now it does nothing. This code is what the compiler writes for you by default.
16	Change the definition of <code>faceValue</code> to return the new instance data. A question to start asking yourself is this: How does this method know one value from another? If there are 1,000 <code>Die</code> objects in the system, how does this single method distinguish one die from another?

#### **Experiments in Failure**

The code did not link. Now it will link if you define the constructor. Add just line 5, get to compiling. Now you should see the most recently added test failing. Why? Because the distribution of values is not reasonable; the die only rolls 1.

Now, update your `faceValue` method to return value instead of returning 1. Then run your tests. Now ask yourself, “what just happened?!”

### **Returning back to just one failure**

The problem is that the method `faceValue` returns member data. That member data is an `int`, which C++ does not initialize by default. Your particular compiler might initialize it to 0. If it does, the tests will still fail.

What’s really happening is that memory for the die object is allocated somewhere. You cannot tell for sure where without looking at `CppUTest`, but where it is placed is not important. It’s in memory somewhere.

That memory may have previously held a value or it might have been set to all 0’s. Whatever that memory had for a value is what the member data will contain. So you need to initialize data. That is the primary role of a constructor. So here are two equivalent ways to accomplish initialization:

```
05: Die::Die() { value = 1; }
```

And then this form I want you to learn:

```
05: Die::Die() : value(1) {}
```

The first form *assigns* the value 1 to the member data. The second version *initializes* the member data to 1. It may sound like the same thing, and for primitives it is the same thing. However, *prefer initialization over assignment*.

Of course, you’re going to have to learn one from the other. This will come up again, so for now:

- Use the second form
- Understand that C++ treats initialization different from assignment
- In terms of a constructor, the second form is how to use initialization versus assignment.
- Even though for primitives there is no difference, there is a difference for objects. It will typically be better performing, but more importantly, there are cases where it will be required.

Make the change to your constructor and notice that you are back to one failing test.

By the way, the name for the second form is: *Member-wise Initialization List*. It starts with a `:` after the close parenthesis of the formal argument list and ends with the opening curly bracket `{`.

### **Get back to green**

Now it is time to get back to green. We’ll do that with a quick and dirty implementation then a better one after that.

```
0a: int Die::roll() {  
0b:     static int lastValue = 0;  
0c:     lastValue = (lastValue + 1) % 6;  
0d:     return value = lastValue + 1;  
0e: }
```

I'm using letters here because these lines are not going to last very long.

Line	Description
0b	Define a variable called <code>lastValue</code> . This variable is <code>static</code> . This form of <code>static</code> makes <code>lastValue</code> exist for the entire program <sup>8</sup> . The initialization of <code>lastValue</code> to 0 is done once, the first time the method is executed, but never again.
0c	Add one to last value and use modulus with 6. This will limit <code>lastValue</code> to the range 0 – 5.
0d	Assign the member data <code>value</code> to <code>lastValue + 1</code> . Since <code>lastValue</code> is in the range 0 – 5, <code>value</code> will be in the range 1 – 6. Return the result of the assignment. Note that assignment returns the thing assigned to, which is <code>value</code> in this case. Technically, assignment returns an l-value. An l-value is something that can appear on the left-hand side of the assignment operator.

This first version should get your tests passing. Make the change and get to green. Once you are green, read on.

### Refactoring

Remember that refactoring means to change the structure of your code without changing its behavior. Behavior, in our work, is defined by the automated tests. If we plan to change the production code, so long as the tests pass, we are in great shape.

The second version of the C++ standard is called `tr1`, which stands for “Technical Release 1”. OK, it really is not a standard but a recommendation. It recommended adding several classes to the standard library. One group of additions to the library for that version was a number of new random number generators. I'm providing this next version for two reasons:

- So you will be aware of classes available to you in the library
- Students in the classes I've taught seem to like this

So here is a final version of the entire `Die.cpp` file for this section:

```
01: #include "Die.h"
02:
03: #include <tr1/random>
04:
05: Die::Die() : value(1) {}
06:
07: int Die::roll() {
08:     using namespace std::tr1;
09:     static mt19937 engine;
10:     static uniform_int<int> uniform(1, 6);
11:
12:     return value = uniform(engine);
13: }
```

<sup>8</sup> This is somewhat simplified, but it's a good enough model for the discussion.

```

14:
15: int Die::faceValue() const {
16:     return value;
17: }

```

Line	Description
03	Include the new random number generation features via this header file. Notice the use of <code>tr1</code> in the name of the header file. You can probably guess that something was part of <code>tr1</code> if its header file includes <code>tr1</code> .
05	The final version of the constructor. This version initializes the member data value to 1 by using the member-wise initialization list.
08	The classes about to be named are part of <code>tr1</code> . Classes of <code>tr1</code> are typically in the <code>tr1</code> namespace. The <code>tr1</code> namespace is a nested namespace in the <code>std</code> namespace. So, for example, <code>mt19937</code> is a class name. Its full name is <code>std::tr1::mt19937</code> . Rather than typing that, the <code>using</code> expression brings all of the names in the <code>std::tr1</code> namespace into the current scope (the <code>roll</code> method). This makes those names available without fully qualifying them.  Since this <code>using</code> expression is in the body of the definition of the <code>roll</code> method, it only impacts the <code>roll</code> method.
09	The class <code>mt19937</code> is a random number generator. This line defines a single instance of the class, called <code>engine</code> . It is static, meaning there will only be one, which will live between executions of the <code>roll</code> method. The first time the <code>roll</code> method is called, this object will be initialized. Since this is an object, and not a primitive, one of its constructors will be called. There are no parameters provided to the constructor so the no-argument constructor will be called.
10	The template class <code>uniform_int&lt;int&gt;</code> attempts to uniformly distribute the values it is provided across some range of values. This line defines an instance of the class called <code>uniform</code> . The first time the <code>roll</code> method is called, one of <code>uniform</code> 's constructors will be automatically called. The (1, 6) after <code>uniform</code> are parameters provided to the constructor. So a constructor taking two numbers will be called rather than the no-argument constructor.
12	First, generate a random value. Next, assign the random result to the value member data. Finally, return the l-value returned as a result of assignment, which is the member data value.  There is quite a bit going on here. <ul style="list-style-type: none"> <li>▪ The statement <code>uniform(engine)</code> calls a method on the <code>uniform</code> class taking an <code>engine</code> object. This looks like a function call, but it is in fact calling a method called <code>operator()</code>, literally. This expression could be written as <code>uniform.operator()(engine)</code>.</li> <li>▪ The <code>()</code> operator is called the function call operator. It allows an object to look like it is a function. Another name for this is “functor” or function-object.</li> <li>▪ The return from calling <code>operator()</code> is assigned to <code>value</code>.</li> <li>▪ The result of that assignment is <code>value</code> itself.</li> <li>▪ The member field <code>value</code> is returned.</li> </ul>

Line	Description
12	Notice the DRY violation? Two lines of code both returning the value member data. This doesn't seem like much, but as you will see, it causes some subtle problems.
12	<p>There is another design principle that this code violates: The <i>Command/Query Separation Principle</i>. This principle suggests that a function or method should either be a command or a query. A command changes state but does not return a value. A query returns a value but does not change state. Since roll() can change the value of the die, its returning a value violates the principle.</p> <p>There are times when following the command/query separation is problematic. One common example is with multi-threaded programming. Often you need to both perform a check and change a value if that check passes, however both of those things need to be done as a single unit of work.</p> <p>We are not writing a multi-threaded application, however, so violation this principle might cause problems.</p> <p>Now a secret. That violation is by design. It will come up shortly as you will see an actual problem that could have been averted had the code not violated the command/query separation.</p>

Now it is time to change your code. Change the implementation of roll to use the new random number generation classes. Now that you are aware of their existence, you're in a good position to look them up if they tickle your fancy<sup>9</sup>, this will be the last mention of the random number generator.

#### **A note on this C ugliness**

Using a static variable in a method (or function) is an old C hack. However, due to the design of the class I only wanted one instance of both the engine and the uniform object. There are several ways to do this:

1. Static variables in the class (so-called class data, or static member data)
2. Static variables in the source file, but outside of a method
3. Global variables
4. Each die object could have its own copy, each initialized with a different random seed
5. Each die object has its own pointer to an engine and an uniform\_int
6. Not do it and just skip those classes (or lightly mention them)
7. The static hack I used

Fundamentally I didn't really like any of these solutions. My personal definition of design is "Selecting the solution that sucks the least." Often there is not a single best answer, so you take the best answer you can get.

Here is why, case by case, I rejected the first six options:

Option	Description
--------	-------------

<sup>9</sup> That's one of the ways this book avoids being an encyclopedia. Another is simply not covering everything. So you will not win any trivial pursuit contests, but you will be able to write decent OO programs with what you will learn.



Option	Description
1	While I did want a single instance, adding these to the header file would then make all files that include <code>Die.h</code> know about the random number generation stuff. This is strictly implementation only so spreading this knowledge across the source base represents bad and fundamentally unnecessary coupling.
2	This isn't really any better than in the function, and it expands the visibility of the variables from just one method to every line in the compilation unit below where it is introduced. It's better than option 1, but if I'm already considering file-level scope, which is what static on a variable outside of a method represents, then static on a variable in a function (or method) really isn't any more complex. Same idea, more focused scope.
3	Global variables allow for wide-reaching coupling. I will use them in controlled fashions. But in reality, global variables are a great way to have guaranteed employment as you work fixing bugs forever.
4	First, there's the issue of making the random implementation visible to all files that include <code>Die.h</code> . That is unnecessary coupling. Unnecessary coupling causes slower compilation times, slower feedback, and increased time to read and understand what is going on.  Also, this would require the introduction of more C++ features than I wanted to do at this point. Important features, to be sure, but ones that will naturally occur.
5	This reduces the issue of unnecessary coupling because it is possible to use something called a forward-declaration to minimize the unnecessary coupling. However, this would introduce dynamic memory allocation as well.  Both of those subjects are coming up. But as with the previous option, more C++ than I wanted to cover at this point. Things that will come up anyway.
6	Of all the options that I did not pick, this is the best option. It might have been the best option overall, however I have witnessed enough people interested in this that it seems worthy of mention. I don't think it's worthy of a lot of detail because those details are available elsewhere, but to know to look up those details, you need to know it exists. So using it is a great way to raise awareness.
7	This option is somewhat complex, but it's something I will actually do in production code to keep implementation details deeply hidden.

### 3.8.4 Recap

Term	Description
<code>&lt;tr1/random&gt;</code>	The header file used to include support for new random number generation algorithms introduced into technical release 1 of C++.
array	A class in the standard library. It offers a thin wrapper around a raw C array. It provides utility methods and it also makes a raw array look like a standard collection.
assignment	<code>=</code> is the assignment operator. It takes the value on its right side and

<b>Term</b>	<b>Description</b>
	applies it to the thing on its left side. The thing on the left side must be an l-value, such as a variable. The thing on the right side can be an l-value but it can also be an r-value, e.g., 4.
begin	A method on the standard collections. It returns an iterator set to the beginning element of the collection.
command-query separation	A design principle that suggests methods and functions should either be a command or a query, but not both. A command changes state but returns no value; a query returns a value but does not return state.
Constructor(ctor)	A special method on a class used to initialize an object of that class. It is special in that you do not explicitly call it; the compiler inserts code to invoke it. In fact, when creating an object, a constructor will be called. Primitive types, such as pointers do not have constructors.
end	A method on the standard collections. It returns something that can be used for comparison. Iteration across all elements in a collection start at begin(), and continue while the iterator is not equal to end().
function-object	An instance of a class that can respond to the message (). This is an overloaded operator. The full method name is operator(). At a superficial level, it makes an object look like it behaves like a function. When used as a parameter to a template class, it is possible to write code that uses either a function or a function object without having to do anything special, so it is a convenient way to make generic code.
functor	Same as a function object, just an alternative name.
initialization	When an object or primitive is declared and set at the same time, it is being initialized. For non-primitives, initialization means the compiler inserts a call to one of its constructors.
iterator	This is both a design pattern, a name used by the standard library and a nested typedef. All of the collection types in the standard library contain a declaration of a type called iterator. It represents something that code uses to work through all elements in a collection. In the case of array and vector, the type of iterator is simply a pointer to what the array or vectors holds. For more complex collections, the iterator is more complex. However, its use is consistent across the collection classes in the standard library.
l-value	A fully-formed variable or object. It is something that can be on the left-side of an assignment operator. Temporary values or a constants such as 4 or "4" are not l-values, but instead r-values.
Member field	A non-static variable that is part of a class definition.
member-wise initialization list	On a constructor, it is the place where member data can be initialized using non-default initialization. It exists between the ) of the constructor argument list and the { marking the start of the

Term	Description
	method definition. The member-wise initialization list begins with a <code>::</code> .
modulus (%)	Given two integer values, return the remainder of division of the number on the left by the number on the right. <code>5 % 6</code> is 5, whereas <code>6 % 5</code> is 1.
mt19937	A class added to the standard library as part of technical release 1. It provides a sequence of random values using the Mersenne Twister algorithm
namespace	A way to group related classes and variables. The original standard library resides in a namespace called <code>std</code> . Examples of its members include <code>std::cout</code> and <code>std::vector</code> . When technical release 1 was introduced, it was created as a nested namespace under <code>std</code> , giving classes like <code>std::tr1::mt19937</code> .
nested type	Adding a type in a class creates a nested type. All of the standard collections have a nested type called <code>iterator</code> and another nested type called <code>const_iterator</code> for dealing with constant collections.
nested typedef	The <code>iterator</code> and <code>const_iterator</code> nested types are created using a C typedef. Since those typedefs are in a class, they are nested typedefs. This means their full name includes the name of the class in which they reside.
no-argument constructor	A constructor on a class that takes no arguments. When you create a C++ class with no constructor, C++ will attempt to add one taking no parameters. You can choose to write that yourself if the one provided by C++ does not do what you require. We came across this while changing the <code>Die</code> class. We added a primitive member data entry called <code>value</code> , which required initialization.
non-primitive	An object. An instance of a class or struct. Non-primitives have constructors, which are called automatically when the object is created.
object/instance	Defining a variable of a class or struct creates an object. Object and instance are synonyms.
operator()	The function call operator. We saw this on the <code>uniform_int</code> class.
pre-increment	<code>++</code> on the left side of a variable. Pre-increment returns the value of a variable <i>after</i> adding one. The significance of this is that the pre-increment operator returns an l-value rather than an r-value. An r-value requires the creation of a temporary object, whereas an l-value does not.
prefer initialization over assignment	All non-primitives are initialized. Assignment after initialization may cause unnecessary or duplicate work. There is little difference with primitive values, however, there is no harm using initialization over assignment, so applying this to all variable types reduces the amount you need to remember.

<b>Term</b>	<b>Description</b>
primitive	Types known by the compiler. Examples include <code>int</code> , <code>char</code> , <code>void*</code> , to name a few. All pointers are primitives. All references (we've not come across this yet) are also primitives.
primitive initialization	Primitives are not initialized by default. We experienced this when adding the value member data to the <code>Die</code> class. It was not initialized. We added a constructor to force its initialization. We then looked at assigning the value in the body of the constructor versus initializing it in the member-wise initialization list. For primitives, there's little difference. However, there are situations where the member-wise initialization list is mandatory.
private:	Access level. Things that are private cannot be accessed outside of the class unless using "friends".
r-value	A value that can only appear on the right side of an assignment operator. A copy or a temporary.
special member functions	Constructors and the destructor are examples of special member functions.
static variables	Static typically means "one" in some manner. <ul style="list-style-type: none"> <li>▪ Static on a variable in a function means there will only be one, it will be initialized the first time through the function and it will live for the life of the program.</li> <li>▪ Static on a variable in a source file but outside of a function means there is only one, it will be initialized before its first use (typically before <code>main()</code> is called in practice) and it is only accessible from the point in the file where it occurs to the end of the file.</li> <li>▪ Static on a function in a file means the function is only available to this one compilation unit. File-level scope.</li> <li>▪ Static on a variable in a class means there will only be one of them rather than one per object. This is typically called class data.</li> <li>▪ Static on a method in a class means that it is a class method. It does not operate on objects. It does not itself have access to member data unless it happens to have an instance of the class.</li> </ul>
<code>std::tr1</code>	The namespace of classes added as part of technical release 1 of the C++ language.
template class	A class that takes parameters in <code>&lt;&gt;</code> . We have seen two examples up to this point, <code>std::array</code> and <code>std::vector</code> . A template class is a partial or meta class. Providing template parameter makes a complete class. In fact, in C++ a <code>vector&lt;int&gt;</code> and <code>vector&lt;int*&gt;</code> are two different classes. Using a template causes the creation of a complete class.
<code>tr1</code>	Technical release 1, or ISO/IEC TR 19768. This is a set of recommendations, not actually a standard. Several of the classes were taken from the boost library.
<code>typedef</code>	A way in C (and C++) to introduce a synonym for another type. I'll

Term	Description
	be using <code>typedef</code> to remove duplication primarily when using template classes from the standard library.
<code>uniform_int&lt;int&gt;</code>	A class described in tr1 that attempts to provide an even distribution of values across a range.
<code>using</code>	A keyword to bring names from one namespace into another namespace.

### 3.9 C++ Idioms

There's one more group of changes based on things the compiler does by default. The compiler will add a no-argument constructor if you do not add any constructors to your class. The compiler will also add three more methods to your class if you do not do so yourself:

- A destructor
- A copy-constructor
- An assignment operator

This section is about doing this yourself. Ultimately, whether and how these methods are declared and or defined should be an explicit decision. The first step, however, is awareness that C++ is adding these methods and then how to override the default behavior.

#### 3.9.1 Updated Header File

Whether a class will take over the generation of those 4 methods is part of a class' definition, which is in a header file:

```
01: #pragma once
02: #ifndef DIE_H_
03: #define DIE_H_
04:
05: class Die {
06: public:
07:     Die();
08:     virtual ~Die();
09:
10:     int roll();
11:     int faceValue() const;
12:
13: private:
14:     int value;
15:
16: private:
17:     Die(const Die&);
18:     Die& operator=(const Die&);
19: };
20:
21: #endif
```

Line	Description
08	Declare a destructor. This is a special method called when an object is removed from the system. By convention, we'll be making all of our destructors virtual. More on this later. There are situations where this is an inappropriate thing to do, e.g., embedded systems with strong memory restrictions, or putting objects into shared memory. Neither of these is relevant for this problem.
17	Declare a copy constructor. It is private, so it is not available for use outside of the class. In fact, it will not be used inside the class, so there will be no definition for this method. This makes it impossible to accidentally copy an object of this class.
18	Ibid. but for the assignment operator. The assignment operator, by convention, returns a reference to the original receiver. By virtue of the &, the return type is an original, not a copy, so it is an l-value. Assignment operators are meant to return l-values, and this is how to do that.
17, 18	Both of these take in a <code>const Die &amp;</code> . The & means reference. The <code>const</code> suggests that the object passed in will remain unchanged.

### Experiment in Failure

Update the header file and attempt to compile. You'll notice a linking error because your class' definition states that the class is providing a destructor. Since the compiler automatically injects calls to a class' destructor when it leaves the system, you must provide a definition if you declare a destructor in the class' definition.

### 3.9.2 Updated Source

```

01: #include "Die.h"
02:
03: #include <tr1/random>
04:
05: Die::Die() : value(1) {}
06: Die::~~Die() {}
07:
08: int Die::roll() {
09:     using namespace std::tr1;
10:     static mt19937 engine;
11:     static uniform_int<int>
        uniform(1, 6);
12:
13:     return value = uniform(engine);
14: }
15:
16: int Die::faceValue() const {
17:     return value;
18: }

```

Line	Description
06	Define the destructor. There's nothing to do, so the compiler-provided destructor

Line	Description
	would have been adequate. However, there's still a reason to provide it. Doing so can reduce your object module size and your linking time. Not by much, but on a large system, you might notice a difference.  It is also not possible to make the destructor virtual if you do not declare it as such yourself. That is, a compiler-provided destructor will never be virtual.

**Back to Green**

Add the destructor definition and get back to green

## 3.9.3 Recap

Term	Definition
Assignment Operator	<p>A method with the following characteristics:</p> <ul style="list-style-type: none"> <li>▪ Its name is <code>operator=</code></li> <li>▪ It must take one parameter, an object of the class, the typical parameter type is <code>const Type&amp;</code>.</li> <li>▪ It should return a reference to an object of the class, so the typical return type is <code>Type&amp;</code>.</li> </ul> <p>By default, classes will be given an assignment operator. By declaring this method, you are stopping the compiler from adding this method itself. So long as it is not used, you do not need to define the method. By making it private, only your class can use it, which makes it unlikely that it will get used.</p>
Copy Constructor	<p>A constructor used when copying the object. Like the assignment operator, the compiler provides one of these. If you declare it in your class, the compiler will not provide it for you.</p> <p>A copy constructor has the following characteristics:</p> <ul style="list-style-type: none"> <li>▪ Its name is the same as your class name. This is a requirement of all constructors.</li> <li>▪ It must take one parameter, a reference to an object of your class. It should be a <code>const</code> reference, but in any case it must be a reference.</li> <li>▪ It has no return type because constructors do not have return types.</li> </ul>
Destructor (dtor)	<p>A destructor is named <code>~ClassName</code>. It is automatically added by the compiler if you do not write one. A call to the destructor is automatically added by the compiler upon an object being removed. We added a destructor and made it virtual. For now, this is meant to get you practicing a habit. We'll come across this again.</p>
Operator Overloading	<p>You can write operators for your class. There have been several examples of this so far:</p> <ul style="list-style-type: none"> <li>▪ The assignment operator declaration you just added to your class</li> <li>▪ The <code>uniform_int</code> class had an overloaded <code>operator()</code>.</li> <li>▪ The vector class and the array class both have an <code>operator[]</code>, which makes both classes look like raw arrays.</li> </ul>
reference	<p>A reference is a primitive type. It allows a variable to refer to another</p>

	<p>variable rather than be a copy. In C, which does not have references, you accomplish the same thing by using pointers. Here are a few things to keep in mind with references:</p> <ul style="list-style-type: none"> <li>▪ Their primary use is as parameters sent into methods. While it is possible to use one in the middle of a function, it is atypical.</li> <li>▪ They are primitive types. This means you can use one without including a header file; though you'll have to forward-declare non-primitive types (classes) for which you use a reference.</li> <li>▪ When they are created, they must be initialized, and once initialized always refer to the same thing.</li> <li>▪ Assignment to a reference changes the value of the thing referred to, not the reference itself.</li> <li>▪ As we will see later, you cannot put references in template classes. So, for example, neither <code>std::vector&lt;int&amp;&gt;</code> or <code>std::array&lt;int&amp;&gt;</code> are valid C++.</li> <li>▪ A reference is a synonym for another object.</li> <li>▪ You might think of a reference as a dereferenced, initialized pointer. Then again, you might not.</li> </ul>
virtual	<p>A keyword saying that a method might be overridden in a sub-class. The first place where you explicitly used this feature was with the virtual destructor of <code>Die</code>. However, the <code>TEST</code> macro actually adds a virtual method to the class it creates to allow for test execution. We will be seeing much more of this coming up.</p>
Virtual dtor	<p>The virtual destructor is a recommendation for any class that might be a base class. For now, form the habit. As you work through the book, more will be revealed.</p>

### 3.10 What's coming up?

Next up is the `DiceGame` itself. However, to effectively test the `DiceGame`, we will need to learn several new things:

- Inversion of Control & Dependency Injection
- Test Doubles
- The Mechanics of dynamic binding in C++ (virtual methods)
- A light spattering of pointers and references
- Subclass/inheritance and command/query separation
- More on iterators in the standard library.

### 3.11 Review Game Rules

The `DiceGame` has a few simple rules:

- Roll < 7, the player loses
- Roll > 7, the player wins
- Roll == 7, it's a push, neither a win nor a loss

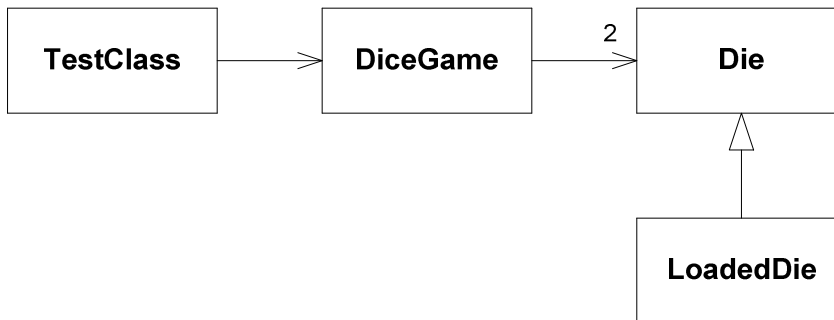
The `DiceGame` ultimately uses 2 die objects, both of which roll 1 – 6 (with an even distribution), so the range of possible totals is 2 – 12.



If you are taking a typical approach to this problem, you might simply write the production code, and be done with it. However, throughout this book we're only writing production code with tests in place first. This introduces a challenge: How can you write a test for a particular situation when what affects the results is a random event? The rules of the game are fixed, but depend on something outside of our control. Or is it?

### 3.11.1 Test Control

Consider this updated class diagram:



Imagine if you created a new class, `LoadedDie`. This class behaves like a `Die` but it can be set to always return a particular value. `LoadedDie` will take the place of `Die` for tests in the `DiceGame`. This allows a test to decide what's going to happen.

This might seem like cheating. However, you already have tests to verify that `Die` works. Given that we know `Die` works, then if `DiceGame` uses `Die` or something that behaves like `Die`, it will not invalidate our testing regime. Since we can control a `LoadedDie`, we can control the test results. In reality, we are selecting a particular path through the production code by controlling the test. This may be a subtle point, so let me restate it in a different way:

- `DiceGame` does not depend on how the roll method is implemented, it depends on what the roll method returns. So any roll method that returns a valid range of values will do. One that returns a random value is as good as one that returns a controlled or fixed value.

This idea is fundamental. `LoadedDie` is going to substitute for `Die` in the test. `LoadedDie` is called a *Test Double*<sup>10</sup>. Test Double is a general category. It describes anything used as a placeholder for a test, used to bring under control things that might otherwise change the results of the test. In this particular case, `LoadedDie` returns a fixed value, so a more specific name is a Stub.

The idea is basic; providing support in your designs for this is fundamental to better coding. It just so happens that making something testable also tends to lead to better design. However, at this point it's time to make a bold, if not misleading statement:

***Testing Trumps Design***

<sup>10</sup> <http://xunitpatterns.com/>, Meszaros, Gerard.

That is, given a choice between a “great design” and a “design that can be tested”, I’ll pick the testable design without a compelling reason to do otherwise. This is generally a false dichotomy as good designs are generally testable. So how can you take advantage of this idea for the `DiceGame`? It will require a few moving parts:

- Dependency Injection
- Polymorphism

### 3.11.2 Dependency Injection

Objects interact by sending messages. When one object talks to another object, the caller sends a message. A message is associated with a method, which is then executed. There are two parts to this relationship:

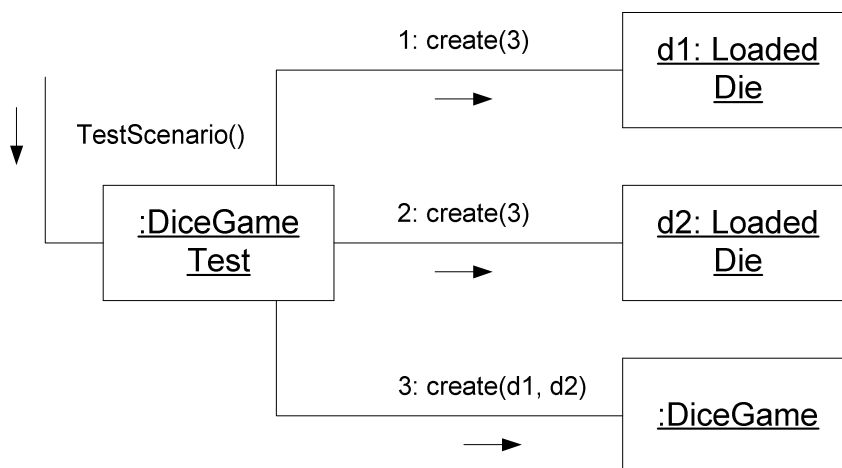
- What message to send
- Which object receives the message

The idea of dependency injection only addresses the second point. It has nothing to do with the first. So dependency injection is about controlling *which* objects are being sent messages.

In our particular situation, we have `DiceGame` and `Die`. `DiceGame` needs to `roll()` two die objects and then get their `faceValue()`. The question then remains, to which die objects will the `DiceGame` talk? Alternatively, how will `DiceGame` get access to `Die` objects? There are two general ways this can happen:

- The `DiceGame` selects which `Die` objects it talks to
- The `DiceGame` is told which `Die` objects to talk to.

The second option is dependency injection. What does this look like from a design perspective? There are several options; here is one called constructor injection:



This is a UML Communication Diagram. A message starts the whole sequence, that’s the unnumbered “`TestScenario()`” message coming “from the outside.” That message results in three more messages (there’s more, I’m just showing the first three). The first and second messages create two instances of `LoadedDie`. Each is created with a value of 3

passed in. The message `create` is a convention to show that an object is being constructed. The `DiceGame` is then created in step three and it is given two handles two `LoadedDie` objects.

Rather than creating `Die` objects itself, the `DiceGame` is given those objects. If the `DiceGame` creates `Die` objects directly, controlling what happens is more difficult (not impossible, just more difficult).

As a sidebar, the first time I was introduced to the idea of Dependency Injection, it seemed so trivial that I rejected it as a significant idea. Sometimes I would pass things into constructors and sometimes I would not. I didn't see its impact on testability. I was confusing the mechanism, passing an object in, with the intent, providing a hook for control of a relationship. It took several months for the importance of this seemingly simple idea to sink in.

Given a `DiceGame` created with two `LoadedDie` objects, both of which were created with the value 3, can you guess the expected result of the test? If the `LoadedDie` objects simply return the value they were given at construction time, then  $3 + 3$  will give 6, which is less than 7. So the player will lose.

However, there are several moving parts to make this work in C++.

### 3.11.3 Polymorphism Moving Parts

Simply passing in a different object is not enough. We need to write the code in such a way that we get *different behavior* with `LoadedDie` than with `Die` objects.

Specifically, we want the sum seen by the `DiceGame` to be under our control. We have not yet written it as such, but assume for the moment that `DiceGame` uses `faceValue()` to calculate a sum. This means we need `LoadedDie` to return a controlled value from the `faceValue` method. For this to work:

- `Die` *should* have a virtual destructor
- `Die` *must* have at least one other virtual method, `faceValue()` in this case
- That virtual method (`faceValue()`) *must* be overridden by `LoadedDie`
- That virtual method (`faceValue()`) *must* be called by `DiceGame`
- `DiceGame` *must* hold either pointers or references to `Die`

The last four bullets enable dynamic binding of the method. That is, the particular method called depends on the type of the object receiving the message. If the `faceValue` message is sent to an instance of the `Die` class, it will call `Die::faceValue`. If the message is instead sent to an instance of the `LoadedDie` class, then it will instead invoke `LoadedDie::faceValue`. This determination is made at runtime, rather than at compile time. Thus, the method is dynamically bound. In C++, most of the work is calculated at compile time, so while there is some overhead, it's small. It's the cost of an indirect jump versus a direct jump.

That is a whole lot to take it. Rather than just jumping right in, we will continue the practice of working through a series of automated tests and changing production code to produce the desired effect.

### 3.12 Testing Into It: LoadedDieTest

Before we can use `LoadedDie`, we'll need to create it. So that's a good place to start. Here's the first version of `LoadedDieTest.cpp`:

```
#include "LoadedDie.h"

#include <CppUTest/TestHarness.h>

TEST_GROUP(LoadedDieShould) {
};

TEST(LoadedDieShould, HaveFaceValueEqualToConstructorParamaterValue) {
    LoadedDie die(5);
    LONGS_EQUAL(5, die.faceValue());
}
```

This test simply demonstrates that:

- There is (or rather will be) a class called `LoadedDie`
- Constructing an instance of `LoadedDie` takes a single parameter
- Calling the `faceValue` method on a `LoadedDie` returns the value passed into the constructor.

Of course, if you create just this test, it will not compile because there is no `LoadedDie` class yet.

#### 3.12.1 Options: Interface/Concrete Inheritance

There are two ways to accomplish our goal:

- Create `LoadedDie` as a subclass of `Die`
- Create a top-level *interface*<sup>11</sup> that both `Die` and `LoadedDie` inherit from

The second approach is more pure, but it also includes more moving parts. For now, we will pick the first option. However, we will use the second option later in this project.

With this decision made, here is `LoadedDie`'s header file:

```
01: #pragma once
02: #ifndef LOADEDIE_H_
03: #define LOADEDIE_H_
04:
05: #include "Die.h"
06:
07: class LoadedDie: public Die {
08: public:
09:     LoadedDie(int value);
10:     int faceValue() const;
11:
12: private:
```

---

<sup>11</sup> C++ does not actually support interfaces, but there is an idiomatic way to create something similar. In any case, an interface is something that describes pure behavior without providing any implementation. That is, it provides method declarations but no method definitions.

```
13:   int loadedValue;
14: };
15:
16: #endif
```

The only new thing in this example is on line 07. The part from the : to the { is how you create a subclass. This states that LoadedDie publicly inherits from the class Die. As we will see, a LoadedDie is “compatible” with or “substitutable for” a Die. Anywhere code wants a Die, we can provide a LoadedDie.

To use inheritance, the code must include the header file for the base class, Die.h in this case. Inheritance is simultaneously powerful and highly coupling. I’ll have more to say on this in a bit.

### 3.12.2 LoadedDie Implementation

```
#include "LoadedDie.h"

LoadedDie::LoadedDie(int value) : loadedValue(value) {
}

int LoadedDie::faceValue() const {
    return loadedValue;
}
```

There is no new syntax in this example. The constructor is provided an int called value. The member data, loadedValue, is initialized with value in the member-wise initialization list. The method faceValue returns loadedValue instead of value, which is what Die::faceValue() does. LoadedDie::faceValue is meant to replace, or override, Die::faceValue.

### 3.12.3 Get your test passing

You have three files to create:

- LoadedDieTest.cpp
- LoadedDie.h
- LoadedDie.cpp

Get to green before moving on.

Here’s a question to ask yourself: In Polymorphism Moving Parts on page 57, there’s a list of bullets. Does this solution conform to those bullets?

### 3.12.4 Experiment in Failure

Make an update to your LoadedDie test:

```
TEST(LoadedDieShould, HaveFaceValueEqualToConstructorParamaterValue) {
    LoadedDie die(5);
    LONGS_EQUAL(5, die.faceValue());
    Die &d = die;
    LONGS_EQUAL(5, d.faceValue());
}
```

Earlier I mentioned that you do not typically use references in the middle of a method and here I am violating that. It's to demonstrate a failure. The variable `d` is a reference to the variable `die`. That is, `die` and `d` are the same variable. If they are the same value, then it seems to make sense that the `faceValue` method should give the same result. If that's the case, then this test will pass.

Check, are you green?

```
TEST(LoadedDieShould, HaveFaceValueEqualToConstructorParamaterValue)
```

```
..\LoadedDieTest.cpp:12: error: Failure in TEST(LoadedDieShould, HaveFaceValueEqualToConstructorParamaterValue)
```

```
    expected <5 0x5>
```

```
    but was <1 0x1>
```

The test fails. The second assertion shows that the `faceValue` is 1 instead of the expected (or desired 5). So there's a problem with the current implementation.

### 3.12.5 Fixing It

The problem is that C++ does not pick the method at runtime unless the method is declared virtual in the base class' definition. So the `faceValue` method needs to be declared virtual. As soon as there's one virtual method in a class, the destructor should be declared virtual as well<sup>12</sup>. While nothing bad will happen here, there is a possibility of a memory leak if you do not do this. So form the habit of making destructors virtual unless you have a good reason not to.

Simply change `Die.h` so that the destructor and the `faceValue` methods are declared virtual:

```
class Die {
public:
    Die();
    virtual ~Die();
    int roll();
    virtual int faceValue() const;
```

Make this change and see that your tests are now passing. Congratulations, you've just used a test to confirm the need for a virtual method.

### 3.12.6 Overloading `faceValue` versus `roll`

Why did `LoadedDie` override `faceValue` versus `roll`? A quick answer is "because that's how we wrote it." More fundamentally, how would you choose given this situation? Alternatively, how can we render the decision moot?

---

<sup>12</sup> This is a surface-level rule. In fact, if sub-classes do no dynamic memory allocation then the destructor does not need to be virtual. There's even more. However, the overhead of making the destructor virtual when you have another virtual method is much lower than adding the first virtual method.

**Command-query separation** suggests that a method should either produce a change or return a value, but not do both. The `roll()` method violates this as it, presumably, changes the value of a `Die` and returns the value just rolled.

That means that a client of a `Die` could simply call `roll()` and never call `faceValue`. Do you see a problem with this? In the section *Polymorphism Moving Parts*, on 57, one of the bullets states that the client must call the virtual method. If the only method that is virtual is `faceValue`, and it is possible that a client could call `roll` and get the same result, then it is possible that a client will not call the virtual method.

There are several ways to address this problem:

- Ignore it. Buyer beware. Some assembly required. Not a great option, but a common one.
- Change the `roll()` method to call `faceValue`.
- Change `roll()` to not return a value, forcing clients to call `faceValue`.

Let's look at those last two options:

#### **Fixing DRY violation**

The current implementation violates DRY. There are two ways in which the `faceValue` can be acquired by client code. Here's the current implementation:

```
int Die::roll() {
    using namespace std::tr1;
    static mt19937 engine;
    static uniform_int<int> uniform(1, 6);

    return value = uniform(engine);
}
```

Here is a version with the DRY violation removed:

```
int Die::roll() {
    using namespace std::tr1;
    static mt19937 engine;
    static uniform_int<int> uniform(1, 6);

    value = uniform(engine);
    return faceValue();
}
```

This solves the problem because `faceValue` is always called. There were two ways in which the `faceValue` was returned. Now there is only 1 way; there is always a path through the `faceValue`.

This is a good solution; however, it may not be not an obvious one until you consider the possibility of a subclass.

#### **Fix command-query separation violation**

Another way to fix the problem is to change `roll` to not return a value. This forces the client to call `faceValue` if it needs the value. The updated `roll()` method becomes:

```
void Die::roll() {
    using namespace std::tr1;
    static mt19937 engine;
    static uniform_int<int> uniform(1, 6);

    value = uniform(engine);
}
```

Notice that these two approaches result in nearly the same code.

### 3.12.7 Fixing Die: Command Query Separation

Update the `Die` class to remove the command-query separation. The definition is above. You will also need to change the declaration (in the header file). Make sure you are green before continuing.

### 3.12.8 Review

Inheritance is defining one class in terms of another. `LoadedDie`'s definition depends on `Die`'s definition, so much so that the header file of `LoadedDie` must include the header file of `Die`. In fact, inheritance is the highest form of coupling in class-based, statically typed languages (e.g., C++, Java, C#). This is ironic in the sense that inheritance offers the possibility of flexibility at the cost of being fragile.

If you plan to use inheritance and polymorphism in your solution, there are several steps required in C++ to make it happen:

- There must be a base class.
- There must be a derived class
- There must be method declared virtual in the base class
- The derived class must override the base class method
- A client must invoke the method through a pointer or reference to the base class
- You should make the destructor virtual in the base class.

That's a bit to remember. Practice it.

Even if you understand the mechanics, however, knowing when to do this is something that only comes with experience. You need to try things out to see if they work. One thing that helps is trying to put methods with objects that have the information to make decisions or do the work. This is one of Craig Larman's GRASP patterns; specifically Information Expert. If you find yourself using a lot of if then else blocks or switch statements, you may be missing an opportunity.

Once you figure out how, and then even when, there's still the issue of avoiding a DRY violation and watching out for command-query separation violations. You'll see several more examples later.

## 3.13 What's on Deck?

It's time to create the actual `DiceGame` class. There's some work to be done, so this is what's coming up:

- Creating the `DiceGame` from scratch, strictly using tests
- Generalizing production code by adding tests



- Refactoring code by extracting classes
- Making tests easier by changing the APIs of classes
- We'll revisit dependency injection
- Dynamic memory allocation
- Using destructors

### 3.14 Test-Driven Walkthrough

Here's a first test to cover one of the conditions of the `DiceGame`:

```
01: #include "LoadedDie.h"
02: #include "DiceGame.h"
03: #include <CppUTest/TestHarness.h>
04:
05: TEST_GROUP(DiceGame) {};
06:
07: TEST(DiceGame, BalanceDecreasesForLoss) {
08:     LoadedDie *d1 = new LoadedDie(3);
09:     LoadedDie *d2 = new LoadedDie(3);
10:     DiceGame game (d1, d2);
11:     game.play();
12:     LONGS_EQUAL(-1, game.getBalance());
13: }
```

Line	Description
08, 09	Create an instance of a <code>LoadedDie</code> using <code>new</code> . The <code>new</code> operator allocates enough memory to hold the thing being created, calls a constructor (if what is being created is non-primitive), and returns a pointer to the allocated memory. Objects created with <code>new</code> live until the code uses the <code>delete</code> operator to release the memory. This code does not use <code>delete</code> , so either there is a memory leak or the game calls <code>delete</code> to free memory.
10	Create a <code>DiceGame</code> . This is a local variable, so it will go away once this block of code finishes. The <code>DiceGame</code> 's constructor is passed two <code>LoadedDie</code> pointers. Initially we will do nothing with those pointers, and then we'll release the memory properly.
11	Call the game's <code>play</code> method. The <code>play</code> method will decide if the game is a win, lose or draw based on the sum of the die objects passed into the constructor.
12	This test controlled what's going to happen. Another way to think about this is that the test has picked one particular path through the production code. The expected result is a loss, so the balance, assumed to be initially 0, is reduced by one. The test defines its own reality.

#### 3.14.1 What's required to make this work?

First, of course, is the `DiceGame` class. It does not exist. A review of the test suggests several demands on the `DiceGame` class:

- It must have a constructor that takes two parameters. Pointers to something. While the code creates `LoadedDie` objects, we'll be treating them as `Die` objects.
- The game must have a play method.
- The game must have a `getBalance` method.
- The game, apparently, is responsible for properly handing the dynamically allocated memory of the `LoadedDie` objects.

### 3.14.2 DiceGame Header

Here's a minimal header that, with the provided source, will get the code back to compiling:

```

01: #pragma once
02: #ifndef DICEGAME_H_
03: #define DICEGAME_H_
04:
05: class Die;
06:
07: class DiceGame {
08: public:
09:     DiceGame(Die *d1, Die *d2);
10:
11:     void play();
12:     int getBalance() const;
13: };
14:
15: #endif

```

Line	Description
05	Declare that there is a class <code>Die</code> . This is also known as a forward-declare. There is a header file that contains the <code>Die</code> definition. Including header files is expensive. A review of the rest of this header file only shows one more use of the <code>Die</code> type (so far). Line 09 makes a reference to <code>Die*</code> . <code>Die*</code> is a pointer. All pointers are primitives, and more importantly, the same size. So the compiler does not need size information, therefore it does not require a definition of <code>Die</code> , it just needs to know that the class exists.
09	The constructor takes in two <code>Die</code> pointers. The test creates <code>LoadedDie</code> with <code>new</code> . <code>new</code> returns pointers, which we are providing to <code>DiceGame</code> . Since <code>LoadedDie</code> inherits from <code>Die</code> (using so-called public inheritance), a pointer to a <code>Die</code> can also point to a <code>LoadedDie</code> . You already saw something similar in a test where a reference to a <code>Die</code> (also a primitive type) referred to a <code>LoadedDie</code> .

### 3.14.3 DiceGame Source

This will get the code compiling but with a failing test:

```
#include "DiceGame.h"
```

```
DiceGame::DiceGame(Die *d1, Die *d2) {  
}  
  
void DiceGame::play() {  
}  
  
int DiceGame::getBalance() const {  
    return -1;  
}
```

#### 3.14.4 Get it Compiling

You have a little bit of work:

- Create `DiceGameTest.cpp`
- Create `DiceGame.h`
- Create `DiceGame.cpp`

Get the code compiling and run all tests. The code should fail, but the failure may be a surprise:

```
TEST(DiceGame, BalanceDecreasesForLoss)  
  
..\DiceGameTest.cpp:7: error: Failure in TEST(DiceGame,  
BalanceDecreasesForLoss)  
  
    Memory leak(s) found.  
  
Leak size: 12 Allocated at: ..\DiceGameTest.cpp and line: 8. Type:  
"new" Content: "ËÇÀ"  
  
Leak size: 12 Allocated at: ..\DiceGameTest.cpp and line: 9. Type:  
"new" Content: "ËÇÀ"  
  
Total number of leaks: 2
```

There's a memory leak. This is the primary reason I chose to use CppUTest for this book. It has simple memory leak detection built in. There are other solutions I could have used, but all of them require more moving parts, so I decided to keep it simple.

You have two quick options to fix this memory leak:

- Have the test free the memory
- Have the `DiceGame` free the memory

If we have the test free the memory, then every test will have to do it. That's a DRY violation. It also seems like a poor assignment of responsibility. If we have the `DiceGame` free the memory, it resolves the DRY violation, but it complicates the `DiceGame` class and it requires "tribal knowledge" of your clients. Your clients will have to know to only pass in objects created using `new`. There is no language-defined

way to determine if a given pointer can safely be deleted or not<sup>13</sup>, so you must just “be careful” (at least for now).

We are going to have `DiceGame` free the memory. We will remove the requirement of tribal knowledge for successful use through a better API design. We’re going to do that later.

### 3.14.5 Handle the memory leak, fix the test

`DiceGame` needs to hold on to two `Die` pointers. We have several options to accomplish that:

- Hold two attributes
- Use an `std::array`
- Use an `std::vector`

Since I want you to learn how to use some of the standard libraries, we’ll use the `vector` class. However, any of these solutions would work.

#### *Update the header*

```

01: #pragma once
02: #ifndef DICEGAME_H_
03: #define DICEGAME_H_
04:
05: class Die;
06: #include <vector>
07:
08: class DiceGame {
09: public:
10:     typedef std::vector<Die*> DiceCollection;
11:     typedef DiceCollection::iterator iterator;
12:
13:     DiceGame(Die *d1, Die *d2);
14:     virtual ~DiceGame();
15:
16:     void play();
17:     int getBalance() const;
18:
19: private:
20:     DiceCollection theDice;
21: };
22:
23: #endif

```

Line	Description
06	Use the <code>std::vector</code> class.
10	Use a nested <code>typedef</code> to make using the <code>std::vector</code> just a bit easier.

<sup>13</sup> That is, given a pointer, there is no language-defined way to tell if it points to something created with `new` or not. You can only delete things created with `new`.

Line	Description
	DiceCollection is a synonym for a vector of pointer to Die.
11	The vector class has an iterator. Create a synonym in the DiceGame class with the same name as the nested typedef. An iterator is included with all of the standard collections and it is one way to work with each of the elements in a given collection.
20	Store an std::vector<Die*>, but make it read well.

### Update the source

```

01: #include "DiceGame.h"
02:
03: #include "Die.h"
04:
05: DiceGame::DiceGame(Die *d1, Die *d2) {
06:     theDice.push_back(d1);
07:     theDice.push_back(d2);
08: }
09:
10: DiceGame::~~DiceGame() {
11:     for(iterator
12:         iter = theDice.begin(); iter != theDice.end(); ++iter)
13:         delete *iter;
14: }
15: void DiceGame::play() {
16: }
17:
18: int DiceGame::getBalance() const {
19:     return -1;
20: }

```

Line	Description
06, 07	Add d1 and then d2 to the end of the vector. This is in the body of the constructor. By the time the code reaches this point, the vector's constructor has already been called. It is not possible to use a constructor to automatically insert these, so it is done after initialization of the vector.
11	Iterate over the entire vector, one element at a time.
12	Call delete, one by one, on each item in the vector.

### Get to green

Make these changes, you should be green.

### Follow the idiom

There is one more problem with the class. It can be copied. This might not seem like a bad idea, but if a copy is made, the copy will get copies of the pointers stored in the contained vector. Let me say that again: The copied object will have *copies of the pointers to the Die* objects, not copies of the Die objects. Eventually the original (or the

copy) will go away, freeing memory. The remaining version will either use the now deallocated objects or attempt to delete them as well. What happens in both cases is not defined, but it falls under the category of “things your mother told you to avoid.”

The easiest thing to do is not allow copies. Simply add this section to your `DiceGame`'s header file just before the close curly bracket:

```
private:
    DiceGame(const DiceGame&);
    DiceGame& operator=(const DiceGame&);
```

At this point it may be worth mentioning that the `private:` is not necessary if there's already another `private:` above. I do this by convention. The bottom of my class has `private:`, but unimplemented methods.

Make sure your solution is still green, and then continue.

### 3.14.6 Always losing is no fun

The implementation leaves a lot to be desired. On the one hand, since the value -1 is already returned, the worse you can ever do is losing once. On the other hand, you always lose once. To infinitely improve the odds of winning, it's time for another test:

```
TEST(DiceGame, BalanceIncreasesForWin) {
    LoadedDie *d1 = new LoadedDie(4);
    LoadedDie *d2 = new LoadedDie(4);
    DiceGame game (d1, d2);
    game.play();
    LONGS_EQUAL(1, game.getBalance());
}
```

Create this test and verify that it fails.

#### ***What all do we need?***

This one simple test is going to force a bit of coding:

- The code needs to actually check for winning or losing
- The code needs either a balance or a way to remember if the previous time was a win or a loss

Here is one way to get this test to pass:

#### ***Add a new field to the class:***

```
private:
    DiceCollection theDice;
    int balance;
```

#### ***Don't forget to initialize that balance***

Add a member-wise initialization list to set balance to 0:

```
DiceGame::DiceGame(Die *d1, Die *d2) : balance(0) {
```

You do not need to do anything with the other member data, `theDice`. Its no-argument constructor is called automatically.

**Now do something in play**

```

01: void DiceGame::play() {
02:     int sum = 0;
03:     for(iterator
           iter = theDice.begin(); iter != theDice.end(); ++iter) {
04:         (*iter)->roll();
05:         sum += (*iter)->faceValue();
06:     }
07:
08:     if (sum > 7)
09:         ++balance;
10:     if (sum < 7)
11:         --balance;
12: }

```

Line	Description
03	Iterate over every element in the collection.
04	Roll the current die object. Since the vector holds pointers and an iterator is a pointer to what the vector holds, the iterator type is actually <code>Die**</code> . The expression <code>(*iter)</code> results in a pointer to a die object. <code>-&gt;</code> then sends a message through a pointer. You could also write this <code>(**iter).roll()</code> .
05	Add to the sum the <code>faceValue()</code> of the current die object.
08 – 11	Conditionally increment or decrement the balance depending on the sum. Note I could have used an <code>else if</code> on line 10, but the code is clear enough that I don't do so.

**Return the balance**

Since `balance` now holds the results, `getBalance()` needs to return it:

```

int DiceGame::getBalance() const {
    return balance;
}

```

**Get back to green**

Make these changes and get back to green.

**One final test**

There's no test for a "push". Even though reviewing code suggests it is handled correctly, let's write a test. This test serves as a reminder of the rules of the game and it constrains the implementation to work for all the conditions we know about.

```

TEST(DiceGame, BalanceRemainsSameForPush) {
    LoadedDie *d1 = new LoadedDie(4);
    LoadedDie *d2 = new LoadedDie(3);
    DiceGame game (d1, d2);
    game.play();
    LONGS_EQUAL(0, game.getBalance());
}

```

Add this final test and verify that your solution is still green.

### 3.14.7 Experiments in Failure

Now for a few experiments to see what might happen if you make a few common mistakes. Here's what's on deck:

- Removing the body of the `DiceGame` destructor
- Removing `virtual` from the declaration of the `Die` destructor
- Removing `virtual` from the declaration of `Die::faceValue`
- Removing `const` from the declaration and definition of `LoadedDie::faceValue`
- Adding `virtual` to the declaration of `Die::roll`

#### **Body of `~DiceGame`**

Make the following change to the body of `DiceGame`:

```
DiceGame::~DiceGame() {  
    // for(iterator  
    //     iter = theDice.begin(); iter != theDice.end(); ++iter)  
    //     delete *iter;  
}
```

Run your tests and notice what happens. In my solution, I end up with three failures. All of those failures related to memory leaks. If you do a little research, you'll notice that there are three tests using the `DiceGame` class.

This is a demonstration of a memory leak, but why is there a memory leak?

In a nutshell:

- The `std::vector` class is a generic container, so it knows nothing about what you are choosing to put into it. This includes the difference between pointers and non-pointers.
- There is nothing built into the language making it possible to tell if something that happens to be a pointer also happens to be dynamically allocated or not. There are things you can do under the covers to strongly suggest such a thing, and it is possible to use custom libraries to make it possible, but no language-defined way.
- Since the `std::vector` class can only depend on standards, there's no way to know if what is holds is a pointer to dynamically allocated memory.
- Even if there were a way to make that determination, there's no way the `std::vector` could know that it is responsible for cleaning up that memory. It would be possible add some kind of flag, but then you're back to the issue of not having a standard way to know if something is both a pointer and that it points to dynamically allocated memory.
- Therefore, the vector has no policy regarding what is put in it.

That's not to say that the `std::vector` does not handle its own memory allocation. A vector holds onto a block of memory that it dynamically allocated. The vector manages an initial block, which can change size over time. Here's a logical image of some sample code:

```
void foo() {
```

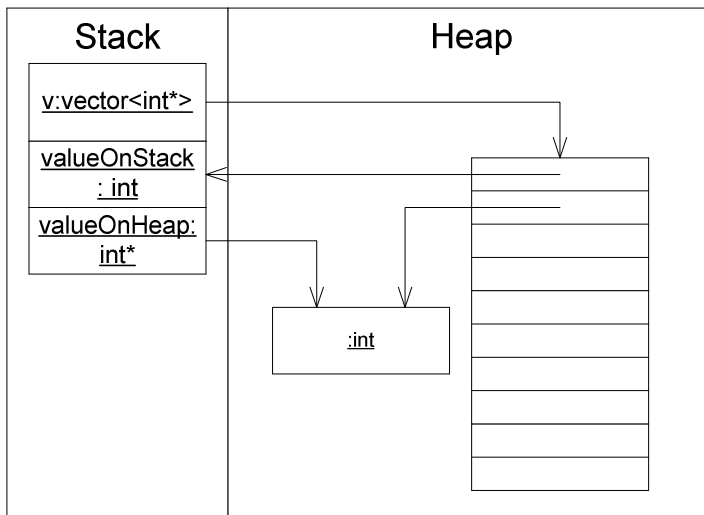


```

std::vector<int*> v;
int valueOnStack;
int *valueOnHeap = new int;
v.push_back(&valueOnStack);
v.push_back(valueOnHeap);
}

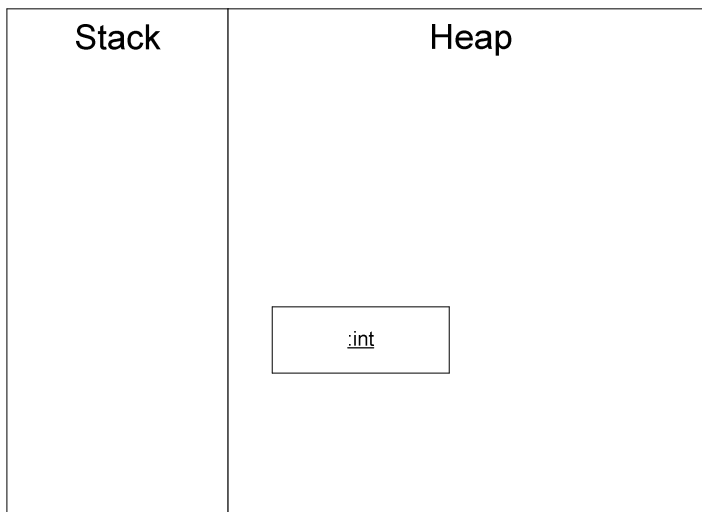
```

Given this simple code, what does this logically look like in memory?



The code example includes three local variables, `v`, `valueOnStack` and `valueOnHeap`. Those variables are stored on the program stack. The `std::vector` holds onto a block of memory (its default size is typically 10). After putting two values onto the vector using `push_back`, the value at index 0 is an address to the local variable `valueOnStack`. The value at index 1 is a copy of the address held onto by the variable `valueOnHeap`.

When this method exits, the program stack is cleared. Anything that is an object on the program stack will have its destructor called. Primitive values like `int` and `int*` do not have destructors. So given this code example, this is what the stack and the heap will resemble after the program terminates:



The `std::vector` class assumes no responsibility for what is added to it. It assumes responsibility for the memory it allocates, which is what it uses to hold on to the things you add to it. However, if you add a pointer to a dynamically allocated object, then you have to take care of releasing that memory.

The body of the destructor was doing that. We'll revisit this in a bit to see another way to make this happen automatically.

Before moving on to the next experiment, uncomment the body of the destructor and verify your code is still green.

### **Removing virtual on ~Die declaration**

In this next experiment, you're going to simply remove `virtual` to "see" what happens. So remove the keyword `virtual` from the declaration of the `Die` destructor and run your tests. What happens?

It appears as if nothing happens, and in this particular case nothing bad happens. To have this kind of change cause a problem requires several steps:

- A derived class inheriting from a base class (we have this)
- Creating an instance of a derived class using `new` (we have this)
- Holding on to that newly-created object via a pointer (we have this)
- Releasing memory to that newly-created object through its pointer (we have this)
- The base class' destructor must be non-virtual (we just did this)
- The derived class must dynamically allocate memory (missing)

In fact, it takes quite a few moving parts to have a non-virtual destructor in a base class cause problems for a derived class. However, to convince you of the problem, make the following change to your `LoadedDie.h` (add an `std::vector`):

```
#include <vector>

class LoadedDie: public Die {
public:
    LoadedDie(int value);
```

```
    int faceValue() const;

private:
    int loadedValue;
    std::vector<int> problemChild;
};
```

Now change your LoadedDie.cpp (just the constructor):

```
LoadedDie::LoadedDie(int value) : loadedValue(value) {
    problemChild.push_back(value);
}
```

Make these changes, run your tests. You'll notice three tests failing due to memory leaks.

Now, make the destructor in Die virtual and see that the tests pass. Finally, remove the use of `std::vector` in LoadedDie (both the header and the source).

### ***What just happened?***

Your LoadedDie class had a destructor even if you do not write one for it. Even if you do write one for it, the call to the destructor of contained objects is added by the compiler (remember, destructors are special). The `std::vector` class allocates memory. Exactly when is left to a given implementation, however adding the line to the constructor to put a single value on the vector forces the vector to perform dynamic memory allocation. When the DiceGame's destructor is called, it calls delete on each of its contained pointers to Die objects:

```
DiceGame::~~DiceGame() {
    for(iterator
        iter = theDice.begin(); iter != theDice.end(); ++iter)
        delete *iter;
}
```

The question is this, "which destructor gets called?" If the destructor is declared virtual in Die, then the compiler will insert just enough code to make sure that the correct destructor is called<sup>14</sup> at runtime. If the destructor is not declared virtual, then the compiler inserts a call to the destructor of the type returned by \*iter). Since iter is a pointer to a Die, \*iter is a Die, so the destructor called is ~Die, regardless of what the pointer actually points to.

Adding virtual to even one method comes with a cost, which is discussed later on. However, adding a second virtual method when there is already one is a much smaller cost. So if you have even one virtual method, just make the destructor virtual. If you are not using inheritance, then you do not need a virtual destructor. If you're not certain that either:

- The class you are working on will never serve as a base class
- The class won't be substituted with a test double during test
- Subclasses, if there are any, will not need to perform any dynamic allocation

Then you can safely leave off the virtual destructor.

---

<sup>14</sup> This has to do with virtual method dispatch. More on this later.

On the other hand, if you are writing unit tests as you write the rest of your code, you'll probably find pretty quickly if there's a missing virtual destructor, so you can probably get away with leaving this until you need it.

Make sure your code is green before moving on to the next experiment.

### **Removing virtual from the declaration of `Die::faceValue`**

This is an experiment you should have already performed. In any case, make this change and see what happens.

You'll notice several failing tests, 4 in my case. Why is this? For the same reason that the wrong destructor is called, the wrong version of `faceValue` is called.

Here's the offending code:

```
01: void DiceGame::play() {
02:     int sum = 0;
03:     for(iterator
04:         iter = theDice.begin(); iter != theDice.end(); ++iter) {
05:         (*iter)->roll();
06:         sum += (*iter)->faceValue();
07:     }
```

Notice the call to `faceValue` on line 06? Which version of `faceValue` is called? If the method is declared virtual, then the compiler inserts just enough code to make sure that the correct version of `faceValue` is called. If the `faceValue` method is not declared virtual, then the compiler inserts enough code to call the method based on the type of `*iter`. Since `*iter` is equal to `Die**`, `*iter` results in a pointer to a `Die`, or `Die*`. The compiler selects the method `Die::faceValue` as a result. Making `faceValue` virtual forces the compiler to insert a little bit more to make this work.

At this point it is worth mentioning a few more groups of terms:

Term	Description
static type (or) compiler-time type	The type known by the compiler based on a static analysis (compile-time reading) of the type information. This is what can be determined strictly from looking at the compilation unit.
dynamic type (or) run-time type	The actual type of the object at runtime. This may or may not be different from the static type. If the static type is either a pointer or a reference to an object, then the static type and dynamic type can be different.

In this most recent example, the static type of `iter` is `Die**`. The static type of the expression `*iter` is `Die*`. However, given that our tests created instances of `LoadedDie`, here's how the analysis goes. The static type of `iter` is still `Die**`, as is the dynamic type. However, while the static type of `*iter` is `Die*`, the actual type is `LoadedDie*`. This only becomes important upon calling a method that is virtual. For non-virtual methods, the compiler selects the method based on the static type. For virtual methods, the compiler selects the method based on the dynamic type. Since the compiler cannot possibly know what the actual run-time type is, it instead inserts a level of

indirection that, when the program runs, causes the correct method to be called. This is called virtual function dispatch. There's more on this subject later.

Add the virtual keyword back on to the declaration of `faceValue`. Get your code back to green, and then move to the next section.

### **Removing const from the declaration and definition of `LoadedDie::faceValue`**

When you mean to override methods, the signatures need to match exactly<sup>15</sup>. Remove the `const` keyword from both the declaration and definition of `faceValue` in `LoadedDie` and run your tests. If you only remove it from one but not the other, your code will not compile.

Interestingly, you'll have the same tests failing as before. In this case, while the base method is virtual, the derived method, as written, does not match so it does not override. That is, it's an unrelated method and it does not have any impact on dynamic method selection.

In general, when I override a method from a base class in a derived class, I'll copy the method signature from the base class' definition into the derived class' definition. Another thing to consider, when writing automated tests meaning to invoke overridden methods, hold a reference or pointer to the base class rather than the derived class. This is a sure way to make sure you've actually overridden a base class method as intended.

Return your code to green. Add the `const` keyword back on to the method declaration in the header file and the method definition in the source file.

### **Adding virtual to the declaration of `Die::roll`**

Is there any value in making `roll` virtual? There are no subclasses that override `roll`, so making it virtual will have no noticeable effect. You could look at the size of the executable before and after. When I do that, before I make the change, my executable is 784,060 bytes. After the change, the size is unchanged. Without creating a subclass, or doing some sophisticated timings, it will be hard to tell whether or not `roll` is in fact virtual.

Originally, the `roll()` method had a return value. This violated command-query separation, and you fixed it. Now it returns no value, so it makes it less likely that a client can depend on its effect directly. So this experiment is not too fruitful.

You can return `roll` to non-virtual or leave it virtual. The end result will not change.

## 3.15 Recap

Term	Description
<code>: public die</code>	<p>The syntax for making one class inherit from another. After <code>class LoadedDie</code> and before the open curly bracket <code>{</code>, add this to define that one class is a subclass of another.</p> <p>In all of the examples in this book we will use <code>: public</code>, but <code>private</code> and <code>protected</code> are available as options and outside our</p>

<sup>15</sup> This is somewhat simplified, return types can be co-variant, but this is the only mention of that idea in this book. It's too advanced for this book.

<b>Term</b>	<b>Description</b>
	scope. Using this for, so-called public inheritance, makes the class in the definition substitutable for the base class.
base class	A class used as part of the definition of another class, specifically in an inheritance relationship. All of a base class becomes part of the derived class. Whether what comes from the base class is available to the derived class depends on the whether the thing in question (method or attribute) is declared to be public, protected or private. Private data, for example, is part of the overall structure but a derived class cannot directly access it.
child class	In an inheritance relationship, it is the class that is defined in terms of another class. A child class knows its parent (or base class). The reverse is not true.
communication diagram	A type of UML diagram that shows objects, connections between objects and interactions, or message flows. Before UML 2.0, this was called a collaboration diagram.
compile-time type	The type of a variable that can be determined strictly looking at the source code. It is independent of the order of execution (run time information).  The compile-time type limits what is available to be used. Only things in the public interface of the compile-time type are available to be used by code.
create	A special method used on UML diagrams. It typically suggests the execution of a constructor. This is a standard convention on dynamic diagrams in UML, which includes sequence and communication diagrams.
delete	Delete is somewhat analogous to free in C with several key differences. <ul style="list-style-type: none"> <li>▪ It is built into the language and it is known as the delete operator. Whereas free() is standard function in a library.</li> <li>▪ If delete is used on a pointer to an object, e.g., <code>Die*</code>, it will automatically call a destructor. If it instead points to a raw time, then there is no destructor to call.</li> <li>▪ There are three forms but I only use 1 in this book. The three forms are delete, delete[] and delete(). The second and third form are array delete and placement delete.</li> </ul> <p>In C++ you should use new for all allocation and delete for all deallocation. Mixing new with malloc is not defined. Using delete on something allocated with malloc or free on something allocated with new is also not defined. It's easy to predict what will typically happen, but the standard says the results are undefined.</p>
delete versus free	See the discussion on delete.
dependency	Making it possible for a client class to provide dependent objects.

<b>Term</b>	<b>Description</b>
injection	For example, the <code>DiceGame</code> requires two <code>Die*</code> for its constructor, so its dependent objects must be provided.
derived class	A class that has a parent or base class. <code>LoadedDie</code> is an example of a derived class. This is a synonym for child class and subclass.
dynamic allocation	Using <code>malloc</code> or <code>new</code> are examples of dynamic allocation. Acquiring space for data at runtime rather than pre-allocating at compile time.
dynamic binding	Selection of a method at runtime rather than compile-time. In C++, virtual methods invoked through pointers or references are dynamically bound rather than statically bound.
dynamic type	A pointer or reference to a class or struct can in fact point (or refer to) an object of its particular type or any class or structure publicly derived from that structure. For example, <code>Die*</code> can legally point to either a <code>Die*</code> or a <code>LoadedDie*</code> .
heap	A region of memory used for dynamic allocation. Things created with <code>new</code> (or <code>malloc</code> ) are put on the heap rather than the stack. Things placed on the heap remain there until they are explicitly removed using <code>delete</code> (or <code>free</code> if <code>malloc</code> was used). The heap is also cleared when a program terminates.
inheritance	A relationship between classes. A base class serves as part of another class' definition. All structural parts of the base class are in the derived class. Accessibility is determined by whether the thing in particular is public, protected or private. Items that are protected are available for derived classes. Things that are private are there, but not available for access.
interface	C++ does not support interfaces directly. An interface represents a concept that describes behaviour but, provides no implementation for that behaviour.
message	When a client makes a request of an object, it sends the target object a message. A message gets turned into a method at some point. In C++, that translation can be done at compile time or runtime. If the target is an object, then the determination is done at compile time. If the target is a pointer or reference to an object, then the determination is done at runtime if the message is virtual or compile time if the method is non-virtual.
method	A synonym for member function. A method is a behaviour of an object. A class declares a number of methods, some virtual, others non-virtual. Methods that are declared virtual can be overridden by sub-classes.
new	A keyword in C++, it provides a mechanism for allocating memory at runtime rather than compile time. Whereas <code>malloc()</code> is standard function in a library, <code>new</code> is a built-in operator.

<b>Term</b>	<b>Description</b>
new versus malloc	<p>When new is used to create an object, the compiler automatically calls a constructor. The sequence of steps for new used on an object:</p> <ul style="list-style-type: none"> <li>▪ Allocate an appropriately-sized chunk of memory.</li> <li>▪ Call the constructor to initialize the memory.</li> <li>▪ Return the address of the allocated memory.</li> </ul> <p>Unlike new, malloc skips the middle step. There are three forms of new: new, new[], new(). The second and third forms are called array new and placement new, respectively. These forms are out of scope for this book.</p>
non-virtual destructor	<p>By default, destructors are non-virtual. If a class may serve as a base class and it is possible that derived classes might perform dynamic memory allocation, then it is a good idea to make base-class destructors virtual. There are several conditions required to make a memory leak happen, but it is possible.</p>
override	<p>Method overriding is where a derived class replaces an implementation of a virtual base-class method suitable for its purposes. We have one explicit example of this where the <code>faceValue</code> method of <code>LoadedDie</code> returns a predictable result.</p>
parent class	<p>Synonym for base class or superclass.</p>
polymorphism	<p>Pithy: same message, different method. That is, the same message sent to multiple objects results in different methods being executed based on which object receives a message. For example, sending <code>faceValue</code> to a <code>Die</code> object results in the most recent roll value (or 1 if the die has not been rolled) to be returned. Whereas, sending <code>faceValue</code> to a <code>LoadedDie</code> will always return the value passed into its constructor.</p>
request	<p>A synonym for message.</p>
response	<p>A synonym for method.</p>
run-time type	<p>The actual type of an object at program execution time. A pointer or reference to an object in C++ can point to or refer to a publicly derived subclass at runtime.</p>
stack	<p>AKA program stack. The place where local variables are automatically stored by the compiler. A variable on the stack lives as long as the block of code containing that variable lives. Variables on the stack automatically go away when a method or function returns. If a variable is a non-primitive, then its destructor is also called.</p> <p>Be careful here. Remember that pointers and references are primitive types. So, for example, a pointer to a die on the stack, when it goes away, does not result in a destructor call.</p>
static type	<p>Same as compile-time type. It is the type of a variable known to</p>



Term	Description
	the compiler.
stub	<code>LoadedDie</code> is an example of a stub class. It is a particular kind of test double that has a fixed value. You can think of a stub as a “snapshot in time.”
sub class	Synonym for derived class or child class.
super class	Synonym for base class or parent class.
test double	A place-holder for a real object used to put some part of a test under control. We created <code>LoadedDie</code> as a test double for <code>Die</code> to make sure we are able to test all of the rules of the <code>DiceGame</code> .
test isolation	There are several interpretations. Tests should not impact each other. However, in this section we introduce the possibility of using test doubles by using dependency injection to fully control what one particular test is doing.
virtual	A keyword added to member function declarations. A virtual method can be overridden by subclasses. If that method is invoked through a pointer or reference to a base class, the correct method will be selected at runtime based on the dynamic type of the object. That is, if <code>faceValue</code> is sent to a <code>Die</code> , then <code>Die::faceValue</code> is called, whereas if <code>faceValue</code> is sent to a <code>LoadedDie</code> , then <code>LoadedDie::faceValue</code> is invoked.
virtual destructor	Destructors for base classes should be declared virtual. If they are not, there’s a possibility of a memory leak, as demonstrated in <i>Experiment in Failure</i> stating on page 59.
virtual method	A method declared virtual can be overridden by subclasses. See virtual above.
virtual method override	Subclasses wishing to replace a base-class’ method with its own implementation override a virtual method declared in a base class.

### 3.16 What’s coming up?

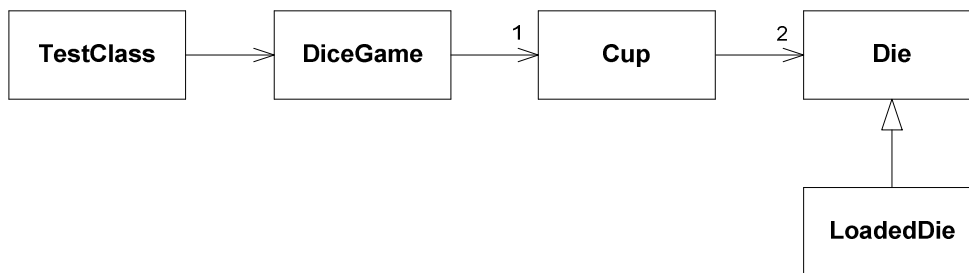
In this section we will revisit:

- Refactoring – specifically Extract Class
- Dependency Inversion Principle

We will additionally improve memory management by using `std::shared_ptr` and then have a first look at a design pattern called the Abstract Factory pattern.

#### 3.16.1 Remember Cup class?

Remember some time ago the mention of a Cup class:



Now it is time to revisit `Cup`. The code already exists, it just happens to be in the `DiceGame`. This burdens the implementation of `DiceGame` with both business rules as well as knowing how to store, roll, and deallocate multiple `Die` objects.

To fix this, you are going to apply an “Extract Class” refactoring. That is, there’s a class in the middle of another, and you are going to make it its own class.

### 3.16.2 Refactoring: Definition

You have already performed some refactoring. Even so, it’s worth remembering just what refactoring means: Changing the structure of a solution without changing its behavior.

In this class, you’ll move code from `DiceGame` into a new class called `Cup`.

### 3.16.3 Updated Cup Header

Here is a header file that captures the `Cup`-related concepts from the `DiceGame` class. Notice that this is a new file. You are not yet making changes to existing code.

```

#pragma once
#ifndef CUP_H
#define CUP_H

class Die;
#include <vector>

class Cup {
public:
    typedef std::vector<Die*> DiceCollection;
    typedef DiceCollection::iterator iterator;

    Cup(Die *d1, Die *d2);
    ~Cup();

    void roll();
    int total();

private:
    DiceCollection dice;
};

#endif
  
```

This code mirrors the code in `DiceGame`. So far, there is nothing new.

### 3.16.4 Updated Cup Source

The source code can be copied from `DiceGame` as well:

```
#include "Cup.h"
#include "Die.h"

Cup::Cup(Die *d1, Die *d2) {
    dice.push_back(d1);
    dice.push_back(d2);
}

Cup::~Cup() {
    for(iterator i = dice.begin(); i != dice.end(); ++i)
        delete *i;
}

void Cup::roll() {
    for(iterator i = dice.begin(); i != dice.end(); ++i)
        (*i)->roll();
}

int Cup::total() {
    int sum = 0;
    for(iterator i = dice.begin(); i != dice.end(); ++i)
        sum += (*i)->faceValue();
    return sum;
}
```

As with the header file, there's nothing explicitly new in this code. The one big difference is that the original code (still) has this coupled for loop in `DiceGame::play`:

```
void DiceGame::play() {
    int sum = 0;
    for(iterator
        iter = theDice.begin(); iter != theDice.end(); ++iter) {
        (*iter)->roll();
        sum += (*iter)->faceValue();
    }
    ...
}
```

This look both rolls the dice and calculates the total. The `Cup`'s implementation of `roll()` and `total()` have had this one loop split into two loops. In fact, Martin Fowler calls this a "split loop" refactoring. The original code unnecessarily coupled rolling and calculating the total.

### 3.16.5 Getting to Compiling

Before changing any of the old production code, get the header file and source file for `Cup` created and compiling. Get you code back to Green.

### 3.16.6 Updating DiceGame

Now it is time to change the `DiceGame` to use the new code. However, rather than just changing all of the code in place, these next steps use a technique often called parallel development. The goal is to get the code as close to complete as possible so that the amount of time that your automated tests fail is as small as possible.

#### **Updating class definition**

First, we'll add some kind of "handle" to a `Cup`. You have three general options (right now):

- A full `Cup`

```
private:
```

```
    Cup cup;
```

- A pointer to a `Cup`

```
private:
```

```
    Cup *cup;
```

- A reference to a `Cup`

```
private:
```

```
    Cup &cup;
```

All three will work but there are some advantages and disadvantages:

- A full `Cup` will be automatically instantiated correctly. Its destructor will be called automatically as well. However, there's no chance for dynamic binding, so testing is a bit more difficult. Also, to hold a full `Cup`, the `DiceGame` header file will have to include the `Cup` header file. Including header files is relatively expensive.
- A pointer is not automatically instantiated or released. Some code somewhere will have to call `new` and `delete` or something more convoluted. However, you do not have to include the header file of `Cup` in the header file of `DiceGame`. A pointer also allows for the possibility of virtual methods to play a role in the game.
- A reference has nearly the same characteristics as a pointer. The primary difference is that a reference must be initialized, which will require you to use a member-wise initialization list. Once initialized, it can never point to anything else.

My preference for testability leads me to the second two options. References are OK as attributes, but I typically reserve them for parameters and return values. For this solution, therefore, I'll go with a pointer. With that in mind, here's an updated `DiceGame` using parallel development:

```
01: #pragma once
02: #ifndef DICEGAME_H_
03: #define DICEGAME_H_
04:
05: class Die;
06: #include <vector>
07: class Cup;
08:
09: class DiceGame {
10: public:
11:     typedef std::vector<Die*> DiceCollection;
12:     typedef DiceCollection::iterator iterator;
```

```

13:
14:     DiceGame(Die *d1, Die *d2);
15:     virtual ~DiceGame();
16:
17:     void play();
18:     int getBalance() const;
19:
20: private:
21:     DiceCollection theDice;
22:     Cup *cup;
23:     int balance;
24:
25: private:
26:     DiceGame(const DiceGame&);
27:     DiceGame& operator=(const DiceGame&);
28: };
29:
30: #endif

```

Line	Description
07	Forward declare the Cup class. This just tells the compiler that there is a class called Die somewhere in the system.
22	Add a pointer to a Cup as member data. This only requires a forward declare because all pointers are primitive types, and therefore the compiler knows their size, which is necessary to calculate the size of a single instance of a DiceGame.

These changes will leave the code in a compiling state and the tests still pass.

### **Initial Updating of Method Definitions**

To introduce this new Cup into the DiceGame eco system, we will update a few of the existing methods to use it as well as the original vector.

First, the construction and destruction:

```

01: #include "Cup.h"
02: DiceGame::DiceGame(Die *d1, Die *d2) : cup(0), balance(0) {
03:     theDice.push_back(d1);
04:     theDice.push_back(d2);
05: }
06:
07: DiceGame::~~DiceGame() {
08:     for(iterator
09:         iter = theDice.begin(); iter != theDice.end(); ++iter)
10:         delete *iter;
11:     delete cup;

```

Line	Description
------	-------------

Line	Description
01	The code will be using Cup, so it must include Cup.h. Remember, Cup was forward declared in the header file. The source file includes it because it needs to know more about the class than the header file.
02	Initialize the cup pointer to 0. This is how you initialize pointers in C++, use 0; do not use NULL.
10	Delete the pointer held in the member data. Right now this pointer is 0. By definition, it is safe to delete a pointer initialized to 0, so this is safe.

This change keeps the code compiling and running. However, there's a bit of an issue going any further. The destructor in `DiceGame` calls `delete` on the pointers passed into the constructor. If we add those same `Die` pointers to the `Cup`, which also calls `delete`, then the same address will be deleted twice. This is forbidden, or rather undefined.

Here's a quick experiment to convince yourself that this is a bad idea:

```
01: DiceGame::DiceGame(Die *d1, Die *d2)
02:     : cup(new Cup(d1, d2)), balance(0) {
03:     theDice.push_back(d1);
04:     theDice.push_back(d2);
05: }
```

On line 5, construct a new `Cup`, passing in `d1` and `d2`. This will cause some kind of segmentation violation or some other undefined behavior. It will fail in some way that will vary by your platform and by what you have installed. In my particular situation I am using Eclipse and I have Visual Studio installed, so I get a prompt to bring up the Visual Studio debugger for this violation. On an XP installation without Visual Studio installed, I'll get a dialog saying that a program terminated and I'll be given the option of mailing a report somewhere.

However, this does suggest an unorthodox intermediate form. Simply update the constructor as shown and then remove the code that releases memory in the destructor:

```
DiceGame::~~DiceGame() {
    delete cup;
}
```

The `Cup` now owns that memory and its destructor releases it.

Make these two changes and your code should be green.

At this point, if you update the `DiceGame::play` method, then the original solution, the `std::vector`, won't really be partaking in the solution:

```
void DiceGame::play() {
    cup->roll();
    int sum = cup->total();

    if (sum > 7)
        ++balance;
    if (sum < 7)
        --balance;
}
```

Simply roll the cup and then get its total. Making this change leaves bits of the previous solution:

- In the header file
  - include of <vector>
  - two typedefs
  - member data called theDice
- In the source file
  - include of Die.h
  - call to `push_back` in the constructor

Removing these bits results in the following `DiceGame` header:

```
#pragma once
#ifndef DICEGAME_H_
#define DICEGAME_H_

class Cup;
class Die;

class DiceGame {
public:
    DiceGame(Die *d1, Die *d2);
    virtual ~DiceGame();

    void play();
    int getBalance() const;

private:
    Cup *cup;
    int balance;

private:
    DiceGame(const DiceGame&);
    DiceGame& operator=(const DiceGame&);
};

#endif
```

And the final `DiceGame` source file:

```
#include "DiceGame.h"
#include "Cup.h"

DiceGame::DiceGame(Die *d1, Die *d2)
    : cup(new Cup(d1, d2)), balance(0) {
}

DiceGame::~DiceGame() {
    delete cup;
}
```

```

void DiceGame::play() {
    cup->roll();
    int sum = cup->total();

    if (sum > 7)
        ++balance;
    if (sum < 7)
        --balance;
}

int DiceGame::getBalance() const {
    return balance;
}

```

Notice that these files are quite a bit smaller, which makes sense since much of the work was dealing with the details of an `std::vector<Die*>`, which has been delegated to the Cup class.

Make these changes; get your solution back to green.

### 3.16.7 What of the idiom?

Remember the copy constructor and assignment operator? Should you hide these two methods? In fact, if you do not hide them, you could end up with serious problems. This code would cause the same `Die` object to get deleted multiple times:

```

01: Die *d1 = new Die;
02: Die *d2 = new Die;
03: Cup c1(d1, d2);
04: Cup c2(c1);
05: Cup c3 = c1;
06: Cup c4(0, 0);
07: c4 = c1;

```

Line	Description
01	Create a die, nothing new here.
02	Ibid.
03	Create a cup called c1, it holds onto (and therefore owns) d1 and d2.
04	Create a cup called c2, using the compiler-provided copy constructor. c2 will hold a copy of d1 and d2 and it will think that it owns d1 and d2. Now there are two objects, c1 and c2 that own d1 and d2. The second cup will get destroyed first, and then the first cup will get destroyed. When that happens you'll see some kind of serious program failure.
05	Adding insult to injury, construct another cup called c3 using the copy constructor. This line is equivalent to line 04, just a different syntax. You can distinguish this from assignment (line 07) because this line involves a variable definition. Oh, now three objects think they own d1 and d2.
06	Create a cup called c4, it initially points to two non-die objects. This will work, but don't roll the cup, or the program fails.



Line	Description
07	Assign c1 to c4. This uses the compiler-provided assignment operator. Now c4 will think it owns d1 and d2. So there are 4 objects, all taking responsibility for releasing the same memory. This is a crash waiting to happen.

How can you fix this? First, make the assignment operator and copy constructor private:

```
private:
    Cup(const Cup&);
    Cup& operator=(const Cup&);
```

Another thing you can do is coming up; use an `std::shared_ptr`. Even so, unless you have a compelling reason to do otherwise, make these methods private. If you do so, then lines 04, 05 and 07 will not compile.

### 3.16.8 A Logical Fix to Cup

Remember `faceValue` on `Die`? It is constant. The `Cup`'s analog, `total`, is not the same. It could do with a change:

```
int total();
```

This method does not change the `Cup`, so it should be `const`:

```
int total() const;
```

Of course, to make this change, you will have to change the method definition:

```
int Cup::total() const {
    ...
}
```

If you try just this, your code will not compile and the error probably won't give you much of a clue what's wrong with your code. Here's the compilation error from my machine (yours may vary):

```
..\Cup.cpp: In member function 'int Cup::total() const':
..\Cup.cpp:22: error: conversion from
'__gnu_cxx::__normal_iterator<Die* const*, std::vector<Die*,
std::allocator<Die*> > >' to non-scalar type
'__gnu_cxx::__normal_iterator<Die**, std::vector<Die*,
std::allocator<Die*> > >' requested
```

Containers have a nested typedef for iterator. This defines a type you use when iterating over the contents of a collection. However, adding `const` to the method makes the current object `const`. When this happens, the collection is considered `const`. The iterator is for non-`const` collections. However, the standard collections have a second nested typedef, `const_iterator`. So you can make the following changes to get this back to compiling and working:

```
01: class Cup {
02: public:
03:     typedef std::vector<Die*> DiceCollection;
04:     typedef DiceCollection::iterator iterator;
05:     typedef DiceCollection::const_iterator const_iterator;
```

Line	Description
03, 04	The original nested typedefs.
05	A new nested typedef for a <code>const_iterator</code> . This is the thing you'll use in any methods denoted as <code>const</code> .

The `total()` method definition changes to use `const_iterator`:

```
01: int Cup::total() const {
02:     int sum = 0;
03:     for(const_iterator i = dice.begin(); i != dice.end(); ++i)
04:         sum += (*i)->faceValue();
05:     return sum;
06: }
```

Line	Description
03	Switch from <code>iterator</code> to <code>const_iterator</code> , and viola the code compiles and passes.

### 3.17 What is going on with `const`?

The `const` keyword says something is `const`. What exactly is `const` depends on where the `const` is located. Here are the typical examples you'll see under a discussion of `const`:

Code	Description
<code>int *pi = 0;</code>	Just a pointer to an integer.
<code>const int *pci = 0;</code> <code>int const *pci = 0;</code>	A pointer to an integer constant. That is, the pointer can point anywhere, and where it points to can change, but you cannot change the underlying integer to which it points. These are equivalent definitions.
<code>int *const cpi = 0;</code>	A constant pointer to an integer. That is, this pointer will always point to the same place in memory (0 in this case). The value pointed to can be changed (if it didn't point to 0 that is).
<code>const int * const cpic = 0;</code> <code>int const * const cpic = 0;</code>	A constant pointer to a constant integer. Nothing can change, neither the value pointed to nor the address in memory. These are equivalent definitions.

This is a typical introduction to this subject, but it's very hard to figure out how to remember all of these details. I learned a secret reading something by Andrew Koenig called the right-left rule. The basics to reading declarations and definitions goes something like this:

- Find the name – that's the one potential hard part.

- Once you have the name, read right until either:
  - You've reached the end of the statement, a ; – or an = works the same way
  - You have one more open parentheses ) than close parentheses (
- Now read to the left until either:
  - If you just found a ; going right, read all the way to the left
  - If you just found an unmatched ) going right, read to the matching (.

Here are those definitions read using this rule:

Code	Description
<code>int *pi = 0;</code>	The name is pi. Going to the right there's an =, so read all the way to the left, which gives pi is a pointer to an int.
<code>const int *pci = 0;</code> <code>int const *pci = 0;</code>	The name is pci. Going to the right, there's an =, so read all the way to the left, which gives either: <ul style="list-style-type: none"> <li>▪ pointer to an integer constant</li> <li>▪ pointer to a constant integer</li> </ul>
<code>int *const cpi = 0;</code>	The name is cpi. Going to the right is an =, so now reading to the left, cpi is a constant pointer to an integer.
<code>const int * const cpic = 0;</code> <code>int const * const cpic = 0;</code>	The name is cpic. Going to the right is an =, so now reading back to the left: <ul style="list-style-type: none"> <li>▪ constant pointer to an int that is constant</li> <li>▪ constant pointer to a constant int</li> </ul>

Those are relatively simple examples. Here's something that is a bit more complex:

```
18: int getBalance() const;
```

The name is getBalance(), what of the rest?

- Going to the right is (, the rules don't say anything about that, but it means that getBalance appears to be a function.
- Continuing right, there's a ). However, that matches the (, so getBalance is a function that takes no arguments. You don't start reading left because the () are balanced, there's not an extra close parentheses.
- Continuing right, is const. So this is a method on something that is const – the object, it turns out.
- Continuing right, is the ;, so finish to the left;
- Going left is int, which is the return type of the method.

So getBalance is a method taking no parameters, which is const and it returns an int. So what does the const mean?

### **The current Object**

How does any method know “the current object”? For example:

```
Die d1;
Die d2;
d1.roll();
d2.roll();
```

The roll method sets the value of a die, but there's only one roll method and in this example there are two `Die` objects. How does this work?

Here's a key to thinking about C++, everything in C++ can be, and often is, represented in C. That is, everything you write in C++ can be translated to structs, functions and pointers.

The member function `Die::roll()` appears to take no arguments, but in fact it takes one argument, the current object. Logically there is a method whose name includes both roll and the name of the class. Let's call this method `Die__roll`. This method takes one argument, a pointer to a die. That pointer is called "this", so here's a more complete function declaration<sup>16</sup>:

```
void Die__roll(Die *this);
```

This is not quite correct, because in a method, it is not possible to change to which object the so-called this pointer points to, so it should be a constant pointer:

```
void Die__roll(Die *const this);
```

Using the right-left rule to read this: "this" is a constant pointer to a `Die`.

In the body of the roll method is the code assigns to value:

```
void Die::roll() {
    ...
    value = uniform(engine);
}
```

This is equivalent to the following<sup>17</sup>:

```
void Die::roll() {
    ...
    this->value = uniform(engine);
}
```

So logically (and in some cases nearly literally), the compiler converts the simple example above to:

```
Die d1;
Die d2;
Die__roll(&d1);
Die__roll(&d2);
```

So now it is hopefully clear how methods know the current object. It's passed in as a hidden parameter. In fact, that's what makes a method a method, a hidden first parameter.

### **So, again, what does a const method do?**

Consider the method `faceValue`:

```
int Die::faceValue() const {
```

<sup>16</sup> Note that this is close to valid C++. There are 2 problems. First, this is a reserved word, so you cannot name a variable this. Second, the name of the method includes two `_`. While technically valid, the compiler and preprocessor reserve names with two or more `_` (typically at the beginning of a name). So in practice I would not put two consecutive underscores in a name.

<sup>17</sup> In fact, this is literally equivalent as this form will compile and work just fine.

```
    return value;
}
```

What's constant in this? The current object, or the `this` pointer. Here's a logical rewrite:

```
int Die__faceValue(const Die *const this) {
    return value;
}
```

So `this` is a constant pointer (which is always is) to a `Die` that is itself constant (which it normally is not). What this does is prevents changes to the current object. It's still possible to make changes. The object is not physically constant, but you have to do something special in modern compilers to remove the `const` characteristic<sup>18</sup>.

### 3.18 Taking Small Steps, Recap

This started off as an exercise to reduce the complexity in `DiceGame` by refactoring. Specifically, you extracted a class out of `DiceGame` and called in `Cup`. In general, refactoring starts with creating new code rather than changing existing code. You can certainly change code in place, but that is a bit riskier. Taking small steps and keeping your code green leaves you in a better state to be interrupted. Since a modern work day involves interruption, this style of code modification can make your daily live a touch easier.

The steps for extracting class are:

- Create a new class by copying existing code
  - Create a standard header file
  - Create the source file
  - Get to compiling
- Add the `Cup` into the `DiceGame` using parallel development
  - Get as much changed in `DiceGame` as you can manage while keeping the solution green
  - Finally make the plunge
  - Clean up after the change
- Quick review
  - We made the copy constructor and assignment operator private
  - We made `total()` `const`

In general, refactoring starts with creation rather than making inline changes. You do not have to do this, but if you change code first and get to a non-compiling state, when you get interrupted, and you will get interrupted, you might lose your train of thought. If you leave the code compiling and the tests passing most of the time, getting interrupted means you still have a working system with some possibly unnecessary code.

Any reasonably mature code base is inconsistent, contains unnecessary code and is in a constant state of decay. That's the reality of any successful system. So if you happen to get interrupted but tests are still passing, then you've simply added a little white noise until you get back to the work.

---

<sup>18</sup> You can use `const_cast<Die*>(this)` to remove the `const`-ness. That's the last you'll hear of it in this book as it is out of scope.

### 3.19 Memory Allocation

How can clients of `DiceGame`, or `Cup` for that matter, know that these classes expect a dynamically allocated pointer? Consider the constructors for `DiceGame`:

```
class DiceGame {
public:
    DiceGame(Die *d1, Die *d2);
```

Either of these examples successfully creates a `DiceGame`:

```
Die d1;                               Die *d1 = new Die;
Die d2;                               Die *d2 = new Die;
DiceGame troubleAhead(&d1, &d2);      DiceGame noWorries(d1, d2);
```

The one on the left will lead to problems at the time of destruction and there's no good way to with the current API to make that clear. You could use a nested typedef such as:

```
class DiceGame {
public:
    typedef Die* dyno_die;
    DiceGame(dyno_die d1, dyno_die d2);
```

This is better because it documents an expectation and it does so in a way that must compile. Can we do better? Can we also clean up the implementation and take care of the allocation a bit more automatically?

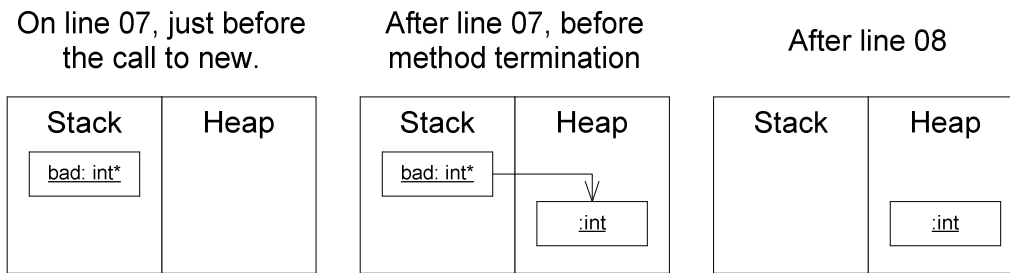
#### 3.19.1 `std::shared_ptr`

There are classes in the standard library that help with allocation. We'll have a look at one such class, though there are several. As with the rest of the book, let's start with a failing test:

##### *MemoryLeakDemonstration.cpp*

```
01: #include <CppUTest/TestHarness.h>
02:
03: TEST_GROUP(MemoryLeakDemonstration) {
04: };
05:
06: TEST(MemoryLeakDemonstration, ThisLeaks) {
07:     int *bad(new int);
08: }
```

This allocates memory on the heap on line 07. There's no other code that releases the memory, and in fact without doing something with writing a custom memory allocation solution, there no way to recover after the test terminates. This is what memory logically looks like over time:



The problem is that the variable is a raw type and raw types don't have constructors. What we need to do is convert the raw type into a non-raw type so that the pointer will have delete called automatically when the method terminates. Here's a similar test, but this one passes:

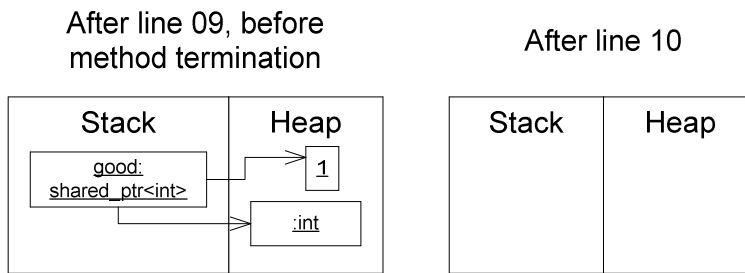
```

01: #include <memory>
02:
03: #include <CppUTest/TestHarness.h>
04:
05: TEST_GROUP(MemoryLeakDemonstration) {
06: };
07:
08: TEST(MemoryLeakDemonstration, ThisWorks) {
09:     std::shared_ptr<int> good(new int);
10: }

```

Line	Description
01	Include the header file for <code>std::shared_ptr&lt;&gt;</code> .
09	Create a shared pointer instead of a raw pointer. Since this is a non-raw type, it has a destructor, which will get called when <code>good</code> goes out of scope.
09	Notice that the template parameter type is <code>&lt;int&gt;</code> and not <code>&lt;int*&gt;</code> . The class is designed to work with pointers to dynamically allocated memory. So a <code>std::shared_ptr&lt;int&gt;</code> is a shared pointer to an integer. The name of the class suggests it is a pointer already. The second example, <code>std::shared_ptr&lt;int*&gt;</code> is valid, but it is a shared pointer to a pointer to an <code>int</code> . It's like an <code>int**</code> .

The `std::shared_ptr` class maintains a count of the number of times the pointer is shared. The deallocation only happens if the count goes to zero. Here's what memory logically looks like over time:



### 3.19.2 Fixing DiceGame

With this in mind, now you can fix `DiceGame`. Here's the updated header file:

```
01: #include <memory>
02:
03: class DiceGame {
04:     ...
05:
06: private:
07:     std::shared_ptr<Cup> cup;
08:     ...
09: };
```

Line	Description
01	Include the header file for <code>std::shared_ptr&lt;&gt;</code> .
07	Change the <code>cup</code> member data to be a shared pointer instead of a raw pointer. This will make calling <code>delete</code> unnecessary; it won't compile either.

The source file needs one change for this to compile and work:

```
01: DiceGame::~DiceGame() {
02: }
```

Delete the body of the destructor. The `std::shared_ptr` takes care of memory deallocation, so you can (and must) remove it.

The design of the class is meant to make it look just like a pointer.

Notice the `play` method is unchanged:

```
void DiceGame::play() {
    cup->roll();
    int sum = cup->total();
    ...
}
```

How does this work?



### Operator Overloading

You've already seen examples of operator overloading, specifically the assignment operator. It turns out that another operator you can overload is `->`. If you were to create such an operator for one of your classes, say `Die`, it could look like this:

```
Die* operator->() { return this; }
```

Then the following code, while redundant, would work:

```
Die d1;
d1->roll();
```

Don't do this. I'm just demonstrating what this might look like. Here's what the same operator might look like for the `std::shared_ptr` class:

```
01: namespace std {
02:     template<class P> class shared_ptr {
03:     public:
04:         shared_ptr(P *v) : target(v) {}
05:         P* operator->() { return target; }
06:     private:
07:         P* target;
08:     };
09: }
10:
11: exp::shared_ptr<int> example(new int);
```

Line	Description
01	Everything from this line to the matching close <code>}</code> is in the namespace <code>std</code> .
02	Define a template class with one parameter <sup>19</sup> . The class is called <code>shared_ptr</code> .
04	Declare and immediately define a constructor. This kind of method is called an implicit inline method. The combination of declaring and then immediately defining a method is what makes it implicit.
05	Declare and then immediately define a method called <code>operator-&gt;</code> . The method returns the <code>target</code> pointer stored in the constructor.
11	Create an instance of the <code>shared_ptr</code> .

This is in no way near a full implementation of the class. For that you should have a look at the `<memory>` header file.

#### How can C++ tell the difference?

How can the compiler decide between a "regular" `->` and one you have declared for your class? You cannot change the definition of the language. This means that while you can write your own operators for your class, you cannot write your own operators for primitive types. All pointers are primitive types, therefore you cannot change the behavior of `->` on a pointer. For example:

<sup>19</sup> The actual class has more than one template argument; this is logically showing what's necessary to get a working `operator->` method added to a class like the `shared_ptr` class.

```
Die d1;
d1->roll();
```

There's only two things that can happen here:

- If your `Die` class has declared an `operator->` method, it will compile.
- If your `Die` class does not declare an `operator->` method, this will not compile.

Contrast that with this example:

```
Die *d1 = new Die;
d1->roll();
delete d1;
```

There's only one thing that happens here. The type of `d1` is pointer to `Die`. All pointers are primitive types, so this uses the built-in version of `operator->`. There is no ambiguity in what happens with either of these examples.

### 3.19.3 Fixing Cup

#### *The Header File*

Now it's time to get a little more creative and fix the `Cup` class. Since `Cup` holds a vector of `Die` objects, we're going to have to do a bit more work:

```
01: class Die;
02: #include <vector>
03: #include <memory>
04:
05: class Cup {
06: public:
07:     typedef std::shared_ptr<Die> spDie;
08:     typedef std::vector<spDie> DiceCollection;
```

Line	Description
03	Include the necessary header file for the shared pointer.
07	Unlike the last example, define a nested typedef because it will make the changes to the source code easier.
08	Update the typedef to use <code>spDie</code> instead of <code>Die*</code> . Note the use of typedefs minimize the changes in the header file to one line.

#### *The Source File*

The source changes in terms of construction and destruction only:

```
01: Cup::Cup(Die *d1, Die *d2) {
02:     dice.push_back(spDie(d1));
03:     dice.push_back(spDie(d2));
04: }
05:
06: Cup::~~Cup() {
07: }
```

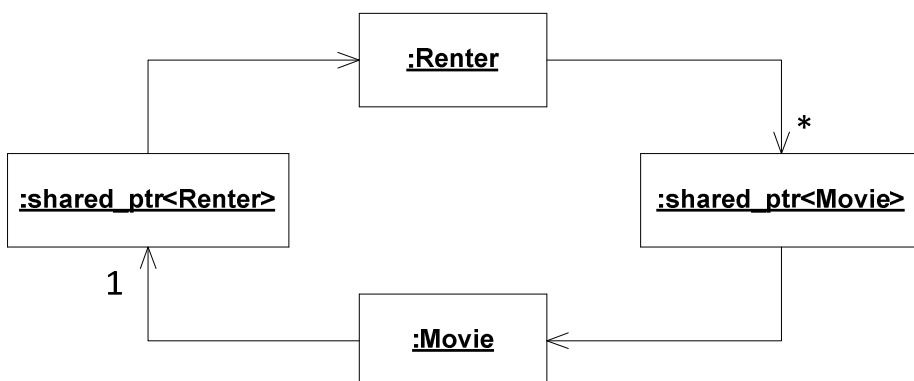
Line	Description
02, 03	Instead of putting raw <code>Die</code> pointers into the <code>std::vector</code> , put <code>spDie</code> . Why you have to do this has to do with the design of the <code>shared_ptr</code> class. The <code>push_back</code> method on <code>vector</code> is expecting an <code>spDie</code> but you have a <code>Die*</code> . There is a constructor in <code>shared_ptr</code> that would work, but the design of <code>shared_ptr</code> makes the implicit use of that forbidden (by use of the <code>explicit</code> keyword). So you must explicitly create an instance of <code>std::shared_ptr</code> , for which we have already created a nested typedef. This is by design because accidentally creating a second shared pointer on something that already has a shared pointer will cause the program to crash when memory is either released multiple times or if memory that was released is then read.
06	All memory is now taken care of by <code>std::shared_ptr</code> , simply remove the body of the destructor.

### 3.20 Warning: Circular References

While `std::shared_ptr` solves the basic problem of releasing memory, as with all solutions, it comes with problems. Specifically, circular references will not get removed.

#### 3.20.1 The Problem: A concrete example

Here is a simple domain using shared pointers:



Here's a version of this diagram in code (`CircularReferenceExample.cpp`):

```

01: #include <memory>
02:
03: struct Movie;
04: struct Renter {
05:     std::shared_ptr<Movie> movie;
06: };
07: struct Movie {
08:     std::shared_ptr<Renter> checkedOutBy;
09: };
10:
11: struct RentAMovieSystem {
12:     std::shared_ptr<Renter> createRenter() {
13:         return std::shared_ptr<Renter>(new Renter);
  
```

```

14:     }
15:
16:     void rentAnyMovieTo(std::shared_ptr<Renter> &renter) {
17:         Movie *someMovie = new Movie;
18:         renter->movie.reset(someMovie);
19:         someMovie->checkedOutBy = renter;
20:     }
21: };
22:
23: #include <CppUTest/TestHarness.h>
24:
25: TEST_GROUP(CircularReference) {
26: };
27:
28: TEST(CircularReference, Broken) {
29:     RentAMovieSystem system;
30:     std::shared_ptr<Renter> renter = system.createRenter();
31:     system.rentAnyMovieTo(renter);
32: }

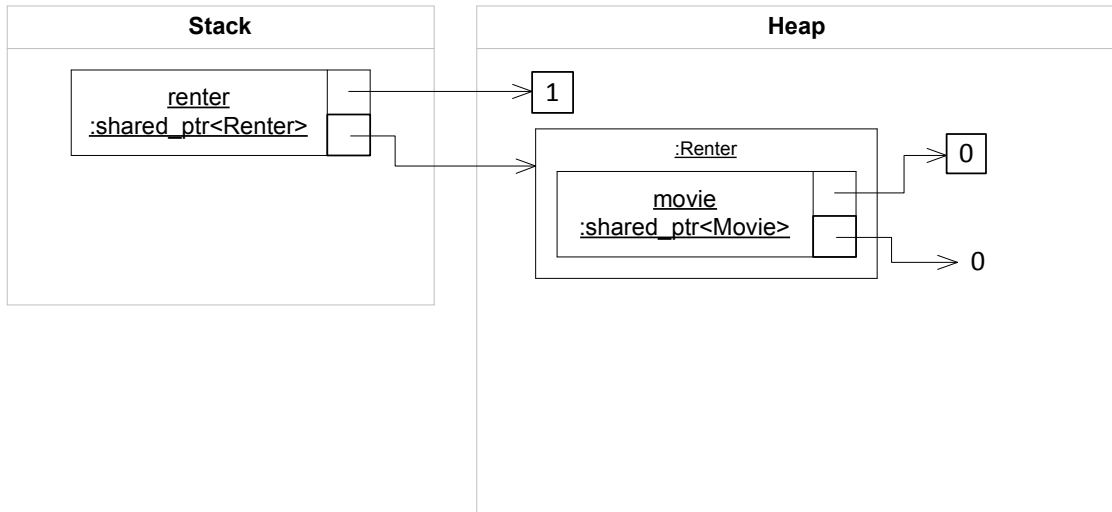
```

Line	Description
01	Include the required header file.
03	Forward-declare the Movie class, so it can be used on line 05.
04	For this example, use <code>struct</code> . Everything is <code>public</code> by default.
05	A Renter has a single movie. This is a simplified version of the diagram where a Renter can either 0 or 1 movies rented rather than 0 to many. Adding a collection here simply complicates the example without making it any more broken.
07	Observation, this single file has both a declaration of Movie and Definition. That's OK, as stated above, declarations can repeat, definitions cannot.
08	A movie knows who it was checked out by, which is 0 by default.
13	This is like a simulated lookup method. Just creating rather than looking up a Renter object.
16	This is where the magic happens
17	Create a new movie
18	Store the newly-created movie in the <code>std::shared_ptr</code> , replacing the current value in that object. It was built using its no-argument constructor, which sets its internal pointer to 0, and its count to 0.
19	Now set the movie's Renter <code>shared_ptr</code> back to the current Renter. This is the line that creates the circular reference. At this point, unless the circle is broken, neither object will be released from memory.
29	Create a rental system used in the rest of the test.
30	Ask the rental system to create a Renter. This is like a lookup, I just didn't want to have to simulate too much to demonstrate the principle.

Line	Description
31	Call the method that forms the circle. Now memory will not be deallocated.

What follows are some logic views of memory at critical points in the code.

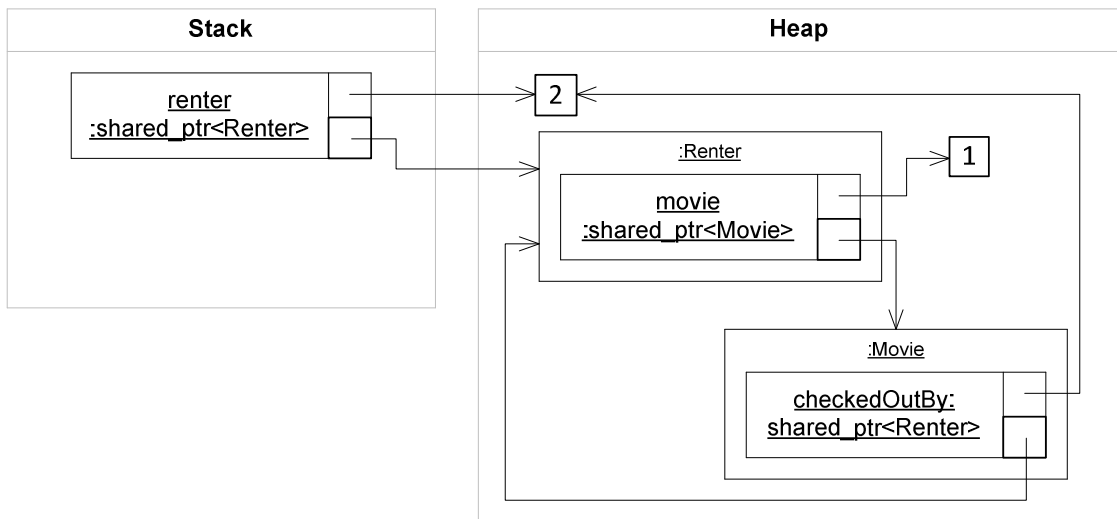
**After Line 30 Completes**



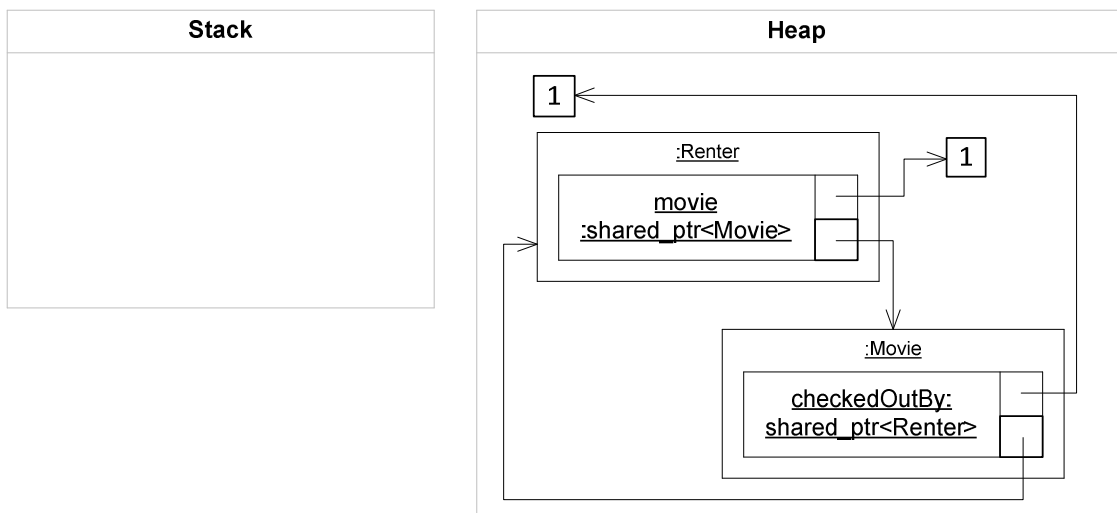
Line 30 requests the creation of a `Renter` by calling `createRenter`, which returns an `std::shared_ptr`. The method first creates a new `Renter`, which is on the heap. A `renter` holds an instance of an `std::shared_ptr<Movie>`. That object is fully contained within the `Renter` object, but that object itself holds a shared count, initially 0 and a pointer to a `Movie`, initially assigned to `0`<sup>20</sup>. The return `std::shared_ptr` object is then copied (via the copy constructor) into a local variable called `renter`, which is wholly on the stack, but it holds a shared counter, 1 after the line completes (but for a short time 2) and a pointer to dynamically allocated `Renter` object. Remember, these are logical views. The actual implementation of `std::shared_ptr` is different. This representation is analogous.

Next, the test calls `rentAnyMoveTo`. Internally this creates a new `Movie`, stores that pointer (thereby resetting the `Renter`'s `movie` `std::shared_ptr`) and then tells the `Movie` who the `renter` is checking it out.

<sup>20</sup> This is logical; an implementation won't allocate the shared count until necessary, typically.

**After rentAnyMovieTo Completes**

When all of this is done, the renter's shared pointer to a Movie has a count of 1 and it now refers to a newly-created Movie. That Movie is wholly on the heap. It physically contains an `std::shared_ptr<Renter>`, which has a shared count of 2 and refers back to the first (and only) renter object created.

**After The Test Finishes**

The test finishes and its program stack is cleared. The renter variable goes out of scope, which means its destructor is called. During destruction, it decrements the shared count from 2 to 1. While you may have expected the Renter on the heap to be released, its count is 1, so it is still around. It holds a reference to a Movie, with a count of 1. These objects are forever pegged in memory.

**3.20.2 Options**

There are several ways to fix this.

- Change your design to remove the circular reference
- Use a `std::weak_ptr`

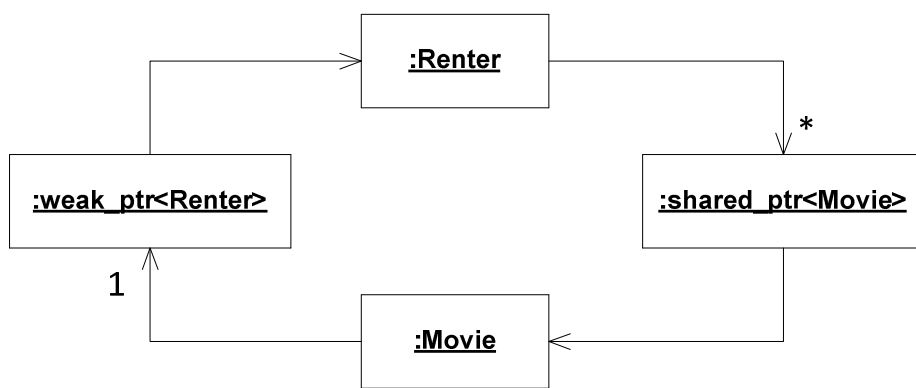
- Use a raw pointer

### **Changing your design**

Circular references are real problems. They are certainly a code smell, meaning they are worth reviewing. If it happens to be possible to simply break the circular reference then that is your best option. This may not be a viable option for any number of reasons, so what can you do if that is not an option?

### **Fixing With weak\_ptr**

One option is to use an `std::weak_ptr` on one side of the relationship. Make a decision about which side is the primary side. This is the side that tends to be the primary or most common path of access. If there's not a clear winner, then go back and review your design. In this example, assume that `Renter` is the primary side of the relationship:



One issue with this is that to create a `weak_ptr` requires the original `shared_ptr`. We have this handy, in the `rentAnyMovieTo` method, so here are a few changes to fix the circular reference:

```

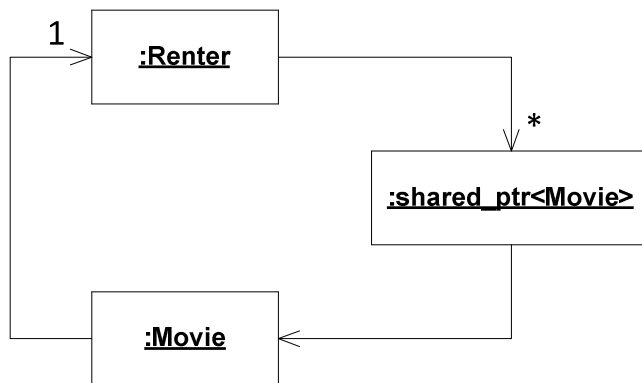
struct Movie {
    std::weak_ptr<Renter> checkedOutBy;
};
  
```

So long as you have the original `shared_ptr` available, simply changing from the type of `checkedOutBy` from a `shared_ptr` to a `weak_ptr` fixes the problem. However, this solution happens to have the correct `shared_ptr` around. If it did not, then you'd need to find it first. So using a `weak_ptr` can be easy if your program makes it easy.

If for some reason getting access to the original `shared_ptr` of `Renter` is difficult or impractical, then use a raw pointer.

### **Fixing with raw pointer**

The diagram is even easier with a raw pointer:



And the code is nearly the same:

```

struct Movie {
    Movie() : checkedOutBy(0) {}
    Renter *checkedOutBy;
};
  
```

And:

```

void rentAnyMovieTo(std::shared_ptr<Renter> &renter) {
    ...
    someMovie->checkedOutBy = renter.get();
}
  
```

Even though these two solutions are not too bad, removing circular references will make your code age better over time.

### 3.21 Recap

The standard library offers a shared pointer to automatically release memory at the right time. It does this by keeping a reference count. When the reference count goes to 0, the memory is released.

However, you need to be careful:

- Do not give an address to a shared pointer that was not created using `new` (not `malloc`, or `address-of`, just `new`)
- Do not give the same address to two different shared pointers you create yourself. They will copy correctly if passed around, but do not create two different shared pointers that both point to the same address.
- Do not have circular references between shared pointers. Either remove the circularity or break the shared pointer chain with a weak pointer or a raw pointer.

If you use shared pointers and follow these rules, you can remove quite a bit of manual memory releasing and you're more likely to have a well behaved system.

### 3.22 What's Coming Up?

Now we'll have a look at refactoring by using built-in features of the standard library to improve the implementation of the `Cup` class. Specifically, we'll look at:

- `std::for_each`
- `std::accumulate`



- `std::bind` to call member functions
- Pointers to member functions
- `std::tr1::placeholders` and `_1`

This section introduces a few of the build-in algorithms in the standard library.

### 3.23 A Few Built-In Algorithms

#### 3.23.1 Updated `roll()`

Here is `Cup::roll()` in its current form:

```
void Cup::roll() {
    for(iterator i = dice.begin(); i != dice.end(); ++i)
        (*i)->roll();
}
```

There is a method in the standard library called `for_each` that accomplishes something like this. It takes two iterators, `dice.begin()` & `dice.end()` and a function or function object that should be applied to each element in the collection.

Here is an example of just that:

```
01: #include <algorithm>
02:
03: static void rollADie(Cup::spDie &die) {
04:     die->roll();
05: }
06:
07: void Cup::roll() {
08:     std::for_each(dice.begin(), dice.end(), rollADie);
09: }
```

Line	Description
01	Include the header file that contains, among other things, <code>std::for_each</code>
03 – 05	Define a function that will be used by <code>std::for_each</code> on each element in its collection. This method is declared static, meaning it is only available in this compilation unit. It takes a reference to an element of the vector containing <code>shared_ptr</code> to <code>Die</code> . Luckily, there's already a nested typedef, so use it.  This is a place where you can use a reference to avoid making an unnecessary copy of an <code>std::shared_ptr</code> . If you do not use a reference, then when <code>rollADie</code> is called from the <code>for_each</code> method, it will pass a copy. This will increase the reference count and copy the pointer. Then the function ultimately calls <code>roll</code> via a pointer to a <code>Die</code> and returns. Upon return, the copy of the <code>std::shared_ptr</code> is removed, its destructor is called, it decrements the reference count by one.  So while not using a reference will work because the thing being copied is a handle to a pointer to a <code>Die</code> object, it required unnecessary work.
04	Actually roll a die. Remember, <code>die</code> is an <code>std::shared_ptr</code> to <code>Die</code> . This uses the overloaded <code>operator-&gt;</code> method, which returns a <code>Die*</code> , to which the built-in <code>-&gt;</code> operator applies to the <code>Die*</code> and invokes the <code>roll</code> method.

Line	Description
08	Call a template function called <code>std::for_each</code> . This function starts at the first parameter ( <code>dice.begin()</code> ), an iterator, and continues while its internal copy of the iterator does not equal the second parameter ( <code>dice.end()</code> ). For each element in the collection, call the function <code>rollADie</code> .

**Make this change**

This change is relatively self-contained. Make this change and get to green.

**Removing the `rollADie` method**

The third parameter to `std::for_each` is either a pointer to a function or an object that has an `operator()` method defined on it. You have seen such a beast in the engine class<sup>21</sup>. Here is a hand-rolled version of that, which we'll migrate into the final form using more built-in standard library features.

Here is the same code, with the function written as a function object, or a functor:

```
01: #include <algorithm>
02:
03: struct RollFunctor {
04:     void operator()(Cup::spDie &die) {
05:         die->roll();
06:     }
07: };
08:
09: void Cup::roll() {
10:     std::for_each(dice.begin(), dice.end(), RollFunctor());
11: }
```

Line	Description
03	Define a struct called <code>RollFunctor</code> . It has a single method, <code>operator()</code> taking the same parameter type as the previous function. This example uses struct since I want everything to be public by default.
05	This is the same code as the previous function. It's just represented as a member function instead of a stand-alone function.
10	For the third parameter, do not pass in a pointer to a function, instead pass in an instance of the class <code>RollFunctor</code> . The expression <code>RollFunctor()</code> creates a temporary instance, which the compiler passes into the <code>std::for_each</code> method. This works because the <code>std::for_each</code> method requires something that can respond to <code>()</code> in its third parameter. A pointer to a function does, but so does an object with an overloaded <code>operator()</code> method. The input parameter type must match the type stored in the underlying vector.

Try this version. Make the change and confirm your solution is green

<sup>21</sup> Review the code in Updated Source on page 53. Specifically the discussion of the line including `uniform(engine)`.

### Removing even the functor

The boost library introduced the bind library to make this kind of thing a bit more automatic. The standard library had some built-in support for this that was seldom used. Those classes still exist, but tr1 recommended the inclusion of the boost::bind functionality, which is a bit clearer, having the advantage of being second instead of first.

Here's the same thing using bind:

```
01: #include <algorithm>
02: #include <tr1/functional>
03: using namespace std::tr1;
04: using namespace std::tr1::placeholders;
05:
06: void Cup::roll() {
07:     std::for_each(
08:         dice.begin(), dice.end(), bind(&Die::roll, _1));
09: }
```

Line	Description
02	This is the header file that provides std::tr1::bind and std::tr1::placeholders.
07	Use the bind function to build a function object that calls the method Die::roll. _1 represents the parameter passed from the std::for_each function into the operator() method.

Give this a try, verify that it also works. Make sure your solution is green before trying the final version.

### Using lambdas

All of this leads up to something that many compilers do not yet support, and even the ones that do support it are not quite “there” yet. Even so, this gives you an idea of what this will look like with modern compilers maybe by 2012<sup>22</sup>:

```
01: void Cup::roll() {
02:     std::for_each(
03:         dice.begin(),
04:         dice.end(),
05:         [](spDie &die){ die->roll(); });
06: }
```

Line	Description
05	Pass in a lambda expression. There are three parts to this expression (there can be more): <ul style="list-style-type: none"> <li>▪ [] – this is a way to refer to local variables; there's no need in this example use, so it is empty. It's required.</li> <li>▪ (spDie &amp;die) – this is the signature of the method, like its predecessors, it</li> </ul>

<sup>22</sup> This code requires gcc 4.5 or later or VS 2010. So it might not work with your configuration.

Line	Description
	takes a reference to an spDie. <ul style="list-style-type: none"> <li>▪ { ... } – the body of the code, which has been constant throughout</li> </ul>

This is one place where you might not be able to make this example work in your system. If you are using gcc 4.5 or later, it will work<sup>23</sup>.

### 3.23.2 Updated total()

The total method can also be improved a bit. Like roll(), there's an appropriate template function for accumulating (summing values):

```
01: #include <numeric>
02: int sumIt(int currentSum, const Cup::spDie &die) {
03:     return currentSum + die->faceValue();
04: }
05:
06: int Cup::total() const {
07:     return std::accumulate(dice.begin(), dice.end(), 0, sumIt);
08: }
```

Line	Description
01	Include the header file that includes, among other template functions, std::accumulate.
02	The accumulate method takes a function with two parameters. The first parameter is the kind of value we are accumulating into, an int. The second parameter is an element type of the underlying collection type, a vector. Since this method is called from total, which is a const method, the value type is actually a const Cup::spDie.
03	Take whatever the current total is (it starts at 0 as we'll see), add to that value the current die's faceValue and return that value. The value returned is passed in the next time through the loop.
07	Use std::accumulate to sum up the faceValue's of all of the die objects. This method takes 4 parameters: <ul style="list-style-type: none"> <li>▪ The beginning of the collection</li> <li>▪ The end of the collection</li> <li>▪ A seed value, sums start at 0, so 0 is the seed value.</li> <li>▪ A function taking two parameters, as described above.</li> </ul>

For the first element in the collection, std::accumulate calls sumIt with two parameters:

- 0 – the seed value
- dice[0]

The sumIt method returns - + dice[0]->faceValue. Let's say for argument the first die has a face value of 6 and the second has a face value of 3. At the second element in the

<sup>23</sup> Most of this book is written using gcc 4.4, but for this example I used 4.5. I stick to 4.4 throughout the book because it represents a good middle ground in what you can expect in modern compilers.

collection, `std::accumulate` passes in the value returned from the first time through the loop and `dice[1]`:

- 6 – the value of the seed and `dice[0]->faceValue`
- `dice[1]`

The result of that addition is 9, which is returned by `sumIt`. Since this is the last time through the loop (there are only 2 `spDie` in the vector), the current sum, 9, is returned. That value is then immediately returned from `Cup::total()`.

### **The bind version**

The version of this using `bind` is nearly universally shunned. So I'll provide it for your review and then point you to a discussion of how to derive at it:

```
int Cup::total() const {
    return std::accumulate(
        dice.begin(),
        dice.end(),
        0,
        std::tr1::bind(
            std::plus<int>(),
            _1,
            bind(&Die::faceValue, _2)
        )
    );
}
```

For a detailed description of this, please review:

<http://schuchert.wikispaces.com/cppttraining.SummingAVector>

### **The lambda version<sup>24</sup>**

The lambda version fares much better:

```
01: int Dice::total() const {
02:     return std::accumulate(
03:         dice.begin(),
04:         dice.end(),
05:         0,
06:         [](int v, const spDie &d){ return v + d->faceValue(); }
07:     );
08: }
```

In this case, the lambda takes two parameters, just as the original `sumIt` function. The body of the code is the same. This may seem a bit obscure. Here is a version similar to the original method using `std::for_each` instead:

```
01: int Dice::total() const {
02:     int sum = 0;
03:     std::for_each(
```

<sup>24</sup> As with the previous lambda version, this example requires gcc 4.5 or later or Visual Studio 2010. It might work with other compilers as well, your mileage may vary.

```
04:     dice.begin(),
05:     dice.end(),
06:     [&sum](const spDie &die){ sum += die->faceValue(); }
07: );
08: return sum;
09: }
```

This version may seem more familiar. The only new syntax is [&sum], which makes the sum local variable by reference into the lambda. This makes the lambda a so-called block closure because access data in its containing scope.

### 3.23.3 Recap

This section introduces two template functions from the standard library. Both of these functions operate over some range of elements in a collection. You define the range by passing in the starting element and the ending element. The range is closed on the left side and open on the right side, meaning it includes the first element, but it excludes the right element. It goes just up to but does not include the end element.

The first template method is `std::for_each`. Use it to iterate over a collection and do something to or with each element. We looked at several forms:

- In the first form, you provided the name of a function (`rollADie`) that the template function calls on each element in the collection.
- In the second form, you accomplished the same thing using a so-called function object, or functor, which is a class that declares an `operator()` method.
- The third form uses the `std::bind` method to call a member function directly. This was taken from the boost library. The standard implementation is a bit more long-winded than the original boost version, but it's certainly usable for simple examples.
- The final form uses a lambda expression, which is new to C++0x. You may not have access to a compiler that supports this syntax, so treat this as what's ahead in the near future for C++.

In all cases, the final parameter was:

- Something that can respond to `()`, either a function or an instance of a class with an `operator()` method.
- The parameter passed into either the function or functor you provided is a reference to the element type in the vector.

Next, you worked with `std::accumulate`. This template method takes an additional parameter, a seed, to start the accumulation. This additional parameter means that the function passed in (or function object) needs to take two parameters, the seed value, and a reference to the element type in the vector.

For `std::accumulate`, you saw three forms:

- The first form took a function called `sumIt`, which calculates the current total and returns it.
- The second form went directly to the `bind` version. Like `std::for_each`, you can hand-write a function object taking the same signature as the `sumIt` function.
- The third form uses a lambda expression.

There are many more functions in the standard library; a good place to start is Effective STL by Scott Meyers.

### 3.24 Improved Test Writing?

Review the `DiceGame`'s constructor, here's what you have right now:

```
DiceGame(Die *d1, Die *d2);
```

Looking at this constructor, there is no way to tell that those die objects must be dynamically allocated. You have tests that demonstrate that. This one signature exposes a few issues with the design of the class:

- Strongly suggests the number of die objects used, which is a weak DRY violation because of the implied rules for win, lose, push.
- There's no clear way to guarantee that the actual arguments are dynamically allocated, exposing the class to a failure at destruction time. This fails another design principle: fail fast – it doesn't fail until long after the defect is introduced.

How can you improve upon this?

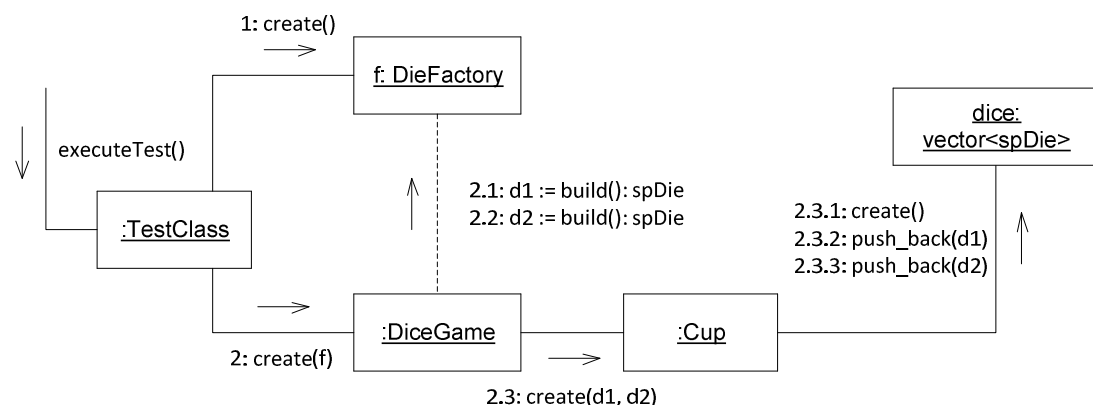
One way to improve at least part of this is to simply change the signature of the constructor to take shared pointers:

```
DiceGame(std::shared_ptr<Die> d1, std::shared_ptr<Die> d2);
```

This addresses the ambiguity of memory allocation as that the `std::shared_ptr` class is not only well defined, but specified as part of a standard. Rather than take that approach, however, we'll instead use a factory.

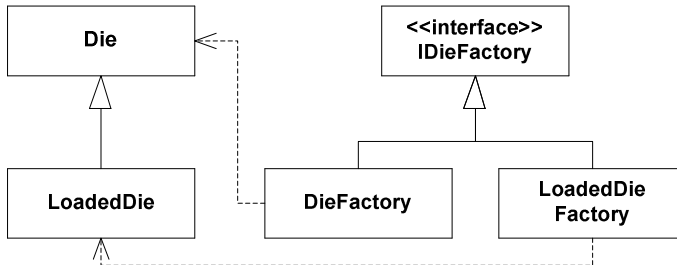
#### 3.24.1 Pass a factory into DiceGame

For better or worse, the implementation requires dynamic allocation. Rather than force that requirement out to the calling class, we can instead force the client to provide an object that does that work instead. Of course, the current design supports testability in the form of providing test doubles, so we don't want to lose that ability in a redesign. Here is one such design using an object called a factory:



### 3.24.2 Oh Wait, testability

The existing solution allows `LoadedDie` or `Die` objects during construction. To mimic this, we either need to construct both kinds of `Die` Objects in a single factory, or, better yet, have two different kinds of factories:



`Die` is a base class, `LoadedDie` is a derived class. The `LoadedDie` class is really part of the test solution. When you created this relationship, it was an example of inheriting from a concrete class, which has its issues. The other option is to inherit from an abstract class, or a class that cannot be instantiated. Generally, inheriting from a concrete class is more fragile than inheriting from an abstract class. Why?

- Concrete things, by their very nature, have more details. So more things can change, increasing the likelihood of something breaking down the road.
- Abstract classes tend to be stable both because they have fewer details to change but also by their very nature. Since they are abstract, other classes depend on them. Since other classes depend on them, they are stable, a self-fulfilling prophecy. Since classes depend on them, they have a certain amount of inertia to resist change; changing them might break dependent classes. This is called the stable dependencies principles. Things are stable because they are depended upon.

As this diagram shows, there will be an interface called `IDieFactory`. It will serve as an API only. It contains the stereo type `<<interface>>` which says that this class will not have any concrete methods. How can we accomplish that? Also, this is going to require quite a bit of change. How can we manage to migrate to this new solution while still living in our production house?

Before we get into all of that, let's digress into why re refactor code as we do.

### 3.25 The 4-contact points of software development

The three laws of TDD are:

- Write no production code without a failing test
- Write just enough of a test to fail
- Write just enough production code to get the test to pass

This list doesn't include refactoring, which is typically an assumed activity. In fact, some people refer to these rules as "red, green, refactor". An even older version of this, from the Smalltalk community, is Red, Green, Blue. (Why Blue for refactor? I think someone was thinking RGB for a color space, luckily they didn't try to use CMYK or LAB!)



In this simple model, there two kinds of code: test & production. There are two kinds of activity: writing & refactoring. Interestingly, at one level it is all code. The thing that distinguishes both sets is intent.

The intent of a test is to demonstrate or maybe specify behavior. The intent of production code is to implement (hopefully) business-relevant functionality.

The intent of writing code is creation. The intent of refactoring code is to change (hopefully improve) its structure without changing its behavior (this is oversimplified but essentially correct).

If you mix those combinations you have the 4-limbs of development:

- Writing a test
- Writing production code
- Refactoring a test
- Refactoring production code

An important behavior to practice is doing only one of these at a time. That is, when you are writing tests, don't also write production code. Sure, you might use tools to stub out missing methods and classes, but the heart of what you are doing is writing a test. Finish that train of thought before focusing on writing production code.

On the other hand, if you are refactoring production code, do just that. Don't change tests at the same time; try to only do one refactoring at a time, etc.

### 3.25.1 Why?

First an analogy that almost always misses since most developers don't additionally rock climb.

When rock climbing, a good general bit of advice is to only move one contact point at a time. For this discussion, consider your two hands and two feet as your four contact points. Sure, you can use your face or knee, but neither are much fun. So just considering two hands and two feet, that suggests that if, for example, you move your right hand, then leave your left hand and both feet in place.

This gives you stability, a chance to easily recover by simply moving the most recent appendage back in place and, when the inevitable happens, another appendage slips, you have a better chance of not eating rock face. If you move more than one thing at a time, you are in more danger because you've taken a risky action and reduced the number of points of contact, or stability.

Will you sometimes move multiple appendages? Sure. But not as a habit. Sometimes you need to take risks. The rock face may not always offer up movement patterns that make applying this recommendation possible. Since you know the environment will occasionally work against you, you need to maintain some slack for the inevitable.

Practicing Test Driven Development is similar. If you change production code and tests at the same time, what happens if a test fails? What is wrong? The production code, the test, both, neither? An even more subtle problem is that tests pass but the test is fragile or heavily implementation-dependent. While not necessarily an immediate threat, it represents design debt that will eventually cause problems. (This also happens frequently

when tests are written after the production code as it's seductively easy to write tests that exercise code, expressing the production's code implementation but fundamentally hiding the intent.)

Notice, if you had only refactored code, then you know the problem is in one place. When you change both, the problem space actually goes from 1 to 3 (4 if you allow for neither). Furthermore, if you are changing both production and test code at the same time and you get to a point where you've entered a bottomless pit, you'll end up throwing away more work if you choose to restore from the repository.

Are there going to be times when you change both? Sure. Sometimes you may not see a clear path that gives you the option to do only one thing at a given time. Sometimes the tests and code will work against you. Often, you'll be working in a legacy code base where there are no tests. Given that the environment will occasionally (or frequently) work against you, you need to maintain some slack.

Essentially, be focused on a single goal at any given time: write a test. then get it to pass. clean up production code & keep the tests first unchanging and then passing.

I find that this is a hard thing both to learn and to apply. I frequently jump ahead of myself. Unfortunately I'm "lucky" enough when I do jump ahead that when I fail, I thoroughly fall flat on my face.

This approach is contextual (aren't they all?). Every time you start working on code, you'll be faced with these four possibilities. Each time you are, you need to figure out what is the most important thing in the moment, and do that one thing. Once you've taken care of the most important thing, you may have just promoted the second most important thing to first place. Even so, reassess. What is the most important thing now? Do that.

### 3.26 Create a concrete Factory

We have existing tests and the solution is green. So we'll start by writing a new test to create a concrete factory. Which one, the `DieFactory` or `LoadedDieFactory`? We have automated tests that require the creation of `LoadedDie`, so that's where we will start.

#### 3.26.1 First Test against the Factory

Here's a test that confirms we can build `LoadedDie` out of a factory. This uses several things you've already seen:

```
#include "LoadedDieFactory.h"
#include "Die.h"
#include <memory>
#include <CppUTest/TestHarness.h>

TEST_GROUP(LoadedDieFactoryShould) {
};

TEST(LoadedDieFactoryShould, ReturnLoadedDie) {
    LoadedDieFactory factory(5);
    spDie d = factory.build();
    LONGS_EQUAL(5, d->faceValue());
}
```

This shows the creation of a `LoadedDieFactory` that returns `LoadedDie` objects always rolling 5. This is a good start, what of its implementation?

### 3.26.2 Define the class: `LoadedDieFactory`

The header file has nothing you've not already seen in other examples at this point:

```
#pragma once
#ifndef LOADEDIEFACTORY_H_
#define LOADEDIEFACTORY_H_

class Die;
#include <memory>
typedef std::shared_ptr<Die> spDie;

class LoadedDieFactory {
public:
    LoadedDieFactory(int value);
    virtual ~LoadedDieFactory();
    spDie build();
private:
    int faceValue;
};

#endif
```

### 3.26.3 Define the methods: `LoadedDieFactory`

The implementation offers no surprises either:

```
#include "LoadedDieFactory.h"
#include "LoadedDie.h"

LoadedDieFactory::LoadedDieFactory(int value) :
    faceValue(value) {
}

LoadedDieFactory::~LoadedDieFactory() {
}

spDie LoadedDieFactory::build() {
    return spDie(new LoadedDie(faceValue));
}
```

Create the new test, header and source file. Get your solution to green.

## 3.27 Update the cup

The Cup needs to be created using shared pointer to die objects, so it's time for a new test.

### 3.27.1 The Test

Here's a simple test that grows the Cup by requiring an overloaded constructor:

```
#include "Cup.h"
```

```

#include "LoadedDieFactory.h"

#include <CppUTest/TestHarness.h>

TEST_GROUP(CupShould) {
};

TEST(CupShould, BeConstructableWithSharedPointers) {
    LoadedDieFactory factory(3);
    Cup cup(factory.build(), factory.build());
    LONGS_EQUAL(6, cup.total());
}

```

Notice that this is a new test file. The Cup came into existence as a refactoring exercise. Now there's a simple test file for it. This class is tested, just not in a classical unit test style. If this class becomes more heavily used, it might be a good idea to remediate the missing tests. For now, all we'll do is test the new code for this class.

### 3.27.2 A new constructor

The constructor looks almost the same as the last one (and in fact, it could be written using the same code):

```

class Cup {
public:
    ...

    Cup(spDie d1, spDie d2);
}

```

The constructor declaration uses the existing nested typedef. The constructor definition is just a few lines:

```

Cup::Cup(spDie d1, spDie d2) {
    dice.push_back(d1);
    dice.push_back(d2);
}

```

Create the new test and missing constructor. Get back to green before moving on.

## 3.28 Dice Game Instantiation

Now is where we need to move slowly, choosing to make one change at a time before moving to another change. There are three tests to verify the game's rules when rolling greater than, less than and equal to 7. Rather than try to fix all of those at once, we'll change one. This will force us to add a constructor, but it will allow the old and new approach to co-exist, keeping code compiling and tests passing more often.

### 3.28.1 First a test

Update the first test in DiceGameTest.cpp:

```

#include "LoadedDie.h"
#include "DiceGame.h"
#include "LoadedDieFactory.h"
#include <CppUTest/TestHarness.h>

```

```

TEST_GROUP(DiceGame) {};

TEST(DiceGame, BalanceDecreasesForLoss) {
    LoadedDieFactory factory(3);
    DiceGame game (factory);
    game.play();
    LONGS_EQUAL(-1, game.getBalance());
}

```

This test refers to a to-be-defined constructor.

### 3.28.2 Notice a pattern? New Constructor

We're adding a lot of constructors all over the place. This is pretty standard since we are dealing with object creation. Here's the updated constructor:

```

class Die;
class LoadedDieFactory;

class DiceGame {
public:
    DiceGame(LoadedDieFactory &factory);
}

```

The new constructor takes in a reference to a LoadedDieFactory. References are primitive, so the header file only needs to forward declare LoadedDieFactory rather than include the header file.

Of course, that defers the including the header file to the source file:

```

#include "DiceGame.h"
#include "LoadedDieFactory.h"
#include "Cup.h"

DiceGame::DiceGame(LoadedDieFactory &factory) : balance(0) {
    spDie d1 = factory.build();
    spDie d2 = factory.build();
    cup.reset(new Cup(d1, d2));
}

```

The last line of the constructor uses the reset method to replace the existing pointer stored by the shared pointer with a new pointer. The existing pointer is 0.

Update the test, add the constructor declaration and definition and get back to green.

### 3.28.3 Update the second test

With the work form the first test, the second test should smoothly translate:

```

TEST(DiceGame, BalanceIncreasesForWin) {
    LoadedDieFactory factory(5);
    DiceGame game (factory);
    game.play();
    LONGS_EQUAL(1, game.getBalance());
}

```

Get your solution to green.

You might have noticed that this test is actually checking for a different value. The original test used 4 and 5, for a total of 9. This test uses 5 and 5. That's OK since those values are in the same range of values for the rules of the game; they are in the same equivalence-class. Even so, it suggests a problem for the next test.

### 3.28.4 Oops, not there yet

The next logical thing to try is conversion of the third and final test, but we hit a brick wall. The current test uses 7 for a total. The current `LoadedDieFactory` class doesn't support that, so before we change that test, we need to extend the definition of the `LoadedDieFactory` to support this.

## 3.29 Extending Loaded Die Factory

This class is a test double. This doesn't mean we just write bad code and get it over with, but it does suggest that we only have to make it as flexible as the tests require. That is, you already have a good idea of what you need:

- Creating a factory with one value is convenient.
- Creating a factory that can produce two different loaded die, one for 3 and one for 4, is all the flexibility we need.

You could create an entirely different `LoadedDieFactory`, or just update the current one in place. Assuming you want to update in place, the next thing is a test.

### 3.29.1 Here's a test

```
TEST(LoadedDieFactoryShould, BeAbleToTakeTwoValues) {
    LoadedDieFactory factory(3, 4);
    spDie d1 = factory.build();
    spDie d2 = factory.build();
    LONGS_EQUAL(3, d1->faceValue());
    LONGS_EQUAL(4, d2->faceValue());
}
```

This test expresses one way that will give us what we need. Notice that it introduces (adds) a constructor rather than changing the existing constructor. Why? Fewer moving parts; change a test, get it to work. Then consider if removing the old constructor makes sense or not (we're not going to).

### 3.29.2 The Updated Class

This was the first solution I came up with:

```
#pragma once
#ifndef LOADEDIEFACTORY_H_
#define LOADEDIEFACTORY_H_

class Die;
#include <memory>
typedef std::shared_ptr<Die> spDie;

class LoadedDieFactory {
public:
    LoadedDieFactory(int firstValue, int secondValue);
```

```

LoadedDieFactory(int value);
virtual ~LoadedDieFactory();
spDie build();

private:
    int values[2];
    int lastIndex;
};

#endif

```

The class will hold an array of two values and an index indicating the last one returned. The idea is that it will toggle between the first and second values. The original constructor will still work; it will just populate the two values with the same value.

```

#include "LoadedDieFactory.h"
#include "LoadedDie.h"

LoadedDieFactory::LoadedDieFactory(int value) : lastIndex(-1) {
    values[0] = value;
    values[1] = value;
}

LoadedDieFactory::LoadedDieFactory(int firstValue, int secondValue)
    : lastIndex(-1) {
    values[0] = firstValue;
    values[1] = secondValue;
}

LoadedDieFactory::~LoadedDieFactory() {
}

spDie LoadedDieFactory::build() {
    lastIndex = (lastIndex + 1) % 2;
    return spDie(new LoadedDie(values[lastIndex]));
}

```

### 3.29.3 Return to green

This is a bit of a jump from the previous version; however the recommendation of doing the simplest thing that could possibly work is for situations where you don't know how to proceed. I've used this kind of thing before, so it's really not complex to me.

Get your solution back to green.

### 3.29.4 Back to that final test

Now you can update the final test in `DiceGameTest.cpp`:

```

TEST(DiceGame, BalanceRemainsSameForPush) {
    LoadedDieFactory factory(4, 3);
    DiceGame game (factory);
    game.play();
    LONGS_EQUAL(0, game.getBalance());
}

```

Since the factory works, this test should work. Make the change, get to green.

### 3.29.5 Why modify this final test at all?

The goal is to migrate the design to a new approach, to do that all existing code depending on the old constructor needs to be updated. Now that it is, you can remove the old constructor can generally clean up the code before moving to the next section.

## 3.30 Final Cleanup

The current implementation requires a little housekeeping. It might seem that more methods make a class more flexible but the exact opposite can be true depending on what you're trying to manage. If you want to build systems that are easier to maintain over time, smaller classes with minimal interfaces age better than large classes. Generally, the vast majority of time and money over the life of a project is spent in so-called maintenance, so anything you can do to help maintenance will be a big win. Keeping things clean is certainly a big help. Bigger still is a trail of automated tests as you work.

With that in mind, it's time to clean up unnecessary code.

### 3.30.1 DiceGame

There are three unnecessary things in the class:

- Forward declaration of `Die`;
- Declaration of a constructor taking two die pointers
- Definition of that same constructor.

Make these changes and confirm your solution is green.

### 3.30.2 Cup

Once you've cleaned up the `DiceGame`, this enables cleaning up `Cup`:

- `Cup` has an unused constructor
- `Cup` has a duplicated typedef for `std::shared_ptr<Die>`

For now, we can safely remove the constructor taking two `Die` pointers.

Make these changes and confirm your solution is green,

### 3.30.3 Common typedef

Now there's the duplicate typedef. While it does not cause problems, it is a DRY violation. So first introduce a new header file (`spDie.h`):

```
#pragma once
#ifndef SPDIE_H_
#define SPDIE_H_

#include <memory>
class Die;
typedef std::shared_ptr<Die> spDie;

#endif
```

Now include that where that type is needed and remove the duplicated typedef:



- Cup.h
- Cup.cpp (replace Cup::spDie with spDie)
- LoadedDieFactory.h

Make these changes and get back to green.

### 3.31 Is this better?

So is this better? Constructing with a factory rather than two Die objects? What about hiding the dynamic memory allocation deeper in the system?

#### 3.31.1 Can we even play a real game

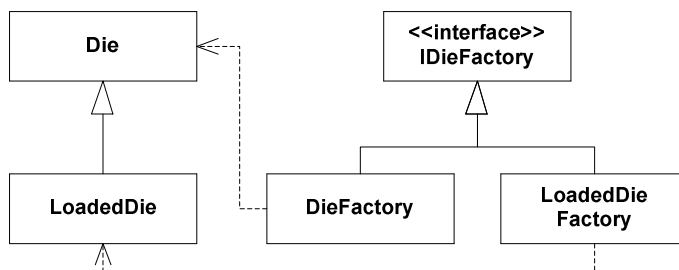
Right now there is a definitive answer, the current solution is clearly worse in one key respect. It's not possible to build a system with real Die objects! By making the most recent changes, it's no longer possible to build a system without a factory and there's only one kind of factory, LoadedDieFactory.

#### 3.31.2 Problem with Test Doubles

This represents something you need to watch out for when using test doubles; you can build a system with missing real code. That's what remains and along the way you'll see an important feature of C++, pure virtual member functions.

### 3.32 Refactor: Extract Interface

To get started, we need to build the DieFactory. Before doing so, review the class diagram:



This shows a top-level interface as the base of the two concrete factories. This is the common form of the abstract factory design pattern; a common abstraction, which one or more concrete classes implement. We have a leg up on this because we have a working class already in place, so we can extract an interface.

#### 3.32.1 The Class Definition

First, create a new class, IDieFactory by extracting what needs to be common:

```

01: #pragma once
02: #ifndef IDIEFACTORY_H_
03: #define IDIEFACTORY_H_
04:
05: #include "spDie.h"
06:
07: class IDieFactory {
  
```

```

08: public:
09:     virtual ~IDieFactory() = 0;
10:     virtual spDie build() = 0;
11: };
12:
13: #endif

```

Line	Description
09	This is a base class, so declare a virtual destructor. The = 0 syntax at the end indicate this as a pure virtual function. This makes the method abstract and it also makes the class abstract; it is not possible to make an instance of this class. Normally, a subclass will have to provide a declaration and definition of all pure virtual methods or it too will be abstract. In the special case of destructors, this is not true. More on this with the source file.
10	The build method also pure virtual. Subclasses will have to provide a declaration and definition of this method or be abstract.

### 3.32.2 Implementing the pure virtual destructor

```

#include "IDieFactory.h"

IDieFactory::~IDieFactory() {
}

```

Notice the definition of the destructor? If you do not include such a definition for a pure virtual destructor, the subclasses will be forced to provide one. To be safe, a base class should declare a virtual destructor. Why make it pure virtual?

In this particular case, the intention of this class is to serve as a behavior-only abstraction – an interface. C++ does not have interfaces, but it can have a class with all pure virtual methods, the next best thing.

So a pure virtual destructor declaration suggests the intent of the class. Providing a definition for the destructor minimizes the requirements for a base class; they will not have to write a destructor, but if they do, the correct one will get called.

### 3.32.3 Update LoadedDieFactory

With an extracted interface, it's quick to update `LoadedDieFactory` to implement that interface:

```

#include "IDieFactory.h"

class LoadedDieFactory : public IDieFactory {

```

Make these changes and get your solution back to green.

### 3.33 Now DieFactory

Time to create the class we need to build a proper system, the `DieFactory`. As with other examples, we'll start with a test. This will use new C++ syntax.

### 3.33.1 First the test

```
#include "DieFactory.h"
#include "Die.h"
#include <typeinfo>
#include <CppUTest/TestHarness.h>

TEST_GROUP(DieFactoryShould) {
};

TEST(DieFactoryShould, ReturnOnlyDie) {
    DieFactory factory;
    spDie die = factory.build();

    CHECK(typeid(Die) == typeid(*die.get()));
}
```

There's something new in this code, the use of `typeid`. The `typeid` operator returns back a thing called a `type_info` object. This comparison verifies that the kind object held onto by the shared pointer returned from the factory is actually a `Die` and not a `LoadedDie`.

### 3.33.2 The Implementation

Here's a minimal implementation of `DieFactory`. First the header file:

```
#pragma once
#ifndef DIEFACTORY_H_
#define DIEFACTORY_H_

#include "IDieFactory.h"

class DieFactory: public IDieFactory {
public:
    spDie build();
};

#endif
```

Now for a source file:

```
#include "Die.h"
#include "DieFactory.h"

spDie DieFactory::build() {
    return spDie(new Die);
}
```

That's it. You can certainly add:

- A no-argument constructor
- A destructor
- A copy constructor
- An assignment operator

This will give you a canonical form for the class.

### 3.33.3 Get to Green

Create this next to final test and the `DieFactory` class. Make sure your source is green before moving on.

### 3.34 A Smoke Test

Can you build a real system? It's time for a different kind of test. Here's a simple test to exercise a system as it is meant to be used:

```
#include "DiceGame.h"
#include "DieFactory.h"
#include <stdio.h>
#include <CppUTest/TestHarness.h>

TEST_GROUP(DiceGameSmokeTest) {
};

TEST(DiceGameSmokeTest, StandardUse) {
    DieFactory factory;
    DiceGame game(factory);

    for(int i = 0; i < 33; ++i)
        game.play();

    char balance[32];
    snprintf(balance, 64, "Balance = %d", game.getBalance());
    UT_PRINT(balance);
}
```

This test attempts to create a `DiceGame` using a `DieFactory`. Try to create this test, you'll find that it's a good idea you did so, because this won't compile.

#### 3.34.1 Make the required updates

The signature of the constructor is incorrect; it needs to refer to `IDieFactory`:

```
#include <memory>
class Cup;
class IDieFactory;

class DiceGame {
public:
    DiceGame(IDieFactory &factory);
```

You'll also need to update the definition:

```
#include "DiceGame.h"
#include "IDieFactory.h"
#include "Cup.h"

DiceGame::DiceGame(IDieFactory &factory) : balance(0) {
```

### 3.34.2 Back to green

Make these changes and confirm that your system is back to green.

### 3.34.3 Where does this test belong?

This test is different in a few ways.

- It uses a new macro, `UT_PRINT`, to output some information.
- It uses only production classes, no test doubles

This is a fully-wired system. You will need to write tests like these to make sure anything about the configuration of your system that might be broken is discovered automatically.

This particular test takes little time to run. Even so, it might be a good idea to organize tests with different intentions into different projects. As a developer, I want to be able to run all the various automatic tests on my personal machine. I want to be able to do that and not have to worry about shared resources like databases or message queues. This suggests certain kinds of design considerations that lead to good test isolation.

In any case, if you decided to include this test in an automated test suite meant to be run by developers often throughout the day, remove the output. Why? In practice, output has a few negative consequences:

- It slows test execution. Anything that unnecessarily slows tests leads to tests getting run less frequently, which reduces their value considerably. You won't find failures fast.
- It encourages manual checking of something that can probably be automated.
- It leads to weaker testing. If there's output, then people can check things just in case.

This does not suggest that a production system should not have logging. I'm just saying that unit tests should produce no output in general.

In any case, you've finished this project. Congratulations

### 3.35 Wrap-up

This last section was primarily about experimenting with a design pattern and the downstream ramifications associated with that. It was also about refactoring in small steps, keeping the code compiling and the tests passing while changing the structure of your solution.

There were a few new things as a result:

Term	Description
Abstract Factory	Designate a class whose primary responsibility is to build one or a family of objects. Create an abstract class that captures the interface and then have multiple implementations for different situations.
Pure Virtual	A virtual method can be pure virtual. This has several effects <ul style="list-style-type: none"> <li>▪ The method is abstract</li> <li>▪ The class is abstract</li> <li>▪ The class cannot be instantiated</li> <li>▪ Subclasses must implement the pure virtual method or</li> </ul>

Term	Description
	themselves be abstract <ul style="list-style-type: none"><li>▪ If you have a pure virtual destructor, it can be defined in the base class and subclasses will not have to implement it.</li></ul> We made all methods in one class pure virtual, the intent of which is to suggest that a class is not just abstract but is in fact an interface.
typeid	An operator used to determine the type of an object. You can only use typeid on classes with at least one virtual method. We used this to confirm that the kind of object retrieved from a DieFactory was in fact an actual Die and not a subclass.

### 3.36 Final Recommendations

C++ is a huge language and if you're just starting out or even if you've been using it for a few years, consider it an ever expanding project to really learn the language. We have skipped most of the language in this project, yet you know enough to get starting writing decent Object Oriented solutions using the language. The rest of this section is a collection of next steps and recommendations.

#### 3.36.1 Books

There are many books you might consider reading over the next several months and years. Here's a short list of some you might want to consider:

- Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions
- More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions
- Accelerated C++: Practical Programming by Example
- Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)
- Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library

Here are a few more books you might want to consider as you dig a bit deeper, or to give you some context:

- Advanced C++ Programming Styles and Idioms
- Ruminations on C++: A Decade of Programming Insight and Experience
- C Traps and Pitfalls
- The Design and Evolution of C++

#### 3.36.2 Katas

A common practice is to take a simple problem and practice it over and over. I've collected a number of katas from other people and a few I've developed myself. Have a look at: <http://schuchert.wikispaces.com/Katas>

#### 3.36.3 Practice

That goes without saying; you need to practice with the language. Learning the language only while on the job, while useful, might actually limit your learning. I started using the

language August of 1989 and I used it nearly daily until around June of 1990. I then took the summer off and did not do any programming for about 2 months. In that 2-month interval, I was able to figure out many things I only knew by rote. Because I was buried in the problem and worried about deadlines, it limited what I was learning about the language.

A way to mitigate that is to practice with katas. An even better way is to pick one substantial problem and practice it over and over, making slight variations on your approach each time. The idea is to become familiar with the domain to the point where you are able to take individual variables and change them. For example, there's a design recommendation called "tell don't ask." You might simply do a problem and universally apply this one design idea. You'll find doing this kind of active experimentation will give you a deeper appreciation of the language. It will also translate to other languages as well, so you're not really just learning C++.

### 3.37 What's coming up?

It's time to start a new problem. The approach will be similar, but the design forces will be a bit more important. In this next problem, you'll won't encounter much new C++, instead you'll be reapplying what you've already seen in the dice game to a problem with richer design issues.

## 4 RPN Calculator

According to Wikipedia ([http://en.wikipedia.org/wiki/Reverse\\_Polish\\_notation](http://en.wikipedia.org/wiki/Reverse_Polish_notation)) Reverse Polish Notation was created in 1954 by Burks, Warren, and Wright. In reverse polish notation, also known as postfix notation, operands appear before an operator. Here are a few examples of such notation:

Expression	Result
30 4 +	34
30 4 -	26

At the beginning of the century I was taking an internal class on Test Driven Development where the instructor used a “regular” calculator as the main example. For some reason, I chose to implement an RPN calculator instead. While working in the class, it occurred to me that my pairing partner and I were having an easier time at it than the other students. In fact, this is because it easier to write an RPN calculator than a “regular” calculator. An RPN calculator:

- Does not require as much “memory” – when the user selects an operator, you can immediately perform a calculation. Contrast this with a regular calculator where upon entering a number and an operator, the calculator has to remember both and wait for the next number before doing any work.
- Does not require () – operators happen immediately

In fact, according to HP Museum (<http://www.hpmuseum.org/hp9100.htm>) this is what made it possible for HP to build the first electronic RPN calculator in 1968. The first HP calculator:

- Weighted about 40 pounds
- Cost around \$5,000 USD
- And was considered a modern miracle

We are going to use a problem similar to an HP calculator as the basis for our second problem. Whereas in the first problem we were looking primarily at C++, the purpose of this problem is to look at:

- Object Oriented Design
- Design Principles
- Design Patterns

Also, unlike the first problem where we worked bottom up, in this problem we will work top down, or out to in. We will start with a series of examples for a first iteration. We’ll create automated checks for those examples and clean up the code as we work our way through the problem.

Side Note: I’ve used the problem many times. I’ve worked it with “raw” TDD – no design up front, see where students take the problem. I’ve also used it as a design problem where, I have students work on up-front, detailed design. For this book I had to choose one path through the problem and I somewhat arbitrarily decided on raw TDD. We will go from examples to tests. As we develop tests, I’ll mention design decisions you need to make or consider.



## 4.1 Project Description

For this problem you will create a programmable RPN calculator. Your calculator will have:

- Several functions,
  - Basic functions such as add, subtract, multiply, divide, less than, greater than
  - Bigger functions such as sum, factorial, prime factors
  - Stack operations such as duplicate, drop, rotate up, rotate down
- It should be easy to add new operations to the calculator
- The calculator should be “programmable”, meaning you can create a new operation that is a combination of any existing operations
  - Those new operations should execute in the same way as the built-in operations
  - It should be possible to have one program refer to another program as well

The calculator will deviate from a standard HP calculator in at least the following ways:

- Our calculator will allow more than 4 numeric entries on its stack
- We will use integer math for simplicity

## 4.2 What’s Coming Up?

Given this preliminary problem description, it’s time to get started on the problem Here’s what to expect:

- Story selection
- Example development
- Automated check writing

To do this well, we’ll also consider

- GRASP patterns to help with developing automated checks
- Actor-system interaction and its impact on top-level API design
- The lost-art of system events

## 4.3 Biting off just enough

This problem already has a number of potential stories including:

- Operators: Adding numbers, calculating factorials, division
- Stack Manipulation: dropping values, duplicating part of the stack
- Programming

This is too much to attempt all at once so we should pick a small set of stories for a first demonstration. Story writing and selection is beyond the scope of this book. However, here are thoughts on a first cut:

- Add, Subtract – these are easy to understand and a calculator without these features would be too surprising. If we were to only pick one, I’d choose subtract. Why? Because unlike add, the order of the numbers is important. However, these are close enough that we’ll do both of them.
- Multiply, Divide – these two finish off the set of operations that even the most basic calculators include.

- Factorial, Drop – these may seem like odd choices, however, they have a fundamental difference from the previous four operations; factorial only needs one operand, drop needs one operand as well, but it produces no results.

Given this list of operations, the next thing to do is develop some concrete examples for each of these stories.

#### 4.4 Develop Examples

What is the difference between a story and an example? A story describes a use of the system. An example fills in that story with specific values. Often, we write these examples using a standard language, or set of keywords. Here are some examples for each of our chosen stories:

Given the user enters	When the user selects	Then the result is
30 4	+	34
30 4	-	26
4 6	*	24
8 2	/	4
5	!	120
5 3 1	drop	5 3

These are all “happy-path” examples. They are simple, complete and represent successful scenarios. What are some examples that represent potential failures? One example might be overflow, another underflow. In a more complete example we would not use integers but some numeric library to provide better precision. So overflow and underflow, while legitimate test cases, are out of consideration since we’re using integers to keep things simple; what we’d write is an artifact of a simplifying decision.

What about the following situations:

- Using any of these with “too few” parameters
- Divide by zero with division
- Factorial of a value less than 1

An artifact of a real HP calculator is that there are always 4 numbers available. When you turn on the calculator, you have 4 values. Those values could be all 0 or they might be values from the last time you used the calculator. In either case, there’s no such thing as “too few parameters”. This can be assumed knowledge, or we can capture a few examples to demonstrate this idea:

Given the user enters	When the user selects	Then the result is
4	+	4
4	-	-4
4	*	0
4	/	0
	+ <or> - <or> *	0
	/	<error: divide by 0>

	drop	
	!	1

These examples are still incomplete (they always will be). There's only one example that shows what happens where there are more than the "perfect" number of values available. What happens when you attempt to add values and there are extra values?

Given the user enters	When the user selects	Then the result is
3 2 6 7 2	+	3 2 6 9
3 2 6 7 2	-	3 2 6 5

Given these examples, it seems that a user can enter several values. Using add or subtract works with the two most recent values. Also, when the order matters, the most recent value is on the right side of the operator and the previous value is on the left.

This seems like a good start. We will probably end up with more examples, but we'll consider this first release done when all of these examples work in automated checks.

#### 4.5 Project Setup

This is a new project so it's time to create a new project setup from scratch. If you need assistance, please refer to section 2.2.1 starting on page 10.

#### 4.6 The first unit check

It's time to start writing our first unit check. To start writing it, we need to make a few decisions. Here are some questions and answers:

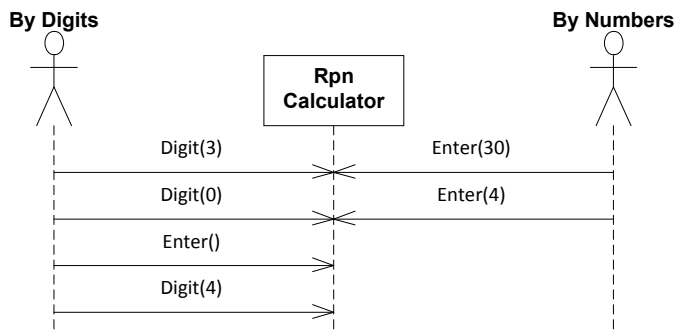
Question: What are we writing this first unit check against?

Answer: Since this is an RPN calculator problem, and we are working out to in, we will target the RPN calculator class with our first unit check. This is an example of using the Controller GRASP pattern (ref). The controller pattern is unfortunately named; it would have better been called facilitator or coordinator. We have messages coming in from the outside via some actor. The actor could be a human but since we are using TDD, the actor is a test.

Question: What are the logical steps of this unit check?

Answer: Enter a number, enter a second number, perform some operation, and verify the results.

Question: Does "enter a number" mean a full number, e.g. 30, or just one digit of a number?



Answer: This is a somewhat arbitrary decision. Rather, the answer is somewhat arbitrary, making the decision is necessary. We could:

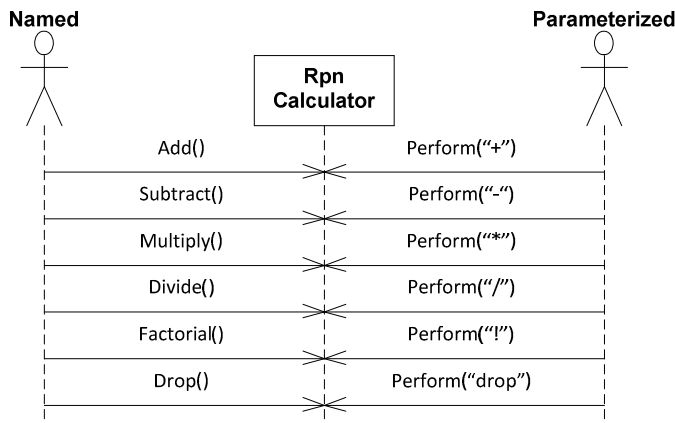
- Enter individual digits like pressing buttons on a calculator
- Enter full numbers, like collecting digits in a text box in the UI and only presenting the final number to the calculator

I have done this problem both ways and for how far we are going to take the problem, there won't be much of a difference. From personal experience, collecting individual digits leads to making it easier to maintain the state of the system in a single location. This becomes important when you start using a numeric library as opposed to using built-in (primitive) numbers. It makes more work for us if we take in individual digits but that initial work isn't really too much.

Notice, as with most questions, experience can be a good guide. When I use this problem in a class setting, I let the students decide. Since I don't have someone to make the decision, I flipped a coin and it came up "numbers" so for this example, we'll be taking in full numbers.

Here's another problem to consider. If we built a complete system with some kind of user interface, the values coming into system would probably not be numbers but instead be strings of digits. In fact, one of the things a controller does is translate requests from the outside world into something the system can understand. It then decides where to send the request and delegates the rest of the work to some kind of plain old C++ object (a POJO). This is another decision I let the students make. This is another thing that comes from writing many systems and having to design the system-level interaction. While this is a necessary problem, the work actually required is dependent on something that is out of the scope for this problem. So in addition to taking in numbers as opposed to digits, the representation of those numbers will in fact be integers and not strings.

Question: How will we select add, versus subtract, etc.?



Answer: What may seem obvious is to have one named method for each operation. This approach has a few characteristics:

- It leads to an API that is open-ended. As you grow the system, the API will continue to expand.
- Additionally, the client has to know which method to call. If the user types “+” or clicks the “+” button, both of these actions must be mapped to the “Add()” method. As the API grows, this mapping also grows.
- If the code compiles, you know you’ll be calling a particular method, which in a sense is like static (compile-time) checking.
- It is easy to understand.
- This kind of interface will not work very well when we consider programming the calculator.

While this is typical, familiar even, that doesn’t mean it’s a good idea.

Using the parameterized approach:

- The API is locked down – this is an example of protected variation and it also is an example of applying the open-closed principle to the problem.
- The client can allow the user to type “+” or press a button with “+” as the label. The mapping is now done deeper in the system.
- Even if the code compiles, you don’t know until you run it if a particular operation will happen.
- This approach maps better to the requirement that programs run the same as other operations.

We could choose either solution, but we must choose one. This is probably one of the most important lessons: Picking a “wrong” solution is often better than waiting to pick the “right” solution. To have a chance seeing how a design can move from one API design to another, we’ll choose the named method approach.

Question: How do we know the most recent result?

Answer: An HP calculator has 4 registers: x, y, z, and t. The x register is the “top” or most recent value. This register is also known as the accumulator. To keep things simple, we could simply ask for either the x register or the accumulator. Both terms are

mentioned in an HP calculator manual, so they come right from the domain. For simplicity, we'll use `getX()`; it comes right from the domain.

#### 4.6.1 Was all of this necessary?

The short answer is yes. If you agree, skip to the next section.

If you are still here, then ask yourself the question, "What is our goal?" We are trying to write an executing piece of code that will show we can add two numbers. For this to work, it will have to compile. That means we needed to make all of those decisions. We could just write the first test and see what happens. In practice, you'll be doing that. However, as you get more experience, the first test will take more and more into consideration. Even when we practice "raw" TDD, our previous experience feeds into the overall approach. So when I'm working on problems, these kinds of things are going through my head either consciously or subconsciously. So you really do need to make these decisions. You can do so as you write a test. However, since I'll be writing the test for you, you'll be missing out on some of this if I don't take the time to walk through it.

#### 4.6.2 Add: The First Test

Having setup your project, it's finally time to create a first test. Here is one such test that takes the previous section into consideration:

##### *RpnCalculatorShould.cpp*

```
#include <CppUTest/TestHarness.h>

TEST_GROUP(RpnCalculatorShould) {
    RpnCalculator *calculator;
    void setup() {
        calculator = new RpnCalculator;
    }
    void teardown() {
        delete calculator;
    }
};

TEST(RpnCalculatorShould, AddTwoNumbers) {
    calculator->enter(30);
    calculator->enter(4);
    calculator->add();
    LONGS_EQUAL(34, calculator.getX());
}
```

A quick recap from the previous project may be in order:

- To use CppUTest you need a `TEST_GROUP` for a set of related tests and a `TEST` for an individual test.
- The `TEST_GROUP` is more commonly known as a test fixture. We put things in the test fixture that are common for all tests.
- In a language with refactoring tools, I'd normally start with just a test method and refactor my way into a common test fixture.

- Additionally, CppUTest does memory leak detection. However, if your class uses any kind of dynamic memory allocation, you might get false positives on memory leaks.
- Given this, I typically start my test code with a TEST\_GROUP that has an instance of the thing it is testing stored as a pointer. Then I use setup and teardown to allocate and release that object. This avoids most of the false positives that might be recorded by CppUTest.

The body of the TEST is a direct interpretation of the previous few pages of discussion. To make this work we'll need to create the missing class, RpnCalculator. To do that, we'll probably want to follow some basic C++ guidelines. With that in mind, here's a header file for RpnCalculator:

#### **RpnCalculator.h**

```
01: #pragma once
02: #ifndef RPNCALCULATOR_H_
03: #define RPNCALCULATOR_H_
04:
05: #include "RpnCalculator.h"
06:
07: class RpnCalculator {
08: public:
09:     RpnCalculator();
10:     virtual ~RpnCalculator();
11:     void enter(int value);
12:     void add();
13:     int getX() const;
14:
15: private:
16:     RpnCalculator(const RpnCalculator&);
17:     RpnCalculator &operator=(const RpnCalculator&);
18: };
19:
20: #endif
```

There is nothing new in this header file if you've read the rest of the book. There's no good reason to copy objects of this class so lines 16, and 17 make that unlikely. The destructor, line 10, is virtual because I used Eclipse to create the class and that's its default setting. It is unlikely we'll create a subclass of this class, so the virtual on the destructor is not necessary. I did it anyway.

The test required enter(), add() and getX(), lines 11 – 13. This header file with no source file will allow the test to compile (but not link).

To get the code to link – and the test to pass, here's a first cut at an implementation:

#### **RpnCalculator.cpp**

```
#include "RpnCalculator.h"

RpnCalculator::RpnCalculator() {}
RpnCalculator::~RpnCalculator() {}
void RpnCalculator::enter(int value) {}
void RpnCalculator::add() {}
```

```
int RpnCalculator::getX() const {  
    return 34;  
}
```

Sure enough, this gets passing results:

```
.
```

```
OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 0 ms)
```

Before moving on, we need to check if there are any places where we might refactor the code. There's not much right now. Sometimes we need to allow code to fester a bit before there's enough to know how to generalize it. So we'll move on to the next example.

#### 4.7 Subtract: The Second Test

The second example is subtraction. We can take the first automated check as a baseline and create our second one:

```
TEST(RpnCalculatorShould, SubtractTwoNumbers) {  
    calculator->enter(30);  
    calculator->enter(4);  
    calculator->subtract();  
    LONGS_EQUAL(26, calculator->getX());  
}
```

For this to compile, we need to update the header file. To get the code to link we'll need to update the source file:

*Added to RpnCalculator.h*

```
void subtract();
```

*Added to RpnCalculator.cpp*

```
void RpnCalculator::subtract() {  
}
```

This gets us to the new test failing while the original test passes. Now the challenge, we must update the code to keep the original test passing and get the new test passing as well.

One thing we can do is covert the hard-coded value returned by `getX()` into a variable. A variable will allow us to generate different results. Those different results can be performed by the `add()` and `subtract()` methods. Moving from a hard-coded value to either a calculated value or introducing a variable is a good second step as you build solutions. In fact, you might consider reviewing a recommendation by Robert Martin he called the transform priority premise (<http://cleancoder.posterous.com/the-transformation-priority-premise>).

In our case, we need to make four changes:

- Add a variable
- Set the result in `add()`
- Set the result in `subtract()`
- Return the result in `getX()`



While not strictly necessary, we should also initialize the member data value in the constructor because C++ won't do it for us. Here are those updates:

#### **Added to *RpnCalculator.h***

```
private:  
    int x;
```

#### **Updated Constructor**

```
RpnCalculator::RpnCalculator() : x(0) {  
}
```

#### **Updated *add()*, *subtract()*, *getX()***

```
void RpnCalculator::add() {  
    x = 34;  
}  
  
void RpnCalculator::subtract() {  
    x = 26;  
}  
  
int RpnCalculator::getX() const {  
    return x;  
}
```

These changes will return you to compiling, linking and tests passing.

## 4.8 What about Actual Values?

Right now the implementation does not use the actual values provided when calling the `enter()` method. When you are working on a problem, you will often come across these challenges. A quick review of the examples shows that the first six each describe the use of a different operation. Nothing in those examples will force use to actually store any values. The first time that occurs is when we write a second automated check against an operator that already has an implementation. This is an unsatisfying result to many programmers. We could jump ahead in the list of examples, but the next example that duplicates the use of an operation is `add` where there is only one item provided. This would force us to tackle two problems at the same time: really writing `add()`, dealing with the situation when there are less than two provided values.

A danger at this point is speculative design. We could just do something and see where it goes. However, if we start that, we'll probably end up writing code that has no automated checks for it. Alternatively, we could write another test for `add()` or `subtract()` that uses two different values. A problem with that is that the second test is a duplicate of the first test. Duplication is not a good thing. Duplicated tests represent rework when things change. Too much duplication might lead to enough inertia to make a change seem too costly.

Referring back to the transform priority premise (<http://cleancoder.posterous.com/the-transformation-priority-premise>), the recommendation is to pick automated checks that will choose simple code changes over more complex code changes. Based on that, we want to either consider one of the provided examples or create another example that will help us. We currently have an empty implementation of `enter()`, so something that will

force us to grow the implementation of `enter()` and that also make sense to for the other operators seems to be warranted.

In an earlier discussion we noted that the most recently entered value was on the right side of the operator while the previously entered value was on the left side. What if we simply write that test? We will enter two values and make sure we can get back those two values.

Here is one such test:

```
TEST(RpnCalculatorShould, ReturnValuesInReverseOrderOfEntry) {
    calculator->enter(30);
    calculator->enter(4);
    LONGS_EQUAL(4, calculator->getX());
    calculator->drop();
    LONGS_EQUAL(30, calculator->getX());
}
```

This is OK, but notice that it also forces us to introduce `drop`. That might be OK, let's see what we can do to make this work:

**Add member function declaration to `RpnCalculator.h`**

```
void drop();
```

**Update `enter()` and `drop()`**

```
void RpnCalculator::enter(int value) {
    x = value;
}

void RpnCalculator::drop() {
    x = 30;
}
```

Try this, you will see the following execution results:

```
...
OK (3 tests, 3 ran, 4 checks, 0 ignored, 0 filtered out, 1 ms)
```

Not quite enough. However, what if we simply write the example for `drop`? Doing so will force `drop` to become more complicated; maybe it will also force us to remember all the values entered as well.

## 4.9 Drop

Here's an automated example for `drop`:

```
TEST(RpnCalculatorShould, SupportDroppingValues) {
    calculator->enter(5);
    calculator->enter(3);
    calculator->enter(1);
    calculator->drop();
    LONGS_EQUAL(3, calculator->getX());
}
```

This automated check fails. Also, it will be hard to make this work without doing some “real” work. In fact the work we’ve been doing is real. Once you’ve been practicing TDD for some time, you’ll probably agree. Even so, this makes things more interesting.

Notice that we enter several values but we only store the last. That’s a clue for what we could do next. Convert our single value, *x*, into many values, some kind of collection. That seems like not too much of a leap. We need to do that in such a way as to keep existing tests passing, get the failing test to pass and we don’t want to spend too much time doing it.

Before we jump in and change it, what collection should we use? The values need to come out in the reverse order entered. That suggests a stack. Conveniently, C++ has a stack class, so we can use that.

Changing from a simple variable to a collection seems like refactoring. We have a failing test. Refactoring with a failing test is frowned upon as we discuss elsewhere. So before we start making that change, let’s first “remove” this test by replacing TEST with IGNORE\_TEST:

```
IGNORE_TEST(RpnCalculatorShould, SupportDroppingValues) {
```

Now we have all passing tests and we are ready to change our implementation.

#### 4.9.1 First introduce the stack

Here’s an updated RpnCalculator.h introducing the stack. Notice that the *x* attribute is still there. When we are done, we’ll remove it.

```
#include <stack>

class RpnCalculator {
public:
    ...

private:
    typedef std::stack<int> RpnStack;
    RpnStack values;
    int x;
...
}
```

Now we can update all of the existing methods to use both the stack and *x*:

```
void RpnCalculator::enter(int value) {
    x = value;
    values.push(value);
}

void RpnCalculator::add() {
    x = 34;
    values.push(34);
}

void RpnCalculator::subtract() {
    x = 26;
    values.push(26);
}
```

```
}  
  
void RpnCalculator::drop() {  
    x = 30;  
    values.pop();  
}
```

Run your automated checks, verify that everything is passing with one ignored test:

```
!...
```

```
OK (4 tests, 3 ran, 4 checks, 1 ignored, 0 filtered out, 1 ms)
```

Now you can remove X from the header file and source file; you'll also need to update getX() to return the top of the stack:

*RpnCalculator.cpp minus x member data*

```
RpnCalculator::RpnCalculator() {  
}  
  
RpnCalculator::~~RpnCalculator() {  
}  
  
void RpnCalculator::enter(int value) {  
    values.push(value);  
}  
  
void RpnCalculator::add() {  
    values.push(34);  
}  
  
void RpnCalculator::subtract() {  
    values.push(26);  
}  
  
void RpnCalculator::drop() {  
    values.pop();  
}  
  
int RpnCalculator::getX() const {  
    return values.top();  
}
```

Interestingly, if you simply update the IGNORE\_TEST to be a TEST again and execute your tests:

```
....
```

```
OK (4 tests, 4 ran, 5 checks, 0 ignored, 0 filtered out, 0 ms)
```

That worked out pretty well.

#### 4.10 Getting Factorial Working

We only have one example for factorial: 5 -> 120. The factorial of 0 is 1. For anything less than 1, we do not have any examples. For the purposes of this example, we'll say that anything less than 0, the result is simply consumed. Here's a sequence of tests for those examples:

```
TEST(RpnCalculatorShould, CalculateTheFactorialOf5As120) {
    calculator->enter(5);
    calculator->factorial();
    LONGS_EQUAL(120, calculator->getX());
}
```

To get this to pass we'll update the header file:

```
void factorial();
```

And add a member function definition to the source:

```
void RpnCalculator::factorial() {
    values.push(120);
}
```

Not to worry, the next text makes us do a little more work:

```
TEST(RpnCalculatorShould, CalculateTheFactorialOf0As1) {
    calculator->enter(0);
    calculator->factorial();
    LONGS_EQUAL(1, calculator->getX());
}
```

At this point, we can simply write a simple factorial implementation:

```
void RpnCalculator::factorial() {
    int operand = values.top();
    int result = 1;
    while(operand > 1)
        result *= operand--;
    values.push(result);
}
```

And finally, what happens if the value is negative?

```
TEST(RpnCalculatorShould, ConsumeValueForFactorialOfNegative) {
    calculator->enter(-4);
    calculator->factorial();
    LONGS_EQUAL(0, calculator->getX());
}
```

Before looking at the implementation, why 0? Remember that the calculator always has values. The real calculator has 4 hardware registers, all initially 0 (or whatever value they had when you last used the calculator). So there are always values. This calculator is brand new and only had ever had a single value entered. Therefore, upon consuming that value, there should be "no entered numbers", which means the calculator will have only 0s available.

Here's one way to do this:

```
void RpnCalculator::factorial() {
    int operand = values.top();
    values.pop();
    if (operand >= 0) {
        int result = 1;
        while (operand > 1)
            result *= operand--;
        values.push(result);
    }
}

int RpnCalculator::getX() const {
    if(values.size() > 0)
        return values.top();
    return 0;
}
```

In the factorial() method, we actually consume the value entered by calling values.pop(). However, this leaves the stack empty for our most recent test. If you simply run the test, it will fail in some way that may not be obvious. It turns out that if you call top() on an empty stack, the method throws an exception.

#### 4.11 Revisit Add

Now that we have finally written an actual implementation for factorial, let's revisit the other operators before adding new ones. First add. We have another example for add, here's a test for that example:

```
TEST(RpnCalculatorShould, AddWhenTheresASingleValue) {
    calculator->enter(4);
    calculator->add();
    LONGS_EQUAL(4, calculator->getX());
}
```

This test fails because the underlying add() hard-codes the result to 34. It is time to fix this. However, to fix this we'll have to consider what factorial did: it called top() and pop() for one operand, so we'll need to do that for add as well:

```
void RpnCalculator::add() {
    int v1 = values.top();
    values.pop();
    int v2 = values.top();
    values.pop();
    values.push(v1 + v2);
}
```

Whoops, this fails. Problem is, top() and pop() really need to be guarded like in the getX() method. Rather than calling top(), we can call getX() and then guard pop():

```
void RpnCalculator::add() {
    int v1 = getX();
    if(!values.empty())
        values.pop();
    int v2 = getX();
```

```
if(!values.empty())
    values.pop();
values.push(v1 + v2);
}
```

This works, which is a qualified success.

#### 4.11.1 Feature Envy

Looking at the `add()` method it makes 4 direct calls to `values`, which is a stack. Furthermore, it makes two calls to `getX()`, which makes 2 calls to the stack as well. All told, that's 8 calls to a stack from `add()`. You can assume that we'll have to do the same thing in `subtract()`, `factorial()`, etc., all operations that use the stack.

When one method makes heavy use of data or methods in another object, this is called feature envy (<http://c2.com/cgi/wiki?FeatureEnvySmell>). Martin Fowler describes feature envy as "...a method that seems more interested in a class other than the one it is in." This seems to fit that description well. How can we fix it? Luckily, Martin Fowler has something to say about that as well. His first recommendation is move method. That is, move the method into the target class.

What does this mean for us? It means we want to move the check for size into the stack. But the stack is not our class; we are using it from the standard library. Fine, we will create our own stack class that is built upon the standard stack class. That's the next section.

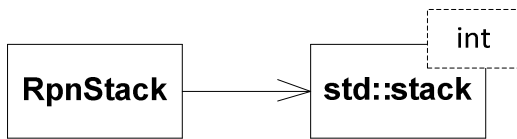
#### 4.12 Resolving Feature Envy: Writing Our Own Stack

This is not a surprising result to me. Collection classes are handy and they are also a common source of duplicated code. Why? Collections such as the stack class can only have general behavior. That behavior should be simple and complete. The stack is a good example. You can push until it is full, which takes a lot of values. You can pop values off it or look at its top only when it has values on it; its size is not zero. That's a typical stack data type behavior, but we are using a stack to implement something that does not behave exactly like a stack. In our case, the values are last in, first out (LIFO) like a stack. Unlike a stack, however, we always have values. The idea of `empty()` doesn't really apply. When a user has not entered any values onto a calculator, the calculator has 0s.

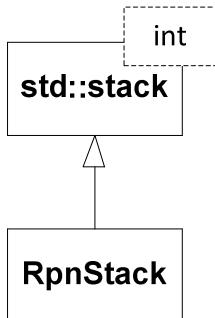
This discrepancy between our domain and what built-in classes have for default behavior is actually the norm in my experience. I saw one project where an effort to remove duplication just within individual source files across a 15,000,000 million line source base of C++ resulted in a 30% reduction of code size. Just over half of that reduction related to the use of collections. Does that mean collections are bad? No, it just means that they serve as a starting point only.

In our case, we want something that behaves like a stack but one that is never empty. We can do this in a number of ways:

- Roll our own implementation
- Use an existing one



- Inherit from an existing one



The first option is obvious, create the solution from scratch. The next two options are more typical. The last option, inheritance, is normally the one I'd choose last. Inheritance is the single highest form of coupling in C++. If there are methods in the base class you do not want exposed, you have other options such as protected or private inheritance, but I do not use these language features. If, for some reason, you want to change method signatures, you might find it hard to do so (depending on the change). In this case, the base class does not expose any virtual methods, so even if we do use inheritance, we won't want to pass around a reference or pointer to the base class when using the derived class. Inheriting from concrete class such as this can cause problems such as calling the wrong method.

The middle option, an RpnStack holds onto an instance of an std::stack, is an example of using delegation instead of inheritance has mentioned in the book Design Patterns. There's no chance to confuse an RpnStack with an std::stack because the types would not be compatible, whereas using inheritance, they would be. So with no other reason, and in general, I'd pick the second option over the third option.

In this case, however, I'm going to demonstrate the third option to show how to inherit from template classes and also how to invoke base-class methods.

#### 4.12.1 First test: top works on an empty stack

Before jumping in, we'll use tests to verify our implementation. Then we'll go back and update the RpnCalculator to use our custom RpnStack. Here's a first test:

##### *RpnStackShould.cpp*

```

#include <CppUTest/TestHarness.h>
#include "RpnStack.h"

TEST_GROUP(RpnStackShould) {
    RpnStack *values;
    void setup() {
        values = new RpnStack;
    }
}
  
```



```

    void teardown() {
        delete values;
    }
};

TEST(RpnStackShould, HaveATopAfterCreation) {
    LONGS_EQUAL(0, values->top());
}

```

To get this to compile, link and pass:

#### *RpnStack.h*

```

#pragma once
#ifndef RPNSTACK_H_
#define RPNSTACK_H_

#include <stack>

class RpnStack : public std::stack<int> {
public:
    RpnStack();
    virtual ~RpnStack();
    int top() const;

private:
    RpnStack(const RpnStack&);
    RpnStack& operator=(RpnStack&);
};

#endif

```

#### *RpnStack.cpp*

```

01: #include "RpnStack.h"
02:
03: RpnStack::RpnStack() {}
04: RpnStack::~~RpnStack() {}
05:
06: int RpnStack::top() const {
07:     if(!empty())
08:         return std::stack<int>::top();
09:     return 0;
10: }

```

Line 8 demonstrates how to call a base-class method. Our version of top() returns 0 if the stack is empty otherwise it returns whatever the base-class top() method would have returned.

To get pop() well behaved is more of the same:

```

TEST(RpnStackShould, HaveASizeOf0AfterPopWhenEmpty) {
    values->pop();
    LONGS_EQUAL(0, values->size());
}

```

If you run this test without updating RpnStack it fails:

```
../RpnStackShould.cpp:21: error: Failure in TEST(RpnStackShould,
HaveASizeOf0AfterPopWhenEmpty)

    expected < 0 0x0000000000000000>

    but was <-1 0xffffffffffffffff>
```

Now update pop() as well, add a method declaration:

```
void pop();
```

And definition:

```
void RpnStack::pop() {
    if(!empty())
        std::stack<int>::pop();
}
```

#### 4.12.2 Update RpnCalculator

Now simply update the RpnCalculator header file to use RpnStack:

```
#include "RpnStack.h"

class RpnCalculator {
    ...
private:
    RpnStack values;
```

Now it should not be necessary to check the size anywhere in the source file:

```
void RpnCalculator::add() {
    int v1 = getX();
    values.pop();
    int v2 = getX();
    values.pop();
    values.push(v1 + v2);
}

int RpnCalculator::getX() const {
    return values.top();
}
```

And you may have noticed that factorial() called top() without checking the size. There was a bug there, we just had not yet noticed it.

#### 4.13 Finish subtract

Now we can write a test for subtract that starts with just one value instead of two on the stack:

```
TEST(RpnCalculatorShould, SubtractWhenTheresASingleValue) {
    calculator->enter(4);
    calculator->subtract();
    LONGS_EQUAL(-4, calculator->getX());
}
```

This test fails when you run it. Now you can update the subtract implementation. Try just copying add() and replacing the + with -:

```
void RpnCalculator::subtract() {
    int v1 = getX();
    values.pop();
    int v2 = getX();
    values.pop();
    values.push(v1 - v2);
}
```

Run the tests:

```
../RpnCalculatorShould.cpp:72: error: Failure in
TEST(RpnCalculatorShould, SubtractWhenTheresASingleValue)

    expected <-4 0xffffffffffffffc>

    but was < 4 0x0000000000000004>

../RpnCalculatorShould.cpp:26: error: Failure in
TEST(RpnCalculatorShould, SubtractTwoNumbers)

    expected < 26 0x000000000000001a>

    but was <-26 0xffffffffffffffe6>
```

Not only did that not work, it caused another test to fail! In this case we need to reorder the operands:

```
values.push(v2 - v1);
```

That fixes it. This makes sense. Subtract cares about the order of operands whereas add does not. But this gives us something ugly: two methods that are almost duplicates of each other:

```
void RpnCalculator::add() {          void RpnCalculator::subtract() {
    int v1 = getX();                int v1 = getX();
    values.pop();                   values.pop();
    int v2 = getX();                int v2 = getX();
    values.pop();                   values.pop();
    values.push(v1 + v2);            values.push(v2 - v1);
}                                    }
```

If this were the only duplication, then it might be ok. We have multiply and divide as well for our first release. They will be exactly the same as well. Four copies of the same code is not a good idea. It leads to duplicated maintenance. It is also error prone. For example, what happens if someone forgets to pop the second value? The operation will appear to work but the next operation will be left with the wrong values on the stack. This is a problem with attributes in general. Every attribute is a means of one method causing another method to fail. In this case, the methods are not exact duplicates, which means we might miss it as well as tools that look for duplicated code.

#### 4.14 Dreaded Duplication or DRY

A fundamental principle of software development is Don't Repeat Yourself – DRY. We have duplication between add and subtract, but more generally we'll have this problem across all operations that consume two values and produce a single value.

There are several ways to fix this problem, here are a few ideas:

- Ignore it, we're paid by the number of lines so duplication is a good thing
- Write a utility method that does everything but + or – and pass in a pointer to a function which does just the + or –, e.g.:

```
void binaryOp(RpnStack &values, int (*pf)(int, int)) {
    int v1 = values.top();
    values.pop();
    int v2 = values.top();
    values.pop();
    int result = pf(v2, v1);
    values.push(result);
}
int addIt(int lhs, int rhs) {
    return lhs + rhs;
}
void RpnCalculator::add() {
    binaryOp(values, addIt);
}
```

- Create a class that does + or – instead of a pointer to a member function:

```
class BinaryOperator {
    virtual int calculate(int lhs, int rhs) = 0;
};
class Add : public BinaryOperator {
    int calculate(int lhs, int rhs) { return lhs + rhs; }
};
class Subtract : public BinaryOperator {
    int calculate(int lhs, int rhs) { return lhs - rhs; }
};
void binaryOp(RpnStack &values, BinaryOperator &op) {
    int v1 = values.top();
    values.pop();
    int v2 = values.top();
    values.pop();
    int result = op.calculate(v2, v1);
    values.push(result);
}
```

This last option has a name; it is called the Template Method Pattern. Its name is unfortunate as it was named before template methods were added to C++. Even so, this last option is moving in a good direction. However, notice that we'll have to do something special for each of the different kinds of operators. The method `binaryOp` above does not belong to `RpnCalculator`. In fact, it demonstrates feature envy. In this case, however, it does not make sense to move the method into `RpnStack` as in the last case with `top()` and `pop()`, but we can still look at feature envy for ideas. There are three suggestions to resolve feature envy:

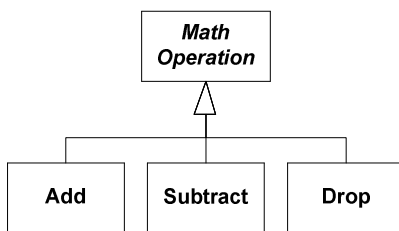
- Move method, which is what we did with RpnStack
- Extract method, which is what binaryOp is an example of
- Extract class, which is what we are going to do

The calculator has a number of operations. Each operation:

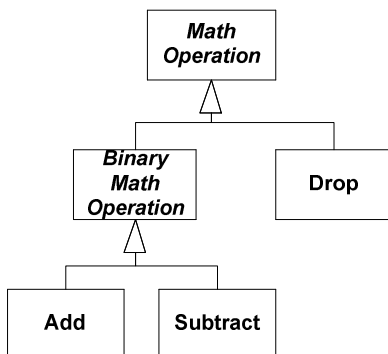
- Consumes a number of operands (0 or more)
- Optionally performs a calculation
- Produces a number of operands (0 or more)

We need to put the responsibility where it belongs and it best belongs with the operation. Feature envy suggests extracting a class, and general assignment of responsibility suggests what that class should do. We can also look to the GRASP patterns for inspiration. One of the GRASP patterns is “information expert” – put the responsibility with the thing that has the data. This does not exactly match because we are not talking about who has the data so much as the thing that selects the variation – the operation. Another GRASP pattern might match better: protected variation – put the responsibility where it varies.

Notice that we can do this for each of our operations. If each of our operations becomes a class and we treat them as variations on a general theme, then we are using a design pattern known as the Strategy Pattern. This might look something like this:



This is a good intermediate step. This alone does not solve the duplication problem. If we combine this with the example of the binary operator we get this:



Wow, that is a lot to do. How can we possibly do this while keeping our tests passing?

#### 4.14.1 Extract Classes

We start by extract classes, one for each existing operation. Notice that we’ll be doing this by refactoring. When we are done, we might consider moving automated checks around.

Since we are taking an existing member function and we want to extract it to another class, a good first start is to simply extract a local function and get it to compile:

```
void addIt(RpnStack &values) {
    int v1 = values.top();
    values.pop();
    int v2 = values.top();
    values.pop();
    values.push(v1 + v2);
}

void RpnCalculator::add() {
    addIt(values);
}
```

Now that we've made this method have no direct dependency on the RpnCalculator class, we can easily extract a class called Add right from the add method:

#### **Add.h**

```
#pragma once
#ifndef ADD_H_
#define ADD_H_

class RpnStack;

class Add {
public:
    void perform(RpnStack &values);
};

#endif
```

Notice the name change from addIt to perform. The method name makes more sense for all operations.

#### **Add.cpp**

```
#include "Add.h"
#include "RpnStack.h"

void Add::perform(RpnStack &values) {
    int v1 = values.top();
    values.pop();
    int v2 = values.top();
    values.pop();
    values.push(v1 + v2);
}
```

And a final update to RpnCalculator:

```
#include "Add.h"
void RpnCalculator::add() {
    Add op;
    op.perform(values);
}
```

```
}

```

All tests should pass.

#### 4.14.2 Keeping it the same

Now that we have one example of what all operations will look like in the Add class, we can extract an interface and then use that as the bases for the other operations. Here is such an interface extracted:

##### **MathOperation.h**

```
#pragma once
#ifndef MATHOPERATION_H_
#define MATHOPERATION_H_

class RpnStack;

class MathOperation {
public:
    virtual ~MathOperation() = 0;
    virtual void perform(RpnStack &values) = 0;
};

#endif

```

The destructor is a so-called special member function. We'll need to define it:

##### **MathOperation.cpp**

```
#include "MathOperation.h"

MathOperation::~MathOperation() {
}

```

Now that this interface exists, we can update add to use it:

```
#include "MathOperation.h"

class Add : public MathOperation {
public:
    void perform(RpnStack &values);
};

```

#### 4.14.3 Updating Subtract

Making the change for Subtract is nearly a copy of Add:

##### **Subtract.h**

```
#pragma once
#ifndef SUBTRACT_H_
#define SUBTRACT_H_

#include "MathOperation.h"

class Subtract : public MathOperation {
public:

```

```
    void perform(RpnStack &values);
};

#endif

Subtract.cpp
#include "Subtract.h"
#include "RpnStack.h"

void Subtract::perform(RpnStack &values) {
    int v1 = values.top();
    values.pop();
    int v2 = values.top();
    values.pop();
    values.push(v2 - v1);
}

RpnCalculator::subtract
#include "Subtract.h"
void RpnCalculator::subtract() {
    Subtract op;
    op.perform(values);
}
```

#### 4.14.4 Drop

The changes for drop follow the same pattern:

##### *Drop.h*

```
#pragma once
#ifndef DROP_H_
#define DROP_H_

#include "MathOperation.h"

class Drop : public MathOperation {
public:
    void perform(RpnStack &values);
};

#endif
```

##### *Drop.cpp*

```
#include "Drop.h"
#include "RpnStack.h"

void Drop::perform(RpnStack &values) {
    values.pop();
}
```



**RpnCalculator::drop**

```
#include "Drop.h"
void RpnCalculator::drop() {
    Drop op;
    op.perform(values);
}
```

**4.14.5 Factorial**

And finally, the changes for factorial are also the same thing. Practicing class extraction is a useful thing. You'll often do it in new development. It tends to be even more useful when working with legacy code. You want to get to a point where doing it is second nature; practice, practice, practice.

**Factorial.h**

```
#pragma once
#ifndef FACTORIAL_H_
#define FACTORIAL_H_

#include "MathOperation.h"

class Factorial : public MathOperation {
public:
    void perform(RpnStack &values);
};

#endif
```

**Factorial.cpp**

```
#include "Factorial.h"
#include "RpnStack.h"

void Factorial::perform(RpnStack &values) {
    int operand = values.top();
    values.pop();
    if (operand >= 0) {
        int result = 1;
        while (operand > 1)
            result *= operand--;
        values.push(result);
    }
}
```

**4.15 Removing Duplication**

Now we have enough in place to remove the duplication between add and subtract. While this is overkill for 2 copies, we're going to have many more than 2, so this effort will be time well spent.

We need a class that:

- Acquires two values from the stack

- Defers the actual calculation to a subclass
- Stores the result

There are at least a few paths we could take to getting this done:

- Refactor – Copy code from Add or Subtract and change the inheritance structure in place. The existing tests will keep us on track.
- Check First – Develop a number of automated checks, one for each of the requirements of this base class.
- Hybrid – Create the class using a copy of code and write all of the tests we'll need anyway.

Any of these ways will get us where we need to go. In class people generally prefer the check first approach, so that's what I'll go with here.

#### 4.15.1 It consumes two values

Here's a quick recap of an earlier suggestion:

First, the BinaryOperator class:

```
class BinaryOperator {
    virtual int calculate(int lhs, int rhs) = 0;
};
class Add : public BinaryOperator {
    int calculate(int lhs, int rhs) { return lhs + rhs; }
};
```

Notice that in this simple hierarchy, the base class is abstract and the derived class has one method. The one method, calculate, has two parameters. Here's where we use it:

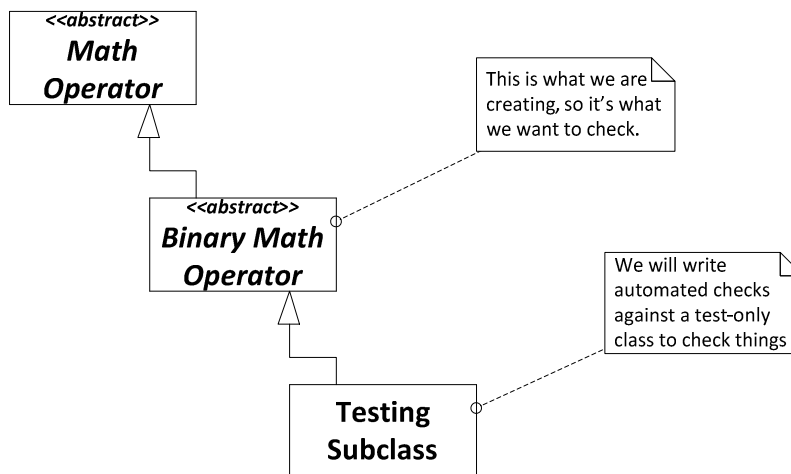
```
01: void binaryOp(RpnStack &values, BinaryOperator &op) {
02:     int v1 = values.top();
03:     values.pop();
04:     int v2 = values.top();
05:     values.pop();
06:     int result = op.calculate(v2, v1);
07:     values.push(result);
08: }
```

The calculate() message is sent on line 6. Notice that the object receiving the calculate() message does not have access to the RpnStack, just two int parameters.

Here are our challenges:

- We are going to create an abstract class, how can we check an abstract class?
- The real subclasses don't yet exist – that is, the Add and Subtract do exist, but they do not yet use the abstract base class.
- Even if these classes did exist, they do not in general have access to the RpnStack, but we want to check its size. Unfortunately, the whole sequence both consumes and produces values, if we check the size, the check is indirect.

This is a common problem. It is one of the costs of inheritance; testing extensions of abstract classes can require more work. However, we can create a testing subclass to fix the problem:



What does this offer?

- We can create the testing subclass before changing Add or Subtract; thereby isolating the work until we have what we think is a finished product.
- We can give our testing subclass access to things that a normal production class would be able to access, opening up possibilities not reasonably available to a production subclass.

However, it does have some cost:

- It takes getting used to – this is resolved with time and experience.
- It might seem like we are testing test code as opposed to production code. If that's actually the case, then we've failed in our efforts, so we must be vigilant to this possibility.
- We have to create a test-only class, this might seem like work. There are some tools for doing this in C++; some free, some commercial. Rather than introduce more moving parts, we'll just create them by hand.

Even with these disadvantages, this is still a solid option. More importantly, this is something you'll use often in adding automated checks to legacy code, so it's good to practice this technique.

Here's our first test for this new BinaryMathOperation class:

```

01: #include <CppUTest/TestHarness.h>
02:
03: #include "BinaryMathOperation.h"
04: #include "RpnStack.h"
05:
06: TEST_GROUP(BinaryMathOperationShould) {
07:     RpnStack *values;
08:     void setup() {
09:         values = new RpnStack;
10:         values->push(4);
11:         values->push(2);
12:     }
13:     void teardown() {
14:         delete values;
  
```

```

15:     }
16: };
17:
18: struct BinaryMathOperationSpy : public BinaryMathOperation {
19:     BinaryMathOperationSpy(RpnStack &values)
20:         : values(values), size(-1) {}
21:     int calculate(int lhs, int rhs) {
22:         size = values.size();
23:         return 13;
24:     }
25:     RpnStack &values;
26:     int size;
27: };
28:
29: TEST(BinaryMathOperationShould, ConsumeTwoValues) {
30:     BinaryMathOperationSpy spy(*values);
31:     spy.perform(*values);
32:     LONGS_EQUAL(0, spy.size);
33: }

```

Line	Description
01 - 16	Mostly boilerplate code. Create a test group. These tests will need an RpnStack so store it as a pointer, allocate and release it in the TEST_GROUP. Additionally, put two known values on it. These values will be used to check that the right values are getting sent in the correct order to the eventual subclasses.
18	Introduce a testing subclass. This is called a Spy because it will be watching what happens and recording. The automated check will use this class to record what happens and then verify that what was supposed to happen actually happened. As a testing subclass it must inherit from BinaryMathOperation – the class we are really trying to create. Rather than use a class use a struct so that everything is public by default.
19	This is a constructor. Notice that this class requires an instance of an RpnStack? This is the special sauce we add that a normal production subclass would not have. Doing this allows this testing subclass to record things that normally would not be accessible to a production subclass.
20	Use member-wise initialization to hold onto the stack and initialize a size attribute to -1. Since -1 is never a valid stack size, it represents a good initial value. We could have used #include <limits.h> and INT_MIN instead.
21	Subclasses have an extension point called calculate that takes two integer parameters.
22	Our implementation will record the size of the stack just after the call to calculate() from the abstract base class.
23	This method has a return value so I pick one of my favourite numbers 13.
29	Finally an automated check.
30	Create the spy and give it a reference to the RpnStack created just before this code executes (remember, it is allocated in the setup() method and released in

Line	Description
	the teardown() method). The setup method always puts two values on it, so we know at this point the size is 2. The test as written guarantees/controls that.
31	Call perform on the base class. Note: we are trying to verify that the perform method is well behaved. Even though we are sending a message to a testing subclass, the method we invoke is in the base class. We are actually exercising code in the base class, which is our ultimate goal.
32	The stack started with a size of 2. The spy records the size upon entry into the calculate() method. Presumably, the lhs and rhs parameters will be the values removed from the stack, but that is another automated check we have yet to write. If the size is 0, then, apparently, two values were taken off the stack

Now that we have this complex test written, we have to write some code to get it to compile:

### **BinaryMathOperation.h**

```

01: #pragma once
02: #ifndef BINARYMATHOPERATION_H_
03: #define BINARYMATHOPERATION_H_
04:
05: #include "MathOperation.h"
06:
07: class BinaryMathOperation: public MathOperation {
08: public:
09:     void perform(RpnStack &values);
10:     virtual int calculate(int lhs, int rhs) = 0;
11: };
12:
13: #endif

```

BinaryMathOperation is our intermediate, abstract base class. All operations that take two parameters and produce a single result will inherit from this base class. Examples might include: add, subtract, multiply, divide, less than, greater than, y to the x, etc. Line 9 is the method all MathOperation classes must write. That's what we are checking now. All subclasses will ultimately need to write a calculate method to do the actual work of calculation. Notice that this is a pure virtual method. There is no reasonable default behavior and we want subclasses to write this member function. That is exactly what pure virtual methods do, along with making the class itself abstract.

To get it to link:

```

#include "BinaryMathOperation.h"

#include "RpnStack.h"

void BinaryMathOperation::perform(RpnStack &values) {
}

```

Now that the code compiles and links, you can check to see that the automated check is failing. Next, to get it to pass:

```
void BinaryMathOperation::perform(RpnStack &values) {
    values.pop();
    values.pop();
    calculate(0, 0);
}
```

This is enough code to get our first automated check passing. We could have copied code from Add, I somewhat arbitrarily chose not to but doing so would have been fine. So long as we verify all of this method's responsibilities with automated checks, we have lots of options on getting to the final version.

#### 4.15.2 It calls an extension point with the correct parameters

One of the responsibilities of our method is to actually call an extension point, `calculate()`. We have done this indirectly because our last check would fail if it were not called. Even so, we need to verify that the parameters are correct.

The `setup()` method pushes two values, 4 and 2. The value pushed first should be treated as the left hand side value while the value pushed second should be treated as the right hand side value. We can make a few changes to our existing test subclass to support this idea or we can create an entirely new testing subclass.

If I use a language with better tool support or if I have some kind of library that supports making these so-called test doubles (though the libraries are typically called mocking libraries), then I make more fine-grained classes on a per-test basis. Since I am and writing my test classes by hand I'll tend towards more chunky testing classes. With that in mind, here's an updated version of our testing subclass:

```
struct BinaryMathOperationSpy : public BinaryMathOperation {
    BinaryMathOperationSpy(RpnStack &values)
        : values(values), size(-1), actualLhs(0), actualRhs(0) {}
    int calculate(int lhs, int rhs) {
        size = values.size();
        actualLhs = lhs;
        actualRhs = rhs;
        return 13;
    }
    RpnStack &values;
    int size;
    int actualLhs;
    int actualRhs;
};
```

This update introduces three changes:

- Introduction of two attributes, `actualLhs` and `actualRhs`
- Both of those new attributes are initialized to 0
- Both of those attributes are updated in the `calculate` method with the actual parameters passed in at runtime.

Notice that these changes leave the class working fine for the existing test. Now a test to use this updated class:

```
TEST(BinaryMathOperationShould, SendTwoParametersInCorrectOrder) {
    BinaryMathOperationSpy spy(*values);
    spy.perform(*values);
    LONGS_EQUAL(4, spy.actualLhs);
    LONGS_EQUAL(2, spy.actualRhs);
}
```

Run this test, it fails. Now to update the production code to passing:

```
void BinaryMathOperation::perform(RpnStack &values) {
    int rhs = values.top();
    values.pop();
    int lhs = values.top();
    values.pop();
    calculate(lhs, rhs);
}
```

Back to all checks passing.

#### 4.15.3 It stores the result

Finally, the result returned from calculate must be stored. To get the original version of the testing subclass to compile without warnings, the calculate method had to return a value. I chose 13, so now we can check for that:

```
TEST(BinaryMathOperationShould, StoreCalculatedResult) {
    BinaryMathOperationSpy spy(*values);
    spy.perform(*values);
    LONGS_EQUAL(13, values->top());
}
```

This test initially fails, a quick update to perform() fixes it;

```
void BinaryMathOperation::perform(RpnStack &values) {
    int rhs = values.top();
    values.pop();
    int lhs = values.top();
    values.pop();
    int result = calculate(lhs, rhs);
    values.push(result);
}
```

Success.

#### 4.15.4 Updating Add and Subtract

Now we can update Add and Subtract, one at a time, while keeping the tests passing:

**Add.h**

```
#include "BinaryMathOperation.h"

class Add : public BinaryMathOperation {
public:
    int calculate(int lhs, int rhs);
};
```

**Add.cpp**

```
#include "Add.h"

int Add::calculate(int lhs, int rhs) {
    return lhs + rhs;
}
```

Check that your checks are still passing.

**Subtract.h**

```
#include "BinaryMathOperation.h"

class Subtract : public BinaryMathOperation {
public:
    int calculate(int lhs, int rhs);
};
```

**Subtract.cpp**

```
#include "Subtract.h"

int Subtract::calculate(int lhs, int rhs) {
    return lhs - rhs;
}
```

Check your checks are still passing.

#### 4.15.5 Not updating drop

Notice that drop is not included in this update. It does not behave the same as Add or Subtract. Also, there is no need for an intermediate base class for it. The hierarchy is asymmetrical – this is a sign of a healthy hierarchy. Few problems collapse into a fully-balance tree. If we find duplication later than warrants another intermediate base class we'll add it then.

#### 4.16 All those methods

The next thing might seem like adding multiply and divide. However, notice how the API of this class continues to grow? It is an open-ended class that has too much responsibility. Now is a good time to lock down the API. We know we are going to continue to add new operations. The API and underlying class keeps changing. It is time to close this class to changes but open up the system to adding new operations easier. We already have much of the required infrastructure in place. We need to come up with how we want the new class to look, using an automated check, and then migrate tests over to that new approach – preferably one at a time.

##### 4.16.1 An example based migration

We'll begin by copying an existing test and making it look like we want it to look. Notice we are not getting rid of the old test. We want the new and old to coexist until we have the new in place.

```
TEST(RpnCalculatorShould, AddTwoNumbers_v2) {
    calculator->enter(30);
    calculator->enter(4);
}
```



```
calculator->execute("+");
LONGS_EQUAL(34, calculator->getX());
}
```

This won't compile yet because the method does not exist. Notice the name. It needs to be unique. Once this test works, I will remove the original test. In general, refactoring starts with a "copy" operation and not a "move" operation.

To get this to compile, link and pass:

**Declare new member function**

```
#include <string>

class RpnCalculator {
public:
    ...
    void execute(const std::string &operatorName);
}
```

And a "complete" implementation – complete for all the tests requiring it that is:

```
void RpnCalculator::execute(const std::string &operatorName) {
    add();
}
```

Verify that this gets the test passing and then:

- Delete the original test
- Rename the new test by removing the \_v2

Now update the test "AddWhenTheresASingleValue" to use the new approach:

```
TEST(RpnCalculatorShould, AddWhenTheresASingleValue) {
    calculator->enter(4);
    calculator->execute("+");
    LONGS_EQUAL(4, calculator->getX());
}
```

Now that we have migrated all uses of add() we can remove it but caution. If we were in a legacy setting where the class we are changing is used by other applications in a library, you cannot so easily remove public methods. In our case, we know all the uses of our class, so we have the freedom (luxury?) of being able to change its public API. In a legacy environment, it's more of a migration – or you do it all yourself. In any case, let's make that transition after we've completed writing the execute() method.

#### 4.16.2 Migrating the subtract() method

This is more of the same. However, we could change an existing test rather than copy. The method we want to call is already there, we just want to use it. So updating the test in place seems OK to me:

```
TEST(RpnCalculatorShould, SubtractTwoNumbers) {
    calculator->enter(30);
    calculator->enter(4);
    calculator->execute("-");
    LONGS_EQUAL(26, calculator->getX());
}
```

Now you have a failing test, time to fix it:

```
void RpnCalculator::execute(const std::string &operatorName) {
    if(operatorName == "+")
        add();
    else if(operatorName == "-")
        subtract();
}
```

And like that, we are back to passing. While I did change a test in place, notice I only changed one. This is a simple enough example we could change everything in one fell swoop. However, you tend to continue as you start, so no better time than the present to do things carefully. In your real projects the code will be more complex and probably less familiar, so learning how to do things carefully from the beginning is the way to go.

There is only one more test using `subtract()`, `SubtractWhenTheresASingleValue`. Update it and get back to passing.

#### 4.16.3 Finish the transformation

Continue this until you've also updated the tests using `drop()` and `factorial()`. You'll end up with something like the following:

```
void RpnCalculator::execute(const std::string &operatorName) {
    if(operatorName == "+")
        add();
    else if(operatorName == "-")
        subtract();
    else if(operatorName == "!")
        factorial();
    else if(operatorName == "drop")
        drop();
}
```

Now you can safely make the methods `add()`, `subtract()`, `factorial()`, `drop()` private. But do those methods even need to be there in the first case? How about we “inline” those methods – simply copy their contents into the `execute()` method and remove them altogether:

```
#include "Add.h"
#include "Subtract.h"
#include "Drop.h"
#include "Factorial.h"
void RpnCalculator::execute(const std::string &operatorName) {
    if (operatorName == "+") {
        Add op;
        op.perform(values);
    } else if (operatorName == "-") {
        Subtract op;
        op.perform(values);
    } else if (operatorName == "!") {
        Factorial op;
        op.perform(values);
    } else if (operatorName == "drop") {
```

```

    Drop op;
    op.perform(values);
}
}

```

Notice that we have much of the work necessary to accommodate polymorphism across a hierarchy of math operations but we are not actually using it? What is more important? Getting an ever-growing API locked down or improving the internal design? I'm not sure there's a clear winner. My preference is locking down the API first.

#### 4.17 Type un-safe

A nice thing about the perform method is that it simplifies the API, so problem solved, right? The first rule of solving problems according to Weinberg is "Every solution introduces problems." What happens when you attempt to execute an unknown operation? In our case, the code silently does nothing. I prefer "fail-fast", so let's add a check that expects such a situation to generate an "UnknownMathOperationException":

##### *The Test*

```

#include "UnknownMathOperationException.h"
TEST(RpnCalculatorShould, ThrowAnExceptionForUnknownOperation) {
    try {
        calculator->execute("--unknown--");
        FAIL("Should have thrown UnknownMathOperationException");
    } catch(UnknownMathOperationException &e) {
        CHECK(1);
    }
}

```

And to get it to compile:

##### *UnknownMathOperationException.h*

```

#pragma once
#ifndef UNKNOWNMATHOPERATIONEXCEPTION_H_
#define UNKNOWNMATHOPERATIONEXCEPTION_H_

#include <exception>

class UnknownMathOperationException: public std::exception {
public:
    UnknownMathOperationException();
    virtual ~UnknownMathOperationException() throw();
};

#endif

```

##### *UnknownMathOperationException.cpp*

```

#include "UnknownMathOperationException.h"

UnknownMathOperationException::UnknownMathOperationException() {
}

```

```
UnknownMathOperationException::~UnknownMathOperationException() throw
() {
}
```

To get this test to pass, we need to update the execute method:

```
#include "UnknownMathOperationException.h"
void RpnCalculator::execute(const std::string &operatorName) {
    if (operatorName == "+") {
        ...
    } else {
        throw UnknownMathOperationException();
    }
}
```

This is an example of closing down an API to changes at the expense of runtime checking. This is often a hard sell. Try it and see if it fits. It is often overkill. If you have a small number of named methods, 5, 10, maybe having a larger API makes sense. Eventually, the changing API becomes a burden. In our case we only have 4, but we know two things:

- There are more on the way – even in this first “release”
- We are going to want macros as well; this API supports the execution of macros in the same way as the built-in operations, which was a stated requirement.

#### 4.18 Long Method

A long method might be one that has a large number of lines. While the execute() method is not long in terms of number of lines yet, it will be. Another interpretation of Long Method is one that exists at different levels of abstraction. Finally, a method that does three different things is long even if it doesn't have a lot of lines:

- Mapping – a string to a MathOperation
- Construction – instantiates a MathOperation to perform the work
- Delegation – sends a message to a MathOperation to actually do the work

Independent of that, however, is this notion of mapping from a string to an object. This is a common problem that arises at the boundary of a system. The RpnCalculator class is a controller object; it waits for systems events or messages from actors. There is yet another design pattern to solve this exact problem: Abstract Factory ([http://en.wikipedia.org/wiki/Abstract\\_factory\\_pattern](http://en.wikipedia.org/wiki/Abstract_factory_pattern)). The abstract factory pattern typically serves to build a suite of objects for a given environment. In our case we want to select from one of many objects for a string. We can move towards such a solution by first extracting the part of the method that does the selection from the part that does the execution. Once we've done that, we can extract a class to do that work.

Here is a stab at such a refactoring:

**Copy the entire method and make it a function:**

```
void findOperationNamed(const std::string &operatorName) {
    if (operatorName == "+") {
        Add op;
        op.perform(values);
    }
}
```

```
} else if (operatorName == "-") {
    Subtract op;
    op.perform(values);
} else if (operatorName == "!") {
    Factorial op;
    op.perform(values);
} else if (operatorName == "drop") {
    Drop op;
    op.perform(values);
} else {
    throw UnknownMathOperationException();
}
}
```

This won't compile as it refers to values, which is member data. However don't want to both select and perform, so remove all references to that member data:

```
MathOperation& findOperationNamed(const std::string &operatorName){
    if (operatorName == "+") {
        Add op;
        return op;
    } else if (operatorName == "-") {
        Subtract op;
        return op;
    } else if (operatorName == "!") {
        Factorial op;
        return op;
    } else if (operatorName == "drop") {
        Drop op;
        return op;
    } else {
        throw UnknownMathOperationException();
    }
}
```

This function compiles with warnings (or maybe errors – somewhat compiler dependent). The code returns reference to temporary objects. In the first project we saw a quick way to get this working using the static keyword. Let's do that here:

```
MathOperation& findOperationNamed(const std::string &operatorName){
    if (operatorName == "+") {
        static Add op;
        return op;
    } else if (operatorName == "-") {
        static Subtract op;
        return op;
    } else if (operatorName == "!") {
        static Factorial op;
        return op;
    } else if (operatorName == "drop") {
        static Drop op;
        return op;
    } else {

```

```

        throw UnknownMathOperationException();
    }
}

```

Now we can update the execute() method to use it:

```

void RpnCalculator::execute(const std::string &operatorName) {
    MathOperation &op = findOperationNamed(operatorName);
    op.perform(values);
}

```

All checks should be passing. This method is now closed to new operations. Meaning, as new math operations are added, this code does not need to change. The code in the extracted function does, but we'll fix that as well. Before we do, however, it is time to move that code into its own class.

#### 4.19 A Concrete Factory

As written, the findOperationNamed() function can become a member function simply by copying it as is. You might be tempted to make it a static method, but do not. Generally, static methods make writing automated checks more difficult. Why? If you need to swap one out at runtime for a particular check, it's not possible in the language to do so. You can link in a different version of the static method and there are other things you can do as well. Even so, what might seem convenient is really asking for trouble as the road to hell (supporting legacy code) is paved with conveniences.

Here's a simple extract class refactoring applied to that code:

##### *MathOperationFactory.h*

```

#pragma once
#ifndef MATHOPERATIONFACTORY_H_
#define MATHOPERATIONFACTORY_H_

#include <string>
class MathOperation;

class MathOperationFactory {
public:
    MathOperationFactory();
    virtual ~MathOperationFactory();
    virtual MathOperation&
        findOperationNamed(const std::string &name);
};

#endif

```

##### *MathOperationFactory.cpp*

```

#include "MathOperationFactory.h"

#include "UnknownMathOperationException.h"
#include "Add.h"
#include "Subtract.h"
#include "Drop.h"

```

```

#include "Factorial.h"

MathOperationFactory::MathOperationFactory() { }
MathOperationFactory::~MathOperationFactory() { }

MathOperation& MathOperationFactory::findOperationNamed(
    const std::string &operatorName) {
    if (operatorName == "+") {
        static Add op;
        return op;
    } else if (operatorName == "-") {
        static Subtract op;
        return op;
    } else if (operatorName == "!") {
        static Factorial op;
        return op;
    } else if (operatorName == "drop") {
        static Drop op;
        return op;
    } else {
        throw UnknownMathOperationException();
    }
}

```

#### 4.19.1 Actually using the factory

The class is easily extracted – we copied some code, but it while it compiles and links, it is not getting used. This factory class should be a dependent object in the RpnCalculator class. Traditionally, the abstract factory pattern, which we are yet to fully follow since we do not have an abstract class, has variations. We don't have a need for variations yet, but it might come up. To follow that pattern, we should hook the factory back into the calculator in a way that allows for overriding.

We do not have to do this, however all the patterns in the Design Patterns book were found in real applications so to not follow something that has worked seems a bit “not build here” ish. An intermediate compromise is to store a pointer and use new and delete for now. Later, if we find a need for variations on the factory, we can introduce an interface and change the pointer type.

With this in mind, here is an updated version of RpnCalculator:

##### ***RpnCalculator.h***

```

class MathOperationFactory;

class RpnCalculator {
    ...

private:
    RpnStack values;
    MathOperationFactory *factory;

```

***RpnCalculator.cpp (the whole thing as it has shrunk)***

```

#include "RpnCalculator.h"
#include "MathOperationFactory.h"
#include "MathOperation.h"
RpnCalculator::RpnCalculator()
    : factory(new MathOperationFactory) {
}

RpnCalculator::~RpnCalculator() {
    delete factory;
}

void RpnCalculator::enter(int value) {
    values.push(value);
}

int RpnCalculator::getX() const {
    return values.top();
}

void RpnCalculator::execute(const std::string &operatorName) {
    MathOperation &op = factory->findOperationNamed(operatorName);
    op.perform(values);
}

```

Notice that because we chose a pointer, the header file only mentions the type in a forward declare rather than including the header file. That's a good thing. Minimizing header file inclusion in other header files leads to more maintainable systems.

**4.20 Retargeting Automated Checks**

Something that should happen as you refactor code with existing automated checks is checking to see if they are still against the right class. How do you know?

In general, the “closer” an automated check is to the thing it is checking, the simpler and more focused it can be. For example, consider the following automated check:

```

TEST(RpnCalculatorShould, AddWhenTheresASingleValue) {
    calculator->enter(4);
    calculator->execute("+");
    LONGS_EQUAL(4, calculator->getX());
}

```

What is this checking exactly? It appears to be checking that there will be values available to the “+” Math Operation even when values have not been entered. Remember that we introduce the RpnStack class to handle that. In fact, we have the following test to make sure that is the case:

```

TEST(RpnStackShould, HaveATopAfterCreation) {
    LONGS_EQUAL(0, values->top());
}

```



Since we know that an `RpnStack` will always have values and we know that the `RpnCalculator` in fact uses `RpnStack`, are we safe in assuming that the above check is no longer necessary?

The answer is, “it depends.” If you view what you are doing as strictly black-box, then you need both checks. This is a tricky question with no clear answer. If we treat what we are doing as a way to:

- Drive development
- Reduce the risk of releasing a defect into the wild

Then it might be OK.

If you think we should have the check, then it seems we should have that check for every operation, which is currently: add, subtract, drop and factorial. In fact we should have it for every future operation as well. Maybe a reasonable middle ground is one or two smoke-like tests that verify the basic feature is in fact there but not do it for all operations.

What about moving the checks for each operation into an a specific source folder for each Operation? E.g., `AddShould`, `SubtractShould`, `FactorialShould`, `DropShould`?

This seems like the right place to put it, however what about making sure the actual operations are in the factory? Should that be in the factory, the calculator or with the checks on the math operation itself? Is the calculator responsible for knowing which operations it supports? Does it even know its supported operations?

There are no clear answers to these questions. If we check in the first place, that’s a great thing. If we have duplicated checks, that is probably better than no checks. Duplicated checks, however, represent waste or inertia against refactoring. Checking the same thing multiple times really doesn’t add to a sense of security about the system.

Even though there is no clear answer, I want to put a stake in the ground for this project. Here are a few recommendations:

- Examples created at the beginning of the overall effort (sprint) will have direct corollaries in automated checks. Essentially we will treat these as Rejection Checks (acceptance tests in older terminology).
- We will create automated checks for each production class. We are currently missing several.
- One case where we may deviate is not writing direct checks against exception classes. If those classes have logic, sure. Right now we simply instantiate and throw them. That’s not worth checking directly; it’s built into the language.

This will lead to duplication of some checks but in many cases we might be able to more directly check the inner classes differently, so they won’t be direct copies.

Given this set of rules, it is now time to remediate checks. In the current solution, we are missing direct checks against the following classes: `Add`, `Drop`, `Factorial`, `MathOperation`, `MathOperationFactory`, and `Subtract`. That’s quite a list, no better time than now to plant a tree.

### 4.20.1 Add

Before we write an automated check for add, we need to ask a question: Should we check against the API published in `MathOperation` or `BinaryMathOperation`? The only thing that is essential is the `calculate()` method, so that seems more direct. It happens to be public, so we can do that. Is that a reasonable thing to have done? Personally, I like public extension points precisely because I can check closer to the thing I care about. That being said, this is a point of contention for many people. In C++ I could have made the method public, private or protected; virtual-ness is orthogonal to access. If the method were protected, would I consider making it public to make checking easier? In C++ I would make the methods public. In Java I would not have to because access rules are different. So I am going to check Add through its `calculate` method because I can:

#### *AddShould.cpp*

```
#include <CppUTest/TestHarness.h>

#include "Add.h"
TEST_GROUP(AddShould) {
    Add op;
};

TEST(AddShould, AddTwoNumbers) {
    LONGS_EQUAL(12, op.calculate(5, 7));
}
```

This is simple and direct. What about verifying that Add is registered in the Factory? This is again something for which there is no definitive answer. This is something worth checking. If we put the check here, we have all of the checks associated with the work on Add in one place. We could put the check in another place, say:

`RegisteredOperationsAre.cpp`. That opens that file up to continual changes and, more importantly, it becomes a bottleneck. We could create a second check source file such as `AddRegistered.cpp`. This keeps it small and focused and it really is a separate concern. This increased the number of files but once written, that file won't need to be recompiled say if we decide to add another check for Add. That seems like the way to go:

#### *AddShouldBeRegistered.cpp*

```
#include <CppUTest/TestHarness.h>

#include "Add.h"
#include "MathOperation.h"
#include "MathOperationFactory.h"

#include <typeinfo>
TEST_GROUP(AddShouldBeRegistered){};

TEST(AddShouldBeRegistered, IsIt) {
    MathOperationFactory factory;
    MathOperation &op = factory.findOperationNamed("+");
    CHECK(typeid(op) == typeid(Add));
}
```

This check verifies that Add is registered as “+” and that the actual type is correct. This may seem like a bit of duplication; however we can either use a base class, utility class or macro to remove all of the duplication. We will do that after in just a bit.

#### 4.20.2 Drop

Unlike Add, Drop directly inherits from MathOperation. To check it, we will need to use an RpnStack.

##### *DropShould.cpp*

```
#include <CppUTest/TestHarness.h>

#include "Drop.h"
#include "RpnStack.h"

TEST_GROUP(DropShould) {
};

TEST(DropShould, DecreaseStackSizeByOne) {
    RpnStack values;
    values.push(4);
    values.push(2);
    Drop op;
    op.perform(values);
    LONGS_EQUAL(1, values.size());
}
```

##### *ShouldBeRegistered.h*

This is a bit unwieldy. Even so, it removes duplication so I’m going with it.

```
#pragma once
#ifndef SHOULDBEREGISTERED_H_
#define SHOULDBEREGISTERED_H_

#include <CppUTest/TestHarness.h>
#include <typeinfo>
#include "MathOperation.h"
#include "MathOperationFactory.h"
#include "UnknownMathOperationException.h"

#define CHECK_REGISTRATION(Classname, OperationName) \
TEST_GROUP(Classname##ShouldBeRegistered){}; \
\
TEST(Classname##ShouldBeRegistered, IsIt) { \
    try { \
        MathOperationFactory factory; \
        MathOperation &op = factory.findOperationNamed(OperationName);\
        CHECK(typeid(op) == typeid(Classname)); \
    } catch(UnknownMathOperationException &e) { \
        FAIL(#Classname " not registered as " OperationName); \
    } \
}
```

```
#endif
```

### *DropShouldBeRegistered.cpp*

The result of using the macro seems good:

```
#include "Drop.h"
#include "ShouldBeRegistered.h"
```

```
CHECK_REGISTRATION(Drop, "drop")
```

You can also update AddShouldBeRegistered to use this macro as well:

```
#include "Add.h"
#include "ShouldBeRegistered.h"
```

```
CHECK_REGISTRATION(Add, "+")
```

#### 4.20.3 Factorial

Like Drop, Factorial requires an RpnStack to check it:

### *FactorialShould.cpp*

```
#include <CppUTest/TestHarness.h>
```

```
#include "Factorial.h"
#include "RpnStack.h"
```

```
TEST_GROUP(FactorialShould) {
};
```

```
TEST(FactorialShould, Calculate120For5) {
    RpnStack values;
    values.push(5);
    Factorial().perform(values);
    LONGS_EQUAL(120, values.top());
}
```

```
TEST(FactorialShould, ConsumeValueWhenValueLessThan0) {
    RpnStack values;
    values.push(-1);
    Factorial().perform(values);
    LONGS_EQUAL(0, values.size());
}
```

```
TEST(FactorialShould, Calculate1For0) {
    RpnStack values;
    Factorial().perform(values);
    LONGS_EQUAL(1, values.top());
}
```

### *FactorialSholdBeRegistered.cpp*

```
#include "Factorial.h"
#include "ShouldBeRegistered.h"
```

```
CHECK_REGISTRATION(Factorial, "!");
```

#### 4.20.4 MathOperation

MathOperation has no implementation. It is the closest thing C++ has to an interface. This is another exception to our simple set of rules.

#### 4.20.5 MathOperationFactory

Given that we've decided to check registration of operations in other places, what is left for this class? Right now it throws an exception when an operation is not found, so that will do for now. As we grow our system, there will be opportunities to add features. As a result, we will add additional automated checks.

**MathOperatorFactoryShould.cpp**

```
#include <CppUTest/TestHarness.h>

#include "MathOperationFactory.h"
#include "UnknownMathOperationException.h"

TEST_GROUP(MathOperationFactoryShould) {
};

TEST(MathOperationFactoryShould,
      ThrowExceptionForUnknownOperation) {
    MathOperationFactory factory;

    try {
        factory.findOperationNamed("--bad_unknown--");
        FAIL("Should have thrown exception");
    } catch(UnknownMathOperationException &) {
        CHECK(1);
    }
}
```

#### 4.20.6 Subtract

There is nothing new for Subtract:

**SubtractShould.cpp**

```
#include <CppUTest/TestHarness.h>

#include "Subtract.h"
TEST_GROUP(SubtractShould) {
};

TEST(SubtractShould, SubtractTwoNumbers) {
    LONGS_EQUAL(-2, Subtract().calculate(5, 7));
}
```

**SubtractShouldBeRegistered.cpp**

```
#include "Subtract.h"
#include "ShouldBeRegistered.h"
```

```
CHECK_REGISTRATION(Subtract, "-")
```

#### 4.21 Adding Multiplication

We have finished remediation for now. Should we have considered doing this as we refactored the code? Yes. This is going to happen. On the plus side, all of the refactoring was supported by existing tests.

The automated check for Multiplication looks much like Add:

***MultiplicationShould.cpp***

```
#include <CppUTest/TestHarness.h>

#include "Multiplication.h"
TEST_GROUP(MultiplicationShould) { };

TEST(MultiplicationShould, MultiplicationTwoNumbers) {
    LONGS_EQUAL(35, Multiplication().calculate(5, 7));
}
```

We need to write the Multiplication class. It looks like Add or Subtract:

***Multiplication.h***

```
#pragma once
#ifndef MULTIPLICATION_H_
#define MULTIPLICATION_H_

#include "BinaryMathOperation.h"

class Multiplication : public BinaryMathOperation {
public:
    int calculate(int lhs, int rhs);
};

#endif
```

***Multiplication.cpp***

```
#include "Multiplication.h"

int Multiplication::calculate(int lhs, int rhs) {
    return lhs * rhs;
}
```

For this to be usable by the calculator, it must be registered:

***MultiplicationShouldBeRegistered.cpp***

```
#include "Multiplication.h"
#include "ShouldBeRegistered.h"

CHECK_REGISTRATION(Multiplication, "*")
```

For this to pass, you will need to update MathOperationFactory.cpp:

```
#include "Multiplication.h"
...
```

```

MathOperation& MathOperationFactory::findOperationNamed(
    const std::string &operatorName) {
    if (operatorName == "+") {
        ...
    } else if (operatorName == "*") {
        static Multiplication op;
        return op;
    } else {
        throw UnknownMathOperationException();
    }
}

```

To finish Multiplication, we need to add the missing checks from the preliminary rejection checks. We wrote this bottom-up and checked both the functionality of Multiplication as well as its registration, so these acceptance checks should pass:

**Added to RpnCalculatorShould.cpp**

```

TEST(RpnCalculatorShould, BeAbleToMultiplyTwoNumbes) {
    calculator->enter(4);
    calculator->enter(4);
    calculator->execute("*");
    LONGS_EQUAL(16, calculator->getX());
}

TEST(RpnCalculatorShould, MultiplyWhenTheresASingleValue) {
    calculator->enter(4);
    calculator->execute("*");
    LONGS_EQUAL(0, calculator->getX());
}

```

## 4.22 Adding Division

Division is similar to the other BinaryOperators, though there is one additional test, divide by Zero. We'll start with the happy path first and add the divide by zero check second:

**DivisionShould.cpp**

```

#include <CppUTest/TestHarness.h>

#include "Division.h"
TEST_GROUP(DivisionShould) {
    Division op;
};

TEST(DivisionShould, DivideTwoNumbers) {
    LONGS_EQUAL(5, op.calculate(15, 3));
}

```

We need to create the Division class for this to pass:

**Division.h**

```

#pragma once
#ifndef DIVISION_H_

```

```

#define DIVISION_H_

#include "BinaryMathOperation.h"

class Division : public BinaryMathOperation {
public:
    int calculate(int lhs, int rhs);
};

#endif

```

**Division.cpp**

```

#include "Division.h"

int Division::calculate(int lhs, int rhs) {
    return lhs / rhs;
}

```

Once this test passes, it's time to add a check for dividing by zero:

**DivisionShould.cpp**

```

#include "DivideByZeroException.h"
TEST(DivisionShould, ThrowExceptionForDivideByZero) {
    try {
        op.calculate(1, 0);
        FAIL("Should have thrown DivideByZeroException");
    } catch (DivideByZeroException &) {
        CHECK(1);
    }
}

```

**DivideByZeroException.h (note, no .cpp)**

```

#pragma once
#ifndef DIVIDEBYZEROEXCEPTION_H_
#define DIVIDEBYZEROEXCEPTION_H_

#include <exception>

class DivideByZeroException : public std::exception {
};

#endif

```

The automated check will fail. Note that how it fails is somewhat platform dependent. On my platform, the entire test suite simply stops running. You will need to update Divide.cpp:

**Updated Division.cpp**

```

#include "Division.h"

#include "DivideByZeroException.h"
int Division::calculate(int lhs, int rhs) {

```



```

    if(rhs == 0)
        throw DivideByZeroException();

    return lhs / rhs;
}

```

Finally, we need to make sure that Divide is registered in the factory:

#### *DivisionShouldBeRegistered.cpp*

```

#include "Division.h"
#include "ShouldBeRegistered.h"

```

```

CHECK_REGISTRATION(Division, "/")

```

Update the factory to make to get your check to pass:

```

#include "Division.h"
...

MathOperation& MathOperationFactory::findOperationNamed(
    const std::string &operatorName) {
    if (operatorName == "+") {
        ...
    } else if (operatorName == "/") {
        static Division op;
        return op;
    } else {
        throw UnknownMathOperationException();
    }
}

```

To complete work on Division, we need to add the missing rejection checks. As with Multiplication, we've worked bottom up so everything should be in place for these checks to immediately pass:

#### *Added to RpnCalculatorShould.cpp*

```

TEST(RpnCalculatorShould, BeAbleToDivideTwoNumbes) {
    calculator->enter(4);
    calculator->enter(4);
    calculator->execute("/");
    LONGS_EQUAL(1, calculator->getX());
}

TEST(RpnCalculatorShould, DivideWhenTheresASingleValue) {
    calculator->enter(4);
    calculator->execute("/");
    LONGS_EQUAL(0, calculator->getX());
}

```

### 4.23 MathOperationFactory refactoring: Storing Math Operations

Notice how you need to keep adding to a long if-then-else structure for every new math operation? That is somewhat error prone, repetitive, and it won't support adding new math operations while the system is running (programmable calculator). There is a simple

class in the standard library that can do this better, a map. In this section you will migrate the current solution to use a map. The number of steps may be more than you expect to keep the code compiling and test passing more often than otherwise:

**Updated MathOperationFactory.h: introduce map**

```
#include <map>
class MathOperation;

class MathOperationFactory {
    ...
private:
    typedef std::map<std::string, MathOperation*> OperationMap;
    OperationMap operationsByName;
};
```

**Update MathOperationFactory constructor**

```
MathOperationFactory::MathOperationFactory() {
    operationsByName["+"] = new Add;
    operationsByName["-"] = new Subtract;
    operationsByName["drop"] = new Drop;
    operationsByName["!"] = new Factorial;
    operationsByName["*"] = new Multiplication;
    operationsByName["/"] = new Division;
}
```

Run your automated check suite. There are many failures; memory leaks are causing those failures. Update the header file by adding another nested typedef:

```
class MathOperationFactory {
    ...
private:
    typedef std::map<std::string, MathOperation*> OperationMap;
    typedef OperationMap::iterator iterator;
    OperationMap operationsByName;
};
```

Update the destructor to use this new typedef:

```
MathOperationFactory::~MathOperationFactory() {
    for(iterator i = operationsByName.begin();
        i != operationsByName.end();
        ++i)
        delete (*i).second;
}
```

Now we can update the findOperationNamed method to use the map:

```
MathOperation& MathOperationFactory::findOperationNamed(
    const std::string &operatorName) {
    iterator candidate = operationsByName.find(operatorName);

    if(candidate == operationsByName.end())
        throw UnknownMathOperationException();
}
```

```
    return *(*candidate).second;
}
```

Now this member function is closed to adding new Math Operations. The constructor is still an issue, but that's the subject of the next section.

#### 4.24 MathOperationFactory refactoring: Automatic Math Operation Registration

Adding a new Math Operation requires several things:

- Create automated check(s) for the behaviour of the math operation
- Create the math operation header and source file
- Create an automated check to verify that the math operation is registered in the factory
- Update the constructor to add the registration
- Write a rejection check to cover any examples provided at the start of the sprint

We can make registration a bit more automatic, but more importantly, we can make it so the factory is closed to change when adding new math operations by using a touch of static magic.

##### 4.24.1 An object for registration

What if we could write a new operation and not have to update the factory? Automatic registration is possible with a few tricks. One of the standard parts of this trick is to have a static variable in the body of a method. We have seen this a few times now. However, now we want to do this for something that is both complex and meant to remain in the code. We will use a series of automated checks to get to an end product:

```
#include <CppUTest/TestHarness.h>

#include "Add.h"
#include "MathOperationRegistrant.h"

TEST_GROUP(MathOperationRegistrantShould) {
};

TEST(MathOperationRegistrantShould, RecordObjectUponCreation) {
    std::string name("+");
    Add *op = new Add;
    MathOperationRegistrant register_add(name, op);

    MathOperationRegistrant::iterator candidate =
        register_add.begin();

    CHECK((*candidate).first == name);
    CHECK((*candidate).second == op);
}
```

To get this to pass:

```
#pragma once
#ifndef MATHOPERATIONREGISTRANT_H_
#define MATHOPERATIONREGISTRANT_H_
```

```

#include <map>
#include <string>
#include "MathOperation.h"

struct MathOperationRegistrant {
    typedef std::map<std::string, MathOperation*> RegistrationMap;
    typedef RegistrationMap::iterator iterator;

    const std::string name;
    MathOperationRegistrant
        (const std::string name, MathOperation *op) : name(name) {
        registered()[name] = op;
    }

    ~MathOperationRegistrant() {
        delete registered()[name];
        registered().erase(name);
    }

    iterator begin() { return registered().begin(); }

private:
    static RegistrationMap &registered() {
        static RegistrationMap registeredOperations;
        return registeredOperations;
    }
};

#endif

```

***static considered harmful***

Static code makes writing reliable automated checks hard. In this case we need a static map somewhere, so the registered() method is fine. We just have too many direct dependencies upon it. This is quickly fixed using a default argument:

```

RegistrationMap &map;
const std::string name;
MathOperationRegistrant(const std::string name,
    MathOperation *op, RegistrationMap &map = registered())
    : map(map), name(name) {
    map[name] = op;
}

~MathOperationRegistrant() {
    delete map[name];
    map.erase(name);
}

iterator begin() { return map.begin(); }

```

Now this code can be overridden if necessary. It turns out it may be necessary, but there's nothing obvious about this. Consider the purpose of this class: operations will be registered here automatically, when we create the math operation factory, it will get its operations from here rather than building them itself. There's no guarantee that the static map stored in `registered()` will in fact be empty when this test executes. We'll eventually come across this problem. Given that we know there's a potential problem, let's fix it now. Note, the only reason I notice this now is from previous experience using this kind of automatic registration feature in other applications in C++.

```
MathOperationRegistrant::RegistrationMap testMap;
MathOperationRegistrant register_add(name, op, testMap);
```

That's it. The change before made it so that none of the methods directly refer to the static reference and allow for dependency injection. This injects a dependent object so that this automated check has no direct connection to the underlying static map. The production code has a single path of execution; it is unaware of the particular map with which it works, just that it works with the same map throughout its life.

We have a few more things to check:

***Do not allow the same name to be used twice***

Since people can now create math operations independently, there's a possibility that two operations will attempt to use the same name. Rather than silently ignore this, let's disallow that:

```
#include "NameInUseException.h"
TEST(MathOperationRegistrantShould, DisallowDuplicatedNames) {
    std::string name("+");
    Add *op = new Add;
    MathOperationRegistrant::RegistrationMap testMap;
    MathOperationRegistrant register_add(name, op, testMap);
    try {
        MathOperationRegistrant r2(name, op, testMap);
        FAIL("Should have thrown an exception");
    } catch(NameInUseException &) {
        CHECK(1);
    }
}
```

For this to compile, we'll need a new exception class:

***NameInUseException.h***

```
#pragma once
#ifndef NAMEINUSEEXCEPTION_H_
#define NAMEINUSEEXCEPTION_H_

#include <exception>

class NameInUseException : public std::exception {};

#endif
```

This test initially fails. We need to update the constructor:

```
#include "NameInUseException.h"
struct MathOperationRegistrant {
    ...
    MathOperationRegistrant(const std::string name,
        MathOperation *op, RegistrationMap &map = registered())
        : map(map), name(name) {
        if(map.find(name) == map.end())
            map[name] = op;
        else
            throw NameInUseException();
    }
}
```

*Do not allow for 0 (null) math operations*

Next, let's make sure that the arguments are OK. First, the passed-in math operation:

```
#include "InvalidArgumentException.h"
TEST(MathOperationRegistrantShould, DisallowANullMathOperation) {
    try {
        MathOperationRegistrant register_add("name", 0);
        FAIL("Should have thrown exception");
    } catch(InvalidArgumentException &e) {
        CHECK(e.name == "mathOperation");
    }
}
```

This needs another exception class:

```
#pragma once
#ifndef INVALIDARGUMENTEXCEPTION_H_
#define INVALIDARGUMENTEXCEPTION_H_

#include <string>
#include <exception>

struct InvalidArgumentException : public std::exception {
    InvalidArgumentException(const std::string name) : name(name) {}
    ~InvalidArgumentException() throw() {}
    const std::string name;
};
```

```
#endif
```

Getting this to pass requires another change to the registrant:

```
#include "InvalidArgumentException.h"
struct MathOperationRegistrant {
    ...
    MathOperationRegistrant(const std::string name,
        MathOperation *op, RegistrationMap &map = registered())
        : map(map), name(name) {
        if(op == 0)
            throw InvalidArgumentException("mathOperation");
    }
}
```

*Operations must have a non-zero length*

```
TEST(MathOperationRegistrantShould, DisallowAZeroLengthName) {
```

```

Add a;
try {
    MathOperationRegistrant register_add("", &a);
    FAIL("Should have thrown exception");
} catch(InvalidArgumentException &e) {
    CHECK(e.name == "name");
}
}

```

And a final update to get this automated check passing:

```

MathOperationRegistrant(const std::string name,
    MathOperation *op, RegistrationMap &map = registered())
    : map(map), name(name) {
    if(op == 0)
        throw InvalidArgumentException("mathOperation");
    if(name.size() == 0)
        throw InvalidArgumentException("name");
}

```

This code has some room for cleaning it up, that is left as an exercise to the reader.

#### 4.24.2 Automatically Register Multiplication

Now we need to try this with an existing math operation. To make this happen, remove the following lines from MathOperationFactory.cpp:

```

#include "Multiplication.h"

operationsByName["*"] = new Multiplication;

```

This will cause the automated check suite to fail with an exception. Update Multiplication:

```

#include "Multiplication.h"

int Multiplication::calculate(int lhs, int rhs) {
    return lhs * rhs;
}

#include "MathOperationRegistrant.h"
MathOperationRegistrant register_multiply("*", new Multiplication);

```

Running your automated checks at this point will produce an interesting result. The problem is the destructor of the MathOperationFactory assumes it is OK to delete everything but it did not allocate everything. So on more change:

```

MathOperationFactory::~MathOperationFactory() {
    MathOperationRegistrant r;
    for(MathOperationRegistrant::iterator i = r.begin();
        i != r.end(); ++i)
        operationsByName.erase((*i).first);
    ...
}

```

This removes any operations registered from the registrant; those objects will be removed automatically when the system shuts down.

Now all of your checks should be passing. It is time to apply this change to all of your math operations so that they are all registered in their source files rather than in the factory.

Once you've made these updates, the factory simplifies:

```
#include "MathOperationRegistrant.h"
MathOperationFactory::MathOperationFactory() {
    MathOperationRegistrant r;
    for(MathOperationRegistrant::iterator i = r.begin();
        i != r.end(); ++i)
        operationsByName[(*i).first] = (*i).second;
}
```

#### 4.24.3 Split registrant

One final change is in order. Rather than have everything for the registrant in its header file, let's split the header into a header and source.

##### *MathOperationRegistrant.h*

```
#pragma once
#ifndef MATHOPERATIONREGISTRANT_H_
#define MATHOPERATIONREGISTRANT_H_

#include <map>
#include <string>
class MathOperation;

struct MathOperationRegistrant {
    typedef std::map<std::string, MathOperation*> RegistrationMap;
    typedef RegistrationMap::iterator iterator;

    RegistrationMap &map;
    const std::string name;
    MathOperationRegistrant(RegistrationMap &map = registered());

    MathOperationRegistrant(
        const std::string name,
        MathOperation *op, RegistrationMap &map = registered());

    ~MathOperationRegistrant();
    iterator begin();
    iterator end();

private:
    static RegistrationMap &registered();
};

#endif
```

##### *MathOperationRegistrant.cpp*

```
#include "MathOperationRegistrant.h"
```



```

#include "MathOperation.h"
#include "NameInUseException.h"
#include "InvalidArgumentException.h"

MathOperationRegistrant::MathOperationRegistrant(
    RegistrationMap &map) : map(map) {}

MathOperationRegistrant::MathOperationRegistrant(
    const std::string name, MathOperation *op, RegistrationMap &map)
    : map(map), name(name) {
    if (op == 0)
        throw InvalidArgumentException("mathOperation");
    if (name.size() == 0)
        throw InvalidArgumentException("name");

    if (map.find(name) == map.end())
        map[name] = op;
    else
        throw NameInUseException();
}

MathOperationRegistrant::~MathOperationRegistrant() {
    delete map[name];
    map.erase(name);
}

MathOperationRegistrant::iterator MathOperationRegistrant::begin() {
    return map.begin();
}

MathOperationRegistrant::iterator MathOperationRegistrant::end() {
    return map.end();
}

MathOperationRegistrant::RegistrationMap
&MathOperationRegistrant::registered() {
    static RegistrationMap registeredOperations;
    return registeredOperations;
}

```

#### 4.25 Add Missing Examples

There are a few missing examples from the original list:

```

TEST(RpnCalculatorShould, AddWithNoValuesProvided) {
    calculator->execute("+");
    LONGS_EQUAL(0, calculator->getX());
}

TEST(RpnCalculatorShould, SubtractWithNoValuesProvided) {
    calculator->execute("-");
    LONGS_EQUAL(0, calculator->getX());
}

```

```
TEST(RpnCalculatorShould, MultiplyWithNoValuesProvided) {
    calculator->execute("*");
    LONGS_EQUAL(0, calculator->getX());
}

#include "DivideByZeroException.h"
TEST(RpnCalculatorShould, GenerateDivideByZeroWhenNoValuesProvided) {
    try {
        calculator->execute("/");
    } catch (DivideByZeroException &) {
        CHECK(1);
    }
}

TEST(RpnCalculatorShould, OnlyAddTwoMostRecentValues) {
    calculator->enter(3);
    calculator->enter(2);
    calculator->enter(6);
    calculator->enter(7);
    calculator->enter(2);
    calculator->execute("+");
    LONGS_EQUAL(9, calculator->getX());
    calculator->execute("drop");
    LONGS_EQUAL(6, calculator->getX());
}

TEST(RpnCalculatorShould, OnlySubtractTwoMostRecentValues) {
    calculator->enter(3);
    calculator->enter(2);
    calculator->enter(6);
    calculator->enter(7);
    calculator->enter(2);
    calculator->execute("-");
    LONGS_EQUAL(5, calculator->getX());
    calculator->execute("drop");
    LONGS_EQUAL(6, calculator->getX());
}
```

#### 4.26 Sprint Summary

The sprint is complete, all examples pass and there are a number of unit checks to back up the rejection checks. We covered a lot of ground:

- Initial project creation
- Adding a few operations
- Removing duplication
- Several examples of extracting class
- Strategy Design Pattern
- Template Method Design Pattern
- Factory Design Pattern
- Automatic registration of new Operations

- Single Responsibility Principle
- Open/Closed Principle
- Calling base-class member functions from a derived-class method of the same name
- Checking for exceptions in automated checks
- Writing basic exception classes and code to throw them
- `std::exception` class
- `std::map`
- `std::pair`
- Delegation versus inheritance

The next sprint is not as feature rich; instead we will add a few new operations and look at another style of writing automated unit checks.

## 5 Rpn Calculator – Sprint 2 – Growing Features

The description of this sprint will be strictly in the form of examples. There are several new operations:

Given the user enters	When the user selects	Then the result is
3 2 6 7 2	sum	20
4 1	<	0
3 4	<	1
4 4	<	1
3 2	==	0
3 3	==	1
3 1	>	1
4 4	>	0
4 7	>	0
3 5 1	swap_xy	3 1 5
3 4 1	dup	3 4 1 1
4 3 2 1 5 2	n_dup	4 3 2 1 5 2 1 5
any value < 2	pf	
2	pf	2
3	pf	3
4	pf	2 2
5	pf	5
6	pf	2 3
7	pf	7
8	pf	2 2 2
9	pf	3 3

### 5.1 Adding Sum

Sum consumes all values on the stack and produces a single result. This operation maps many values to one; it consumes all values on the stack, adds them up and puts a single value back. We saw something like this using `std::accumulate` back on page 103, section 3.23. What are things we want to check:

- It consumes all and produces 1
- It adds correctly
- Overflow would be reasonable but there's a general decision to ignore overflow
- That this operation is registered – this applies to all operations

With that in mind, we can check each of these things against the same sequence. A question to ask is should we put all of the checks in one place or many. In other places we tend to keep to one or a maybe a few checks together. This will be no different:

```
#include <CppUTest/TestHarness.h>
```

```

#include "Sum.h"
#include "RpnStack.h"

TEST_GROUP(SumShould) {
    RpnStack *values;
    void setup() {
        values = new RpnStack;
        values->push(5);
        values->push(3);
        values->push(2);
        values->push(9);
        Sum op;
        op.perform(*values);
    }
    void teardown() {
        delete values;
    }
};

TEST(SumShould, AddAllValues) {
    LONGS_EQUAL(19, values->top());
}

TEST(SumShould, ProduceASingleValue) {
    LONGS_EQUAL(1, values->size());
}

```

Notice how all of the work happens in setup() and the two methods simply check different results. It is possible that both of these could fail, either one or could fail or, eventually, none of them fail. The granularity makes for better targeting of problems when they occur.

For this to work:

#### **Sum.h**

```

#pragma once
#ifndef SUM_H_
#define SUM_H_

#include "MathOperation.h"

class Sum: public MathOperation {
public:
    void perform(RpnStack &values);
};

#endif

```

#### **Sum.cpp**

```

#include "Sum.h"

```

```

#include "RpnStack.h"
void Sum::perform(RpnStack &values) {
    int result = 0;
    while(values.size() > 0) {
        result += values.top();
        values.pop();
    }
    values.push(result);
}

```

### **Sum Registration**

Sum needs to be registered; that work is done in Sum.cpp. Rather than creating a separate source file for that automated check, let's experiment with putting it in the SumShould.cpp:

**Added to bottom of SumShould.cpp:**

```

#include "ShouldBeRegistered.h"
CHECK_REGISTRATION(Sum, "sum");

```

**Added to bottom of Sum.cpp:**

```

#include "MathOperationRegistrant.h"
MathOperationRegistrant register_sum("sum", new Sum);

```

## 5.2 Less Than

What are the cases for Less Than:

- One value less than another
- One value greater than the other
- Two equal values
- Less than is actually registered

Less than consumes two values and produces a single value, so it behaves like a binary math operation and will therefore inherit from that class instead of MathOpeation:

**LessThanShould.cpp**

```

#include <CppUTest/TestHarness.h>

#include "LessThan.h"

TEST_GROUP(LessThanShould) {
    LessThan op;
};

TEST(LessThanShould, Be1For2Versus4) {
    LONGS_EQUAL(1, op.calculate(2, 4));
}

```

**LessThan.h**

```

#pragma once
#ifndef LESSTHAN_H_
#define LESSTHAN_H_

```

```
#include "BinaryMathOperation.h"

class LessThan: public BinaryMathOperation {
public:
    int calculate(int lhs, int rhs);
};

#endif

LessThan.cpp
```

```
#include "LessThan.h"

int LessThan::calculate(int lhs, int rhs) {
    return lhs < rhs ? 1 : 0;
}
```

A few more checks:

*Added to LessThanShould.cpp*

```
TEST(LessThanShould, Be0For4Versus4) {
    LONGS_EQUAL(0, op.calculate(4, 4));
}

TEST(LessThanShould, Be0For4Versus2) {
    LONGS_EQUAL(0, op.calculate(4, 2));
}
```

It should be registered:

*Added to LessThanShould.cpp*

```
#include "ShouldBeRegistered.h"
CHECK_REGISTRATION(LessThan, "<");
```

Add the required registration to LessThan.cpp:

```
#include "MathOperationRegistrant.h"
MathOperationRegistrant register_lessThan("<", new LessThan);
```

### 5.3 Equal To and Greater than

Equal to and greater than have the same set of checks as less than, the actual results are different. Both of these consume two values and produce a single result, as with Less Than. Given that information, create the required checks for these classes and make sure they are registered.

### 5.4 Swap XY

Swap XY consumes two values and produces two values, so it does not fit under the Binary Math Operation class. Here is another example of an automated check where all of the setup and execution happens in the setup() method, followed by a number of checks, each in their own method:

```
#include <CppUTest/TestHarness.h>

#include "SwapXy.h"
```

```

#include "RpnStack.h"

TEST_GROUP(SwapXyShould) {
    RpnStack *values;
    void setup() {
        values = new RpnStack;
        values->push(-3);
        values->push(5);
        values->push(1);
        SwapXy().perform(*values);
    }
    void teardown() {
        delete values;
    }
};

TEST(SwapXyShould, ResultInSameStackSize) {
    LONGS_EQUAL(3, values->size());
}

TEST(SwapXyShould, MakeXEqualTo5) {
    LONGS_EQUAL(5, values->top());
}

TEST(SwapXyShould, MakeYEqualTo1) {
    values->pop();
    LONGS_EQUAL(1, values->top());
}

TEST(SwapXyShould, LeaveNegative3WhereItWas) {
    values->pop();
    values->pop();
    LONGS_EQUAL(-3, values->top());
}

#include "ShouldBeRegistered.h"
CHECK_REGISTRATION(SwapXy, "swap_xy");

```

The implementation is straightforward:

#### **SwapXy.h**

```

#pragma once
#ifndef SWAPXY_H_
#define SWAPXY_H_

#include "MathOperation.h"

class SwapXy: public MathOperation {
public:
    void perform(RpnStack &values);
};

```



```
#endif
```

```
SwapXy.cpp
```

```
#include "SwapXy.h"
```

```
#include "RpnStack.h"
```

```
void SwapXy::perform(RpnStack &values) {
```

```
    int x = values.top();
```

```
    values.pop();
```

```
    int y = values.top();
```

```
    values.pop();
```

```
    values.push(x);
```

```
    values.push(y);
```

```
}
```

```
#include "MathOperationRegistrant.h"
```

```
MathOperationRegistrant register_swapXy("swap_xy", new SwapXy);
```

## 5.5 Dup

Dup is left as an exercise.

## 5.6 N Dup

This operation uses the top of the stack as a count and the duplicates that many items from the remainder of the stack back on top of the stack:

```
#include <CppUTest/TestHarness.h>
```

```
#include "NDup.h"
```

```
#include "RpnStack.h"
```

```
TEST_GROUP(NDupShould) {
```

```
    RpnStack *values;
```

```
    void setup() {
```

```
        values = new RpnStack;
```

```
        values->push(4);
```

```
        values->push(3);
```

```
        values->push(2);
```

```
        values->push(1);
```

```
        values->push(5);
```

```
        values->push(2);
```

```
        NDup().perform(*values);
```

```
    }
```

```
    void teardown() {
```

```
        delete values;
```

```
    }
```

```
};
```

```
TEST(NDupShould, HaveSameTwoValuesAtTop) {
```

```
    LONGS_EQUAL(5, values->top());
```

```
    values->pop();
```

```
    LONGS_EQUAL(1, values->top());
```

```
}
```

```

TEST(NDupShould, StillHaveOriginalTwoValues) {
    values->pop();
    values->pop();
    LONGS_EQUAL(5, values->top());
    values->pop();
    LONGS_EQUAL(1, values->top());
}

TEST(NDupShould, LeaveRemainderOfStackAlone) {
    values->pop();
    values->pop();
    values->pop();
    values->pop();
    LONGS_EQUAL(2, values->top());
    values->pop();
    LONGS_EQUAL(3, values->top());
    values->pop();
    LONGS_EQUAL(4, values->top());
    values->pop();
}

TEST(NDupShould, IncreaseStackSizeCorrectly) {
    LONGS_EQUAL(7, values->size());
}

#include "ShouldBeRegistered.h"
CHECK_REGISTRATION(NDup, "n_dup");

```

#### *NDup.h*

```

#pragma once
#ifndef NDUP_H_
#define NDUP_H_

#include "MathOperation.h"

class NDup: public MathOperation {
public:
    void perform(RpnStack &values);
};

#endif

```

#### *NDup.cpp*

```

#include "NDup.h"
#include "RpnStack.h"

#include <vector>
void NDup::perform(RpnStack &values) {
    int count = values.top();
    values.pop();
    std::vector<int> toCopy;
    for (int i = 0; i < count; ++i) {

```

```

    toCopy.push_back(values.top());
    values.pop();
}

for (int i = 0; i < 2; ++i)
    for (std::vector<int>::reverse_iterator i = toCopy.rbegin();
         i != toCopy.rend(); ++i)
        values.push(*i);
}

#include "MathOperationRegistrant.h"
MathOperationRegistrant register_nDup("n_dup", new NDup);

```

## 5.7 Prime Factors

For this operation, we will take a slower approach and attempt to get back to check driven development. We'll start with a single check and try to minimally modify the code to get the next check working. This will also demonstrate yet another way to express automated checks:

### *PrimeFactorsShould.cpp*

```

#include <CppUTest/TestHarness.h>

#include "PrimeFactors.h"
#include "RpnStack.h"

TEST_GROUP(PrimeFactorsOf) {
    RpnStack *values;
    void setup() {
        values = new RpnStack;
    }
    void teardown() {
        delete values;
    }
    void givenTheValue(int value) {
        values->push(value);
    }
    void whenCalculatingItsPrimeFactors() {
        PrimeFactors().perform(*values);
    }
    void expectNoResults() {
        LONGS_EQUAL(0, values->size());
    }
};

TEST(PrimeFactorsOf, 1AreEmpty) {
    givenTheValue(1);
    whenCalculatingItsPrimeFactors();
    expectNoResults();
}

```

To get this passing:

**PrimeFactors.h**

```
#pragma once
#ifndef PRIMEFACTORS_H_
#define PRIMEFACTORS_H_

#include "MathOperation.h"

class PrimeFactors: public MathOperation {
public:
    void perform(RpnStack &values);
};

#endif
```

**PrimeFactors.cpp**

```
#include "PrimeFactors.h"
#include "RpnStack.h"

void PrimeFactors::perform(RpnStack &values) {
    values.pop();
}
```

## 5.7.1 Of 2 ...

```
TEST_GROUP(PrimeFactorsOf) {
    ...
    void expect(int value) {
        LONGS_EQUAL(value, values->top());
    }
    void andThen() {
        values->pop();
    }
    ...
};
TEST(PrimeFactorsOf, 2Are2) {
    givenTheValue(2);
    whenCalculatingItsPrimeFactors();
    expect(2);
    andThen();
    expectNoResults();
}
```

**Updated PrimeFactors.cpp**

```
void PrimeFactors::perform(RpnStack &values) {
    int value = values.top();
    values.pop();

    if(value == 2)
        values.push(2);
}
```

## 5.7.2 Of 3...

```
TEST(PrimeFactorsOf, 3Are3) {
    givenTheValue(3);
    whenCalculatingItsPrimeFactors();
    expect(3);
    andThen();
    expectNoResults();
}
```

*Updated PrimeFactors.cpp*

```
if(value >= 2)
    values.push(value);
```

## 5.7.3 Of 4 ... multiple values

```
TEST(PrimeFactorsOf, 4Are2And2) {
    givenTheValue(4);
    whenCalculatingItsPrimeFactors();
    expect(2);
    andThen();
    expect(2);
    andThen();
    expectNoResults();
}
```

*Updated PrimeFactors.cpp – starting to get ugly*

```
if (value >= 2) {
    if (value % 2 == 0) {
        values.push(2);
        value /= 2;
    }
    if (value > 1)
        values.push(value);
}
```

## 5.7.4 Of 5 ...

This one just works because it's the same as 3.

```
TEST(PrimeFactorsOf, 5Are5) {
    givenTheValue(5);
    whenCalculatingItsPrimeFactors();
    expect(5);
    andThen();
    expectNoResults();
}
```

## 5.7.5 Of 6 ... two values, but they are different

```
TEST(PrimeFactorsOf, 6Are3And2) {
    givenTheValue(6);
    whenCalculatingItsPrimeFactors();
    expect(3);
    andThen();
    expect(2);
}
```

```

    andThen();
    expectNoResults();
}

```

Surprisingly, or not, this one also works.

#### 5.7.6 As will 7 ...

```

TEST(PrimeFactorsOf, 7Are7) {
    givenTheValue(7);
    whenCalculatingItsPrimeFactors();
    expect(7);
    andThen();
    expectNoResults();
}

```

#### 5.7.7 But 8 is different, 3 values, instead of just 2.

```

TEST(PrimeFactorsOf, 8Are2And2And2) {
    givenTheValue(8);
    whenCalculatingItsPrimeFactors();
    expect(2);
    andThen();
    expect(2);
    andThen();
    expect(2);
    andThen();
    expectNoResults();
}

```

*Finally an update to PrimeFactors.cpp*

```

    while (value % 2 == 0) {

```

Simply changing the if to a while fixes this.

#### 5.7.8 Is 9 different?

```

TEST(PrimeFactorsOf, 9Are3And3) {
    givenTheValue(9);
    whenCalculatingItsPrimeFactors();
    expect(3);
    andThen();
    expect(3);
    andThen();
    expectNoResults();
}

```

This requires that we vary the divisor:

```

    if (value >= 2) {
        for (int divisor = 2; divisor <= value; ++divisor)
            while (value % divisor == 0) {
                values.push(divisor);
                value /= divisor;
            }
    }
    if (value > 1)

```

```
    values.push(value);
}
```

But wait, the loop starts at 2, so do we need the outer if statement? Not at all.

```
for (int divisor = 2; divisor <= value; ++divisor)
    while (value % divisor == 0) {
        values.push(divisor);
        value /= divisor;
    }
if (value > 1)
    values.push(value);
```

What about the bottom if statement?

```
void PrimeFactors::perform(RpnStack &values) {
    int value = values.top();
    values.pop();

    for (int divisor = 2; divisor <= value; ++divisor)
        while (value % divisor == 0) {
            values.push(divisor);
            value /= divisor;
        }
}
```

It was also not necessary, and that finishes it. Try with a few larger values.

### 5.7.9 Register It

We spent so much time on checking the operation we nearly forget to make sure it is registered in the factory:

**One more automated check in PrimeFactorsOf.cpp**

```
#include "ShouldBeRegistered.h"
CHECK_REGISTRATION(PrimeFactors, "pf");
```

**Actual Registration in PrimeFactors.cpp**

```
#include "MathOperationRegistrant.h"
MathOperationRegistrant
    register_primeFactors("pf", new PrimeFactors);
```

## 5.8 Examples as Rejection Checks

Notice that all of the automate checks written targeted the implementation class directly? If we follow the stated policy, then we should have several more checks written against the RpnCalculator class. Before you write any of those, do you expect any of them to fail? If not, how might you work this information into your project work?

In writing these checks, I observed a lot of duplication but also a lot of unnecessary detail. I spend just a touch of time removing some duplication:

```
TEST_GROUP(RpnCalculatorShould) {
    RpnCalculator *calculator;
    void setup() {
        calculator = new RpnCalculator;
```

```
    }
    void teardown() {
        delete calculator;
    }
    void topWas(int value) {
        LONGS_EQUAL(value, calculator->getX());
        calculator->execute("drop");
    }
    void enter(int value) {
        calculator->enter(value);
    }
    void execute(const std::string &opName) {
        calculator->execute(opName);
    }
};

TEST(RpnCalculatorShould, AddTwoNumbers) {
    enter(30);
    enter(4);
    execute("+");
    topWas(34);
}

TEST(RpnCalculatorShould, SubtractTwoNumbers) {
    enter(30);
    enter(4);
    execute("-");
    topWas(26);
}

...
TEST(RpnCalculatorShould, NDupCorrectly) {
    enter(4);
    enter(3);
    enter(2);
    enter(1);
    enter(5);
    enter(2);
    execute("n_dup");
    topWas(5);
    topWas(1);
    topWas(5);
    topWas(1);
    topWas(2);
    topWas(3);
    topWas(4);
    topWas(0);
}

TEST(RpnCalculatorShould, CalculatePrimeFactorsOf100Correctly) {
    enter(100);
    execute("pf");
    topWas(5);
}
```



```
topWas(5);  
topWas(2);  
topWas(2);  
topWas(0);  
}
```

The remainder of the missing automated rejection checks is left as an exercise.

## 6 Rpn Calculator – Sprint 3 – Macros

It is finally time to begin programming the calculator. First some examples:

Example 1	Example 2	Example 3	Example 4	Example 5
<pre>start + * - save macro1 6 4 9 3 macro1 -42</pre>	<pre>start save macro3 &lt;error&gt; too few steps</pre>	<pre>start ^^% &lt;error&gt; unknown operation 4 2 + 6</pre>	<pre>start + + save + &lt;error&gt; operation name in use</pre>	<pre>start / ! save m1 start swap_xy m1 save m2 2 8 m2 24</pre>

This is not a complete description of macro recording but it is a good start.

- Example 1 demonstrates a simple macro. Create a macro with three math operations, enter 4 values and execute that macro. The result of -42 shows that the execution of the steps is in the order entered.
- Example 2 demonstrates that there must be at least 1 step in any macro – a somewhat arbitrary requirement, but one nonetheless. Generate an error if this condition is not met.
- Example 3 demonstrates that a macro can only be built with known operations; if an unknown operation is provided, stop recording. This is simple behavior. It makes creating a circular set of macros more difficult.
- The fourth example demonstrates that you cannot use a name that is already in use. This particular example only demonstrates that with a build-in operation, but it applies for the ones you create yourself. Notice, with this additional limitation, it is not possible to create circular macros (the proof is left as an exercise).
- The final example demonstrates that one macro can refer to another. Since there is a requirement that macros should execute like regular operations, this may not seem significant. What is significant, however, is that there are 2 macros in the system. In general, a user can create any number of macros. That's what this example suggests.

If we take these examples at face value, what are the actual messages coming into our system? It looks like the follow methods must be added: start, save. Are these methods on the Rpn Calculator, on a different façade or are they simply new kinds of math operations? Any of the above will work. For now, we'll take the easier road and say these are new methods on the RpnCalculator. What of the operations, do we need to create a new method or can we use the existing one? The decision isn't arbitrary: add a new method, change an existing method, but both will work. Rather than belabor the decision, we'll use the existing method and see what happens.

### 6.1 Happy Path

Here's a happy path rejection check:

**Added to RpnCalculatorShould.cpp**

```
TEST(RpnCalculatorShould, BeAbleToRecordAndExecuteMacro) {
    calculator->start();
    calculator->execute("+");
    calculator->execute("*");
    calculator->execute("-");
    calculator->save("macro1");
    enter(6);
    enter(4);
    enter(9);
    enter(3);
    calculator->execute("macro1");
    topWas(-42);
}
```

**Get to compiling: Update RpnCalculator.h**

```
void start();
void save(const std::string &macroName);
```

**Get to linking: Update RpnCalculator.cpp**

```
void RpnCalculator::start() {
}

void RpnCalculator::save(const std::string &macroName) {
}
```

Now we have a failing rejection check. We have a few options:

- Get this working out-to-in
- Get this working in-to-out

There are a number of moving parts to this and we already have several things in place:

- This new construct should operate like a regular math operation, so it should inherit from the math operation interface (abstract base class).
- Math operations reside in the math operation factory, which is also where they are looked up. So it seems that any new operations should end up there.
- Currently, math operations self-register. This option is for ones that are known when the system is written, not when it is executing, so we might need to update the factory to allow for new operations while the system is running.

Rather than getting this check to pass out-to-in, let's start working in-to-out (or bottom-up). We have a decision; do we leave this check failing while we do our other work or do we "remove" it somehow? There's an easy way to note this as "not ready to check yet":

```
IGNORE_TEST(RpnCalculatorShould, BeAbleToRecordAndExecuteMacro) {
```

When we execute our automated checks, the summary will show one ignored check. We'll leave this in place until we think we're ready to give it a go.

**6.1.1 A Macro**

First, we need to create something that can hold on to a number of math operations but itself behaves like a math operation. Here's one such example:

```

#include <CppUTest/TestHarness.h>

#include "Macro.h"
#include "RpnStack.h"
#include "Add.h"

TEST_GROUP(MacroShould) {
};

struct MathOperationSpy : public MathOperation {
    MathOperationSpy() : performCount(0) {}
    void perform(RpnStack &values) {
        ++performCount;
    }
    int performCount;
};

TEST(MacroShould, HandleMultipleMathOperations) {
    MathOperationSpy spy;
    Macro op;
    RpnStack values;
    op.append(spy);
    op.append(spy);
    op.perform(values);
    LONGS_EQUAL(2, spy.performCount);
}

```

This check verifies that we can add multiple math operations to a macro and that each is sent the perform message.

#### **Macro.h**

```

#pragma once
#ifndef MACRO_H_
#define MACRO_H_

#include "MathOperation.h"
#include <list>

class Macro: public MathOperation {
public:
    void perform(RpnStack &values);
    void append(MathOperation &op);

private:
    typedef std::list<MathOperation*> MathOperationList;
    typedef MathOperationList::iterator iterator;
    MathOperationList operations;
};

#endif

```

Notice that this class uses the `<list>` class instead of `<vector>`. This is more for you to be aware of the class that for a compelling design reason. Other than `std::list` versus `std::vector`, you won't notice any difference in this simple example.

#### *Macro.cpp*

```
#include "Macro.h"

void Macro::perform(RpnStack &values) {
    for(iterator i = operations.begin(); i != operations.end(); ++i)
        (*i)->perform(values);
}

void Macro::append(MathOperation &op) {
    operations.push_back(&op);
}
```

This is a minimal implementation of Macro that gets the automated check passing.

### 6.1.2 Adding to factory

Now we need to be able to add one of these to the factory. Here is an automated check for that:

#### *Added to MathOperationFactoryShould.cpp*

```
#include "MathOperation.h"
struct MathOperationStub : public MathOperation {
    void perform(RpnStack &values) {}
};

TEST(MathOperationFactoryShould, AllowRegistrationOfNewMathOperations)
{
    MathOperationFactory factory;
    MathOperationStub *op = new MathOperationStub;
    factory.add("newop", op);
    try {
        CHECK(op == &factory.findOperationNamed("newop"));
    } catch(UnknownMathOperationException &) {
        FAIL("Should have found a math operation");
    }
}
```

This almost works. The check passes, but there is a memory leak. The factory gets most of its operations from the math operation registrant; all but this one. We need to update the destructor of the factory to remove this memory leak:

```
MathOperationFactory::~MathOperationFactory() {
    MathOperationRegistrant r;
    for(MathOperationRegistrant::iterator i = r.begin();
        i != r.end(); ++i)
        operationsByName.erase((*i).first);
    for(iterator i = operationsByName.begin();
        i != operationsByName.end(); ++i)
        delete (*i).second;
}
```

```
}

```

This is quite a bit, remove the operations in the factory that come from the math operation registrant and then delete anything else that is left over. While this does seem to work, it's a bit of a mess:

- Allocation in the calculator is passed to the factory and then released: what happens if `save()` is not called?
- The factory has to include `MathOperation.h` again because of the destructor.
- The destructor is ugly.

Auto-registration is to blame for some of this; the destructor is more complex because of it. The split of allocation and deallocation is problematic; we could have the factory perform the allocation and deallocation or put it somewhere else. We could update the factory to use the auto registration rather than copy it. Then the factory would look in two places for an operation. Before doing any of that, let's take this a bit further to see how much uglier it gets.

### 6.1.3 Adding it to RpnCalculator

We have two stub methods and we're working on a happy path. Here is something that will work for this first automated check:

```
void RpnCalculator::start() {
    macro = new Macro;
    recording = true;
}

void RpnCalculator::save(const std::string &macroName) {
    factory->add(macroName, macro);
    recording = false;
}

void RpnCalculator::execute(const std::string &operatorName) {
    MathOperation &op = factory->findOperationNamed(operatorName);

    if (!recording) {
        op.perform(values);
    } else {
        macro->append(op);
    }
}

```

**Member Data: Update header**

```
class Macro;

class RpnCalculator {
    ...

private:
    ...
    Macro *macro;
    bool recording;
}

```

**Member Data: Initialization**

```
RpnCalculator::RpnCalculator() :
    factory(new MathOperationFactory), macro(0), recording(false) {
}
```

These changes should work. Now update the rejection check in RpnCalculatorShould by removing IGNORE\_ and verify that it now passes.

**6.2 Empty macros not allowed**

Here's a rejection check for this one:

**Added to RpnCalculatorShould.cpp**

```
#include "IllegalMacroException.h"
TEST(RpnCalculatorShould,
ThrowExceptionWhenAttemptingToSaveZeroLengthMacro) {
    calculator->start();
    try {
        calculator->save("should fail");
        FAIL("Should have thrown exception");
    } catch(IllegalMacroException &e) {
        CHECK(1);
    }
}
```

**IllegalMacroException.h**

```
#pragma once
#ifndef ILLEGALMACROEXCEPTION_H_
#define ILLEGALMACROEXCEPTION_H_

#include <exception>

class IllegalMacroException : public std::exception {
};

#endif
```

This check fails, so we need to update the save() method to check:

```
#include "IllegalMacroException.h"
void RpnCalculator::save(const std::string &macroName) {
    if(macro->stepCountAtLeast(1) == false)
        throw IllegalMacroException();
    factory->add(macroName, macro);
    recording = false;
}
```

For this to compile, link and pass:

**Added to Macro.h**

```
bool stepCountAtLeast(unsigned length) const;
```

**Defined in Macro.cpp**

```
bool Macro::stepCountAtLeast(unsigned length) const {
```

```
    return operations.size() >= length;
}
```

The check passes but there's a memory leak. If the macro has been allocated in `start()`, it should be released either here or in the destructor. We noticed the problem here, we can fix it here; this also points to other problematic paths of execution:

```
#include "IllegalMacroException.h"
void RpnCalculator::save(const std::string &macroName) {
    Macro *candidate = macro;
    recording = false;
    macro = 0;
    if(candidate->stepCountAtLeast(1)) {
        factory->add(macroName, candidate);
    } else {
        delete candidate;
        throw IllegalMacroException();
    }
}
```

After this method is done, either the macro was recorded, in which case its memory is owned by the factory, or the macro was deleted. Regardless, the calculator is no longer recording and the macro attribute is initialized to 0. This points to another problem, what if `start()` was not first called?

### 6.2.1 Must call start first()

```
TEST(RpnCalculatorShould, RequireStartToBeCalledBeforeSave) {
    try {
        calculator->save("should fail");
        FAIL("Should have thrown exception");
    } catch(IllegalMacroException &e) {
        CHECK(1);
    }
}
```

This automated check fails. How it fails is somewhat platform dependent. In my case it simply stops tests from running. What's happening is a null pointer reference on this line:

```
    if(candidate->stepCountAtLeast(1)) {
```

We can fix this by checking for it:

```
#include "IllegalMacroException.h"
void RpnCalculator::save(const std::string &macroName) {
    if(macro == 0)
        throw IllegalMacroException();
    ...
}
```

Notice this is getting a touch ugly. We will work on this in a bit; let's continue with automated rejection checks based on the provided examples.

### 6.2.2 Unknown operation cannot be added to a macro

```
TEST(RpnCalculatorShould, OnlyAllowValidMathOperationsToBeAdded) {
```



```

calculator->start();
try {
    execute("^%");
    FAIL("Should have thrown exception");
} catch(UnknownMathOperationException &e) {
    CHECK(1);
}
enter(4);
enter(2);
execute("+");
topWas(6);
}

```

This check fails. While the system already throws `UnknownMathOperationException`, the code needs to clean up the current macro and move back into record mode after the exception, and it doesn't.

**Update `RpnCalculator.execute`:**

```

#include "UnknownMathOperationException.h"
void RpnCalculator::execute(const std::string &operatorName) {
    if (!recording) {
        MathOperation &op = factory->findOperationNamed(operatorName);
        op.perform(values);
    } else {
        try {
            MathOperation &op = factory->findOperationNamed(operatorName);
            macro->append(op);
        } catch (UnknownMathOperationException &e) {
            delete macro;
            macro = 0;
            recording = 0;
            throw e;
        }
    }
}
}

```

This is getting pretty ugly. There's duplication and there are essentially two different methods in this based on the state of things. We will address this shortly. For now, let's finish the last two examples then we'll come back and consider different approaches to removing this ugliness.

### 6.2.3 Cannot save under existing name

The system should not allow saving a macro name under a name that is already in use:

**Added to `RpnCalculatorShould.cpp`**

```

#include "NameInUseException.h"
TEST(RpnCalculatorShould, DisallowSavingUnderExistingName) {
    calculator->start();
    try {
        execute("+");
        execute("+");
        calculator->save("+");
    }
}

```

```

    FAIL("Should have thrown exception.")
  } catch (NameInUseException &e) {
    CHECK(1);
  }
}

```

#### *NameInUseException.h*

```

#pragma once
#ifndef NAMEINUSEEXCEPTION_H_
#define NAMEINUSEEXCEPTION_H_

#include <exception>

class NameInUseException : public std::exception {};

#endif

```

This check fails. This is really a behavior of the factory, not the rpn calculator, so we can get this check to pass but then we should consider adding another micro-check to the math operation factory should source file.

#### *Update MathOperationFactory.cpp*

```

#include "NameInUseException.h"
void MathOperationFactory::add(const std::string &name, MathOperation
*op) {
    if(operationsByName.find(name) != operationsByName.end())
        throw NameInUseException();
    operationsByName[name] = op;
}

```

While the check passes, there's a memory leak (again). To fix this:

#### *Update RpnCalculator::save*

```

#include "IllegalMacroException.h"
#include "NameInUseException.h"
void RpnCalculator::save(const std::string &macroName) {
    ...
    if (candidate->stepCountAtLeast(1)) {
        try {
            factory->add(macroName, candidate);
        } catch (NameInUseException &e) {
            delete candidate;
            throw e;
        }
    }
    ...
}

```

Notice how the rpn calculator keeps growing? This class violates the single responsibility principle, it is no longer cohesive. We'll address this once we finish the examples.

### 6.2.4 Adding missing check on the factory

The last automated rejection check required changes to the factory, so let's add that missing check on the factory:

*Added to MathOperationFactoryShould.cpp*

```
#include "NameInUseException.h"
TEST(MathOperationFactoryShould, NoAllowRegistrationOfAlreadyUsedName)
{
    MathOperationFactory factory;
    MathOperationStub *op = new MathOperationStub;
    factory.add("newop", op);
    try {
        factory.add("newop", op);
        FAIL("Should have thrown exception");
    } catch(NameInUseException &e) {
        CHECK(1);
    }
}
```

Notice, this one passes as is. That's because we already wrote a more integration-oriented check that required this behavior. This is just keeping us honest.

### 6.2.5 Macros can refer to other macros

This should be no problem, let's see how well we've done up to this point:

```
TEST(RpnCalculatorShould, AllowMacrosToReferToOtherMacros) {
    calculator->start();
    execute("/");
    execute("!");
    calculator->save("m1");
    calculator->start();
    execute("swap_xy");
    execute("m1");
    calculator->save("m2");
    enter(2);
    enter(8);
    execute("m2");
    topWas(24);
}
```

This automated rejection check passes without changing the underlying system. This is no surprise since we treat macros like regular operations.

## 6.3 Cleaning up the calculator

The calculator has become a bit of a mess by adding support for programmability. Here are the offending member functions:

```
#include "UnknownMathOperationException.h"
void RpnCalculator::execute(const std::string &operatorName) {
    if (!recording) {
        MathOperation &op = factory->findOperationNamed(operatorName);
        op.perform(values);
    }
}
```

```
} else {
    try {
        MathOperation &op = factory->findOperationNamed(operatorName);
        macro->append(op);
    } catch (UnknownMathOperationException &e) {
        delete macro;
        macro = 0;
        recording = 0;
        throw e;
    }
}
}

#include "IllegalMacroException.h"
#include "NameInUseException.h"
void RpnCalculator::save(const std::string &macroName) {
    if (macro == 0)
        throw IllegalMacroException();

    Macro *candidate = macro;
    recording = false;
    macro = 0;
    if (candidate->stepCountAtLeast(1)) {
        try {
            factory->add(macroName, candidate);
        } catch (NameInUseException &e) {
            delete candidate;
            throw e;
        }
    } else {
        delete candidate;
        throw IllegalMacroException();
    }
}
```

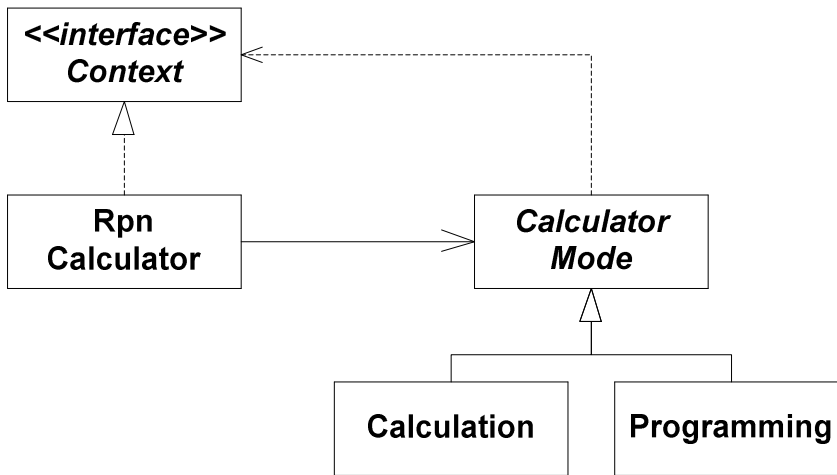
The RpnCalculator serves as the entry point into the system. It shouldn't do too much; instead it should delegate much of the work to other objects. Generally, when a class does too much work, we extract out part of the class into another class. This work is a prime candidate for such a refactoring. We could:

- Create a class that does the programming
- Use the state pattern

The state pattern is a more specific form of factoring out part of the work into another class. The difference is that there are typically multiple states, which we have, and the next state is determined by messages coming into the system. Consider:

- The calculator is initially in calculation mode
- When the start() message is received, the system is in programming mode
- When the save() message is received, the system returns to normal mode

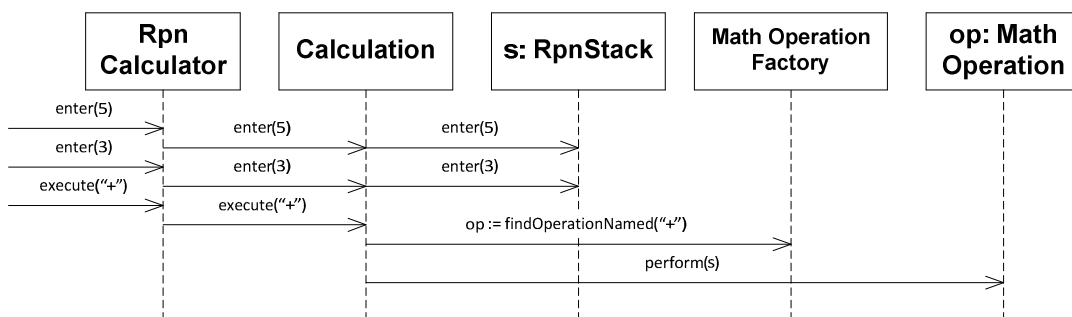
As it is, the state pattern may seem a bit complex for this and it probably is. However, let's go with it and see what we end up with. Here's a model of the state pattern applied to our domain:

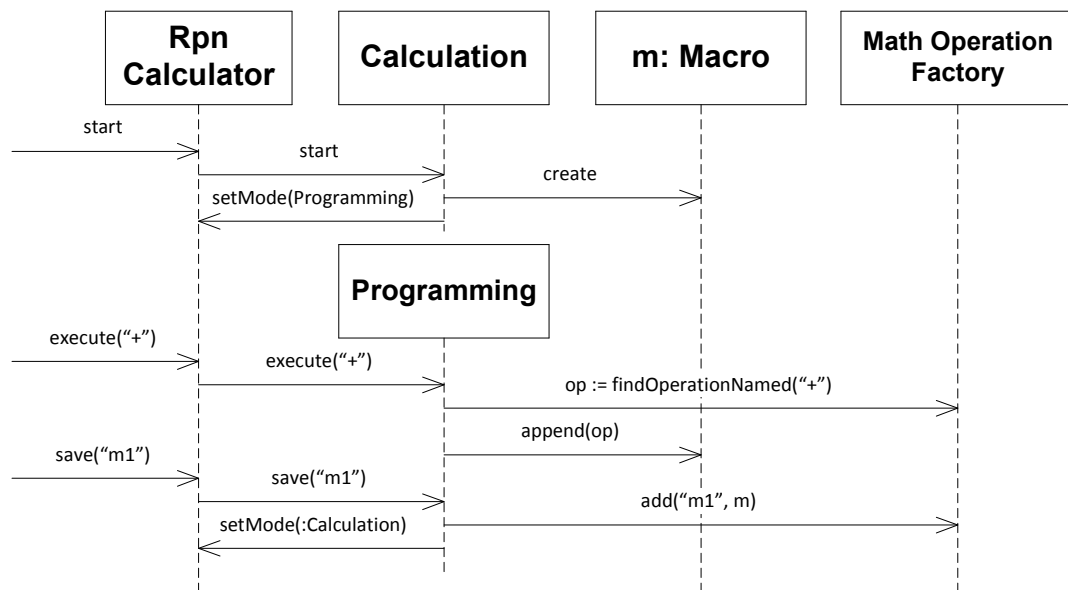


In the state pattern, there's a hierarchy of state objects. In our case, that is the calculator mode and its two derivatives: Calculation and Programming. The calculator has a reference to one of these at all times, that's the solid line from Rpn Calculator to Calculator Mode. The thing with the state is called the "context" object. Typically, the state objects need to work directly with the context, the rpn calculator in this case. If you do so directly, then there is a circular reference between the object with the state and the state hierarchy. Circular references are causes a number of problems, so we'll extract an interface called Context. The calculator will depend on its mode; the mode will depend on the context. This breaks the circular reference.

The Rpn Calculator and its Calculator Mode can be considered one logical grouping of functionality. When a message comes into the calculator, depending on its mode it does one of a few things:

**Calculation Mode**



**Programming**

This is a lot to take in, so we'll take it test-first.

### 6.3.1 Calculation Mode

In calculation mode, we should do what the calculator was doing before we added macros. With that in mind, here is a series of automated checks on the Calculation class:

#### **CalculationShould.cpp**

```

#include <CppUTest/TestHarness.h>

#include "RpnCalculator.h"
#include "Calculation.h"

TEST_GROUP(CalculationShould) {
    RpnCalculator *context;
    void setup() {
        context = new RpnCalculator;
    }
    void teardown() {
        delete context;
    }
};

TEST(CalculationShould, PutNumbersOnStackForEnter) {
    Calculation mode;
    mode.enter(context, 5);
    LONGS_EQUAL(5, context->getX());
}
  
```

For this to compile, you'll need to make several changes:

#### **Create Calculation.h**

```
#pragma once
```

```
#ifndef CALCULATION_H_
#define CALCULATION_H_

class Context;

class Calculation {
public:
    Calculation();
    virtual ~Calculation();
    void enter(Context *context, int value);
};

#endif
```

*Create Calculation.cpp*

```
#include "Calculation.h"

#include "Context.h"
#include "RpnStack.h"

Calculation::Calculation() {
}

Calculation::~~Calculation() {
}

void Calculation::enter(Context *context, int value) {
    context->getStack().push(value);
}
```

*Create Context.h*

```
#pragma once
#ifndef CONTEXT_H_
#define CONTEXT_H_

class RpnStack;
class Context {
public:
    virtual ~Context() = 0;
    virtual RpnStack &getStack() = 0;
    virtual void enter(int value) = 0;
};

#endif
```

Make RpnCalculator inherit from Context and add new method:

```
#include "Context.h"
class RpnCalculator : public Context {
public:
    ...
    RpnStack &getStack() { return values; }
```

The first check now passes.

### 6.3.2 Executes Operations Directly

```
TEST(CalculationShould, ExecuteOperationsDirectly) {
    Calculation mode;
    context->enter(5);
    context->enter(2);
    mode.execute(context, "+");
}
```

*Update Calculation.h*

```
#include <string>

class Calculation {
public:
    ...
    void execute(Context *context, const std::string &name);
}
```

*Update Calculation.cpp*

```
#include "MathOperationFactory.h"
#include "MathOperation.h"
void Calculation::execute(Context *context, const std::string &name) {
    MathOperation &op
        = context->getFactory()->findOperationNamed(name);
    op.perform(context->getStack());
}
```

*Update Context.h:*

```
#include <string>
class Context {
public:
    ...
    virtual void execute(const std::string &name) = 0;
}
```

### 6.3.3 Throw exception when told to save

```
#include "IllegalMacroException.h"
TEST(CalculationShould, ThrowExceptionWhenToldToSave) {
    Calculation mode;
    try {
        mode.save(context, "should fail");
        FAIL("Should have thrown exception");
    } catch(IllegalMacroException &e) {
        CHECK(1);
    }
}
```

*Update Calculation.h*

```
class Calculation {
    ...
    void save(Context *context, const std::string &name);
}
```



**Update Calculation.cpp**

```
#include "IllegalMacroException.h"
void Calculation::save(Context *context, const std::string &name) {
    throw IllegalMacroException();
}
```

## 6.3.4 Change to Programming Mode When Told To Start

**Create an automated check**

```
#include <typeinfo>
#include "Programming.h"
TEST(CalculationShould, ChangeToProgrammingStateWhenToldToStart) {
    Calculation mode;
    mode.start(context);
    CalculatorMode *finalMode = context->getState();
    CHECK(typeid(*finalMode) == typeid(Programming));
}
```

This requires several changes to compile:

- Add a start method to Calculation
- Create CalculationMode interface
- Update Calculation to use CalculationMode interface
- Create Programming class from CalculationMode interface
- Add getState() and setState methods to Context
- Implement getState() and setState methods in RpnCalculator

**Calculation.h**

```
class Calculation {
    ...
    void start(Context *context);
}
```

**CalculatorMode.h (extract interface from Calculation.h)**

```
#pragma once
#ifndef CALCULATORMODE_H_
#define CALCULATORMODE_H_

#include <string>
class Context;

class CalculatorMode {
public:
    virtual ~CalculatorMode() = 0;
    virtual void enter(Context *context, int value) = 0;
    virtual void execute(Context *context, const std::string &name) = 0;
    virtual void save(Context *context, const std::string &name) = 0;
    virtual void start(Context *context) = 0;
};

#endif
```

**CalculatorMode.cpp**

```
#include "CalculatorMode.h"

CalculatorMode::~CalculatorMode() {
}
```

**Update Calculation.h**

```
#include "CalculatorMode.h"
class Calculation : public CalculatorMode {
    ...
}
```

**Create Programming.h**

```
#pragma once
#ifndef PROGRAMMING_H_
#define PROGRAMMING_H_

#include "CalculatorMode.h"

class Programming: public CalculatorMode {
public:
    Programming();
    ~Programming();
    void enter(Context *context, int value);
    void execute(Context *context, const std::string &name);
    void save(Context *context, const std::string &name);
    void start(Context *context);
};

#endif
```

**Create Programming.cpp**

```
#include "Programming.h"

Programming::Programming() {
}

Programming::~Programming() {
}

void Programming::enter(Context *context, int value) {
}

void Programming::execute(Context *context, const std::string &name) {
}

void Programming::save(Context *context, const std::string &name) {
}

void Programming::start(Context *context) {
}
```

**Update Context.h**

```
class CalculatorMode;

class Context {
    ...
    virtual CalculatorMode *getState() = 0;
    virtual void setState(CalculatorMode *newMode) = 0;
};
```

Update RpnCalculator.cpp

```
class RpnCalculator : public Context {
public:
    ...
    CalculatorMode *getState();
    void setState(CalculatorMode *newMode);

private:
    ...
    CalculatorMode *mode;
```

**Update RpnCalculator.cpp**

```
RpnCalculator::RpnCalculator() :
    factory(new MathOperationFactory), macro(0), recording(false),
    mode(0) {
}

RpnCalculator::~~RpnCalculator() {
    delete factory;
    delete mode;
}

CalculatorMode *RpnCalculator::getState() {
    return mode;
}

void RpnCalculator::setState(CalculatorMode *newMode) {
    delete mode;
    mode = newMode;
}
```

That's a lot of mechanical work but it having done this and gotten the final check against Calculation passing, you've well past the half-way mark.

**6.3.5 Programming Mode**

Now we do the same thing with Programming mode (in the same order):

```
TEST(ProgrammingShould, IgnoreEnterForNow) {
    Programming op;
    op.enter(context, 5);
    CHECK(1);
}
```

This is a bit odd. For now enter should do nothing. This is probably OK but it does appear to violate the Liskov substitution principle. This happens when using the state pattern. What it means is that there is no necessary behavior (for now) for this particular request. We will put this to good use in the next section.

Since we had to stub out the enter() method on Programming to get finish CalculationShould, this automated check just works. The use of CHECK(1) at the end is an indication that this test is really a placeholder for now.

### 6.3.6 Record Steps for Execution

```
TEST(ProgrammingShould, RecordOperationsForExecute) {
    Programming op;
    op.execute(context, "+");
    op.execute(context, "-");
    CHECK(op.getMacro()->stepCountAtLeast(2));
}
```

*This requires adding a few things to Programming.h:*

```
class Macro;
class Programming: public CalculatorMode {
public:
    ...
    Macro *getMacro() { return macro; }

private:
    Macro *macro;
};
```

*And updating Programming.cpp*

```
#include "Macro.h"
#include "Context.h"
#include "MathOperationFactory.h"

Programming::Programming() : macro(new Macro) {
}

Programming::~Programming() {
    delete macro;
}

void Programming::execute(Context *context, const std::string &name) {
    MathOperation &op
        = context->getFactory()->findOperationNamed(name);
    macro->append(op);
}
```

*This uses a new method on context, get factory.*

*Update Context.h:*

```
class MathOperationFactory;

class Context {
```

```
public:
    ...
    virtual MathOperationFactory *getFactory() = 0;
```

And, finally, an update to *RpnCalculator.h*:

```
class RpnCalculator : public Context {
public:
    ...
    MathOperationFactory *getFactory() { return factory; }
```

### 6.3.7 Adding macro to factory

Here's an automated check:

```
#include "UnknownMathOperationException.h"
#include "MathOperationFactory.h"
TEST(ProgrammingShould, AddMacroToFactoryUponSave) {
    Programming op;
    op.execute(context, "+");
    op.execute(context, "-");
    op.save(context, "__add_sub__");
    try {
        context->getFactory()->findOperationNamed("__add_sub__");
        CHECK(1);
    } catch(UnknownMathOperationException &e) {
        FAIL("Operation not added to factory");
    }
}
```

To get this to work, we need to update *Programming.cpp*. Note that this work already exists in *RpnCalculator::save*, even so, I'll write a minimal version and then make sure all checks are passing before moving on:

```
void Programming::save(Context *context, const std::string &name) {
    context->getFactory()->add(name, macro);
    macro = 0;
}
```

### 6.3.8 Saving causes state change

Saving should also put the calculator back into calculation mode:

```
#include <typeinfo>
#include "Calculation.h"
TEST(ProgrammingShould, ChangeStateToCalculationUponSave) {
    Programming op;
    op.execute(context, "+");
    op.save(context, "__add__");
    CHECK(typeid(Calculation) == typeid(*context->getState()));
}
```

To get this working:

```
#include "Calculation.h"
void Programming::save(Context *context, const std::string &name) {
    context->getFactory()->add(name, macro);
```

```

macro = 0;
context->setState(new Calculation);
}

```

### 6.3.9 Other checking

A quick review of the `RpnCalculator::save` shows it does more checking than our current `Programming::save` mode. This needs to be fixed eventually, so now is as good of a time as ever. Here's a list of the other checks around saving from `RpnCalculatorShould`:

- Disallow Saving Under Existing Name
- Only Allow Valid Math Operations To Be Added
- Throw Exception When Attempting to Save Zero Length Macro
- Require Start to be called Before Save

Not all of these still make sense. For example, the calculator will only be in programming mode if `start()` has been called, so we can skip this check. The other checks seem to make sense, so let's add them (as a set):

*Disallow...*

```

#include "NameInUseException.h"
TEST(ProgrammingShould, DisallowSavingUnderAnExistingName) {
    Programming op;
    op.execute(context, "+");
    try {
        op.save(context, "+");
        FAIL("Should have thrown exception");
    } catch(NameInUseException &e) {
        CHECK(typeid(Calculation) == typeid(*context->getState()));
    }
}

```

This check fails as the state after `save()` should be `Calculation` but it is not.

Here's an update to `Programming::save()` to make that happen:

```

#include "Calculation.h"
#include "NameInUseException.h"
void Programming::save(Context *context, const std::string &name) {
    try {
        context->getFactory()->add(name, macro);
        macro = 0;
        context->setState(new Calculation);
    } catch(NameInUseException &e) {
        context->setState(new Calculation);
        throw e;
    }
}

```

*Valid Operations*

```

#include "UnknownMathOperationException.h"
TEST(ProgrammingShould, OnlyAllowValidOperations) {
    Programming op;
    try {

```

```

    op.execute(context, "bogus___");
    FAIL("Should have thrown exception");
} catch(UnknownMathOperationException &e) {
    CHECK(typeid(Calculation) == typeid(*context->getState()));
}
}

```

This fails as the last test failed, so we need to fix it as well.

```

#include "Calculation.h"
#include "UnknownMathOperationException.h"
void Programming::execute(Context *context, const std::string &name) {
    try {
        MathOperation &op
            = context->getFactory()->findOperationNamed(name);
        macro->append(op);
    } catch(UnknownMathOperationException &e) {
        context->setState(new Calculation);
        throw e;
    }
}

```

*Zero Length...*

```

#include "IllegalMacroException.h"
TEST(ProgrammingShould, DisallowZeroLengthMacro) {
    Programming op;
    try {
        op.save(context, "name");
        FAIL("Should have thrown exception");
    } catch(IllegalMacroException &e) {
        CHECK(typeid(Calculation) == typeid(*context->getState()));
    }
}

```

This makes the save method a bit unruly but similar to the original in RpnCalculator:

```

#include "NameInUseException.h"
#include "IllegalMacroException.h"
void Programming::save(Context *context, const std::string &name) {
    try {
        if(macro->stepCountAtLeast(1)) {
            context->getFactory()->add(name, macro);
            macro = 0;
            context->setState(new Calculation);
        } else {
            context->setState(new Calculation);
            throw IllegalMacroException();
        }
    } catch(NameInUseException &e) {
        context->setState(new Calculation);
        throw e;
    }
}

```

### 6.3.10 What about the start method?

A quick review of the Programming class reveals that the start() method is empty. It is an error to call start() when already in Programming mode, so let's make that a fact:

```
#include "InvalidRequestException.h"
TEST(ProgrammingShould, DisallowStart) {
    Programming op;
    try {
        op.start(context);
        FAIL("Should have thrown exception.");
    } catch(InvalidRequestException &e) {
        CHECK(1);
    }
}
```

This uses a new exception class:

```
#pragma once
#ifndef INVALIDREQUESTEXCEPTION_H_
#define INVALIDREQUESTEXCEPTION_H_

#include <exception>

struct InvalidRequestException : public std::exception { };

#endif
```

And it requires a little bit of work in Programming.cpp:

```
#include "InvalidRequestException.h"
void Programming::start(Context *context) {
    throw InvalidRequestException();
}
```

### 6.3.11 Ready to finish what we've started...

Now that we have an implementation for the state hierarchy, it is time to update our RpnCalculator to use it.

## 6.4 Updating RpnCalculator to use state...

Calculator already has state member data but it's initialized to 0. Update the constructor:

```
#include "Calculation.h"

RpnCalculator::RpnCalculator() :
    factory(new MathOperationFactory), macro(0), recording(false),
    mode(new Calculation) {
}
```

Most of the method in calculator now delegate to its mode member data:

```
void RpnCalculator::enter(int value) {
    mode->enter(this, value);
}
```



```
void RpnCalculator::execute(const std::string &operatorName) {
    mode->execute(this, operatorName);
}

void RpnCalculator::start() {
    mode->start(this);
}

void RpnCalculator::save(const std::string &macroName) {
    mode->save(this, macroName);
}
```

Verify that all of the automated checks are passing.

#### 6.4.1 Final Cleanup

Notice that the RpnCalculator has member data it no longer uses: macro, recording. Both of these can be removed from the header and source files, along with the forward declaration and #include of Macro.

#### 6.4.2 Summary

This is a somewhat incomplete implementation of the state pattern:

- The two kinds of modes, programming and calculation, know about each other. Often this is extracted out to a factory, but that seems like too much for only 2 states. This suggests, as described above, that this pattern is maybe overkill for this problem.
- The RpnCalculator also creates an instance of Calculate. Again, a factory would solve this problem.
- The places where state switches forces deallocation. This is OK, but notice that the object sending the message to switch state is ultimately the one that gets deleted. That turns out to be OK, but fragile.

## 7 Rpn Calculator – Sprint 4 – More Complex Blocks

Macros are interesting; it is time to add a few blocks. Here are some examples:

Example 1	Example 2	Example 3	Example 4	Example 5
<pre>65 8 . &lt;out&gt;8 emit &lt;out&gt;A cr &lt;out&gt;\n</pre>	<pre>start 2 * save times2 5 times2 10</pre>	<pre>start if 13 else 9 then save m1 8 m1 13 0 m1 9</pre>	<pre>start 2 ndup &lt; if drop else swap then save min 14 2 min 2</pre>	<pre>start 2 ndup &gt; if swap 1 - swap else then save down1 6 3 down1 5</pre>
Example 6	Example 7			
<pre>start 6 timesdo 2 * end save m6 3 m7 192</pre>	<pre>start begin 2 ndup &gt; while swap 1 - swap else then swap dup . swap save downto 6 2 downto &lt;out&gt;5 &lt;out&gt;4 &lt;out&gt;3 &lt;out&gt;2</pre>			

Example 1 demonstrates three new operations: “.”, “emit”, “cr”. These operations send output to the terminal. In our case, we’ll use `std::cout`, but not directly. The first, `.`, simply displays the top of the stack as a number and consumes that value. Emit, on the other hand, treats the number as a character and displays it, thus since 65 is the ASCII value of A, that is what is displayed. Finally, `cr` sends a new line to the output.

The second example demonstrates that a macro can contain constants. Those constants are pushed onto the stack during execution. This simple program simply multiplies what's on the stack by 2.

The third example shows a new composite operation, if—else. If the value at the top of the stack is non-zero, put 13 on the stack, else put 9 on the stack. It's a simple little program, but it demonstrates a whole new operation.

The fourth example uses the previous 2 examples to write a min function.

<fill in if I end up keeping those operations>

## 7.1 Output Operations

These operations produce output in some form. This introduces several problems:

- To what do we send output?
- How do we record it to verify that the correct output is in fact set?
- How do we bolt this into our current system?

The answer to the first question is simple: an object. We could use `std::cout`, but that introduces an unnecessary direct dependency upon `std::cout`. While certainly possible, and even a reasonable default behavior, our software will grow better if we avoid this direct connection to `std::cout`.

The answer to the second question is the same as the first: an object. This second bullet also suggests an additional requirement; we want this dependency to be injected into our system.

The final bullet begs a question and demonstrates a problem with depending on concrete objects. We have three new operations, all of them do things that our system was not designed to accomplish. Right now, all math operations depend on an `RpnCalculator`, which is sent in to the `perform` method. How can we easily fix this?

The work we did to introduce the state pattern has some of what we need. Instead of having operations depend on `RpnStack`, we could instead have them depend on `Context`. We can even accomplish this by migrating rather than redoing everything all at once.

Here's how we're going to do that:

- Create an automated check that uses a new interface on `MathOperation`.
- That math operation will be concrete initially.
- We'll get the three new operations working using the new `perform()` method.
- We'll then update individual automated checks to use the new `perform()` method.
- Once the checks are migrated, we'll slowly migrate individual math operations to use the new method and slowly remove the old version of `perform()`.
- Once we think we've fixed everything, we'll lean on the compiler to tell us what we're missing.

### 7.1.1 The “.” operator

Here's an automated check to get us started. This is a bit of a leap since we've decided to inject a dependent object. We end up creating several classes:

```
#include <CppUTest/TestHarness.h>
```

```

#include "Dot.h"
#include "OutputDestination.h"
#include "RpnCalculator.h"

TEST_GROUP(DotShould) {
};

#include <vector>
struct OutputDestinationSpy : public OutputDestination {
    std::vector<int> writtenInts;
    void writeInt(int value) {
        writtenInts.push_back(value);
    }
};

TEST(DotShould, SendTopValueAsNumber) {
    Dot op;
    OutputDestinationSpy *spy = new OutputDestinationSpy;
    RpnCalculator calculator(spy);
    calculator.enter(42);
    op.perform(&calculator);
    LONGS_EQUAL(42, spy->writtenInts[0]);
}

```

#### *OutputDestination.h*

This is an interface that gives us a level of indirection between the console and our system:

```

#pragma once
#ifndef OUTPUTDESTINATION_H_
#define OUTPUTDESTINATION_H_

class OutputDestination {
public:
    virtual ~OutputDestination() = 0;
    virtual void writeInt(int value) = 0;
};

#endif

```

There is an implementation of the destructor in a file called OutputDestination.cpp (not shown since we've done this several times).

Our RpnCalculator is constructed with a spy but how can we make sure that no existing tests are broken? Overload the constructor:

```

class OutputDestination;

class RpnCalculator : public Context {
public:
    RpnCalculator();
    RpnCalculator(OutputDestination *out);
}

```

```

...
OutputDestination *getOutput() { return out; }

private:
...
OutputDestination *out;

```

We've stuck to the forward declaration of `OutputDestination` and it is stored as a pointer. What do we do by default in the no-argument constructor versus the new constructor taking in an output destination?

#### *RpnCalculator.cpp*

```

#include "ConsoleOutputDestination.h"

RpnCalculator::RpnCalculator() :
    factory(new MathOperationFactory), mode(new Calculation),
    out(new ConsoleOutputDestination) {
}

RpnCalculator::RpnCalculator(OutputDestination *out) :
    factory(new MathOperationFactory), mode(new Calculation),
    out(out) {
}

RpnCalculator::~RpnCalculator() {
    delete out;
    delete mode;
    delete factory;
}

```

I've created a "real" implementation of `ConsoleOutputDestination` that simply writes directly to `cout`. It is the default type used. This gives backwards-compatibility with existing automated checks and allows for dependency injection via overloading.

The destructor assumes it owns the memory associated with the `out` member data. I also took the time to order deletes such that they are in the reverse order of allocation. This is unnecessary, but I like to do this because it makes my class behave more like auto-allocated objects.

#### *ConsoleOutputDestination.h*

```

#pragma once
#ifndef CONSOLEOUTPUTDESTINATION_H_
#define CONSOLEOUTPUTDESTINATION_H_

#include "OutputDestination.h"

class ConsoleOutputDestination: public OutputDestination {
public:
    ConsoleOutputDestination();
    ~ConsoleOutputDestination();
    void writeInt(int value);
};

```

```

#endif

ConsoleOutputDestination.cpp

#include "ConsoleOutputDestination.h"
#include <iostream>

ConsoleOutputDestination::ConsoleOutputDestination() {
}

ConsoleOutputDestination::~~ConsoleOutputDestination() {
}

void ConsoleOutputDestination::writeInt(int value) {
    std::cout << value;
}

```

### 7.1.2 It Should Be Registered...

It's been a little while since we wrote a new Math Operation. It'd be easy to forget to register it:

*Added to DotShould.cpp*

```

#include "ShouldBeRegistered.h"
CHECK_REGISTRATION(Dot, ".");

```

*Added to Dot.cpp*

```

#include "MathOperationRegistrant.h"
MathOperationRegistrant register_dot(".", new Dot);

```

## 7.2 Emit and a problem with growing interfaces...

First an automated check:

```

#include <CppUTest/TestHarness.h>

#include "Emit.h"
#include "RpnCalculator.h"

TEST_GROUP(EmitShould) {
};

#include "OutputDestinationSpy.h"

TEST(EmitShould, WriteCharacter) {
    Emit op;
    OutputDestinationSpy *spy = new OutputDestinationSpy;
    RpnCalculator context(spy);
    context.enter('A');
    op.perform(&context);
    CHECK_EQUAL_C_CHAR('A', spy->writtenChars[0]);
}

```

This check uses a spy. Rather than create a new spy class, I extracted it from the previous test and the used the same one in both places:

#### ***OutputDestinationSpy.h***

```
#pragma once
#ifndef OUTPUTDESTINATIONSPY_H_
#define OUTPUTDESTINATIONSPY_H_

#include <vector>
#include "OutputDestination.h"

struct OutputDestinationSpy : public OutputDestination {
    std::vector<int> writtenInts;
    std::vector<char> writtenChars;

    void writeInt(int value) {
        writtenInts.push_back(value);
    }

    void writeChar(int value) {
        writtenChars.push_back(value);
    }
};

#endif
```

This spy class needed to implement a new method on the OutputDestination interface. Unfortunately, so did the ConsoleOutputDestination.

#### ***OutputDestination.h***

```
class OutputDestination {
    ...
    virtual void writeChar(int value) = 0;
};
```

#### ***ConsoleOutputDestination.h***

```
class ConsoleOutputDestination: public OutputDestination {
    ...
    void writeChar(int value);
};
```

#### ***ConsoleOutputDestination.cpp***

```
void ConsoleOutputDestination::writeChar(int value) {
    std::cout << (char)value;
}
```

### 7.2.1 Emit should be registered

Add the missing check and implementation to EmitShould.cpp and Emit.cpp to make sure this new math operation is registered as “emit”.

### 7.3 Finally, cr

The cr operation sends “\n” to the console. We can repeat what we’ve just done:

```
#include <CppUTest/TestHarness.h>

#include "Cr.h"
#include "RpnCalculator.h"

TEST_GROUP(CrShould) {
};

#include "OutputDestinationSpy.h"

TEST(CrShould, WriteLine) {
    Cr op;
    OutputDestinationSpy *spy = new OutputDestinationSpy;
    RpnCalculator context(spy);
    op.perform(&context);
    CHECK_EQUAL(1, spy->linesWritten);
}

#include "ShouldBeRegistered.h"
CHECK_REGISTRATION(Cr, "cr");
```

#### *Cr.h*

```
#pragma once
#ifndef CR_H_
#define CR_H_

#include "MathOperation.h"

class Cr: public MathOperation {
public:
    void perform(Context *context);
    void perform(RpnStack &values);
};

#endif
```

#### *Cr.cpp*

```
#include "Cr.h"

#include "Context.h"
#include "OutputDestination.h"

void Cr::perform(Context *context) {
    context->getOutput()->writeLine();
}

void Cr::perform(RpnStack &values) {
}
```



```

#include "MathOperationRegistrant.h"
MathOperationRegistrant register_cr("cr", new Cr);

OutputDestination.h
class OutputDestination {
    ...
    virtual void writeLine() = 0;
};
#endif

ConsoleOutputDestination.h
class ConsoleOutputDestination: public OutputDestination {
    ...
    void writeLine();
};

ConsoleOutputDestination.cpp
void ConsoleOutputDestination::writeLine() {
    std::cout << std::endl;
}

OutputDestinationSpy.h
struct OutputDestinationSpy : public OutputDestination {
    ...
    int linesWritten;

    OutputDestinationSpy() : linesWritten(0) {}

    ...
    void writeLine() {
        ++linesWritten;
    }
};
#endif

```

## 7.4 Migrating to new Perform Interface

We need to find all places where the old perform method is called and, where possible, upgrade to the new interface that uses Context. This will be a bit of work since many automated checks use RpnStack.

What follows are a few such examples of the change followed by a list of other places you'll need to update. I'm working alphabetically:

### 7.4.1 BinaryMathOperation

This is the first direct descendant of MathOperation alphabetically so this is where we'll start:

#### **Update BinaryMathOperationShould.cpp**

First thing we can do is change the setup to use a calculator:

```
#include "RpnCalculator.h"

TEST_GROUP(BinaryMathOperationShould) {
    RpnCalculator *context;
    RpnStack *values;
    void setup() {
        context = new RpnCalculator;
        values = &context->getStack();
        ...
    }
    void teardown() {
        delete context;
    }
};
```

Applying this change maintains all checks. Next, we can change just one of the checks to use the new method:

```
TEST(BinaryMathOperationShould, ConsumeTwoValues) {
    BinaryMathOperationSpy spy(*values);
    spy.perform(context);
    LONGS_EQUAL(0, spy.size);
}
```

This does not work without a change to BinaryMathOperation. When you have multiple overloaded virtual methods but you only override one, C++ will stop searching up the hierarchy for overload resolution. To make C++ aware of all overloaded candidates you need to add a using statement:

```
class BinaryMathOperation: public MathOperation {
public:
    using MathOperation::perform;
    void perform(RpnStack &values);
};
```

Note that this makes the names available for overloaded name resolution. To define one of the methods, we still need to explicitly specify it as demonstrated by the last line.

Update the next few checks to use the different version of perform. Replace all:

```
spy.perform(*values);
```

With:

```
spy.perform(context);
```

Now update the test double in the file as well:

```
struct BinaryMathOperationSpy : public BinaryMathOperation {
    BinaryMathOperationSpy(Context *context)
        : context(context), size(-1), actualLhs(0), actualRhs(0) {}
    int calculate(int lhs, int rhs) {
        size = context->getStack().size();
        ...
    }
    Context *context;
    ...
};
```

```
};
```

For this to work we need to update each of the tests. Replace all:

```
BinaryMathOperationSpy spy(*values);
```

With:

```
BinaryMathOperationSpy spy(context);
```

Now the values member data in the TEST\_GROUP is no longer needed so we can go back to the setup and clean it up as well:

```
TEST_GROUP(BinaryMathOperationShould) {
    RpnCalculator *context;
    void setup() {
        context = new RpnCalculator;
        context->enter(4);
        context->enter(2);
    }
    void teardown() {
        delete context;
    }
};
```

This isn't perfect. The last automated check uses the stack, so it needs one final update:

```
TEST(BinaryMathOperationShould, StoreCalculatedResult) {
    BinaryMathOperationSpy spy(context);
    spy.perform(context);
    LONGS_EQUAL(13, context->getStack().top());
}
```

Notice that while we did make changes to several automated checks, we did not additionally change any production code nor did we get rid of any of the checks. So this seems like a stable operation.

#### 7.4.2 Update the calculator

We should be able to update the calculator to use the new perform method. Of course, with the introduction of the state pattern, the actually calling of the operations is in the Calculation mode class:

##### **Calculation.cpp**

```
void Calculation::execute(Context *context, const std::string &name) {
    MathOperation &op =
        context->getFactory()->findOperationNamed(name);
    op.perform(context);
}
```

We would be able to remove the old perform method from BinaryMathOperation if it were no longer required:

##### **MathOperation (starting to remove old perform)**

```
class MathOperation {
    ...
}
```

```
virtual void perform(RpnStack &values) {};  
};
```

Now that this method is no longer required and there's a default implementation, we should be able to update BinaryMathOperation to remove the old perform:

```
class BinaryMathOperation: public MathOperation {  
public:  
    virtual void perform(Context *context);  
    virtual int calculate(int lhs, int rhs) = 0;  
};
```

*Updated BinaryMathOperation.cpp*

```
#include "BinaryMathOperation.h"  
  
#include "Context.h"  
#include "RpnStack.h"  
  
void BinaryMathOperation::perform(Context *context) {  
    int rhs = context->getStack().top();  
    context->getStack().pop();  
    int lhs = context->getStack().top();  
    context->getStack().pop();  
    int result = calculate(lhs, rhs);  
    context->getStack().push(result);  
}
```

### 7.4.3 The magic of checks

I fully expected this to work, but it failed. Why? Well macros still call the old perform. How do I know this now? I ran my automated checks and two failed. The ones that failed are named:

```
TEST(RpnCalculatorShould, AllowMacrosToReferToOtherMacros)  
TEST(RpnCalculatorShould, BeAbleToRecordAndExecuteMacro)
```

So without running the debugger, I'm pretty sure where I need to target my efforts. A quick update to Macro.cpp fixes this as well:

*Macro.h*

```
class Macro: public MathOperation {  
public:  
    void perform(Context *context);  
    void perform(RpnStack &values);  
    void append(MathOperation &op);
```

*Macro.cpp (new perform added)*

```
void Macro::perform(Context *context) {  
    for(iterator i = operations.begin(); i != operations.end(); ++i)  
        (*i)->perform(context);  
}
```

Since we've just got back to all checks passing by updating macro, let's update MacroShould to use the new interface:

```

struct MathOperationSpy : public MathOperation {
    ...
    void perform(Context *context) {
    ...
    };

TEST(MacroShould, HandleMultipleMathOperations) {
    ...
    RpnCalculator context;
    ...
    op.perform(&context);
    ...
}

```

Now you can remove perform(RpnStack &) from the Macro class.

#### 7.4.4 The Newest Math Operations

The Cr, Dot and Emit classes have an unnecessary perform(RpnStack&) method. You can safely update all three classes by removing that version of perform from both places.

#### 7.4.5 Dup

Dup is next alphabetically in the list. First, we'll update DupShould.cpp:

```

#include <CppUTest/TestHarness.h>

#include "Dup.h"
#include "RpnCalculator.h"

TEST_GROUP(DupShould) {
    RpnCalculator *context;
    void setup() {
        context = new RpnCalculator;
        context->enter(4);
        context->enter(3);
        Dup().perform(context);
    }
    void teardown() {
        delete context;
    }
}

TEST(DupShould, LeaveSameValueInX) {
    LONGS_EQUAL(3, context->getX());
}

TEST(DupShould, HaveSameValueInY) {
    context->execute("drop");
    LONGS_EQUAL(3, context->getX());
}

TEST(DupShould, IncreaseStackSizeBy1) {
    LONGS_EQUAL(3, context->getStack().size());
}

```

```

}

TEST(DupShould, LeaveRemainderOfStackAlone) {
    context->execute("drop");
    context->execute("drop");
    LONGS_EQUAL(4, context->getX());
}

```

```

#include "ShouldBeRegistered.h"
CHECK_REGISTRATION(Dup, "dup");

```

*Dup.h*

```

class Dup: public MathOperation {
public:
    void perform(Context *context);
};

```

*Dup.cpp*

```

void Dup::perform(Context *context) {
    context->getStack().push(context->getStack().top());
}

```

#### 7.4.6 What Remains

There are several classes left to update. Note, in a large system, it might be OK to have both methods and do as we did, have one method call the other for backwards compatibility.

The following automated checks need to be updated along with their classes: Drop, FactorialShould, NDupShould, PrimFactorsOf, Sum, SwapXy.

Once you've updated all of these additional files, you can safely remove the old perform from the system and make the new perform pure virtual.

### 7.5 Numeric Constants as Operations

Our next example uses a constant value as part of a macro. Here's an automated check to reflect that example:

```

TEST(RpnCalculatorShould, AllowConstantValuesInMacros) {
    calculator->start();
    enter(2);
    execute("!");
    calculator->save("times2");
    enter(5);
    execute("times2");
    topWas(10);
}

```

This fails, but it shouldn't be too much to make this work:

*Update Programming::enter*

```

#include <sstream>
void Programming::enter(Context *context, int value) {
    std::stringstream name;

```

```
name << value;
if(context->getFactory()->hasRegistered(name.str()) == false) {
    MathOperation *op = new PushConstant(value);
    context->getFactory()->add(name.str(), op);
}
execute(context, name.str());
}
```

Add new method to MathOperationFactory (to header and source file of course):

```
bool MathOperationFactory::hasRegistered(const std::string &name) {
    return operationsByName.find(name) != operationsByName.end();
}
```

Notice, that this method could have been extracted from the add method:

```
void MathOperationFactory::add(
    const std::string &name, MathOperation *op) {
    if(hasRegistered(name))
        throw NameInUseException();
    operationsByName[name] = op;
}
```

## 7.6 If ... then ... else

## 8 Rpn Calculator – Sprint 5 – FitNesse & CSlim

8.1.1 A spec-driven example

8.1.2 A sequence diagram showing flow

8.2 **Adding a basic text ui**

8.3 **Adding several more operators**

8.3.1 ifelse

8.3.2 ntimesdo

8.3.3 ConditionWhileDo

8.4 **Programming the Calculator with a string**

8.4.1 Example forth program

8.4.2 Breaking it into parts

*Tokenization using spaces*

*Tokenization using regular expressions*

8.4.3 Building a Basic Sequence

8.4.4 Building a conditional sequence

8.4.5 Building a complex sequence

8.4.6 Adding the behavior into the calculator

8.4.7 Exercising the new behavior from the text ui

8.4.8 Saving your extensions



## 9 **Where to go next?**

### 9.1 **TDD Is Not Enough**

#### 9.1.1 GRASP

#### 9.1.2 SOLID + D

#### *Packaging Metrics*

#### 9.1.3 Code Smells

#### 9.1.4 WELC

#### 9.1.5 Test Doubles

#### 9.1.6 Coding Katas

#### 9.1.7 The 4 Actions (should be sooner)

## 10 **Appendix A: Revealing the Magician**

### 10.1 **Arrays versus pointers**

#### 10.1.1 Koenig's `i[3] == 3[i]` trick

### 10.2 **Methods versus functions**

### 10.3 **Operator Overloading**

### 10.4 **Overloading `<<`**

### 10.5 **Overloading `++i` versus `i++`**

### 10.6 **Virtual Functions**

### 10.7 **`new` & `delete` versus `malloc` & `free`**

## 11 **Appendix B: More Complex Composition with Bind**

## 12 **Appendix C: FitNesse, a quick introduction**

## 13 To Be Deleted

### 13.1 My Story Until 2010

My name is Brett L. Schuchert. I pronounce my last name as “shoe – heart” but I’m happy with many variations on that name. You can review my CV at <http://schuchert.wikispaces.com/MyCv>.

I started using computers on a paper terminal playing start trek in 7<sup>th</sup> grade. During 8<sup>th</sup> grade I nearly failed a typing class on manual typewriters and I started learning BASIC on an Apple II, mostly drawing low-res graphics. I took programming classes in high school; first BASIC, then COBOL and 6502 Assembly language. Because of the COBOL class, I took a typing class and learned to touch type (two spaces after each.) because my COBOL programs were hundreds to thousands of lines long. I learned 6010 Assembly on a Commodore 64 before taking the 6502 Assembly language course in High School. The processors were nearly the same but on the 6510 it was possible to view 64K of RAM as well as the ROM on top of the RAM by disabling interrupts and programming address 0 and 1 with a memory pattern.

I studied both Electrical Engineering and Computer Science at the University of Iowa. I first learned data structures and algorithms in Pascal. I followed that with a class on assembly language programming with two simulated assembly languages. The first assembly language did not have a stack, the second one did. The most important thing I did in that class was learn about activation records (stack frames) and we wrote recursive algorithms in assembly. It was quite useful.

Next in line was discrete mathematics. This gave me an appreciation of logic and a hate of program correctness proofs. At that time I thought them to be bunk. The only difference between then and now is I have a stronger argument that I won’t bore you with. I took operating systems courses, programming language foundations courses, my favorite language from that series being SNOBOL and learning about Bacus Naur Form.

In early 1989 I took an operating systems programming class from Mahesh Dodani. This turns out to be one of the most important classes because of professor Dodani. He allowed me to take that OO programming class, which allowed me to become a research assistant and learn C++. I that job, I worked with Jeff Francis, who knew a lot more about programming than I did. He taught me about revision control, specifically CVS. I’m amazed still today at places I go that don’t use such tools.

Working as a Research Assistant later allowed me to be an undergraduate teaching assistant in the College of Engineering, where I helped port an old embedded system programming class from assembly to C. In doing that, I had to port the cross-compiler. It was a K&R Style 1 compiler and I had to simply write `getc` and `putc`. I started by looking at the generated assembly. It was 68000, which I knew from an engineering course. I started with hand-coded assembly, and then I moved to embedded assembly in the C code. I then realized that all I was doing was reading memory locations, so I just cast the address in memory from a `void*` to a `char*` and did everything with macros. I mention this because it was at that point that I groked pointers fully. I nearly had them down before that experience, but that solidified pointers for me. Later, when I had to interact with an A to D converter, I just read an address. When I realized that function call

overhead was prohibitive on the platform, I redesigned the API to take an array and count. That was a valuable lesson about performance, which is really no longer necessary at that level but still relevant between systems.

While all of this was going on, I was also teaching computer literacy courses at Kirkwood Community college. By the time I was learning C++ and Smalltalk, I had 4 years of teaching under my belt to so-called non-traditional students. In one class I had a range of ages from 17 – 63.

When I started learning C++ I was using CFront 1.1. There were no books and the internet as we know it really didn't exist. The first book I bought was "The Annotated Reference Manual" with two "experimental" chapters, one on templates, and the other on exceptions. Later, I came across "Advanced C++ Programming Styles and Idioms" by James Coplien. I tore into that book and then started an email conversation with the author. That was exceptionally valuable for me because I learned quite a bit from the book and my conversations, and James introduced me to the world of book reviews. I started reviewing books (badly if I'm being honest), and this led to a book review by a relative unknown, Robert Martin. He would become my boss about 13 years later and be better known as Uncle Bob.

I worked 18 months at a startup in Dallas after leaving college. I thought I knew C++ but I was a typical arrogant college graduate who though he knew everything but really didn't know what was important. I did know the language well; I didn't understand the domain at all. I ended up using AWK quite a bit for data conversion and I worked as a technical business analyst.

During this time I managed to teach a few C and C++ classes at The University of Texas at Arlington. That's where I wrote my first TCP/IP application and my first client/server code. I was too stupid to not try to do so in class, live, making heavy use of the man command along the way. I learned at least as much as my students to be sure.

My next job was with a small company called Object Space. I joined as a trainer working with Graham Glass. At first it was C++ training and then Smalltalk training. I took the excellent outlines that Graham had intuitively developed and formalized them so I could teach them. This is when I started writing too many slides for classes. During this time we also added other classes on formal Object-Oriented analysis and design. I worked with Craig Larman on course development. I have a memory of he and I huddled together around a 13" monitor using Visio in Windows NT to design the flow for our C++ and Smalltalk classes to keep them in Sync.

We used an interesting design problem in that class, Monopoly. I learned a lot about analysis, design, programming, design patterns, formal software process and training, among other things, because of Monopoly

I started consulting at Object Space. I had gotten fully burnt out from training. I got to the point where (I thought) I knew all the questions I might get asked and had overly precise answers to each of those questions. My first gig out of training was to work on a Smalltalk project that eventually involved some C programming. I convinced the client to use C++ and all of a sudden I was training several people C++ on a real project using an OO database.

In 1994 I came across the STL from Stepanov and Lee. After learning it a bit, I started using it on this project. I remember having a 10,000 semicolon program that took 7 hours to compile on an HP because of templates. The same program took 15 minutes on a 400 MHz Pentium using Visual C++ 4.2.

Consulting at Object Space gave accelerated my learning. I got to experience things that didn't work without having to commit the error myself. I am certain that given the opportunity, I would have done many of the things my clients did exactly how they did it (I often did). So I managed to learn faster because I was seeing what did not work and then coming up with something that did work.

I worked as the "architect" for a three team distributed development effort starting in 1997 after learning Java. This is where I was introduced to JUnit and I became test-infected. So I've been writing unit tests in one form or another for about 13 years now. I worked on a Java phone project; I helped port a system from using MS tools to Java 1.02 and we used invisible applets for asynchronous DOM updates (it wasn't AJAX, but it was the same idea – everybody was doing it at that point). Later, related to that project, I was that architect that did the over-design of a project and then got pulled off for later development. I went back to see the damage my over design had caused, I did help remove some deadlock issues but that was a failure of over design even if the product did get used.

Near the end of my time at Object Space I was involved in intensive 8-week internal boot camps. This was again a great way for me to learn as well as teach and coach. Looking back I have to assume I was pretty bad in many ways. I know I was so-so handling the soft-skills side of things.

In 1997, along with switching to Java, I came across a book by Jerry Weinberg. I recognized his name from Exploring Requirements, which I loved, so I read this book and that started a journey that is still going. I took Problem Solving Leadership then Change Shop and later the SEM group, a Satir Yearlong program, SEM writer's workshop and the X workshop for developing experiential learning situations.

All of this started to take hold when I left Object Space and joined Valtech. I took classes that had burnt me out in the past, and redesigned them to make them more student-driven. While these were early experiments, they'd turn out to lead to something important later on.

My job at Valtech was similar to the one at Object Space. I taught and consulted. I was at Enron the day it went down (it wasn't my fault). As a result I went to England for 6 months and joined the largest project of my career. 343 people, I was #344. I learned quite a bit about just how bad a government project can go. After coming back from that job I started a 6-month project that turning into a 4-year project for me.

I was one of six people helping to port an existing COBOL solution to Java and train the COBOL programmers at the same time. That was an amazing experience. I learned more than I could have ever hoped. We put many applications into production, many with very low defects. The first project was 12 people, 9 – 10 months. In the first 9 months of production use, we found one defect. We were test infected at the time, which led to a

pretty low defect rate. We got lucky, we made a lot of mistakes and those came back to bite us.

Of the many things I learned on that project, one of them was a confirmation of the importance of automated testing. Even though we did a poor job on our unit tests, they were slow, dependent on data we did not own, etc., we were able to make significant changes without breaking things. On one occasion we need to make an architectural change across a number of applications (about 7). I warned the then 35ish developers that I'd be making a change on the weekend and they wanted to be checked in by Friday. I then changed 1,287 files across around 1.5 million lines of code and didn't break anything.

I made many mistakes but unlike many of my previous projects, I had to live with them and learn from them. I had to eat my own dog food. So it was a great experience with a lot of just amazing people both from Valtech and the Hertz.

While working on this project, I worked with Aspect Oriented Programming and introduced a solution to an ongoing problem. It was the right solution given the context but probably not the right one if we had the option to redo everything. I presented a talk on AOP to our Java group, at a private conference for Valtech and then I did something that would have far-reaching effects.

I attempted to write my talk in a self-paced experiential learning exercise. That got me started writing on Google pages first and then later wikispaces.com. I managed to get accepted at SD West on another subject but I was prepared to go to SD West because I was writing about what I was learning. I was doing that because I had read another book by Jerry Weinberg, "Weinberg on Writing." Since I started that site, I've written around 800 pages of material. From late 2008 to August 2010 I had 700,000 hits to my site. That doesn't count the first 2 years. Because I was presenting at SD West, I once again met Bob Martin. I had applied at Object Mentor in 2001 but did not take the job and instead joined Valtech. However, in 2007 things were different. Valtech made a similar mistake to Object Space that had essentially signed the death knell for Object space; they got rid of the training department.

When I discovered that had happened I was in a tutorial by Bob Martin. I discussed joining Object Mentor; two weeks later I had an offer. 6 weeks after that I finished up an internal boot camp I was teaching at Hertz and I joined Object Mentor.

For three years I worked at Object Mentor honing my Agile software development skills, like Test Driven Development. In those three years I came across many excellent people and projects, which just accelerated my learning.

Around mid-June 2010 my good friend David Nunn offered me the C++ classes. I had been teaching C++ at Object Mentor, primarily Test Driven Development in C++. I took where I was at and wrote a class that was very different from the kind of class I wrote in the early 1990's.

That class is the genesis of this book.



## 14 Index

- #define, 27
- #endif, 24, 27
- #ifndef, 27
- #include, 27
  - "" versus <>, 27
- #pragma, 23, 27
  - once, 27
- <tr1/random>, 47
- array, 47
- Assignment Operator, 53
- Automated Tests
  - Test Control, 55
  - Test Granularity, 33
  - Wisdom, 29
- C++ Idioms, 51
- C++ Recommendations
  - prefer initialization over assignment, 49
- C++ Standard Library
  - array, 47
  - begin, 48
  - end, 48
  - mt19937, 49
  - vector, 17
- C++ Terminology, 28
  - assignment, 47
  - Assignment Operator, 53
  - Compilation Unit, 16
  - const member function, 27
  - Copy Constructor, 53
  - declaration, 23
  - definition, 23
  - Definition, 28
  - Destructor, 53
  - initialization, 48
  - l-value, 48
  - Member function, 28
  - member-wise initialization list, 48
  - namespace, 49
  - non-primitive, 49
  - Object Module, 16
  - Operator Overloading, 53
  - operator(), 49
  - pre-increment, 49
  - primitive, 50
  - r-value, 50
  - static, 50
  - struct, 17
  - template class, 17, 50
  - typedef, 50
  - virtual, 54
- class, 27
- class versus struct, 28
- CommandLineTestRunner, 16
  - CommandLineTestRunner.h, 16
  - RunAllTests, 16
- command-query separation, 48
- Common Errors
  - Forgetting ; at the end of class, 26
  - Forgetting ; at the end of TEST\_GROUP, 26
  - Getting the signature incorrect, 26
- Compilation Unit, 16
- const member function, 27
- constructor
  - no-argument constructor, 49
- Constructor, 48
- Copy Constructor, 53
- CppUTest, 28
  - Auto Test Discovery, 15
  - Building CppUTest, 8
  - CHECK, 27
  - CommandLineTestRunner, 16
  - Downloading CppUTest, 8
  - LONGS\_EQUAL, 16
  - Mechanics of CppUTest, 9
  - Order of Tests, 34
  - RunAllTests, 16
  - TEST, 17, 28
  - Test Fixture, 34
  - TEST\_GROUP, 17, 28, 31
  - TestHarness.h, 17, 28
- Creating a Project, 10
- declaration, 23, 24, 27
- Declaration, 28
- definition, 23, 24, 25, 27
- Definition, 28
- Dependency Injection, 56
- Design Principles
  - command-query separation, 48

- Destructor, 53
- Don't Repeat Yourself, 31
- Downloading CppUTest, 8
- DRY, 31
- Eclipse CDT
  - Library Path, 11
- Eclipse CDT
  - Auto Save and Refresh, 11
  - C++0x, 11
  - Creating a Project, 10
  - Include Path, 10, 11
  - Included Libraries, 11
  - Installing Eclipse CDT, 6
  - Installing the Wascana Plugin, 8
  - Library Path, 10
  - Run Last Thing Execute, 12
  - Starting Eclipse CDT, 6
- Eclipse Shortcuts
  - Ctrl-B, 16
  - Ctrl-F11, 16
- function-object, 48
- functor, 48
- Include Directory, 16
- Include Path, 10
- Installing Eclipse CDT, 6
- Installing the Wascana Plugin, 8
- Library Directory, 16
- Library Path, 10
- LONGS\_EQUAL, 16
- Mechanics of CppUTest, 9
- Member function, 28
  - definition, 28
- member-wise initialization list, 48
- namespace, 49
- nested type, 49
- nested typedef, 49
- no-argument constructor, 49
- non-primitive, 49
- Object Module, 16
- Operator Overloading, 53
- operator(), 49
- Polymorphism, 57, 59, 61
  - Moving Parts, 57
- prefer initialization over assignment, 49
- pre-increment, 49
- primitive, 50
  - initialization, 50
- private:, 24, 50
- protected:, 24
- public:, 24, 28
- Refactoring, 21
- Refactroing, 44
- RunAllTests, 16
- r-value, 50
- Scope ::, 28
- static, 50
- struct versus class, 28
- template class, 50
- TEST, 17, 34
- Test Control, 55
- Test Fixture, 34
- Test Granularity, 33
- TEST\_GROUP, 17, 31, 32, 34
- TestHarness.h, 17
- typedef, 49, 50
- UML, 20
  - Class Diagram, 31, 55
  - Communication Diagram, 56
- vector, 17
- virtual, 54
- Wascana Plugin
  - Installing the Wascana Plugin, 8