Contents

Preface

I decided to write this book because I thought it would be interesting to cover Design Patterns using a test-driven approach—I'm a sucker for "killing two birds with one stone". Because of this choice, the chapter on Observer pattern serves dual purposes with a particular focus on getting the reader up to speed on TDD basics. More advanced TDD topics like proper use of *test doubles*, writing testable code, etc., will not be addressed (although I try to provide links for further investigation where appropriate). All this said, the primary subject of this book is not TDD, it's Design Patterns! Also, please be forwarned that we will not be able to be particularly rigorous in our test coverage and will likely be omitting a lot of defensive code that would be required of a "real system". I have chosen understandibility over robustness for an obvious reason—I don't want the reader to get lost in details that don't really pertain to the point being made. This is a fairly typical approach, but I apologize in advanced if I somehow insult your sensibilities!

Programming Languages

Most of the examples will be in Java, PHP, and JavaScript, but some might use other languages. This should be fine since:

- TDD and Design Patterns should be language agnostic
- The examples will be simple enough to follow (even if in an unfamiliar language)

Resources

I've generally listed any resources as clickable web links that you can learn more from, or, as links to where you might purchase the book that I'm referencing.

Line breaks in code

I've taken the liberty of purposely wrapping long lines that won't fit within the width of the page.

Also, I've found that the generated code samples for PHP do not show up correctly unless I start and end the sample code blocks out with corresponding PHP opening and closing tags. So that's why every darn PHP code snippet has these!

Contributions

I am definitely open to collaborative authorship (hey, I did put it on github!), provided that other authors follow the general style and spirit of the book. At some point, I'll try to define this all in a more concrete way. If you do want to contribute, you'll probably want to have a good look at the commented Makefile, and also notice the use of "extra lines" between code samples. I've managed to find workarounds for the somewhat finicky pandoc/docbook tool-chain (and I'm thankful that it works at all since these tools really make life so much easier!)

Special acknowledgement

I feel obliged to commend Addy Osmani on his countless community contributions, particularly in developer education, and acknowledge that seeing the impact of his work inspired me to write myself. Also, the book: Essential JS Design Patterns is where I shamelessly lifted the pandoc build process being used here!

Introduction

Why should we study design patterns?

As design patterns help us to use object-oriented best practices, it is useful to already have some meaningful experience in object oriented programming before attempting a deep study of design patterns. In fact, we have seen some try to dissuade a study of design patterns until the reader has reached an architect or designer skill level (whatever exactly that means). Obviously, the more prerequisite knowledge you have, the faster and you'll be able to grasp difficult abstract concepts; so perhaps there's some truth to the level of readiness that's ideal.

However, we would assert that you need not yet be an object oriented master to benefit from an initial study of design patterns. In fact, object oriented and good design go hand in hand—therefore, studying good design should reinforce your understanding of some object oriented concepts. Examples of design pattern implementations show object-oriented best practices "in action", and, since many of us learn best by example, the study of the two at the same time just might be complimentary.

Before reading this book, please take some of the following disclaimers and suggestions in to account:

- design patterns are an ongoing study; you'll likely learn new things when you return to the material at a later date
- therefore, assume that you'll need to return to the materials at later stages of your career
- if we don't explain something in a way that gets through to you, don't give up; try to look at a few similar examples on the web, or in other design pattern books, and look for the similarities—it'll start to make sense!
- Our goal is simply to "get your feet wet" with design patterns and not to cover them exhaustively.

TDD

As we have stated, TDD is *not* the focus of this book, just an approach we've used to examine design patterns. However, unlike the majority of the book, the next chapter on Observer pattern **does** try to act as a sort of "TDD: up and running" tutorial—we painstakingly examine a TDD session (as we implement the Observer pattern) and go through each test case one by one in order to quickly bring you up to speed in TDD.

If you already have a lot of experience doing test-driven development, you may be "put off" by the verbosity of that chapter. Feel free to jump ahead, and, rest assured that subsequent chapters will be much more "to the point".

Design Patterns

We would be remiss not to mention the pioneers known as the *Gang of Four* who, with their book Design Patterns: Elements of Reusable Object-Oriented Software, described several reusable patterns that can be used to solve common recurring software design dilemnas.

We will discuss several, but not all, of the GoF patterns, since they lay the framework from which new design patterns have evolved. As every pattern has its pros and cons, we will list the ones we've noticed as applicable. As we cover each particular pattern, we will not be adhering to any rigid *pattern structures* like those presented in the following section. Our goal is to distill the material in an accessible way. Hence, we advise you to view this book as a supplementary material on the subject.

Pattern Structures Formal design pattern descriptions are known to adhere to certain well-defined *pattern structures*. Christopher Alexandar is an architecture academic who, while pioneering the whole design pattern concept in the first place, provided the Alexandrian Form. The *Gang of Four* uses the Portland Form (which includes no less than: *intent, motivation, applicability, structure, implementation, sample code, known uses, and related patterns sections*).

Observer Pattern

Tip: *This is a long chapter that goes through the TDD process with a fine tooth comb. If you're already comfortable with TDD, feel free to skip to the bottom to see the implementation.*

Overview

Wikipedia describes the Observer Pattern as follows:

> The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods…

There are many event based systems that are quite similar in spirit to the Observer pattern such as: Pub/Sub, EventEmitter (node.js), NSNotification (Cocoa), etc. These all have in common the ability to dynamically notify one to many interested parties via some sort of broadcast mechanism. In the case of the Observer, a *subject* calls the update method of one to many *boserver* instances (it does this without needing to know anything specific about these parties other than that they implement a common interface to recieve said notifications). In object oriented circles, this decoupled use of composition and interfaces is called *programming to an interface not an implementation*. This is a key object oriented principle that allows us to maintain orthogonality in our systems.

Another feature of this pattern is that observers can be added or removed at runtime—thus providing a means to "hot swap" components easily when needed.

At its core the pattern involves the following steps:

1. Allow *observers* to register (and also unregister) to recieve notifications.
2. Keep these observer instances in a suitable data structure such as a *List* or *Array*.
3. When state changes (e.g. a new article is published, a user logs in, a track is played, etc.) these observers are notified by the *subject* who calls an interface defined update method on each of the observers.

Observer Pattern Class Diagram

The following is a class diagram that represents the Observer Pattern we'll be discussing in this chapter:

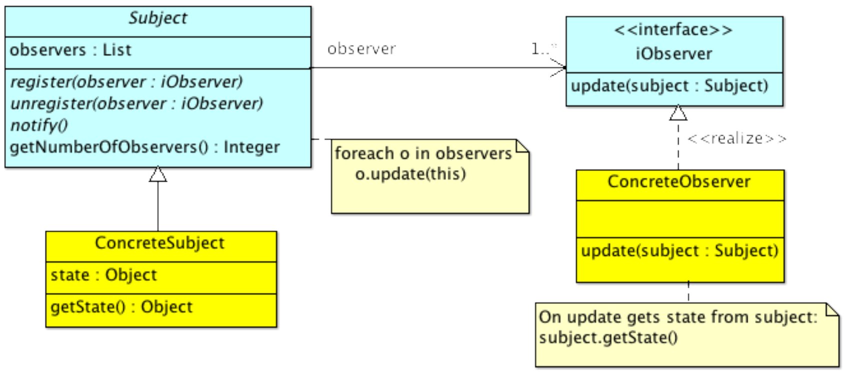

Figure 1: Observer Pattern Class Diagram

Note that there can be 1.. concrete subjects or observers.*

Implementing Observer Pattern using TDD

Let's use TDD to implement this pattern. Specifically, we'll be writing unit tests to test our Observer Pattern implementation as an isolated unit. Please be forwarned that I'll start out going through each and every "Red, Green, Refactor" cycle (so you can get a feel for this process), but will eventually start to condense things as we move forward. We'll be using the phpunit framework for this chapter so make sure you have that installed if you'd like to follow along.

```
<?php
require_once('Observer.php');

class ObserverTests extends PHPUnit_Framework_TestCase
{
    public function testSubscribing()
    {
        $publisher = new Subject();

    }
}
?>
```

And when we run phpunit we get:

```
$ phpunit ObserverTest.php # outputs:
Fatal error: Class 'Subject' not found in
/Users/rlevin/programming/labs/BOOK/research/observer/ObserverTests.php on line 8
```

So now we add the appropriate minimal code to get it green:

```
<?php
abstract class Subject
{
}
class ConcreteSubject extends Subject
{
}
```

And then we move on to finishing up our spec:

```
<?php
class ObserverTests extends PHPUnit_Framework_TestCase
{
    public function testSubscribing()
    {
        $observer = array();
        $subject = new ConcreteSubject();
        $subject->register($observer);
        $this->assertEquals(1, $subject->getNumberOfObservers(),
            "Subscriber should have 1 observer when 1 registered.");
    }
}
?>
```

Running phpunit again:

```
$ phpunit ObserverTest.php # outputs:
Fatal error: Call to undefined method ConcreteSubject::register() in
/Users/rlevin/programming/labs/BOOK/research/observer/ObserverTests.php on line 10
```

So we add the two methods:

```
<?php
public function register($observer)
{
}
public function getNumberOfObservers()
{
}
?>
```

And get following:

```
$ phpunit ObserverTests.php
Should be 1 observer when 1 added.
Failed asserting that null matches expected 1.
```

So we need to create the data structure to hold our list of observers:

```
<?php
abstract class Subject
{
}
class ConcreteSubject extends Subject
{
    private $observers = array();
    public function register($observer)
    {
        array_push($this->observers, $observer);
    }
    public function getNumberOfObservers()
    {
        return count($this->observers);
    }
}
?>
```

Refactoring

So now that we have our first test spec defined (and we're all green), we can take a step back and think about what in this code could be better. This is called **refactoring** (or Code Refactoring). Note that we only start refactoring when our tests are green.

So, is there something about this code that strikes you as odd? It should! We have an abstract class but we're not defining any abstract methods. In this case, the *register* method is really part of our Subject interface so we should add that abstract method to Subject:

```
<?php
abstract class Subject
{
    abstract public function register($observer);
    // abstract public function unregister($observer);
}
?>
```

Notice that we take the liberty of also adding the unregister abstract method, but we comment it out since we want to maintain a fairly fast "iterative rhythm".

You may have also noticed that we made a pretend observer object (we could have used a fake object, but in this case we knew we'd soon be implementing real observers). Let's get rid of that and create a very simple but real observer now:

```
<?php
public function testSubscribing()
{
    $observer = new ConcreteObserver();
    ...
?>
```

And we get a failure saying it's not defined so...

```
<?php
interface iObserver
{
}
class ConcreteObserver implements iObserver
{
}
?>
```

Ok, so we've proven that we can push an object on to a PHP array. "Big deal!", you say. Sure, but we now have the benifit of having some test coverage to build on top of as we introduce more "exciting" features.

"Hot swapping" components

One of the features of Observer pattern is the ability for observers to attach and detach themselves at runtime. There are many uses for this, but let's say you wanted to temporarily re-route a minimal production logging system with a more verbose logging system to track down an issue. Rather than taking down the whole system, you could simply swap out the ProductionLogger observer and swap in a VerboseLogger simply calling *unregister* and *register* respectively. Then, upon fixing the issue, you could do the reverse, and thus, swap the components back as they were.

So let's get to work on that *unregister* method…

```
<?php
public function testUnsubscribing()
{
    $observer1 = new ConcreteObserver();
    $observer2 = new ConcreteObserver();
    $subject = new ConcreteSubject();
    $subject->register($observer1);
    $subject->register($observer2);
    $this->assertEquals(2, $subject->getNumberOfObservers(),
        "Should be 2 observers when 2 added.");
    $subject->unregister($observer1);
    $this->assertEquals(1, $subject->getNumberOfObservers(),
        "Should be 1 observer when 1 when one removed.");
}
?>
```

Of course we get:

```
**Fatal error: Call to undefined method ConcreteSubject::unregister()
```

At this point I'm not going to be listing every step (like when we define a class or method when we have an undefined test error just to get the test to pass) but, if *you're following along, remember to take small steps between coding and running your tests.*

We need to define an *unregister* method which we do as follows:

```php
<?php
public function unregister($observer)
{
    foreach ($this->observers as $k => $v)
    {
        if ($observer == $v)
        {
            unset ($this->observers[$k]);
            return true;
        }
    }
    return false;
}
?>
```

And we're all green again. Our implementation so far is:

```php
<?php
abstract class Subject
{
    abstract public function register($observer);
    abstract public function unregister($observer);
}

class ConcreteSubject extends Subject
{
    private $observers = array();

    public function register($observer)
    {
        array_push($this->observers, $observer);
    }

    public function unregister($observer)
    {
        foreach ($this->observers as $k => $v)
        {
            if ($observer == $v)
            {
                unset ($this->observers[$k]);
                return true;
            }
        }
        return false;
    }

    public function getNumberOfObservers()
    {
        return count($this->observers);
    }
}

interface iObserver
{
}
class ConcreteObserver implements iObserver
{
}
?>
```

Refactoring Pull Up

At this point, it's debatable whether Subject should be abstract or an interface since we aren't supplying any default behavior. Having a quick look, it seems we can "pull up" the getNumberOfObservers method. We pull it up into our abstract Subject class as follows:

```
<?php
abstract class Subject
{
    protected $observers = array();
    abstract public function register($observer);
    abstract public function unregister($observer);
    public function getNumberOfObservers()
    {
        return count($this->observers);
    }
}
?>
```

Note that getNumberOfObservers is a method that I took the liberty of creating to make the Subject more testable. It's not actually part of the Observer pattern.

DRY (don't repeat yourself)

Ok, that looks better but I don't like our testUnsubscribing spec. It's too long for such a simple test and also, if you look caefully at testSubscribing, you'll see that we have duplicate code breaking the DRY Principle. Essentially, the

DRY principle states that we want to avoid duplication since that requires us to maintain duplicate code bases in parallel.

Setup and Teardown

We can pull that duplicate code up in to a setUp method (setUp is used to set up state before each test case run; as expected, tearDown executes just after each test case run). Doing so will mean we have to go through and change our references from the local to class level like so:

```php
<?php
require_once('Observer.php');

class ObserverTests extends PHPUnit_Framework_TestCase
{
    private $observer = null;
    private $observer2 = null;
    private $subject = null;

    public function setUp()
    {
        $this->observer = new ConcreteObserver();
        $this->observer2 = new ConcreteObserver();
        $this->subject = new ConcreteSubject();
    }
    public function tearDown()
    {
        $this->observer = null;
        $this->observer2 = null;
        $this->subject = null;
    }

    public function testSubscribing()
    {
        $observers = $this->registerMany(1);
        $this->assertEquals(1, $this->subject->getNumberOfObservers(),
            "Subscriber should have 1 observer when 1 observer registered.");
    }

    public function testUnsubscribing()
    {
        $this->subject->register($this->observer);
        $this->subject->register($this->observer2);
        $this->assertEquals(2, $this->subject->getNumberOfObservers(),
            "Should be 2 observers when 2 added.");
        $this->subject->unregister($this->observer);
        $this->assertEquals(1, $this->subject->getNumberOfObservers(),
            "Should be 1 observer when 1 when one removed.");
    }
}
?>
```

You should now notice that—since our last refactoring—the test case has become much more readable and easy to understand. Thus we didn't just reduce duplication (but more importantly) made our tests self documenting.

For this exercise, we're using phpunit's setUp/tearDown combination (similar to JUnit if you're a Java developer), but you'll see other similar hooks in other frameworks like beforeEach/afterEach (Jasmine), begin/done (QUnit), etc. These methods are called just before and after each test case allowng you to set up (and tear down) state. This ensures that each test is ran in isolation not depending on any leftover state from previous test case runs.

Helpers

I'd like to do one more refactoring before moving on—I'm not happy with the way we're registering our observers within each test case. Let us try moving that code in to a helper function:

```php
<?php
    public function testUnsubscribing()
    {
        $observers = $this->registerMany(5);
...
    private function registerMany($n)
    {
        $observers = array();
        foreach (range(0, $n-1) as $key) {
            $observers[$key] = new ConcreteObserver();
            $this->subject->register($observers[$key]);
        }
        return $observers;
    }
}
?>
```

The above helper method is a direct lift from the ***rollMany*** *helper function in the famous bowling game kata. If you haven't heard of it you should take a quick look at Uncle Bob's Bowling Game Kata. Read the short article and perhaps schedule time in your calendar to go through the power point presentation step by step. Better yet, play with the kata using your favorite language(s) in your "spare time"... it's quite instructive. Here's an example of using phpunit with BDD to solve Bowling Game.*

A Slight Detour: What should I test?

Determining just what to test is a difficult task but here are some general guidelines to help with that process.

Structured Basis Testing The main idea of Structured Basis Testing is that your test cases, for a particular peice of code, are derived from the code's logic and you try to have tests in place for every possible branch. So at it's simplest:

```php
<?php
function foo() {  // count 1 for the function itself
    if (something) { // count 2 for this condition
        // ...
    } else if (somethingElse) { // count 3 for this condition
        // ...
    }
    // We count one for function itself because this might be
    // all that's executed if neither above are truthy!
    return true;
}
?>
```

This is also sometimes referred to as *logic coverage testing*. The programmer's tome Code Complete 2 discusses this in depth in it's *Chapter 22: Developer Testing*.

Boundary Conditions Testing The Practice of Programming tells us that we should:

> Test code at its boundaries... The idea is that most bugs occur at boundaries.

When you test production code that can take a range of valid values, use just below minimum, minimum, maximum, and just below and exceeding maximum values. This takes discipline and time (but given that boundaries are likely places where errors creep up) it's well worth the effort.

Also keep in mind that there might be compound boundaries when 2 or more variables interact. An obvious example is when you multiply two numbers. In this case, you have the potential to inadvertantly exceed the maximum value.

Known Error Testing Is there an area that is likely to be problemattic? If so, make sure to test for that area.

An example might be if you were to implement a "time picker". Setting aside time zone and daylight savings concerns, we'd want heavy coverage on times known to be error prone such as: 00:00, 12:00am, 12:00pm (keep in mind the user's time format might be 12 or 24 hour format!) In addition, you may also want to use: 23:59, 11:59pm, 11:59am, 00:01, 12:01am, 12:01pm, etc.

Data-Flow Testing This is also discussed in the Code Complete 2 book, and you should turn to that resource for more in depth coverage. However, essentially, data-flow testing provides that *data usage* is just as problemattic as control flow, and that you therefore need to take into account of the various possible states your data structures might be in (e.g. Defined, Used, Killed, Entered, Exited, etc.) Data flow testing is beyond the scope of this book but you can read more in this paper if you're interested.

Cross Checking If you have to implement something that's been reliably implemented elsewhere, you may be able to *cross check* your results against the existing implementation:

```
actual = MyMath.squareRoot(n);
expected = Math.sqrt(n);
assertEquals(actual, expected,
"My implementation of square root should be same as Java's");
```

In this chapter, we are focusing on using TDD to unit test our Observer implementation thus leaving out many other important types of testing: Integration Testing, Systems Testing, Stress Testing, etc. In a real system, you will ideally have a suite of complimentary tests that include all of the aforementioned within an Continuous Integration System. You should also consider using a tool to measure your test code coverage.

Testing both the "happy" and "sad" paths

Gary Bernhardt, of Destroy All Software, uses "happy" and "sad" to describe the normal and abnormal code paths. At minimum, both of these paths—sometimes also called the nominal and non-nominal paths (but I like Gary's happy/sad better!)—should get proper test coverage. Perhaps so far we've been a bit too joyful!

```
<?php
    public function testSubscribingWithFalsy()
    {
        $observers = $this->subject->register(null);
        $this->assertEquals(0, $this->subject->getNumberOfObservers(),
            "Should be 0 observers if register called with a falsy.");
    }
?>
```

Which results in:

```
Failed asserting that 1 matches expected 0.
```

So we change our system under test to be:

```
<?php
    public function register($observer)
    {
        if (!empty(\$observer))
        {
            $this->observers[] = $observer;
        }
    }
?>
```

And we're back to passing. However, not only could we pass in falsy arguments, but we might also pass in a non-observer as well. So we need to defend against that as well:

```
<?php
    public function testSubscribingWithNonObserver()
    {
        $nonObserver = "I'm not a legal observer!";
        $observers = $this->subject->register($nonObserver);
        $this->assertEquals(0, $this->subject->getNumberOfObservers(),
            "Should be 0 observers if register called with a falsy.");
    }
?>
```

Obviously, we need some type hinting help here and so we try to change our abstract Subject's interface to be like:

```
<?php
    abstract public function register(iObserver $observer);
    abstract public function unregister(iObserver $observer);
?>
```

But doing this results in the following goliath error output:

```
1) ObserverTests::testSubscribingWithFalsy
Argument 1 passed to ConcreteSubject::register()
must implement interface iObserver, null given...

2) ObserverTests::testSubscribingWithNonObserver
Argument 1 passed to ConcreteSubject::register()
must implement interface iObserver, string given...
...
```

But if we look carefully at these errors, we see that the type hinting is actually doing it's job thus disallowing us to pass both NULL and String to register! Let's remove the following two test cases:

```
    // public function testSubscribingWithFalsy()
    // {
    //     $observers = $this->subject->register(null);
    //     $this->assertEquals(0, $this->subject->getNumberOfObservers(),
    //         "Should be 0 observers when register called with a falsy.");
    // }
    // public function testSubscribingWithNonObserver()
    // {
    //     $nonObserver = "I'm not a legal observer";
    //     $observers = $this->subject->register($nonObserver);
    //     $this->assertEquals(0, $this->subject->getNumberOfObservers(),
    //         "Should be 0 observers when register called with a falsy.");
    // }
```

After said removal, we're back to passing again, and we've made our production code easier to read:

```php
<?php
    public function register(iObserver $observer)
    {
        $this->observers[] = $observer;
    }
?>
```

Obviously, in a real system we'd need to think of other "sad paths", but it's interesting to note that the mere process of testing has helped us to remove needless code. For example, we don't have to add code like:

```
    if (is_object($newSubject) &&
        $newSubject instanceof Subject) {
    ...
```

Let’s continue to look for other possible “sad paths”:

```php
<?php
    public function testUnsubscribingWhenNone()
    {
        $actual = $this->subject->unregister(new ConcreteObserver());
        $this->assertEquals(false, $actual,
            "Unregister returned status should be false if no observers");
    }
    public function testGetNumberObserversWhenNone()
    {
        $this->assertEquals(0, $this->subject->getNumberOfObservers(),
            "Should be 0 observers when none.");
    }
?>
```

We try the above and they both pass without any changes. Since the getNumberOfObservers is really just a helper to facilitate testing (and is quite simple), let’s remove that test. We only want “useful” tests.

Before we can implement our observers we need to add a capability to the subject. In order to be useful, we need to be able to set and get some sort of meaningful data from the Subject. Let’s do that now:

```php
<?php
    public function testSubjectShouldHoldState()
    {
        $stateAsString = "some state";
        $this->subject->setState($stateAsString);
        $actual = $this->subject->getState();
        $this->assertEquals($stateAsString, $actual,
            "Should get/set state as string");

        $stateAsArray = array('foo' => 'foo val');
        $this->subject->setState($stateAsArray);
        $actual = $this->subject->getState();
        $this->assertSame($stateAsArray, $actual,
            "Should get/set state as array");
    }
?>
```

And the obvious implementation:

```php
<?php
    public function setState($state)
    {
        $this->state = $state;
    }
    public function getState()
    {
        return $this->state;
    }
?>
```

I'm not pleased with the length of the test case and don't think it's too useful to test both for types String and Array. It was the first time through but not for repeatable test suite runs. So I'll remove the String part leaving just:

```
<?php
    public function testSubjectShouldHoldState()
    {
        $stateAsArray = array('foo' => 'foo val');
        $this->subject->setState($stateAsArray);
        $actual = $this->subject->getState();
        $this->assertSame($stateAsArray, $actual,
            "Should get/set state as array");
    }
?>
```

Again, we want readable and meaningful tests so we sometimes have to make these sorts of judgement calls, and I'm going to favor brevity in this case.

Now that we have a way to hold state, we can start implementing our observer interface:

```
<?php
    public function testObserverGetsNotified()
    {
        $subject = new ConcreteSubject();
        $observer = new ConcreteObserver($subject);
        $state = 'yogabbagabba';
        $subject->setState($state);
        $this->assertEquals($state, $observer->getState(),
            "Setting state in subject notifies observer");
    }
?>
```

And then:

```php
<?php
interface iObserver
{
    public function update (iSubject $subject);
}
class ConcreteObserver implements iObserver
{
    private $state;
    public function update (iSubject $subject)
    {
        $this->state = $subject->getState();
    }
    public function getState()
    {
        return $this->state;
    }
}
?>
```

Resulting in:

```
1) ObserverTests::testObserverGetsNotified
Setting state in subject notifies observer
Failed asserting that null matches expected 'yogabbagabba'.
```

Our observer looks ok, but it's likely not getting notified (duh, we didn't implement that yet!)...

Implementing Observers

So we're at a sort of critical point here. We've implemented quite a bit of code, and we still have more (the notify observers routine for example). Moreover, we can feel our blood pressure rise and our confidence level dropping. What to do? Simply back up a step. Get those tests green again!

```
    // public function testObserverGetsNotified()
    // {
    //     $subject = new ConcreteSubject();
    //     $observer = new ConcreteObserver($subject);
    //     $state = 'yogabbagabba';
    //     $subject->setState($state);
    //     $this->assertEquals($state, $observer->getState(),
    //         "Setting state in subject notifies observer");
    // }
```

We've simply commented out our last test case, re-ran our tests and we're green:

```
OK, but incomplete or skipped tests!
Tests: 6, Assertions: 6, Incomplete: 1.
```

We can take a deep breath feeling reassured that we haven't created a 30 minute debugging session for ourselves. Also note that we've left the "partially implemented" changes in our production code. So we've actually made some progress given that, so far, we haven't caused a regression. We'll take a bit of a risk, and leave those change in while we implement notifications:

```php
<?php
    public function testObserverGetsNotified()
    {
        $this->subject->register($this->observer);
        $state = 'yogabbagabba';
        $this->subject->setState($state);
        $this->subject->notify();
        $this->assertEquals($state, $this->observer->getState(),
            "Setting state in subject notifies observer");
    }
?>
```

This was derived from the test case we commented out earlier. If it's not obvious, we've defined $this->observer in setUp. Here's our full implementation so far:

```php
<?php
abstract class Subject
{
    protected $observers;
    protected $state;

    abstract public function register(iObserver $observer);
    abstract public function unregister(iObserver $observer);

    public function getNumberOfObservers()
    {
        return count($this->observers);
    }
}

class ConcreteSubject extends Subject
{
    public function __construct()
    {
        $this->observers = array();
    }
    public function register(iObserver $observer)
    {
        $this->observers[] = $observer;
    }

    public function unregister(iObserver $observer)
    {
        foreach ($this->observers as $k => $v)
        {
            if ($observer == $v)
            {
                unset ($this->observers[$k]);
                return true;
            }
        }
        return false;
    }

    public function notify()
    {
        foreach ($this->observers as $observer) {
            $observer->update($this);
        }
    }
    public function setState($state)
    {
        $this->state = $state;
    }

    public function getState()
    {
        return $this->state;
    }
}

interface iObserver
{
    public function update (Subject $subject);
}
class ConcreteObserver implements iObserver
{
    private $state = null;
    public function update (Subject $subject)
    {
        $this->state = $subject->getState();
    }
    public function getState()
    {
        return $this->state;
    }
}
?>
```

Notice that by using a sprinkle of type hinting, we don't have to resort to using things method_exists checks to determine if the subject passed into *update* actually has a *getState* method... it can be safely assumed since *Subject* defines that as an abstract (thus required) method.

Unique observers We'd like to prevent an exact observer instance getting included in our subject's observer *List* twice. There are several approaches we could take to solve this, but let's go with adding an id property to each observer instance and then utilizing array_unique.

First the ID part:

```
<?php
// we add this test
    public function testObserversHaveIds()
    {
        $this->assertNotNull($this->observer->getID());
    }

// and we add this to observer:
    private $state = null;
    private $id;
    public function __construct()
    {
        $this->state = null;
        $this->id = uniqid (rand(), true);
    }
    public function getID()
    {
        return $this->id;
    }
?>
```

Now that our observers have IDs, we can have our subject enforce uniqueness in *register*. First let's see this fail:

```
<?php
    public function testAddingDuplicates()
    {
        $this->subject->register($this->observer);
        $this->subject->register($this->observer);
        $this->assertEquals(1, $this->subject->getNumberOfObservers(),
            "Should only add a particular observer instance once");
    }
?>
```

This results in:

```
1) ObserverTests::testAddingDuplicates
Should only add observer once
Failed asserting that 2 matches expected 1.
```

In order to use array_uniquewe need to first implement a ___toString in our observer.

We first comment out testAddingDuplicates, verify our tests are again passing, add the **toString method, and once more verify our tests are still passing. After safely adding** toString, we uncomment testAddingDuplicates and continue onwards on our mission to disallow duplicates. All we have to at this point is to call array_unique when we register our observers like so:

```
<?php
    public function register(iObserver $observer)
    {
        $this->observers[] = $observer;
        $this->observers = array_unique($this->observers);
    }
?>
```

Our test is now passing and we've ensured all our observers are, in fact, unique instances.

Final Implementation

At this point, we have a fairly complete Observer implementation that could be further extended easily should we so desire.

ObserverTests.php

```
<?php
require_once('Observer.php');

class ObserverTests extends PHPUnit_Framework_TestCase
{
    private $subject = null;
    private $observer = null;

    public function setUp()
    {
        $this->subject = new ConcreteSubject();
        $this->observer = new ConcreteObserver($this->subject);
    }
    public function tearDown()
    {
        $this->subject = null;
        $this->observer = null;
    }

    public function testSubscribing()
    {
        $observers = $this->registerMany(1);
        $this->assertEquals(1, $this->subject->getNumberOfObservers(),
            "Subscriber should have 1 observer when 1 observer registered.");
    }

    public function testUnsubscribing()
    {
        $observers = $this->registerMany(5);
        $this->assertEquals(5, $this->subject->getNumberOfObservers(),
            "Should be 5 observers when 5 added.");
        $this->subject->unregister($observers[0]);
        $this->assertEquals(4, $this->subject->getNumberOfObservers(),
            "Should be 4 observers when 1 when one removed.");
    }
    public function testUnsubscribingWhenNone()
    {
        $actual = $this->subject->unregister(new ConcreteObserver());
        $this->assertEquals(false, $actual,
            "Unregister returned status should be false if no observers");
    }
    public function testSubjectShouldHoldState()
    {
        $stateAsArray = array('foo' => 'foo val');
```

```php
        $this->subject->setState($stateAsArray);
        $actual = $this->subject->getState();
        $this->assertSame($stateAsArray, $actual,
            "Should get/set state as array");
    }
    public function testCreateObserver()
    {
        $actual = new ConcreteObserver(new ConcreteSubject());
        $this->assertNotNull($actual, "Should create observer");
    }
    public function testObserverGetsNotified()
    {
        $this->subject->register($this->observer);
        $state = 'yogabbagabba';
        $this->subject->setState($state);
        $this->subject->notify();
        $this->assertEquals($state, $this->observer->getState(),
            "Setting state in subject notifies observer");
    }
    public function testObserversHaveIds()
    {
        $this->assertNotNull($this->observer->getID());
    }
    public function testAddingDuplicates()
    {
        $this->subject->register($this->observer);
        $this->subject->register($this->observer);
        $this->assertEquals(1, $this->subject->getNumberOfObservers(),
            "Should only add a particular observer instance once");
    }
    private function registerMany($n)
    {
        $observers = array();
        foreach (range(0, $n-1) as $key) {
            $observers[$key] = new ConcreteObserver();
            $this->subject->register($observers[$key]);
        }
        return $observers;
    }
}
?>
```

Observer.php

```php
<?php
abstract class Subject
{
    protected $observers;
    protected $state;

    abstract public function register(iObserver $observer);
    abstract public function unregister(iObserver $observer);
    abstract public function notify();

    public function getNumberOfObservers()
    {
        return count($this->observers);
    }
}

class ConcreteSubject extends Subject
{
    public function __construct()
    {
        $this->observers = array();
    }
    public function register(iObserver $observer)
    {
        $this->observers[] = $observer;
        $this->observers = array_unique($this->observers);
    }

    public function unregister(iObserver $observer)
    {
        foreach ($this->observers as $k => $v)
        {
            if ($observer == $v)
            {
                unset ($this->observers[$k]);
                return true;
            }
        }
        return false;
    }

    public function notify()
    {
        foreach ($this->observers as $observer) {
            $observer->update($this);
        }
    }
    public function setState($state)
    {
        $this->state = $state;
    }

    public function getState()
    {
        return $this->state;
    }
}

interface iObserver
{
    public function update (Subject $subject);
}
class ConcreteObserver implements iObserver
{
    private $state = null;
    private $id;
    public function __construct()
    {
        $this->state = null;
        $this->id = uniqid (rand(), true);
    }
    public function getID()
    {
        return $this->id;
    }
    public function __toString()
    {
        return "id: " . $this->getID();
    }
    public function update (Subject $subject)
    {
        $this->state = $subject->getState();
    }
    public function getState()
    {
        return $this->state;
    }
}
?>
```

Considerations

Here are some further considerations when implementing Observer pattern that we haven't discussed...

SPL

It should be noted that we could have used the PHP 5 Standard Library's SplObserver and SplSubject instead of rolling our own interfaces. However, doing so was helpful in getting a clear understanding of exactly how the Observer Pattern works. If you're interested learning more about SPL, you may want to have a look at the following resources: SPL, SplSubject, and SplObserver. *Note that instead of register/unregister they use the names attach/detach.*

Push vs. Pull

It should also be noted there are different strategies for how the state is obtained by the observers. In this chapter's example, we used the *pull method* since our observers called:

```
$subject->getState()
```

Thus, upon being notified, we "pulled in" the changes from the subject. However, the inverse is also possible—where the subject *pushes* changes down to the observers—and, approriately enough, is called the *push method.* An example of how *push method* might work is the following:

```
$observer->update($this, $user, $trackPlayed,
    $action, [..more state info..]);
```

Here, the subject has explicitly included the pertinent state information in the call to update.

The up side here is that the observers need not call:

```
subject->getState()
```

However, there's bigger downside—loss of reuse. When we keep the state object in the subject generic enough, it's easy to reuse the Observer code (since the state object abstracts itself in a model, value object, etc.) However, if we (in contrast to using a generic state object) use a specific method signature to push state, we tightly couple our subject to the observers. In general, favor the *pull method* over the *push method.*

Pitfalls

Warning: This section contains some opinionated suggestions. Use your own judgement and form your own opinions!

The following are some potential issues to look out for when implementing Observer pattern:

- **Cyclic dependencies**: Allowing observers to set state reduces orthogonality. It is our opinion that another "impartial module" should be responsible for providing state updates to the subject. If you do need this coupling consider another design approach or implementing a Mediator to manage these interactions.
- **Relationship hierarchies**: Insidious coupling can occur if you allow observers to interact with each other. Strive to design observers to be completely unaware of each other.
- **Wasteful updates**: If an observer only has a particular *interest*, recieving all updates is wasteful. This can be remedied by adding an *interest* input to the *register* signature and checking on a per observer basis before broadcasting to that observer.
- **Spurious updates**: In a high throughput system with many observers, there are potential temporal issues. For example, a call to setState() before a previous state has been fully pushed out could result in notifications being sent to observers that have yet to receieve the previous state. The obvious workaround is to queue these states and only start notifications for the "newest state" once "previous state" notification are complete. This of course adds additional complexity.
- **Unpredictable ordering**: This pattern does not provide predictable ordering of observer updates and doing so increases coupling.
- **Inconsistent state**: If setting state is more than a simple assignment, you must ensure *consistent state* before broadcasting updates.
- **Duplicate observers**: It's easy to inadvertandly create a system where an observer is recieving "double updates" as a result of duplicates observers having been registered.

We have other decisions to make when we design our implementation (who should call notify? what happens when a subject is deleted? will there be orphaned observers?). These require further research and are beyond the scope of this chapter.

Example Uses

What are some example applications that might benefit from the Observer Pattern?

Here are a couple ideas:

- **Mailing List**: Let's say you have a system that periodically adds new articles to be published. Upon a new article becoming available, the system could make a call to *setState* and thereafter *notify* allowing observers: EmailObserver, RSSFeedObserver, GooglePlusObserver, FacebookObserver, etc., to deal with the particulars of broadcasting said article as appropriate for that broadcast method.
- **Login system**: Say you want provide notifications to various sub-systems when a user logs in. For example you want a "Security Observer" and perhaps a "User Stats Observer" (for the marketing department), Logger (for troubleshooting), etc. You could do so easily with the Observer pattern. *This article has a nice example Login system using Observer.*
- **Social Music**: A social music application could choose to broadcast notifications when a user plays, pauses, stops, fast forwards, seeks, etc. Perhaps your company is fortunate enough to be partnered with Facebook and needs to broadcast to the Music Dashboard via the Open Graph API. But then another partner comes along and offers a similar deal to broadcast to the tunein online radio site. And then another…

In the above example, each partner's API is likely to be quite different. You'd want your user tracking system to be completely orthogonal from the various broadcasting modules that will integrate with the above APIs. *The Pragmatic Programmer book has a wonderful chapter on the advantages of writing orthogonal software if you're interested in further investigating the concept.*

- **GUI Notifications**: You have some "live stats" that need to be reflected in real time by various components. For example, when the price of Apple stock changes, you need to update corresponding charts, graphs, text in callout boxes, etc.

Exercises

Do one of the following:

- Implement one of these systems using simple print statements. For example, when the RSSFeedObserver's *update* is called you can just do something like:

```php
<?php
public function update($subject)
{
    $article = $subject->getState();
    print "RSSFeedObserver::update called: " .
        "About to syndicate article " . $article->getTitle();
}
?>
```

- Come up with your own problem that lends itself to the Observer pattern and implement it.

Summary

In this chapter we've taken a look at the *Observer Pattern* and learned the following:

- What the *Observer Pattern* is and how to implement it
- Observer allows us to *program to an interface, not an implementation*—subjects need not know about what the observers will do with state…they just call the *update* method—thus decoupling modules that interact
- Observers can be "hot swapped" via the register and unregister routines
- We discussed the differences and trade offs between the *pull method* and the *push method* strategies
- Programming to an interface can result in less boiler-plate "guard checks" if type hinting used within the *update* routine
- We also learned about some of the subtleties of TDD as we iterated on our Observer implementation

Other references not linked above

Design Patterns in Ruby, Gang of Four, Easles on Observer, IBM Article, GOF Patterns, Pitfalls of Observer Pattern, Derkeiler on Observer, Rice - CPP Resources, Addy Osmani Patterns Book PHP Design Patterns, Andy Pangus Article

Object Oriented Principles

This book assumes fundamental knowledge of basic object oriented concepts—a prerequisite to understanding the patterns we'll be covering in this book—and so the discussion here will be purposely short for for the sake of brevity. If you're already an "object oriented guru", feel free to skip this chapter (although may want to peruse the section on S.O.L.I.D. principles for review).

Abstraction

Dictionary.com provides the following definition for abstraction: > Considering something as a general quality or characteristic, apart from concrete realities, specific objects, or actual instances.

From a programming perspective, an abstraction is a focused representation of the essential properties of a particular thing. Further, an *abstract date type* (ADT), will, typically, provide self-evident and immediate understanding of what that thing is. For example, when we mention the word "Car", it is immediately understood that we are talking about a *specific type of thing* that can be driven, has a steering wheel, breaks, etc. So, although a car mechanic may additionally think of internal combustion systems, crankshafts, cylinders, pistons, etc.—your crazy Aunt Helen surely does not! It's the first set of car qualities described that are at a *higher level of abstraction*, and therefore, easier to understand. When we make good use of abstraction while designing our interfaces, we keep the inner workings and gory details of our implementations hidden; they become "black boxes", that, "just work".

Class

A class can be thought of as a blueprint that defines an abstract data type (or ADT; see above). Thinking in terms of the nouns in a problem statement (e.g. an employee, customer, bank account, etc.), one can derive classes from real world things.

For brevity's sake, we assume you have some knowledge in this area. If not, see Wikipedia's page on Classes.

Interface

> The set of all signatures defined by an object's operations is called the *interface* to the object. An object's interface characterizes the complete set of requests that can be sent to the object … *type* is a name used to denote a particular interface … a type is a *subtype* of another if its interface contains the interface of its *supertype*—GoF

The word type supports the notion that something is a *particular type of thing*, and so it may brings some semantic benefits in that it helps us to visualize that thing in our minds. We will, therefore, use it interchangably with interface .

Dynamic Binding

Through the use of interfaces we can achieve *dynamic binding*—the run-time association of a method call on an object and one of its methods. Since an interface provides a guarantee that certain operations will be available, we can substitute objects that have identical interfaces for each other at run-time. This substitutability is, in fact, known as *polymorphism*, and will be discussed in more detail below.

Inheritance

A *superclass* (also known as a base, or parent class) contains the core properties of a type, and may be "inherited from" by a *subclass* (also known as a derived, extended, or child class). The ability to inherit provides a way to reuse fields and methods since sub-types are able to *derive* properties from their ancestors.

Encapsulation

Grady Booch defines encapsulation as: > "the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior".

Encapsulation is a means of achieving *information hiding* such that the internal properties of an object are only allowed to be accessed or modified in a centralized and controlled manner. This is typically done via accessors and mutators (also known as getters and setters). The encapsulated property can then be changed (or completely replaced, reimplemented, etc.), and—provided the public interface remains the same—external components will not be impacted.

Encapsulation provides another "protective benefit"—the prevention of internal data from being changed throughout a system—and, thus, helps us to maintain proper data integrity. Further, encapsulation reduces system complexity by limiting interdependencies between software components.

To recap, the following are benefits of encapsulation: * data management is centralized * data integrity is maintained * modules remain decoupled since each manage their own data

These concepts are explored in much greater detail in the seminal work by Grady Booch, Object-Oriented Analysis and Design with Applications. The following two pages are also useful: Encapsulation, and Information Hiding Wikipedia pages for more details.

Polymorphism

Wikipedia: >Polymorphism is a programming language feature that allows values of different data types to be handled using a uniform interface.

The word *polymorphism* refers to the ability for one thing to take on many different forms. Similarly, in programming, polymorphism is the ability for one interface to represent many implementations. This is generally achieved by having an *interface* that is implemented by one or more *sub-types* (concrete implementations of that interface). While the sub-types may contain varying implementation details, the fact that they all support the same interface allows them to be used interchangeably.

Computer programs can utilize polymorphism and achieve runtime flexibility. If we take our Car example from earlier, and assume the Car interface has a *drive* method, we can pass instances that implement this interface dynamically. The users of these instances will be able to "blindly" call *drive*:

```
Car honda = new HondaAccord();
Car toyota = new ToyotaCamry();
someObject.doSomethingUsefulWithCars(honda);
someObject.doSomethingUsefulWithCars(toyota);
...
public void doSomethingUsefulWithCars(Car car) {
    // We don't have to care what kind of car it is, we just call drive:
    car.drive();
}
```

The most interesting part here is that—while we have instantiated two completely different car implementations (a Honda and a Toyota)—we have set them both to the type Car (an interface). In doing so, we guarantee that they have both have the *drive* method available. This is known as "programming to an interface not an implementation", and becomes quite important as we strive for more flexibile systems. *See the chapter on Strategy pattern for a more detailed example.*

GoF succinctly states some of polymorphism's key benefits:

> Polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at run-time.

S.O.L.I.D. Principles

The mnemonic acronym SOLID was: > introduced by Michael Feathers for the "first five principles" identified by Robert C. Martin in the early 2000s that stands for five basic principles of object-oriented programming and design. The principles when applied together intend to make it more likely that a programmer will create a system that is easy to maintain and extend over time.

SRP: The Single Responsibility Principle

The SRP principle is related to a broader concept called *cohesion*—the degree to which elements of a class or module are related. If you have a Customer class that deals with saving this customer to a file or database, generating reports for the customer, etc., it would be said to have *low cohesion* since these are not the responsibilities of a customer.

The SRP provides that: A class should have one, and only one, reason to change. Inverting this we could say that: a class with only one responsibility only has one reason to change. For an interesting discussion on the SOLID principle see the Hanselminutes Podcast #145 where Robert Martin interviewed on the topic.

Open/Closed Principle

The open/closed principle provides that: classes may be open for extension but closed for change. So, if we want augment a classes behavior, we must do so without actually modifying the class itself! Strategies for accomplishing this will be discussed in more detail in later chapters, but we may:

- Use *compositional delegation* to compose various abstractions at runtime, all acting in accordance with a common interface. *We will see this technique in action in the chapter on the Strategy pattern.*
- Use an asynchronous messaging scheme, where objects are notified on an "as-needed basis", via a common defined interface. *This technique will be seen in the chapter on the Observer pattern.*
- Use a plugin based archetecture that instantiates dynamic plugins, perhaps, searching a common directory and/or using common naming conventions, etc.

The take away, is that we should strive to design our modules, such that, once they've been fully tested and deployed in to production, we don't have to make any more *direct changes* to them. While this may seem overly ambitious, it turns out to be accomplishable.

Liskov Substitution Principle

In his interview with Scott Hansen (show #145), Robert C. Martin describes how the substitution principle may be understood in terms of how inherited instances may be used within a program: > "If you have an expectation of some object or some entity and there are several possible sub-entities, we'll call them sub-types that could implement that original entity. The caller of the original entity should not be surprised by anything that happens if one of the sub-entities is substituted. So, the simple way to think about this is, if you're used to driving a Chevrolet you shouldn't be too surprised when you got into a Volkswagen, you'd still be able to drive it. That's the basic idea, there's an interface, you can use that interface, lots of things implement that interface one way or another and none of them should surprise you."

He has also covered this in his writings by succinctly stating: "Derived classes must be substitutable for their base classes."

Another example might be if we were to create a Stack class by inheriting from a LinkedList super class. Although we'd be able to conveniently use the list's functionality to *push* and *pop* elements from off the front of the list, we could not pass around the *stack instance* without surprising users with inappropriate methods (those inherited from the linked list like: *contains*, *indexOf*, etc.) These operations are not what one might expect from a stack, and therefore, we'd be violating the substitution principle.

In this case, we should, instead, compose the Stack with an internal property that references an instance of linked list. With that in place, the Stack can then choose to expose only the appropriate *push* and *pop* operations—these would, in turn, delegate to the list "under the hood". The concept of *delegation* can be distilled as:

- a class property holds a reference to another *module* (the delegate)
- that property is used to call one of the delegate's instance methods

By using a list instance (rather than inheriting from list) and then delegating to it, we are *favoring composition over inheritance.* Here's naive but easy to understand example of how that might work (please humor our assumptions):

```
class Stack {
    // Here we are "composing" our stack with a list
    // Assume we later initialized with: list=new LinkedList()
    private List list;

    public void push(Object someObject) {
        // Assume the linked list inserts at index 0
        list.insert(someObject);
    }

    public Item pop() {
        // Assume the linked list returns the first item on delete
        return list.delete();
    }
}
```

If we were to now create an instance of this version of Stack class, it would no longer expose inappropriate list methods as it did before. Only *push* and *pop* are exposed (which is what one would expect of a simple Stack), and thus, it does not *surprise* it's users. Incomplete as this implementation might be, it would not be in violation of the Liskov Substitution Principle.

When ISA "falls down" A heuristic that's popular for determining class relationships is to check if the "ISA" statement holds true. For example, we could classify an Espresso by saying that it *isa* (yes, that's one word!) type of beverage. It holds that if something is a type of something else, it must share some properties. For example, both a Beverage and an Espresso have a *density*. Since density is fairly *low level* thing, it's nice to have that defined in Beverage, so Esspresso doesn't have to deal with such details. We can then focus on more interesting features of Esspresso like *caffeine* and perhaps define an operation like: *getAmountOfCaffeine*. Therefore, we are able to contemplate Esspresso from a higher level of abstraction. Unfortunately, though, performing such classifications using the isa technique has some issues…

Robert Martin provides that the above heuristic can break down with the canonical example: a square *is a* rectangle. While this holds (from a geometry perspective), if you derive a Square class from a Rectangle class, you will get inappropriate behavior. For example, *setHeight* from Rectangle, will not need to do anything to it's width, whereas the Square must be sure to keep its width and height the same. In fact, the square might not even want to separate width and height, and prefer to have a "side" property instead (since a square's sides will always have the same length!). Being lazy, you may decide that, "well, I can override setHeight, and put in a simple conditional check"). As you do so, you realize that you will also need the exact same conditional check in setWidth. Obviously, this scenario will result in an abysmal state of affairs for the poor maintainer of the inconsistent Square.

Interface Segregation Principle

ISP, essentially, provides that one large interface must be "segregated" into (separated), more specific, individual interfaces. This is really a corollary to *single responsibility* and *cohesion* principles which provide that a particular module should have one, and only one, focused responsibility.

"Make fine grained interfaces that are client specific."—Robert Martin

Dependency Inversion Principle

The dependency inversion explicity states that we should: "depend on abstractions, not on concretions."—Robert Martin. So, if you're calling a function, it should be an abstract function; if you're holding a reference to an object, that reference must point to an abstraction, etc. In practice, this is impossible to achieve throughout an entire system, since, at some point, somewhere, we'll need to instantiate an object instance. We can, however, be mindful to encapsulate these object creations using creational patterns we'll discuss later. By doing so, we compartmentalize areas that are likely to change.

Decorator Pattern

Overview

Design Patterns: Elements of Reusable Object-Oriented Software provide the following intent for decorators:

> Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

They further state that decorators may be used to:

> add responsibilities to individual objects dynamically … for responsibilities that can be withdrawn … when extension by subclassing is impractical …

Decorators, also known as *wrappers*, conform to the interface of the component they decorate. When the decorator is called, it in turn, calls the same method for the component it decorates (it can do so since they share a common interface). This arrangement allows us to nest decorators recursively, and add or remove them as needed.

Decorator Pattern Class Diagram

The following is a class diagram that represents the Decorator Pattern we'll be discussing in this chapter:

Decorator Pattern Implementation

Here are the steps to implementing the decorator pattern:

- Define your main component's interface
- Define your concrete implementations of that interface

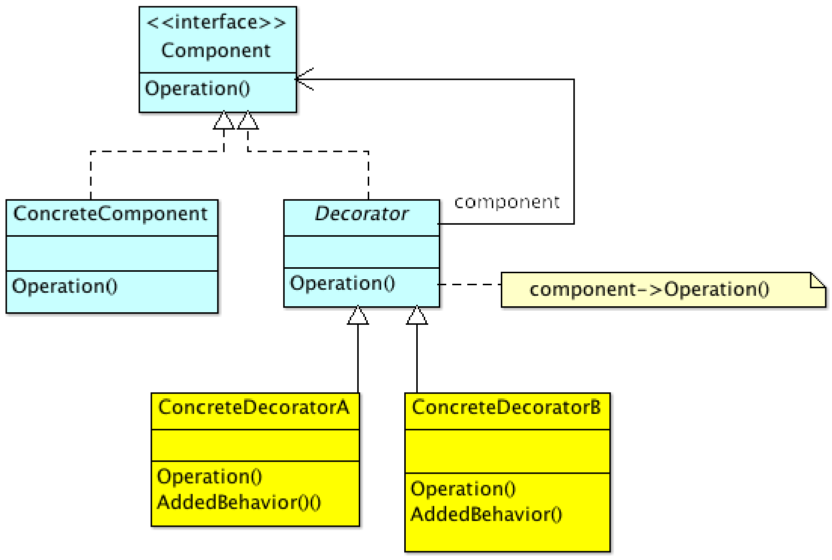

Figure 2: Decorator Pattern Class Diagram

- Define a Decorator interface or abstract class. It must also implement the main component's interface!
- Define concrete decorators that conform to above interface. *You may be able to "pull up" the method in to the above abstract Decorator if the decorated operation is simple and/or there is duplication (we did so in our examples below).*
- In the client code (code that uses the decorator(s) defined above):
  - Instantiate the main component
  - Instatiate one or many decorators being careful to "wrap" each one around the main component
  - Call decorated operations as needed

Java Implementation

Let us imagine that we have a store that sells tennis rackets that are set at various prices as expected. However, a customer may also choice to customize the string, grip, etc., when making their purchase. Thus we need a way to account for the change in price depending on these customizations. We can do so easily by simply wrapping each customization in respective decorators. Let's look at how this might be implemented using Java…

Disclaimers: for the sake of brevity and understandability, we have not been "thorough" in either our tests or implementation. We may continue to take such liberties throughout the rest of the book. Also note that we have combined the tests and implementation below (whereas the project in the book's repo has separate files)

```
/**
 * Tests
 */

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;


public class RacketTests {

    private ConcreteRacket racket;

    @Before
    public void setUp() throws Exception {
        racket = new ConcreteRacket();
    }

    @After
    public void tearDown() throws Exception {
        racket = null;
    }

    @Test
    public void testShouldDecorateComponentWithSynthGut() {
        RacketDecorator decorator = new PrinceSyntheticGutStringDecorator(racket);
        assertEquals("Should wrap with Synthgut decorator and add it's price",
                105.00, decorator.getPrice(), 0.01);
    }
    @Test
    public void testShouldDecorateComponentWithVSGut() {
        RacketDecorator decorator = new VSGutStringDecorator(racket);
        assertEquals("Should wrap with Synthgut decorator and add it's price",
                140.00, decorator.getPrice(), 0.01);
    }
    @Test
    public void testShouldAddMultipleDecoratorTypes() {
        RacketDecorator decorator =
            new WilsonProOvergripDecorator( new VSGutStringDecorator(racket) );
        assertEquals("Should wrap with VSGut decorator and add it's price",
                143.00, decorator.getPrice(), 0.01);
    }
    @Test
    public void testShouldAddSameDecoratorsMultipleTimes() {
        RacketDecorator decorator =
            new WilsonProOvergripDecorator( new VSGutStringDecorator(racket) );
        RacketDecorator wrappedAgain = new WilsonProOvergripDecorator( decorator );
        assertEquals("Should wrap with multiple decorators and adding all to price",
                146.00, wrappedAgain.getPrice(), 0.01);
    }
}


/**
 * Implementation
 */

public interface Racket {
    public double getPrice();
}

public class ConcreteRacket implements Racket {

    // In a "real program", we'd create a "value object"
    private double price;

    public ConcreteRacket() {
        price = 100;
    }

    @Override
    public double getPrice() {
        return price;
    }
}

public abstract class RacketDecorator implements Racket {

    protected Racket racket;
    protected double price;
    public RacketDecorator(Racket component) {
        racket = component;
    }

    @Override
    public double getPrice() {
        return racket.getPrice() + price;
    }

}

public class PrinceSyntheticGutStringDecorator extends RacketDecorator {

    public PrinceSyntheticGutStringDecorator(Racket component) {
        super(component);
        this.price = 5;
    }
}

public class VSGutStringDecorator extends RacketDecorator {
    public VSGutStringDecorator(Racket component) {
        super(component);
        this.price = 40;
    }
}

public class WilsonProOvergripDecorator extends RacketDecorator {

    public WilsonProOvergripDecorator(Racket component) {
        super(component);
        this.price = 3;
    }

}
```

PHP Implementation

Let us now look at the PHP version of this same implementation. It's virtually identical!

```
<?php

interface iRacket
```

```
{
    public function getPrice();
}

class Racket implements iRacket
{
    private $price;

    public function __construct()
    {
        $this->price = 100;
    }

    public function getPrice()
    {
        return $this->price;
    }
}

// We use abstract instead of interface here so we can define
// the properties $component and $price properties just once
// Also note that by implementing iRacket, it "gets" getPrice.
abstract class RacketDecorator implements iRacket
{
    protected $component;
    protected $price;
    public function __construct($racket)
    {
        $this->component = $racket;
    }
    public function getPrice()
    {
        return $this->component->getPrice() + $this->price;
    }
}

class PrinceSyntheticGutStringDecorator extends RacketDecorator
{
    public function __construct($racket)
    {
        parent::__construct($racket);
        $this->price = 5;
    }
}
class VSGutStringDecorator extends RacketDecorator
{
    public function __construct($racket)
    {
        parent::__construct($racket);
        $this->price = 40;
    }
}
class WilsonProOvergripDecorator extends RacketDecorator
{
    public function __construct($racket)
    {
        parent::__construct($racket);
        $this->price = 3;
    }
}

?>
```

We have omitted the PHP tests to save space, but, if you'd like to see them, they are in the book's repo.

We have coded the above implementation quite close to the racket *domain*, but we could have chosen to make a more general Product module, in place of what we called Racket, and thereafter embody various different types of products within Product. This might be useful in a general shopping cart scenario.

Considerations

Let us now examine the benefits and drawbacks of this pattern, and also, look at why it might be a better choice than inheritance.

Decorator vs. Inheritance

Decorators are often contrasted to using static inheritance to achieve the same thing—after all, we do have the option to simply create new sub-classes for each new responsibility discovered. In doing so, however, we increase the complexity of our system and pollute our base class with unrelated responsibilities.

Let's take an example that leads to having child classes that need to share properties with each other. Many derived *Beverage* types might need to have a Milk property, for example: Egg Nog, Shake, Mocha, while others do not (Juice should **not** have a Milk property). This leads to the following quandary: if we add milk to each derived class we will end up with *duplication*, but, if we instead add milk to the base class we violate the *substitution principle* (as one will undoubtedly be *surprised* to find a Milk property in their *orange juice* instance!).

We can avoid these inheritance problems altogether as the decorator pattern allows us to: *change the skin of an object versus changing its guts*—GoF. We simply add or remove decorators as needed. Doing so will simply *extend* the functionality of our classes, while acting in accordance with the open/closed principle—which, as you recall, states that we can extend a class, but must not not change it directly.

Transparency

One key benefit of the decorator pattern is that, by using the component's original interface, we can transparently add capabilities. This unburdens clients from having to keep track of things like: how many times has the component been decorated, in what way, etc. We simply add responsibilities, as needed, by merely wrapping the component with new decorators.

Pitfalls

Explosion of decorator objects

Use of this pattern will often lead to a proliferation of small, similar classes, decreasing both understandability and maintainability. One can imagine a whole product catalog implemented this way (and get an immediate migrane headache as a result!)

Identity tests

Decorators are not identical to the components they wrap, and thus, tests for object types will not work.

Removal/reordering of decorators

As you recall, GoF provides that you might use decorators:

> for responsibilities that can be withdrawn

This would seem to imply the ability to withdraw decorators (perhaps at runtime). If we consider the underlying mechanics of how decorators reference one another—similar to a linked list, where one link has a direct relationship to the next or previous—one might simply manipulate references to achieve this. However, this would result in objects that *need to know too much*, adding complexity and coupling to a system. In practice, removal won't be required, but, if you absolutely must have this functionality, you might want to consider another approach.

It not be idiomatic for the language you're using

The decorator pattern is more pervasive in certain (usually statically typed) languages than others. Java and .NET, both use the decorator pattern extensively in their own I/O Stream implementations. Also, most GUI programming done in Java will involve the wrapping of nested UI components. Thus, if you're on a team that's doing Java, C#, etc., the decorator pattern might be perfectly appropriate. However, if you're using a more dynamic language (like Ruby), it may not be an idiomatic choice. Since such languages allow you to add methods to objects at runtime—known as *duck typing* ("if it walks like a duck, and quacks like a duck, treat it as a duck")—you might entirely avoid the decorator pattern.

Exercises

We mentioned that while our implementation is quite specific to tennis rackets, one could implement a shopping cart system utilizing the decorator pattern to represent the products in that system. Do so. You don't need to implement a "full blown" shopping cart system, simply use print statements to stub out details that are not pertinent to the exercise.

Once you've done the above, add more and more products to your store's catalog. What do you notice?

Factory Pattern

Overview

In this chapter, we will discuss object creation, and specifically, how you can use factories to encapsulate such creational responsibilities. You'll see that instantiation is a rather volatile affair that should be compartmentalized before it gets "out of control". But before we dive in to factory patterns, let's revisit some prerequisite object-oriented principles.

Encapsulate what varies

Any portion of code that instantiates a new object is taking on a creational responsibility; and further becomes coupled to the type being instantiated. This defeats our goal to create classes that are cohesive units that do "one thing well". Even more insidious is the fact that there's no guarantee what new objects we'll need tomorrow... today we need an Oracle connection, tomorrow a MySQL connection... today we're sending Email messages, tomorrow RSS feeds... and so on. We've already learned that we should encapsulate areas likely to change—and creational code, as we've seen, is always likely to change.

Dependency Inversion Principle

If you recall from our object-oriented chapter, we discussed the S.O.L.I.D. principles which included the *Dependency Inversion Principle*. Let's dig a bit deeper into that principle which states that:

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

What are high and low level modules? High-level modules deal with larger sets of functionality, whereas lower level modules deal with more detailed operations. For example, a module that generates SQL, connects to a Database, and then executes that SQL's query, would be a low-level module. A related, but higher-level module, might be a Persistance class responsible for allowing general data persistence. While it may use the aforementioned low-level database module to write to the database, it may also work with other low-level modules like an I/O module for writing to files, a Console module for writing to standard output, and so on. So, if in this example, our Persistance class directly creates instances of any of the lower-level classes (DB, I/O, Console, etc.), it breaks the Dependency Inversion Principle since instantiation causes a direct dependency.

Let's make this all a bit more concrete with another example. Assume you have a Messenger module that's in charge of sending things. Perhaps your first set of requirements state that the Messenger need only send via the post office ("snail mail"):

```
class Messenger {
    private SnailMail mail;
    public Messenger () {
        mail = new SnailMail();
    }
    public void send(String destination, Package package) {
        mail.send(String destination, Package package);
    }
}
class SnailMail {
    public void send(String destination, Package package) {
        System.out.println("Postal service sending package");
    }
}
```

The key violation of DIP above is the 'mail' property of type SnailMail. Since Messenger is a high level module, and it now depends on the SnailMail module, DIP has been violated. We can see why this violation matters if we imagine that tomorrow our Messenger needs to additionally support a Fedex sender. Having a crazy deadline to meet we nervously code up the following:

```
class Messenger {
    private SnailMail mail;
    private Fedex fedex;
    public enum Speed {FAST, SLOW};

    public void send(Speed speed,
            String destination, Package package)
    {
        if (speed == Speed.FAST) {
            this.fedex = new Fedex();
            this.fedex.send(String destination, Package package);
        } else {
            this.mail = new SnailMail();
            this.mail.send(String destination, Package package);
        }
    }
}
```

Hopefully, you find this code objectionable! We now have the following unfortunate code smells:

- the unfortunate addition of the ‘speed’ enum property
- extra type-based fields making code harder to read

The rather inprecise ‘speed’ property will be problematic when we add senders that send things at approximately the same speed, or have new speeds requiring us to add new enum values.

Also, what happens next week when we decide to support international mail and have opportunities to add new carriers? We’ll have to directly modify the conditional code adding these carriers (and we’ll be breaking the open/closed principle since we’ll have to modify our class directly!)

So how can we solve these sorts of problems without breaking DIP? Let’s see how the Factory Method might help to circumnavigate such issues.

Factory Method

The factory method is a creational design pattern that allows you to encapsulate the creation of concrete types by defining an abstraction through which objects are created:

> “Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.”—Gang of Four

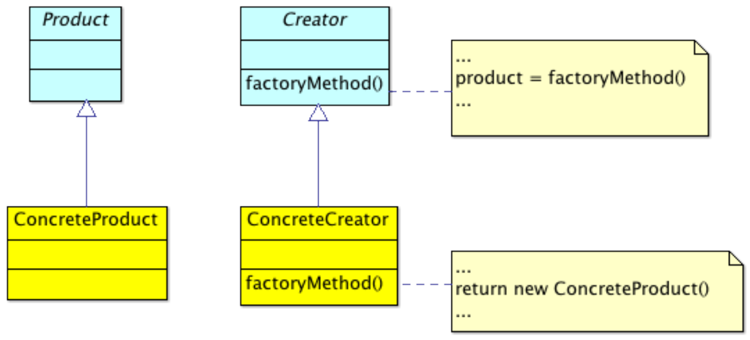

Figure 3: Factory Method

As we'll see shortly, the Factory Method has two main variants, one using sub-classing, the other a parameterized factory method. We'll start out looking at the sub-classing variant which is implemented as follows:

- *Product* defines the interface for products
- *ConcreteProduct* implements the product interface
- An abstract class *Creator* defines a factory method, which creates and returns a *Product*
- A ConcreteCreator overrides the factory method to return actual instances of a *ConcreteProduct*

Let's toss our earlier Messenger code and try implementing a *Sender* that's created using the Factory Method pattern. We'll start with a test that ensures that we can generate an appropriate sender via its corresponding factory:

```php
<?php
// ... code intentionally omitted
    public function testFactoryBegetsProduct() {
        $fedexFactory = new FedexFactory();
        $fedexProduct = $fedexFactory->createSender();
        $this->assertInstanceOf('FedexSender', $fedexProduct,
            "Should get correct sender from factory");
        $this->assertEquals(SenderTypes::Fedex, $fedexProduct->getType(),
            "Should be able to call sender's operations");
    }
// ... code intentionally omitted
?>
```

The test above simply confirms that a “Fedex Factory” should be able to generate a “Fedex Sender” product. Here’s the implemention. . .

```
<?php

// Factories
interface Factory {
    public function createSender();
}
class FedexFactory implements Factory {
    public function createSender() {
        return new FedexSender();
    }
}
class SnailMailFactory implements Factory {
    public function createSender() {
        return new SnailMailSender();
    }
}
class AramexFactory implements Factory {
    public function createSender() {
        return new AramexSender();
    }
}

// Products
class SenderTypes {
    const SnailMail = 0;
    const Fedex = 1;
    const Aramex = 2;
}
interface Sender {
    public function send($destination, $package);
    public function getType();
}
class FedexSender implements Sender {
    public function send($destination, $package) {
        echo ("Fedex sending...\n");
    }
    public function getType() {
        return SenderTypes::Fedex;
    }
}
class SnailMailSender implements Sender {
    public function send($destination, $package) {
        echo ("Snail mail sending...\n");
    }
    public function getType() {
        return SenderTypes::SnailMail;
    }
}
class AramexSender implements Sender {
    public function send($destination, $package) {
        echo ("Aramex sending international...\n");
    }
    public function getType() {
        return SenderTypes::Aramex;
    }
}

?>
```

With the above code in place, our messenger no longer needs to worry about which type of sender it's working with; that burden becomes the client's. For example, if a client wanted to send a package via Fedex it would simply do:

```
$fedexFactory = new FedexFactory();
$sender = $fedexFactory->createSender();
$sender->send($someDestination, $somePackage);
```

While the client *does* have to instantiate the factory itself, the product gets created "behind the scenes", and is therefore encapsulated.

Factory Method Variations

We mentioned earlier that there are two main variations of Factory Method:

- The sub-class approach

The Factory Method is defined in a base abstract class or interface (the Creator), and then overriden by concrete sub-classes (see the above Sender example).

- Parameterized Factory Method approach

In this version, there is generally just one Creator who's factory method takes a "type" parameter which is used to determine which type to of object create:

```php
<?php
// ... code intentionally omitted
class SenderFactory {
    public function createSender($type) {
        $sender = null;

        if ($type == SenderTypes::Fedex) {
            $sender = new FedexSender();
        } else if ($type == SenderTypes::Snail) {
            $sender = new SnailMailSender();
        } else if ($type == SenderTypes::Aramex) {
            $sender = new AramexSender();
        } else {
            // An arbitrary default might be snail mail
            $sender = new SnailMailSender();
        }
        return $sender;
    }
}
?>
```

The above conditional would appear to be a flagrant abuse of open/closed principle (as you'll have to directly modify the SenderFactory as soon as you need to add a new sender type). On the other hand, it might be more straightforward (since we don't need to create sub-classes). In any event, both variants give us the benefit of *encapsulating that which varies*.

Pitfalls

Layers of indirection: The sub-class variant requires us to create an inheritance hierarchy which adds some complexity.

Conditional code: In the parameterized factory method variant we end up with ugly conditional code that violates open/closed principle.

Complicated: Using a constructor to create a new object is immediately understandable, whereas, a factory method might not be. This might be helped by standardizing on factory naming conventions throughout your code base.

Abstract Factory

The GoF definition for Abstract Factory states that it:

> Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

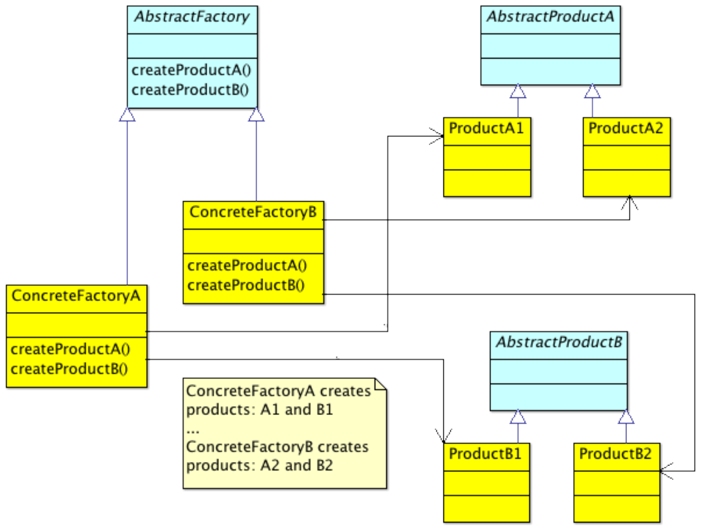

Figure 4: Abstract Factory

- AbstractFactory defines the interface for product creation
- ConcreteFactory creates the concrete products
- AbstractProduct defines the interface for the product
- ConcreteProduct defines the concrete product objects themselves

It is easy to get confused when trying to diffrentiate the Factory Method and the Abstract Factory; especially, since the Abstract Factory may, in fact, use the Factory Method in its implementation! Try to remember that the primary intent of the Abstract Factory pattern is to allow us to return "familes of related objects".

We're going to use the Abstract Factory pattern to model a car factory which allows us to create families of car products. In order to keep things easier to follow, we'll only worry about creating the engine and passenger compartment components.

Let's start with some tests. We want to know that when we use a particular concrete factory, we can, in turn, receive the correct corresponding concrete component:

```php
<?php
// ... code intentionally omitted

    // Abstract Factory - Tests
    public function testFactoriesGetCorrectStandardEnginePart() {
        $standardCarFactory = new StandardCarPartsFactory();
        $this->assertInstanceOf('CombustionEngine',
                $standardCarFactory->createEngine(),
                "Standard gets correct CombustionEngine");
    }
    public function testFactoriesGetCorrectMuscleEnginePart() {
        $muscleCarFactory = new MuscleCarPartsFactory();
        $this->assertInstanceOf('V8Engine',
                $muscleCarFactory->createEngine(),
                "Muscle gets correct V8Engine");
    }
    public function testFactoriesGetCorrectHybridEnginePart() {
        $hybridCarFactory = new HybridCarPartsFactory();
        $this->assertInstanceOf('HybridEngine',
                $hybridCarFactory->createEngine(),
                "Hybrid car gets correct HybridEngine");
    }
    public function testGetCorrectStandardPassengerCompartment() {
        $standardCarFactory = new StandardCarPartsFactory();
        $this->assertInstanceOf('StandardPassengerCompartment',
                $standardCarFactory->createPassengerCompartment(),
                "Standard gets correct StandardPassengerCompartment");
    }
    public function testGetCorrectMusclePassengerCompartment() {
        $muscleCarFactory = new MuscleCarPartsFactory();
        $this->assertInstanceOf('MusclePassengerCompartment',
                $muscleCarFactory->createPassengerCompartment(),
                "Muscle car gets correct MusclePassengerCompartment");
    }
    public function testGetCorrectHybridPassengerCompartment() {
        $hybridCarFactory = new HybridCarPartsFactory();
        $this->assertInstanceOf('HybridPassengerCompartment',
                $hybridCarFactory->createPassengerCompartment(),
                "Hybrid car gets correct HybridPassengerCompartment");
    }
?>
```

And here's the implementation.

```
<?php

abstract class CarPartsFactory {
    abstract public function createEngine();
    abstract public function createPassengerCompartment();
    // ... Doors, Wheels, etc., etc.
}

class StandardCarPartsFactory extends CarPartsFactory {
    public function createEngine() {
        return new CombustionEngine();
    }
    public function createPassengerCompartment() {
        return new StandardPassengerCompartment();
    }
}

class MuscleCarPartsFactory extends CarPartsFactory {
    public function createEngine() {
        return new V8Engine();
    }
    public function createPassengerCompartment() {
        return new MusclePassengerCompartment();
    }
}

class HybridCarPartsFactory extends CarPartsFactory {
    public function createEngine() {
        return new HybridEngine();
    }
    public function createPassengerCompartment() {
        return new HybridPassengerCompartment();
    }
}

interface Engine {
    public function start();
    public function stop();
    public function accelerate();
}

class HybridEngine implements Engine {
    public function start() {
        echo "Starting hybrid engine\n";
    }
    public function stop() {
        echo "Stopping hybrid engine\n";
    }
    public function accelerate() {
        echo "Accelerating hybrid engine\n";
    }
}

class V8Engine implements Engine {
    public function start() {
        echo "Starting V8 engine\n";
    }
    public function stop() {
        echo "Stopping V8 engine\n";
    }
    public function accelerate() {
        echo "Accelerating V8 engine\n";
    }
}

class CombustionEngine implements Engine {
    public function start() {
        echo "Starting Combustion engine\n";
    }
    public function stop() {
        echo "Stopping Combustion engine\n";
    }
    public function accelerate() {
        echo "Accelerating Combustion engine\n";
    }
}

interface PassengerCompartment {}
class StandardPassengerCompartment implements PassengerCompartment {}
class HybridPassengerCompartment implements PassengerCompartment {}
class MusclePassengerCompartment implements PassengerCompartment {}
?>
```

With the above in place, client code can simply instantiate the desired factory, and then use that instance to create the parts as needed:

```
$factory = new MuscleCarPartsFactory();
$this->engine = $factory->createEngine();
$this->interior = $factory->createPassengerCompartment();
// ... and so on
```

An alternative is create a 'product' class that uses the factory to generate a *fully built product*. In the following we create a Car class that takes a parts factory in its constructor, and then, essentially, builds itself:

```php
<?php
class Car {
    protected $engine = null;
    protected $passengerCompartment = null;

    public function __construct(CarPartsFactory $partsFactory) {
        $this->engine = $partsFactory->createEngine();
        $this->passengerCompartment =
            $partsFactory->createPassengerCompartment();
    }
    public function start() {
        $this->engine->start();
    }
    public function stop() {
        $this->engine->stop();
    }
    public function accelerate() {
        $this->engine->accelerate();
    }
    protected function __ensureBuilt() {
        if ($this->engine == null) {
            $this->build();
        }
    }
    public function getEngine() {
        return $this->engine;
    }
    public function getPassengerCompartment() {
        return $this->passengerCompartment;
    }
}
?>
```

Allowing a Car class to essentially build itself, does add a creational responsibility to Car; but given that all objects that implement a constructor are, in essence, intializing themselves, we'd argue this hasn't really weakened cohesion.

In this arrangement, we have the Car's constructor taking the CarPartsFactory in its constructor (letting clients be burdened with deciding which type of car parts factory to use). This is called Dependency Injection. Please do not confuse *injection* with *inversion* as it's so easy to do (see Dependency Injection Is NOT The Same As The Dependency Inversion Principle).

By allowing client code to inject the "depended on component" (DOC), we are afforded more flexibility in our tests (since our tests can create these depended on components and "pass them in"):

```php
<?php
    public function testCarPartsFactoryCreateEngineCalled() {
        $hybridEngine = new HybridEngine();
        $factoryStub = $this->getMock('HybridCarPartsFactory');
        $factoryStub->expects($this->once())
             ->method('createEngine')
             ->will($this->returnValue($hybridEngine));
        // Here we "inject" Car with a factory
        $car = new Car($factoryStub);
        $car->start();
    }
// ... code intentionally omitted
?>
```

In the above test, we are able to create a mock object, which takes place of the hybrid factory, and confirm that it in fact gets called from within the Car's constructor.

Pitfalls

Adding new components: The biggest issue with the Abstract Factory is that anytime you want to add an additional component, you will have to modify your factory and component code as such. For example, if we add a 'createWheels' method, look at the hoops we must jump through:

- Add the new wheels component:
  - Add new abstract wheel product
  - Add new wheel implementation
- Add a 'createWheels' to our abstract factory

- Add a 'createWheels' to *all* derived concrete factories: StandardCarPartsFactory, HybridCarPartsFactory, MuscleCarPartsFactory

This is tedious at best. Also, since we're having to reopen our code, we cannot assert conformance to the open/closed principle.

Complexity: Perhaps your reaction to the Car example above was, "Wow, that's a lot of code there!". We'd agree, and this is arguably another disadvantage of the Abstract Factory—the sheer number of classes involved. Imagine how complicated things will become when we start adding all the common car components you'd expect like the transmission, drive train, break system, wheels, doors, etc.

Static Factory

There is another way in which we may supply factory-like functionality which is not an official design pattern per se, but useful all the same. Joshua Bloch calls it a *static factory method*. In Effective Java, he describes this as simply:

> a static method that returns an instance of the class.

He goes on to provide some advantages of static factories:

- they have descriptive names, unlike constructors
- whereas a class can only have a single constructor with a given signature, you can have as many static factories as needed
- they don't have to create new objects each time called
- they can return subtypes

He goes on to provide an example of how a *service provider framework* might supply a mechanism for registering alternative implementations via a well-known *properties file* (similar to what the Pragmatic Programmers might call a *metadata-driven approach*). Using this metadata-driven approach, a client passes in a key to the provider framework's static factory method, which is then used to look up and instantiate the appropriate class. The following is a variation on this theme:

```php
<?php
// ... code intentionally omitted

    // Tests
    public function testGetDefinedInstance() {
        $returnedInstance = ProviderFramework::getInstance("moduleA");
        $this->assertInstanceOf('ModuleA', $returnedInstance,
            "Should get the correct instance");
    }

    public function testGetUndefinedInstanceReturnsDefault() {
        $returnedInstance = ProviderFramework::getInstance("bogus");
        $this->assertInstanceOf('DefaultModule', $returnedInstance,
            "Should fall back to a default instance if undefined key");
    }

// ... code intentionally omitted

// Provider Framework - Implementation
class Metadata {
    public function getModules() {
        // Let's imagine that in a "real system" we'd be getting
        // these from a known metadata configuration file.
        return array(
            "moduleA" => "ModuleA",
            "moduleB" => "ModuleB",
        );
    }
}

class DefaultModule {
}
class ModuleA {
}
class ModuleB {
}

class ProviderFramework {
    private static $modules = null;

    private static function initMetadata() {
        if (self::$modules == null) {
            $meta = new Metadata();
            self::$modules = $meta->getModules();
        }
    }

    public static function getInstance($key) {
        self::initMetadata();
        if (!isset(self::$modules[$key])) {
            return new DefaultModule();
        }
        $klass = self::$modules[$key];
        return new $klass();
    }
    public static function reset() {
        self::$modules = null;
    }
}

?>
```

Pitfalls

Some disadvantages of the static factory approach are:

- we cannot subclass (due to the private constructor)
- they're not distinguishable from other static methods

Final Thoughts

In this chapter we've presented three patterns to help aid in object creation (well, actually two patterns and one idiom—*static factory* isn't really an "official pattern"). These patterns can help you to maintain cohesion in your modules, pushing any creational responsibility to encapsulated factories.

Singleton Pattern

Overview

The Singleton pattern provides a means of ensuring that only one instance of a particular class is created. This is generally achieved by marking the constructor of that class as private, and then providing a public method that returns the one shared instance (e.g. 'getInstance').

Singletons may be applicable when:

> "there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point"—Gang of Four

Singletons are probably the most controversial of the design patterns, bringing the following concerns:

- Meaningful unit testing becomes far more difficult

If the system under test uses one or many singleton objects, it becomes hard to test that system in an isolated manner. Generally, it is advised to use dependency injection instead.

- Global state

While the issue of multi-threaded applications creating duplicate singletons can be circumnavigated by using *synchronized methods* or "eager instantiation", the global state introduced by this pattern greatly "reduces the chance of parallelism within a program".

Despite this controversy, we've decided to write this short chapter on Singletons, since you'll likely encounter them in many projects and need to, at least, be familiar with how they work. That said, definitely think thrice before adding Singeltons to your project, and, perhaps read these posts first:

- Guide: Writing Testable Code
- Performant Singletons

Implementation

So you've read the "warning labels" and still want to see how this is implemented; well, with much trepidation, we present the Singleton pattern:

```
public class Singleton {

    // Here we've "eager-initialized" our Singleton instance.
    // Using a synchronized getInstance or double locking are
    // other options.
    private final static Singleton instance = new Singleton();

    private Singleton () {}

    public static Singleton getInstance() {
        return instance;
    }
}

// ... code intentionally omitted
```

```
@Test
public void testCreateSingleton() {
    assertSame(Singleton.getInstance(), Singleton.getInstance());
}
```

Builder Pattern

Overview

The Builder design pattern provides a way to:

> separate the construction of a complex object from its representation so that the same construction process can create different representations.—Gang of Four

The Builder pattern is a creational pattern that allows for abstracting the creation of related but complex objects; this is done by utilizing concrete builders that independently construct and assemble products as appropriate.

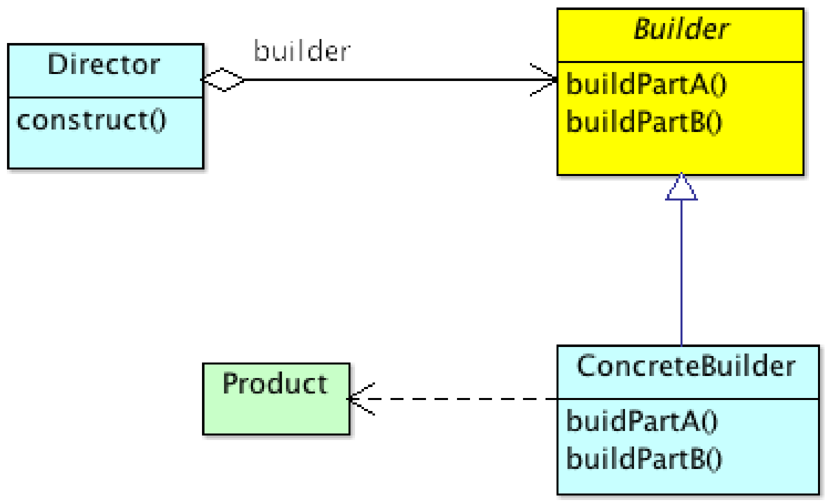

Figure 5: Builder Pattern

Participants and Collaborations

The participants in Builder are:

- **Builder:** Provides an interface for building parts of the complex object
- **ConcreteBuilder:** Assembles the parts that make up the object
- **Director:** Builds complex objects by delegating to the Builder
- **Product:** Represents the complex object being constructed

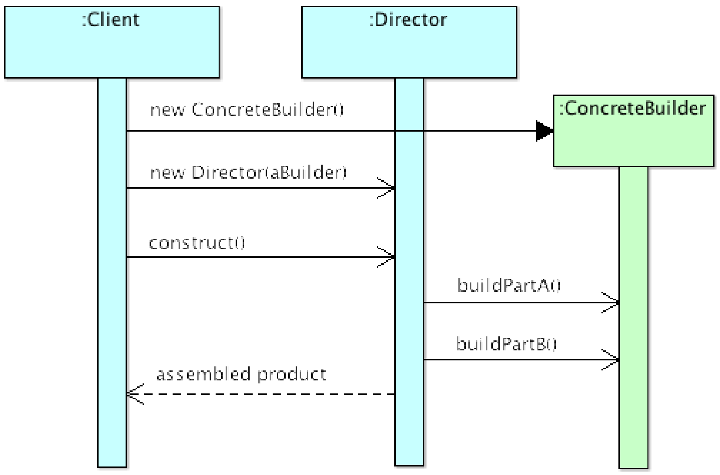

Figure 6: Builder Sequence Diagram

The assembly of the product involves the following collaborations:

- Client creates Builder and Director objects (injecting Director with Builder)
- Client calls *construct* (or similar) on Director
- Director calls appropriate series of Builder *build* operations
- Director calls get on the Builder getting the fully assembled product.

Implementation

Imagine the following *happy hour* scenario:

- You order a drink from a waitress at a pub
- The Waitress (the *Director*), takes down your order
- She then asks the Bartender (the *Builder*) to prepare your drinks
- Once the drinks are ready, she puts them on her tray, walks over and serves them to you

If you think about it, the above happy hour scenario is really quite similar to how Builder pattern works. Let's, in fact, put this example to code…first we'll start with the tests:

```
// Install node.js, npm, mocha.js, and sinon.js:
// $ npm install -g mocha
// $ npm install sinon
// $ mocha --ignore-leaks
var sinon, assert,
    IBartender, Bartender, Waitress, DrinkMenu;

assert = require('assert');
sinon = require('sinon');
IBartender = require("../builder.js").IBartender;
Bartender = require("../builder.js").Bartender;
Waitress = require("../builder.js").Waitress;
DrinkMenu = require("../builder.js").DrinkMenu;

describe("Builder tests", function() {
    var bartender, waitress;
    beforeEach(function() {
        bartender = new Bartender();
        waitress = new Waitress(bartender, DrinkMenu);
    });
    afterEach (function() {
        bartender = null;
        waitress = null;
    });

    it("should take an order for mojito", function() {
        var spy = sinon.spy(bartender, "prepareMojito");
        waitress.takeOrder([DrinkMenu.mojito]);
        assert(spy.called);
        bartender.prepareMojito.restore();
    });
```

```
it("should take order for all drinks on menu", function() {
    var allDrinks=[],
        key,
        allSpies={};

    // Spy on each of the Bartender's prepareXXX methods
    for(key in DrinkMenu) {
        allSpies[key] = sinon.spy(bartender, DrinkMenu[key]);
        allDrinks.push(DrinkMenu[key]);
    }

    waitress.takeOrder(allDrinks);

    // Assert Bartender's prepareXXX methods called; restore
    for(key in allSpies) {
        assert(allSpies[key].called);
    }
    for(key in DrinkMenu) {
        bartender[DrinkMenu[key]].restore();
    }
});

it("should only take Array of valid drink types", function() {
    var result;
    result = waitress.takeOrder("invalid type");
    assert.equal(null, result);
    result = waitress.takeOrder(['not_a_drink!']);
    assert.equal(null, result);
    result = waitress.takeOrder([DrinkMenu.mai_tai,
                                'not_a_drink!',
                                DrinkMenu.pina_colada]);
    assert.equal(null, result);
    result = waitress.takeOrder(undefined);
    assert.equal(null, result);
    result = waitress.takeOrder(null);
    assert.equal(null, result);
});
it("should return correct number of drinks", function() {
    var drinksServed =
        waitress.takeOrder([DrinkMenu.mojito,
                            DrinkMenu.mai_tai]);
    assert.equal(2, drinksServed.length);
});
it("should allow duplicate drinks to be ordered", function() {
    var drinksServed =
        waitress.takeOrder([DrinkMenu.mojito,
                                DrinkMenu.mai_tai,
                                DrinkMenu.mai_tai,
                                DrinkMenu.margarita,
                                DrinkMenu.mai_tai]);
        assert.equal(5, drinksServed.length);
    });
});
```

Implementation

And here's our implementation:

```
var DrinkMenu = {
    beer: "prepareBeer",
    mojito: "prepareMojito",
    kamikaze: "prepareKamikaze",
    margarita: "prepareMargarita",
    mai_tai: "prepareMaiTai",
    pina_colada: "preparePinaColada"
};

var IBartender = function() {};
IBartender.prototype[DrinkMenu.mojito] = function() {
    throw new Error("Method must be implemented.");
};
IBartender.prototype[DrinkMenu.kamikaze] = function() {
    throw new Error("Method must be implemented.");
};
IBartender.prototype[DrinkMenu.mai_tai] = function() {
    throw new Error("Method must be implemented.");
};
IBartender.prototype[DrinkMenu.margarita] = function() {
    throw new Error("Method must be implemented.");
};
IBartender.prototype[DrinkMenu.beer] = function() {
    throw new Error("Method must be implemented.");
};
IBartender.prototype[DrinkMenu.pina_colada] = function() {
    throw new Error("Method must be implemented.");
};


var Bartender = function() {
    this.drinks = [];
};
```

```
Bartender.prototype = new IBartender();
Bartender.prototype[DrinkMenu.beer] = function() {
    this.drinks.push("Beer");
};
Bartender.prototype[DrinkMenu.mojito] = function() {
    this.drinks.push("Mojito");
};
Bartender.prototype[DrinkMenu.kamikaze] = function() {
    this.drinks.push("Kamikaze");
};
Bartender.prototype[DrinkMenu.mai_tai] = function() {
    this.drinks.push("Mai Tai");
};
Bartender.prototype[DrinkMenu.margarita] = function() {
    this.drinks.push("Margarita");
};
Bartender.prototype[DrinkMenu.pina_colada] = function() {
    this.drinks.push("Pina Colada");
};
Bartender.prototype.ordersUp = function() {
    return this.drinks;
};


var Waitress = function(bartender, menu) {
    this.bartender = bartender;
    this.menu = menu;
};

// Precondition: drinks elements must be valid DrinkMenu items
Waitress.prototype.takeOrder = function(drinks) {
    var i, order = [], self = this;

    function isBartenderFunction(key) {
        return typeof self.bartender[key]==="function";
    }

    if(Array.isArray(drinks)) {
        for (i = 0; i < drinks.length; i++) {
            if (isBartenderFunction(drinks[i])) {
                order.push(self.bartender[drinks[i]]());
            } else {
                return null;
            }
        }
        return (order.length) ? self.bartender.ordersUp() : null;
    }
    return null;
};

module.exports.IBartender = IBartender;
module.exports.Bartender = Bartender;
module.exports.Waitress = Waitress;
module.exports.DrinkMenu = DrinkMenu;
```

Pitfalls

The BuilderPattern doesn't seem to commit many flagrant violations of object-oriented principles. However, as with most design patterns, it does add some complexity. Also, there is a more modern version of Builder pattern that you'll see that uses Fluent Interfaces, but this is beyond the scope of this chapter. Here's an interesting article if you'd like to learn more about that approach.

Summary

The Builder pattern allows us to decouple our client code from the construction of complex objects. It can be used to make our code more testable. For example, if we have an object with a "busy constructor"(a constructor doing object instantiation; a big testability no no!), one option we have is to refactor to have the constructor take a builder object. It can then use the builder's operations to initialize its member variables. This is really an alternative form of constructor dependency injection, a technique that allows test code to control how an object under test's dependents are constructed.

Command Pattern

Overview

The Command pattern is used to:

> "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations."—Gang of Four

Command is a behavioural pattern that decouples objects that invoke behaviour from objects that actually carry out those behaviours. Appropriately, we have an *Invoker* object that requests some behaviour, and a *Receiver* object that actually carries out that behaviour.

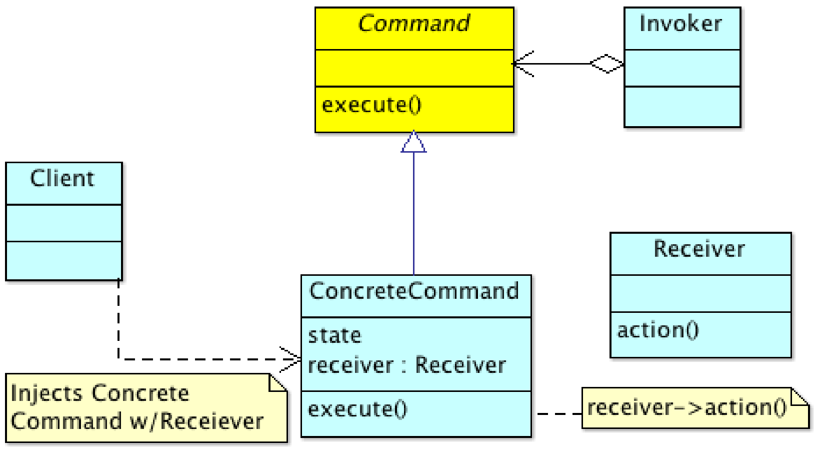

Figure 7: Command Pattern

In the case of the Command pattern, however, the Invoker does not know nor communicate with the receiver directly—instead, it calls a Command object's *execute* method, which takes care of invoking the receiver; the receiever then carries out the desired behaviour.

The command object is what decouples the invoker and receiver objects and provides the following benefits:

- The invoker needn't know about the receiever's interface
- The behavior may be carried out asynchronously

The defining characteristics of the command object are:

- an *execute* operation
- a reference to a *Receiver* that has code to fulfil the request

There are variations of command pattern that add functionality like *undo* and *batch commands*. We will look at some of these later in this chapter.

Implementation

The above sequence diagram should be fairly self-evident:

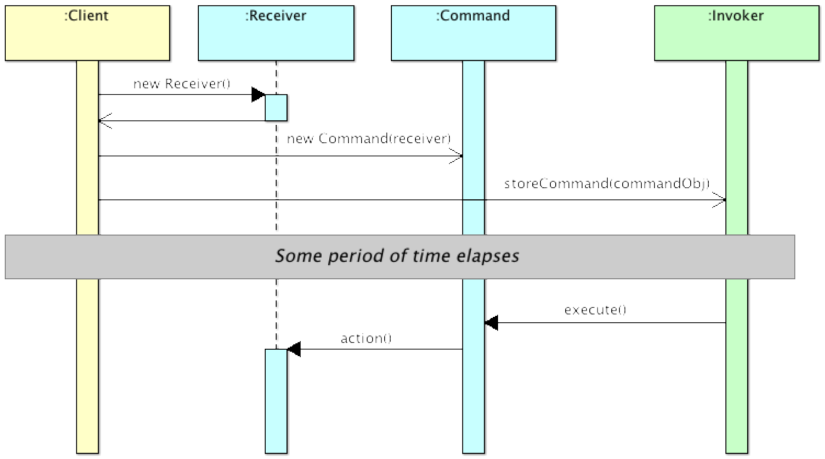

Figure 8: Command Sequence Diagram

- a Client creates a Command Object, injecting it with a Receiever
- the Client stores this "command object" on the Invoker
- some period of time elapses
- the Invoker calls *execute* on the Command Object
- the Command Object, in turn, calls *action* on the Receiever

The "Very Simplest" Command Pattern Implementation

Before we do something useful with the Command pattern, let's first create a "vanilla" Command implementation. As we've described, the core collaborations happen when the Invoker decides to trigger the Command's *execute* or *undo* method. Let's *Mock* the Receiver object and simply ensure that its appropriate operations get called:

```
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
```

```java
// http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html
import static org.mockito.Mockito.*;

public class CommandTests {

    private Invoker invoker;
    private Receiver mockReceiver;
    private Command command;

    @Before
    public void setUp() throws Exception {
        this.mockReceiver = mock(Receiver.class);
        this.command = new ConcreteCommand(this.mockReceiver);
        this.invoker = new Invoker();
        this.invoker.setCommand(this.command);
    }

    @After
    public void tearDown() throws Exception {
        this.command = null;
        this.invoker = null;
        this.mockReceiver = null;
    }

    @Test
    public void testCommandCallsReceiver() {
        // Test acting as "client" for now
        this.invoker.action();
        verify(this.mockReceiver, times(1)).doSomething();
    }

    @Test
    public void testUndo() {
        this.invoker.undo();
        verify(this.mockReceiver, times(1)).undoSomething();
    }

}
```

The above tests are *code smells* since there's no clear *system under test* (we're asserting the our Receiver mock gets called as the result of invoking the Invoker; but Invoker, in turn, depends on ConcreteCommand. Generally, we should not mock dependents more then one level deep). Since our tests are merely acting as a confirmation that the pattern is working as we expect, we won't bother refactoring. The tests pass with the following implementation:

```java
public interface Command {
    public void execute();
    public void undo();
}

public class ConcreteCommand implements Command {
    private Receiver receiver;

    public ConcreteCommand(Receiver receiver) {
        this.receiver = receiver;
    }
    @Override
    public void execute() {
        this.receiver.doSomething();
    }

    @Override
    public void undo() {
        this.receiver.undoSomething();
    }
}

public class Receiver {
    public void doSomething() {
        System.out.println("Doing something useful...");
    }
    public void undoSomething() {
        System.out.println("Undoing 'something useful'...");
    }
}

public class Invoker {
    private Command command = null;

    public void setCommand(Command command) {
        this.command = command;
    }
    public void action() {
        if (this.command != null) {
            command.execute();
        }
    }
    public void undo() {
        if (this.command != null) {
            command.undo();
        }
    }
}
```

Reading the above code, it should be clear the path that the code takes when Invoker's *action* is called; the Concrete Command's *execute* is called, which in turn, calls the Receiver's *doSomething* method. The code path for *undoing* is equally simple. This is a purposely pedentic example, but does show the general collaborations between the Invoker, Command, and Receiver objects.

Naming Conventions

One of the difficulties we're presented with when trying to use design patterns, is how to map the names of a pattern's collaborating objects to those in the system we're trying to build. Should we incorporate the pattern being used as a suffix (e.g. MyFooInvoker, and MyBarReceiver, etc.)? Although this does have the advantage of indicating the pattern objects being used, it may start to look like a form of *Hungarian Notation.* The real question, however, is whether the resulting name properly describes the thing it represents. In some cases, the suffix approach might work; we could have Command objects named DogSitCommand and DogSpeakCommand. These actually do represent the thing being modelled (while still including the pattern object as a suffix).

As Code Complete 2 advocates, we should strive to describe only the thing being represented itself:

> Does the name fully and accurately describe what the variable represents? . . . Names should be specific as possible. Names that are vague enough or general enough to be used for more than one purpose are usually bad names.

The take away (in our opinion) is, to remain flexible, but use your judgement as to whether the suffix approach will detract from a good variable name or not.

Example: A user interface toolkit

The quintessential problem that lends itself to the Command Pattern is a *user interface toolkit.* For example, the toolkit might have widgets that a user interacts with like menu items, toolbar items, keyboard shortcuts, etc., that trigger certain actions. Let's try to implement a subset of such a system.

Designing "on the go"

We need to take a couple of minutes to figure out how we're going to map our GUI's objects to those defined in the Command Pattern. We can use a sort of "watered down" pseudocode programming process approach to come up with something like the following:

```
@Test
public void testGUIImplementationOfCommandPattern() {
    // 1. Instantiate a document "Receiver"
        // We'll call ours: DocumentOperations
        // open, close, cut, paste, etc.
    // 2. Then add menu items (Commands):
        // MenuItemOpen, MenuItemClosed, etc.
        // ToolBarItemOpen, ToolBarItemClosed, etc.
    // 3. UIEventsManager will act as our "Invoker". We'll support:
        // handleMenuPressEvent
        // handleUndoMenuPressEvent
        // handleToolBarPressEvent
        // handleUndoToolBarPressEvent
        // ... etc.
}
```

If we were using PPP formally, we'd put these sorts of comments in a production method, and then replace each section with actual implementation code. But, in this case, we were actually sitting inside our first test case, looking for a way to quickly get our thoughts down; this type of bastardized PPP works fine for that purpose (yes, pencil and paper would work as well).

As you can see, we've decided that our *Receiver* will be a DocumentOperations class, and support open, close, cut, paste, etc.; our *Concrete Commands* will be MenuItemOpen, MenuItemClose, ToolBarItemOpen, ToolBarItemClosed, etc.; and our *Invoker* will be the UIEventsManager (in a real system we'd likely separate this one into more classes as it's taking on too many responsibilities; but for this example, we'll overlook this shortcoming); this will provide the corresponding event registration and handling, etc.

Implementation

Ok, so we have a basic sketch of our design. Let's implement this thing! For brevity's sake, we'll just implement the open, close, cut, and paste operations.

First, here are the tests for the GUI application:

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

//http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html
import static org.mockito.Mockito.*;

public class GUITests {

    private ICommand mockMenuOpenCommand;
    private ICommand mockMenuCloseCommand;
    private ICommand mockMenuCut;
    private ICommand mockMenuPaste;

    private ToolBarItemOpen mockToolbarOpenCommand;
    private ToolBarItemClose mockToolbarCloseCommand;
    private ToolBarItemCut mockToolbarCut;
    private ToolBarItemPaste mockToolbarPaste;

    private UIEventsManager eventManager;
    private DocumentOperations mockDocumentOperations;

    @Before
    public void setUp() {
        mockMenuOpenCommand = mock(MenuItemOpen.class);
        mockMenuCloseCommand = mock(MenuItemClose.class);
        mockMenuCut = mock(MenuItemCut.class);
        mockMenuPaste = mock(MenuItemPaste.class);
        mockToolbarOpenCommand = mock(ToolBarItemOpen.class);
        mockToolbarCloseCommand = mock(ToolBarItemClose.class);
        mockToolbarCut = mock(ToolBarItemCut.class);
        mockToolbarPaste = mock(ToolBarItemPaste.class);
        mockDocumentOperations = mock(DocumentOperations.class);
        eventManager = new UIEventsManager();
    }

    @After
    public void tearDown() {
        mockMenuOpenCommand = null;
        mockMenuCloseCommand = null;
        mockMenuCut = null;
        mockMenuPaste = null;
        mockToolbarOpenCommand = null;
        mockToolbarCloseCommand = null;
        mockToolbarCut = null;
        mockToolbarPaste = null;
        mockDocumentOperations = null;
        eventManager = null;
    }

    @Test
    public void testDocumentOperationsReceiverCalledForMenuOpen() {
        ICommand menuOpenCommand =
            new MenuItemOpen(mockDocumentOperations, "foofile.txt");
        menuOpenCommand.execute();
        verify(mockDocumentOperations, times(1)).open("foofile.txt");
    }

    @Test
    public void testDocumentOperationsReceiverCalledForToolbarOpen() {
        ICommand toolbarOpenCommand =
            new ToolBarItemOpen(mockDocumentOperations, "foo2file.txt");
        toolbarOpenCommand.execute();
        verify(mockDocumentOperations, times(1)).open("foo2file.txt");
    }

    @Test
    public void testDocumentOperationsReceiverCalledForMenuClose() {
        ICommand menuCloseCommand =
            new MenuItemClose(mockDocumentOperations, "foofile.txt");
        menuCloseCommand.execute();
        verify(mockDocumentOperations, times(1)).close("foofile.txt");
    }

    @Test
    public void testDocumentOperationsReceiverCalledForToolbarClose() {
        ICommand toolbarCloseCommand =
            new ToolBarItemClose(mockDocumentOperations, "foo2file.txt");
        toolbarCloseCommand.execute();
        verify(mockDocumentOperations, times(1)).close("foo2file.txt");
    }

    @Test
    public void testDocumentOperationsReceiverCalledForMenuCut() {
        ICommand menuCutCommand =
            new MenuItemCut(mockDocumentOperations);
        menuCutCommand.execute();
        verify(mockDocumentOperations, times(1)).cut();
    }

    @Test
    public void testDocumentOperationsReceiverCalledForToolbarCut() {
        ICommand toolbarCutCommand =
            new ToolBarItemCut(mockDocumentOperations);
        toolbarCutCommand.execute();
        verify(mockDocumentOperations, times(1)).cut();
    }

    @Test
    public void testDocumentOperationsReceiverCalledForMenuPaste() {
        ICommand menuPasteCommand =
            new MenuItemPaste(mockDocumentOperations);
        menuPasteCommand.execute();
        verify(mockDocumentOperations, times(1)).paste();
        menuPasteCommand.undo();
        verify(mockDocumentOperations, times(1)).undoPaste();
    }

    @Test
    public void testDocumentOperationsReceiverCalledForToolbarPaste() {
        ICommand toolbarPasteCommand =
            new ToolBarItemPaste(mockDocumentOperations);
        toolbarPasteCommand.execute();
        verify(mockDocumentOperations, times(1)).paste();
        toolbarPasteCommand.undo();
        verify(mockDocumentOperations, times(1)).undoPaste();
    }

    @Test
    public void testInvokerInvokesMenuOpenConcreteCommand() {
        eventManager.addMenuCommand("open", mockMenuOpenCommand);
        eventManager.handleMenuPressEvent("open");
        verify(mockMenuOpenCommand, times(1)).execute();
    }

    @Test
    public void testInvokerInvokesToolbarOpenConcreteCommand() {
        eventManager.addMenuCommand("open", mockToolbarOpenCommand);
        eventManager.handleMenuPressEvent("open");
        verify(mockToolbarOpenCommand, times(1)).execute();
    }

    @Test
    public void testInvokerInvokesMenuClosedConcreteCommand() {
        eventManager.addMenuCommand("close", mockMenuCloseCommand);
        eventManager.handleMenuPressEvent("close");
        verify(mockMenuCloseCommand, times(1)).execute();
    }
```

```
    @Test
    public void testInvokerInvokesToolbarClosedConcreteCommand() {
        eventManager.addMenuCommand("close", mockToolbarCloseCommand);
        eventManager.handleMenuPressEvent("close");
        verify(mockToolbarCloseCommand, times(1)).execute();
    }

    @Test
    public void testInvokerInvokesMenuCut() {
        eventManager.addMenuCommand("cut", mockMenuCut);
        eventManager.handleMenuPressEvent("cut");
        verify(mockMenuCut, times(1)).execute();
    }

    @Test
    public void testInvokerInvokesToolbarCut() {
        eventManager.addMenuCommand("cut", mockToolbarCut);
        eventManager.handleMenuPressEvent("cut");
        verify(mockToolbarCut, times(1)).execute();
    }

    @Test
    public void testInvokerInvokesMenuPaste() {
        eventManager.addMenuCommand("paste", mockMenuPaste);
        eventManager.handleMenuPressEvent("paste");
        verify(mockMenuPaste, times(1)).execute();
    }

    @Test
    public void testInvokerInvokesToolbarPaste() {
        eventManager.addMenuCommand("paste", mockToolbarPaste);
        eventManager.handleMenuPressEvent("paste");
        verify(mockToolbarPaste, times(1)).execute();
    }
}
```

And here are the various implementation files for the GUI system:

```
// IDocumentOperations acts as a "Receiver" (Command Pattern)
public interface IDocumentOperations {
    public void open(String fileName);
    public void close(String fileName);
    public void cut();
    public void undoPaste();
    public void paste();
}

public class DocumentOperations implements IDocumentOperations {

    public void open(String fileName) {
        System.out.println("Opening " + fileName + "...");
    }

    public void close(String fileName) {
        System.out.println("Closing " + fileName + "...");
    }

    @Override
    public void cut() {
        System.out.println("Cutting some text...");
    }

    @Override
    public void paste() {
        System.out.println("Pasting some text...");
    }

    @Override
    public void undoPaste() {
        System.out.println("Undoing last paste operation...");
    }
}

public interface ICommand {
    public void execute();
    public void undo();
}

// MenuItemOpen acts as a "ConcreteCommand" (Command Pattern)
public class MenuItemOpen implements ICommand {

    private IDocumentOperations documentOperations;
    private String fileName;

    public MenuItemOpen(IDocumentOperations documentOperations, String fileName) {
        this.documentOperations = documentOperations;
        this.fileName = fileName;
    }
```

```java
    @Override
    public void execute() {
        this.documentOperations.open(this.fileName);
    }

    @Override
    public void undo() {} // NOP

}

//MenuItemClosed acts as a "ConcreteCommand" (Command Pattern)
public class MenuItemClose implements ICommand {
    private IDocumentOperations documentOperations;
    private String fileName;

    public MenuItemClose(IDocumentOperations documentOperations, String fileName) {
        this.documentOperations = documentOperations;
        this.fileName = fileName;
    }

    @Override
    public void execute() {
        this.documentOperations.close(this.fileName);
    }

    @Override
    public void undo() {} //NOP

}

public class MenuItemCut implements ICommand {
    private IDocumentOperations documentOperations;

    public MenuItemCut(IDocumentOperations documentOperations) {
        this.documentOperations = documentOperations;
    }

    @Override
    public void execute() {
        this.documentOperations.cut();
    }

    @Override
    public void undo() {} //NOP
}
```

```java
public class MenuItemPaste implements ICommand {

    private IDocumentOperations documentOperations;

    public MenuItemPaste(IDocumentOperations documentOperations) {
        this.documentOperations = documentOperations;
    }

    @Override
    public void execute() {
        this.documentOperations.paste();
    }

    @Override
    public void undo() {
        this.documentOperations.undoPaste();
    }

}

public class ToolBarItemOpen implements ICommand {

    private IDocumentOperations documentOperations;
    private String fileName;

    public ToolBarItemOpen(IDocumentOperations documentOperations, String fileName) {
        this.documentOperations = documentOperations;
        this.fileName = fileName;
    }

    @Override
    public void execute() {
        this.documentOperations.open(this.fileName);
    }

    @Override
    public void undo() {} // NOP

}

public class ToolBarItemClose implements ICommand {
    private IDocumentOperations documentOperations;
    private String fileName;

    public ToolBarItemClose(IDocumentOperations documentOperations, String fileName) {
        this.documentOperations = documentOperations;
        this.fileName = fileName;
    }

    @Override
    public void execute() {
        this.documentOperations.close(this.fileName);
    }

    @Override
    public void undo() {} //NOP

}

public class ToolBarItemCut implements ICommand {
    private IDocumentOperations documentOperations;

    public ToolBarItemCut(IDocumentOperations documentOperations) {
        this.documentOperations = documentOperations;
    }

    @Override
    public void execute() {
        this.documentOperations.cut();
    }

    @Override
    public void undo() {} //NOP
}

public class ToolBarItemPaste implements ICommand {

    private IDocumentOperations documentOperations;

    public ToolBarItemPaste(IDocumentOperations documentOperations) {
        this.documentOperations = documentOperations;
    }

    @Override
    public void execute() {
        this.documentOperations.paste();
    }

    @Override
    public void undo() {
        this.documentOperations.undoPaste();
    }
```

```
}

import java.util.HashMap;
import java.util.Map;

public class UIEventsManager {
    private Map<String, ICommand> menuCommandsMap = new HashMap<String, ICommand>();
    private Map<String, ICommand> toolBarCommandsMap = new HashMap<String, ICommand>();

    public void addMenuCommand(String key, ICommand menuCommand) {
        this.menuCommandsMap.put(key, menuCommand);
    }

    public void addToolBarCommand(String key, ICommand toolbarCommand) {
        this.toolBarCommandsMap.put(key, toolbarCommand);
    }

    public void handleMenuPressEvent(String key) {
        ICommand menuCommand = this.menuCommandsMap.get(key);
        if (menuCommand != null) {
            menuCommand.execute();
        }
    }

    public void handleUndoMenuPressEvent(String key) {
        ICommand menuCommand = this.menuCommandsMap.get(key);
        if (menuCommand != null) {
            menuCommand.undo();
        }
    }

    public void handleToolBarPressEvent(String key) {
        ICommand toolbarCommand = this.toolBarCommandsMap.get(key);
        if (toolbarCommand != null) {
            toolbarCommand.execute();
        }
    }

    public void handleUndoToolBarPressEvent(String key) {
        ICommand toolbarCommand = this.toolBarCommandsMap.get(key);
        if (toolbarCommand != null) {
            toolbarCommand.undo();
        }
    }
}
```

Since we've taken the liberty to add print statements in our DocumentOperations (the Receiver in this case), we've decided to include a manual test (in addition to our unit tests):

```java
public class GUIManualTest {

    public static void main(String[] args) {
        UIEventsManager eventManager =
            new UIEventsManager();
        DocumentOperations docOperations =
            new DocumentOperations();
        ICommand menuOpenCommand =
            new MenuItemOpen(docOperations, "myfile.txt");
        ICommand menuCloseCommand =
            new MenuItemClose(docOperations, "myfile.txt");
        ICommand menuCutCommand =
            new MenuItemCut(docOperations);
        ICommand menuPasteCommand =
            new MenuItemPaste(docOperations);

        ICommand toolbarOpenCommand =
            new ToolBarItemOpen(docOperations, "myfile2.txt");
        ICommand toolbarCloseCommand =
            new ToolBarItemClose(docOperations, "myfile2.txt");
        ICommand toolbarCutCommand =
            new ToolBarItemCut(docOperations);
        ICommand toolbarPasteCommand =
            new ToolBarItemPaste(docOperations);

        eventManager.addMenuCommand("open", menuOpenCommand);
        eventManager.addMenuCommand("close", menuCloseCommand);
        eventManager.addMenuCommand("cut", menuCutCommand);
        eventManager.addMenuCommand("paste", menuPasteCommand);
        eventManager.addToolBarCommand("open", toolbarOpenCommand);
        eventManager.addToolBarCommand("close", toolbarCloseCommand);
        eventManager.addToolBarCommand("cut", toolbarCutCommand);
        eventManager.addToolBarCommand("paste", toolbarPasteCommand);


        // Here we "trigger" some pertinent events
        System.out.println("Manually testing the menu-based commands...");
        eventManager.handleMenuPressEvent("open");
        eventManager.handleMenuPressEvent("cut");
        eventManager.handleMenuPressEvent("paste");
        eventManager.handleUndoMenuPressEvent("paste");
        eventManager.handleMenuPressEvent("close");
        System.out.println("Manually testing the toolbar-based commands...");
        eventManager.handleToolBarPressEvent("open");
        eventManager.handleToolBarPressEvent("cut");
        eventManager.handleToolBarPressEvent("paste");
        eventManager.handleUndoToolBarPressEvent("paste");
        eventManager.handleToolBarPressEvent("close");
    }

}
```

Running this gives the following output:

```
Manually testing the menu-based commands...
Opening myfile.txt...
Cutting some text...
Pasting some text...
Undoing last paste operation...
Closing myfile.txt...
Manually testing the toolbar-based commands...
Opening myfile2.txt...
Cutting some text...
Pasting some text...
Undoing last paste operation...
Closing myfile2.txt...
```

Code walk-through

Although there is quite a bit of code in the above example, if you take a look at the *main* method in GUIManualTest, you should be able to see that we have the exact same "code flow" as we did in the simple pedentic exercise we started with earlier in this chapter. Let's take the recipe we defined earlier for Command Pattern and apply it to the GUI system:

- a Client creates a Command Object, injecting it with a Receiever

In our case, the Receiver is the DocumentOperations class, and this step is carried out as follows:

```
ICommand toolbarOpenCommand =
    new ToolBarItemOpen(docOperations, "myfile2.txt");
```

- the Client stores this “command object” on the Invoker

In our case, the UIEventsManager is our Invoker, and we store the command objects as follows:

```
eventManager.addToolBarCommand("open", toolbarOpenCommand);
```

We’re storing our commands within the UIEventsManager as *Maps*. So the first argument (‘open’ in the above example), is the key that will be used to later look up the appropriate command when we want to trigger *execute*.

- the Invoker calls *execute* on the Command Object

In our case, we’re triggering an event that will cause the Invoker to do this:

```
eventManager.handleToolBarPressEvent("open");
```

- the Command Object, in turn, calls *action* on the Receiever

As we know, our various Menu and ToolBar commands are delegating to the DocumentOperations class (the Receiver) to actually carry out the corresponding operations.

Template Pattern

Overview

The Template Method’s intent is as follows:

> Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure—Gang of Four

In the Template Method, we define an operation in a base class that has a particular sequence of “sub-operations”. This sequencing of calls from the Template Method to these sub-operations is an invariant—in fact, the template method itself, is often marked *final* so that it cannot be changed.

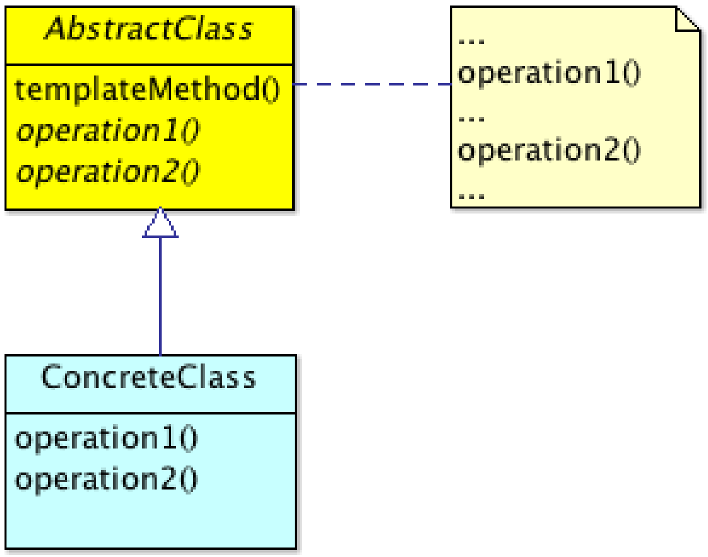
AbstractClass
templateMethod()
operation1()
operation2()
...
operation1()
...
operation2()
...
ConcreteClass
operation1()
operation2()

Figure 9:

Hollywood Principle

The arrangement of having a Template Method in the base class that calls operations defined by sub-classes is an example of using the *Hollywood Principle*: "Don't call us, we'll call you."—an analogy drawn from how a typical casting call process works: A pool of talent shows up to audition for a *part*. Perhaps there's sign in table where the talent leaves a media package (e.g. head shots, credit list, contact information, etc.). They are told to go to a waiting room (where they possibly wait all day), but eventually, get a chance to perform their audition. *Casting directors* watch, take notes, and likely make faces of disapproval. Later, through a process of elimination, the casting directors come up with a *casting list*.

Once the talent has finished their audition, the casting director might say: "don't call us, we'll call you". This is, essentially, a dismissive way of saying "don't bother to pursue the part any further…we'll get a hold of you if we feel like it!". This analogy is similar to the arrangement we have in Template Method: Sub-classes do not call operations on the parent ("don't call us")—they may override operations and even implement hooks—but only the parent gets to decides if and when these overriden operations get called ("we'll call you!").

Features

Template method provides the following benefits:

- The base class provides a single entry point for execution
- The base class has control of the sequence of performed operations for an algorithm
- Sub classes provide implementation details
- The base class may define *hook methods* with empty implementations that lower-level classes can choose to override

Hooks

Hook methods allow lower level components to "hook in to" a system. High-level classes determine exactly at which point these hooks are called. They are generally defined with empty implementations so that, if not overriden, nothing happens when they're called. In fact (in this case), the real difference between the abstract and hook methods is that the former are required, the later optional.

Participants

- **AbstractClass:** Defines the Template Method which will, in turn, call any of the following if defined:
  - **Concrete methods:** Generalized methods that provide any shared functionality
  - **Abstract methods:** Operations that sub-classes must override (required)
  - **Hook methods:** Operations that sub-classes may override (optional)

Implementation

The following is about the simplest implementation of Template Method you could find, but will help us to get a firm grasp of how this pattern works:

“‘java abstract class AbstractClass { public abstract void operation1(); public abstract void operation2(); public void hook1() {} public void hook2() {} public final void templateMethod() { hook1(); operation1(); operation2(); hook2(); } } class ConcreteClass extends AbstractClass { private String className = this.getClass().getSimpleName(); public void operation1() { System.out.println(className + “: operation1 called...”); } public void operation2() { System.out.println(className + “: operation2 called...”); } public void hook1() { System.out.println(className + “: hook1 called...”); } } class ConcreteClass2 extends AbstractClass { private String className = this.getClass().getSimpleName(); public void operation1() { System.out.println(className + “: operation1 called...”); } public void operation2() { System.out.println(className + “: operation2 called...”); } public void hook2() { System.out.println(className + “: hook2 called...”); } }

public class Template { public static void main(String[] args) { ConcreteClass cc1 = new ConcreteClass(); cc1.templateMethod(); ConcreteClass2 cc2 = new ConcreteClass2(); cc2.templateMethod(); } } “‘ Unsurprisingly, running this code prints out the following:

We define the abstract class with both abstract operations and concrete “hook” operations (notice the empty code blocks); it also has the *templateMethod* which calls each of the aformentioned operations. Notice that the hooks will “no op” if not implemented by a particular sub-class. For example, the first ConcreteClass only implements *hook1* (and we see that in the output), whereas ConcreteClass2 only implements *hook2* (also reflected in the output).

Example: Audio Decoder

Let’s take a look at a slightly more interesting (albeit, still not “real world ready”) example of implementing an Audio Decoder abstraction. We want to be able to handle both MP3's and AAC lossy audio formats, and have access to libraries that implement the low-level native details for each. We may want to support Ogg at a later date, and possibly even lossless formats too. Therefore, we need to comply with open/closed, and make sure that any classes we define don't have to later be "opened up".

First our tests:

"'java

import static org.junit.Assert.*;

import org.junit.After; import org.junit.Before; import org.junit.Test;

//http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html import static org.mockito.Mockito.*;

public class AudioDecoderTests { private INativeDecoder mockNativeMP3Decoder; private INativeDecoder mockNativeAACDecoder;

```
@Before
public void setUp() throws Exception {
    mockNativeMP3Decoder = mock(NativeMP3Decoder.class);
    mockNativeAACDecoder = mock(NativeAACDecoder.class);
}

@After
public void tearDown() throws Exception {
    mockNativeMP3Decoder = null;
    mockNativeAACDecoder = null;
}

@Test
public void testPlaysMP3s() {
    MP3Decoder mp3Decoder = new MP3Decoder(mockNativeMP3Decoder, "my_song.mp3");
    mp3Decoder.play();
    verify(mockNativeMP3Decoder, times(1)).decode(null);
}

@Test
public void testPlaysAACs() {
    AACDecoder aacDecoder = new AACDecoder(mockNativeAACDecoder, "my_song.aac");
    aacDecoder.play();
    verify(mockNativeAACDecoder, times(1)).decode(null);
}
```

} "' And here's our partial implementation (we've decided to err on the side of brevity for this example as actually implementing an audio decoder might distract from this chapter's topic—the Template Method pattern):

```java
class AudioInputStream {}

public abstract class AudioDecoder {
    protected String filePath;
    protected INativeDecoder decoder;

    public AudioDecoder(INativeDecoder decoder, String pathToAudioFile) {
        this.filePath = pathToAudioFile;
        this.decoder = decoder;
    }
    public abstract AudioInputStream loadStream();
    public abstract void decode(AudioInputStream ais);

    // Hooks
    public void beforeDecode(){}
    public void afterDecode(){}

    public void play() {
        AudioInputStream ais = loadStream();
        beforeDecode();
        decode(ais);
        afterDecode();
    }
}

interface INativeDecoder {
    public void decode(AudioInputStream ais);
}

class NativeAACDecoder implements INativeDecoder {
    public void decode(AudioInputStream ais) {
        // ... complex decode implementation omitted
        System.out.println("NativeAACDecoder decoding audio stream...");
    }
}

class NativeMP3Decoder implements INativeDecoder {
    public void decode(AudioInputStream ais) {
        // ... complex decode implementation omitted
        System.out.println("NativeMP3Decoder decoding audio stream...");
    }
}

class AACDecoder extends AudioDecoder {
    public AACDecoder(INativeDecoder decoder, String pathToAudioFile) {
        super(decoder, pathToAudioFile);
    }

    @Override
    public AudioInputStream loadStream() {
        // ... complex code to load an AAC audio stream
        return null;
    }

    @Override
    public void decode(AudioInputStream ais) {
        this.decoder.decode(ais);
    }

    public void beforeDecode(){
        System.out.println("AAC staring...");
    }
    public void afterDecode(){
        System.out.println("AAC stopped...");
    }
}

class MP3Decoder extends AudioDecoder {

    public MP3Decoder(INativeDecoder decoder, String pathToAudioFile) {
        super(decoder, pathToAudioFile);
    }

    @Override
    public void decode(AudioInputStream ais) {
        this.decoder.decode(ais);
    }

    @Override
    public AudioInputStream loadStream() {
        // ... complex code to load an MP3 audio stream
        return null;
    }

    public void beforeDecode(){
        System.out.println("MP3 staring...");
    }
    public void afterDecode(){
        System.out.println("MP3 stopped...");
    }
```

```
}
```

The INativeDecoder related classes are not a part of Template Method pattern. However, they are convenient DOC's that we mocked in our tests to confirm that the correct corresponding native implementations get called. For example, when we set up the MP3Decoder, we ensure that the *mockNativeMP3Decoder.decode* method gets called.

The astute reader may notice that our INativeDecoder's look as though they're being used almost like algorithms in the Strategy pattern (don't get confused... we're about to cover this pattern next!). That's sort of true, but realistically, these native audio API's would be much more complex, and also, since we don't control them, they're really not proper "strategy algorithms" per se.

Exercises

FeedReader: Implement a FeedReader that works with Atom, RSS1, RSS2, etc., and create an abstract FeedParser interface and with concrete implementations for each. The *readFeeds* will be the Template Method. You may use the following pseudocode to get started but feel free to "roll your own":

```
abstract class FeedReader {
    public final ArrayList<Feed> readFeeds(String url) {
        // .. omitted
    }
    abstract String getFeedTitle();
    abstract String getFeedAuthor();
    abstract String getFeedContent();
    public void beforeFeedParsed() {}
    public void afterFeedParsed() {}
    // .. omitted
}

class AtomReader extends FeedReader { ...
    public void beforeFeedParsed() {
        // override...
    }
    public void afterFeedParsed() {
        // override...
    }
    abstract String getFeedTitle() {
        // gets title
    }
    abstract String getFeedAuthor() {
        // gets author
    }
    abstract String getFeedContent() {
        // gets content
    }
}
class RSS1Reader extends FeedReader {
    //...
}
class RSS2Reader extends FeedReader {
    // ...
}
```

Pitfalls

The following are some disadvantages to Template Method:

- **Class explosion:** As the needs of an application grows, you will be required to create new classes for each new behavior desired. This can lead to an explosion of template object
- **Understandability:** Since Template Method requires inheritance, it can become hard to reason about the various sub-classes involved
- **Superclass coupling:** As the super-class defines all involved operations, sub-classes are tightly couples to it

Summary

The Template Method pattern lets subclasses define certain steps of an algorithm while preserving the sequence of these steps. It is somewhat similar to the Strategy pattern, in that the implementation details of certain behaviors are encapsulated and interchangeable. However, the Template Method uses *inheritance* to achieve its goals, whereas the Strategy uses *composition*. Template method also provides an opportunity to add hooks that clients can optionally implement. These hooks get called at certain points in the Template Method's run-time execution (e.g. beforeXXX, afterXXX, onXXX, etc.).

Let's look at a somewhat related pattern next… the Strategy Pattern.

Strategy Pattern

Overview

The Strategy pattern allows us to:

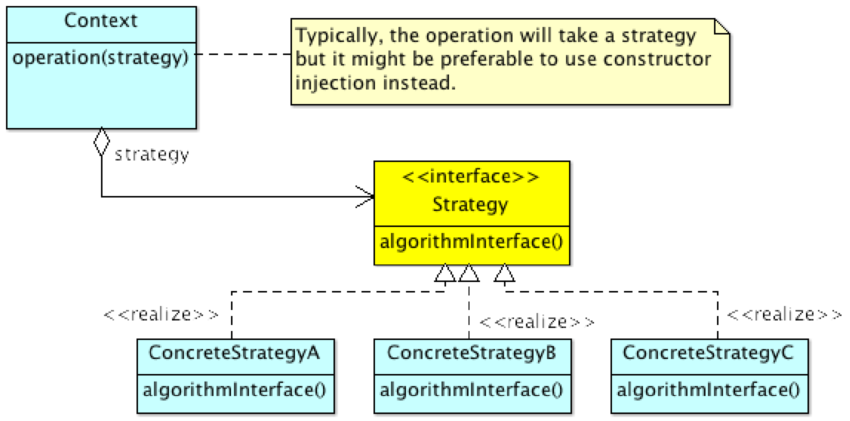

Figure 10:

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it—Gang of Four

Features

Strategy pattern provides the following benefits:

- provides a way to add variants of an algorithm by means of composition (as opposed to using inheritance like we did in Template Method)
- provides an alternative to conditional branching moving the variant algorithms in to their own Strategy classes
- allows you to add algorithms without having to reopen the base class; thus it complies with open/closed principle

Participants

- **Strategy:** Provides the common interface that all strategy algorithms will implement
- **ConcreteStrategy:** The classes that implement above Strategy interface
- **Context:** Gets injected with a particular ConcreteStrategy object at run-time

Implementation

Sticking to the audio analogy, let's imagine a simple web audio player that provides a way to *play* and *pause* (we'll keep it simple). We'd like to be able to switch between using an HTML5 audio implementation and a custom SWF implementation (Flash). Let's assume there's a mechanism in place such that the system knows at run-time whether it should use HTML5 or Flash for playing audio (this might be achieved via browser detection, URL naming conventions, etc.).

In the future we can imagine needing to add additional support for other audio formats (e.g. we may use a Java applet to play Ogg Vorbis audio, etc.). Thus we need an implementation that provides flexibility.

In this example, we'll be using JavaScript. Here are the mocha tests:

```
// Install node.js, npm, mocha.js, and sinon.js:
// $ npm install -g mocha
// $ npm install sinon
// $ mocha --ignore-leaks
var sinon, assert, IStrategy,
    HTML5AudioPlayer, SWFAudioPlayer, AudioPlayer;

assert = require('assert');
sinon = require('sinon');
IStrategy = require("../strategy.js").IStrategy;
HTML5AudioPlayer = require("../strategy.js").HTML5AudioPlayer;
SWFAudioPlayer = require("../strategy.js").SWFAudioPlayer;
AudioPlayer = require("../strategy.js").AudioPlayer;

describe("strategy tests", function() {
    var html5AudioPlayer, swfAudioPlayer;

    beforeEach(function() {
        html5AudioPlayer = new HTML5AudioPlayer();
        swfAudioPlayer = new SWFAudioPlayer();
    });
    afterEach (function() {
        html5AudioPlayer = null;
        swfAudioPlayer = null;
    });

    it("should play html5 audio", function() {
        var playSpy, player;

        playSpy = sinon.spy(html5AudioPlayer, "play");
        player = new AudioPlayer(html5AudioPlayer);

        player.playAudio();
        assert(playSpy.called);
        html5AudioPlayer.play.restore();
    });

    it("should pause html5 audio", function() {
        var pauseSpy, player;

        pauseSpy = sinon.spy(html5AudioPlayer, "pause");
        player = new AudioPlayer(html5AudioPlayer);

        player.playAudio();
        player.pauseAudio();
        assert(pauseSpy.called);
        html5AudioPlayer.pause.restore();
    });

    it("should play Flash audio", function() {
        var playSpy, player;

        playSpy = sinon.spy(swfAudioPlayer, "play");
        player = new AudioPlayer(swfAudioPlayer);

        player.playAudio();
        assert(playSpy.called);
        swfAudioPlayer.play.restore();
    });

    it("should pause Flash audio", function() {
        var pauseSpy, player;

        pauseSpy = sinon.spy(swfAudioPlayer, "pause");
        player = new AudioPlayer(swfAudioPlayer);

        player.playAudio();
        player.pauseAudio();
        assert(pauseSpy.called);
        swfAudioPlayer.pause.restore();
    });
});
```

Here is our implementation (note that our AudioPlayer is the *Context* and gets the Strategy object injected in into its constructor. Another variation is to pass the Strategy object to each operation directly; this provides later binding.

However, binding early in the constructor does ensure our object is consistently initialized *before* any of its methods get called.).

```
var IStrategy = function() {};
IStrategy.prototype = {
    play: function() {
        throw new Error("Method must be implemented.");
    },
    pause: function() {
        throw new Error("Method must be implemented.");
    }
};

var HTML5AudioPlayer = function() {};
HTML5AudioPlayer.prototype = new IStrategy();
HTML5AudioPlayer.prototype = {
    play: function() {
        console.log("Playing HTML5 audio...");
    },
    pause: function() {
        console.log("Pausing HTML5 audio...");
    }
};

var SWFAudioPlayer = function() {};
SWFAudioPlayer.prototype = new IStrategy();
SWFAudioPlayer.prototype = {
    play: function() {
        console.log("Playing SWF audio...");
    },
    pause: function() {
        console.log("Pausing SWF audio...");
    }
};

var AudioPlayer = function(playerStrategy) {
    this.player = playerStrategy;
};
AudioPlayer.prototype = {
    playAudio: function() {
        this.player.play();
    },
    pauseAudio: function() {
        this.player.pause();
    }
};

module.exports.IStrategy = IStrategy;
module.exports.HTML5AudioPlayer = HTML5AudioPlayer;
module.exports.SWFAudioPlayer = SWFAudioPlayer;
module.exports.AudioPlayer = AudioPlayer;
```

Pitfalls

The following are some disadvantages to Strategy patter:

- **Interface coupling:** As the context depends on the Strategy's interface, there's some coupling between the two
- **Complexity:** Sometimes conditional branching is more immediately understandable than dynamic strategy algorithms (while the later is more flexible, you may not need it)
- **Clients must decide which strategy to use:** Clients must be able to select the correct Strategy algorithm to be injected in to the Context; sometimes this knowledge is non-trivial and adds complexity

Summary

The Strategy pattern puts families of algorithms in to classes that can be "plugged in" at runtime. As the strategy pattern uses composition instead of inheritance, it provides a nice alternative to subclassing. Because these strategy algorithms can be injected at run-time, it's fairly trivial to add behaviors to the system. This complies with the open/closed principle which states that objects should be open for change but closed for modification.

State Pattern

Overview

The State pattern allows us to:

> Allow an object to alter its behavior when its internal state changes.
> The object will appear to change its class.—Gang of Four

The State pattern is a behavioral design pattern that, much like the Strategy pattern, uses composition to encapsulate behaviors that must change at run-time.

However, unlike the Strategy pattern, these encapsulated behaviors (or actions) are dependent on a Context object's *current state*. When a certain action is requested of the Context, it simply delegates to its state reference, which takes care of determining how to best handle the request. That state will take in to account any rules for how it should respond to such a request. For example, if a door is in the "locked state", and the action requested is "open door", the appropriate result might be "entry denied" (whereas if the door is in the "closed state" (closed but unlocked), the same "open door" action might result in "door opened").

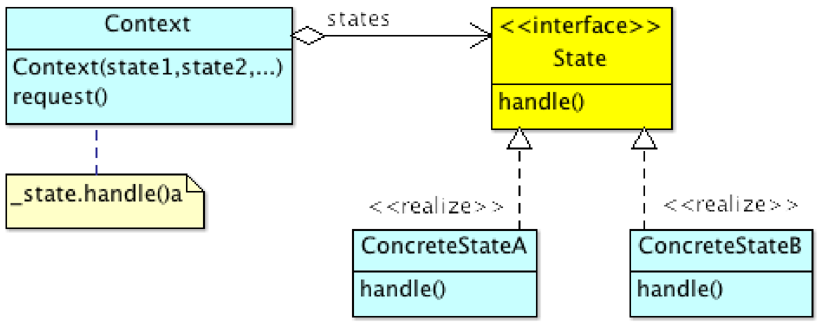

Figure 11: State Diagram

Participants and Collaborations

- **Context:** Delegates to *current state* when requests are made
- **State:** Defines the interface for concrete state classes to implement
- **ConcreteState:** Implements functionality for a particular state

Design Process

The first step to putting this all together is to enumerate the states and actions for the system (transitions can be enumerated later in the process).

States Enumerating the various states for our door example we come up with:

- Opened
- Closed

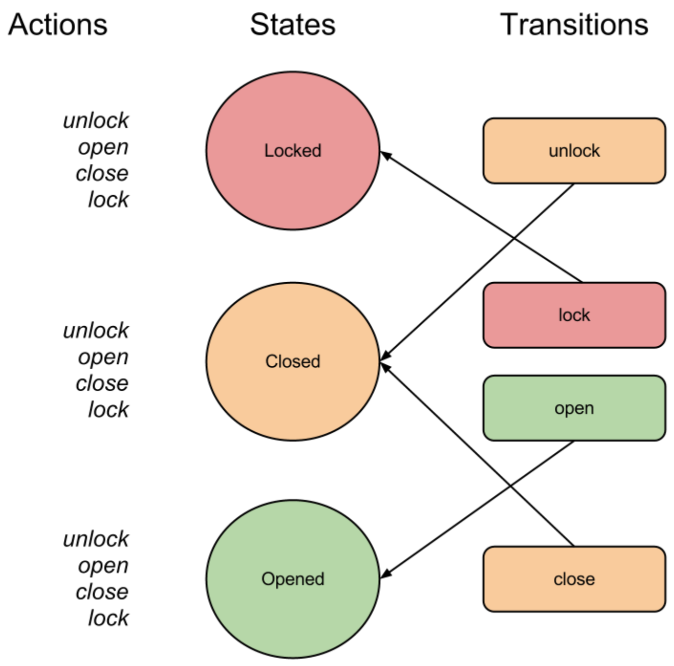

Figure 12: Actions, States, and Transitions for a typical Door

- Locked

Actions The actions required for our door example might be:

- unlock door
- turn door knob
- push door open
- close door
- lock door . . . and so on.

Putting it all together Now that we've determined the *states* and *actions*, we next need to determine the *transitions* (and get a picture of how these interactions will work together). Using a pen and pad (or graphics program), we can take the following steps:

- create a three column diagram ordered with actions, states, and transitions
- under actions, list all the actions you'll need
- under the state column use circles to represent your states
- for each action for a given state, determine if successfully completing that action results in a transition to another state; if so, add a rectangle with the name of the action under your transitions column adjacent to the state the action took place in
- draw an arrow from the transition to the state it transitions to

This process will give a good first glance at how the states will evolve. We used this approach to create the figures in this chapter.

Case Study: Audio Application

If you read the previous chapter on Strategy pattern, you may have noticed that we overlooked something when implementing the audio player—what happens if I hit play, but I'm already playing audio? Correspondingly, what happens if I hit pause, when not in the play mode? As we'll see, the State pattern handles these types of quandaries wonderfully.

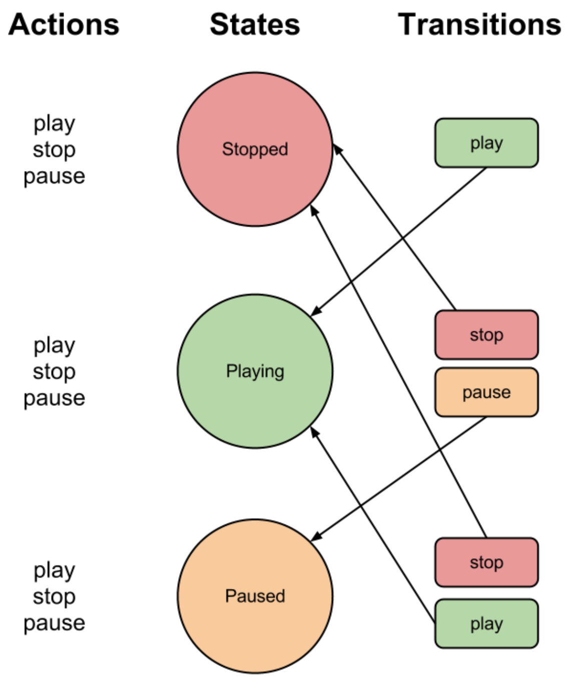

Figure 13: Audio State Design

Implementation

For our tests, we need to test each action for each state to confirm that the behavior is correct. So in the Playing state we test that:

- **play:** does nothing
- **stop:** stops audio
- **pause:** pauses audio

And of course we need similar tests for our Stopped and Paused states. We won't show all the tests we came up with (see the book's repo for that), but here are the ones for just the Paused state:

```
describe("Paused", function() {
    it("should transition from paused to playing state",
    function() {
        var spyPausedPlay, spyLibPlay, player;

        spyPausedPlay = sinon.spy(paused, "play");
        spyLibPlay = sinon.spy(lib, "play");
        player = new AudioPlayer({  'playing': playing,
                                    'paused': paused},
                                    'paused');
        player.playAudio();
        assert(spyPausedPlay.called);
        assert(spyLibPlay.called);
        assert(player.getState()['name'] === 'Playing');
        spyPausedPlay.restore();
    });
    it("should transition from paused to stopped state",
    function() {
        var spyPausedStop, spyLibStop, player;

        spyPausedStop = sinon.spy(paused, "stop");
        spyLibStop = sinon.spy(lib, "stop");
        player = new AudioPlayer({  'playing': playing,
                                    'stopped': stopped,
                                    'paused': paused},
                                    'paused');
        player.stopAudio();
        assert(spyPausedStop.called);
        assert(spyLibStop.called);
        assert(player.getState()['name'] === 'Stopped');
        spyPausedStop.restore();
    });
    it("should not pause if already paused",
    function() {
        var spyPausePause, spyLibPause, player;

        spyPausePause = sinon.spy(paused, "pause");
        spyLibPause = sinon.spy(lib, "pause");
        player = new AudioPlayer({  'playing': playing,
                                    'stopped': stopped,
                                    'paused': paused},
                                    'paused');
        player.pauseAudio();
        assert(!spyLibPause.called);
        assert(player.getState()['name'] === 'Paused');
        spyPausePause.restore();
    });
});
```

As we've mentioned in an earlier disclaimer, our test and production code are "overly optimistic" as we don't test for bad inputs, etc. Again, we've purposely sacrificed robustness for more understandable code. In practice, we'd aim for much more thorough code!

The flow of the three tests are quite similar (with the last one simply negating that audio will be paused if already in the paused state):

- We create two test spies: One that corresponds with the low-level underlying audio library that is carrying out *play*, *pause*, and *stop*; and the other that corresponds to our ConcreteState's action (e.g. the Paused state's *play* method, etc.). With these two spies, we can simply assert whether or not they got called as appropriate.
- We instantiate the AudioPlayer (our Context) injecting a map of our instantiated states as our first argument, and the key to the initial state as our second argument. Arranging the initialization of these objects outside of the AudioPlayer's constructor makes it easy to create test doubles with state appropriate for that particular test case's needs.
- Once we have the above set up, we call the production method on the AudioPlayer (e.g. player.pauseAudio, etc.), and then assert that our spies got called as expected.
- The last line of each test restores the spied on method to its original state. Sinon.js, essentially, hijacks these spy methods by intercepting any calls and then proxying to the original.

These restore calls weren't actually necessary for the above tests (we're nulling out these objects in our teardown and re-creating them in subsequent setup methods), but we've left them in to show how this is done. For example, we would need to do restore if the spy pointed to the jQuery.ajax method (since jQuery is really a one time globally loaded lib).

Here's our Audio State implementation:

```
// 3rd Party Lib
var AudioLib = function() {};
AudioLib.prototype = {
    play: function() {
        console.log('Playing audio...');
    },
    pause: function() {
        console.log('Pausing audio...');
    },
    stop: function() {
        console.log('Stopping audio...');
    }
};

// Context
var AudioPlayer = function(states, initialState) {
    this.states = states;
    this.currentState = states[initialState];
};
AudioPlayer.prototype = {
    playAudio: function() {
        this.currentState.play(this);
    },
    pauseAudio: function() {
        this.currentState.pause(this);
    },
    stopAudio: function() {
        this.currentState.stop(this);
    },
    getState: function() {
        return this.currentState;
    },
    setState: function(newState) {
        this.currentState = this.states[newState];
    }
};

// State
```

```
var IState = function(name) {
    this.name = name;
};
IState.prototype = {
    play: function(context) {
        throw new Error("Method must be implemented.");
    },
    pause: function(context) {
        throw new Error("Method must be implemented.");
    },
    stop: function(context) {
        throw new Error("Method must be implemented.");
    }
};

// Concrete States
var Playing = function(name, audioLib) {
    this.name = name;
    this.audioLib = audioLib;
};
Playing.prototype = new IState();
Playing.prototype = {
    play: function(context) {
        console.log("Already playing ... nothing to do...");
    },
    pause: function(context) {
        this.audioLib.pause();
        context.setState('paused');
    },
    stop: function(context) {
        this.audioLib.stop();
        context.setState('stopped');
    }
};

var Paused = function(name, audioLib) {
    this.name = name;
    this.audioLib = audioLib;
};
Paused.prototype = new IState();
Paused.prototype = {
    play: function(context) {
        this.audioLib.play();
        context.setState('playing');
    },
    pause: function(context) {
        console.log("Already paused ... nothing to do");
    },
    stop: function(context) {
        this.audioLib.stop();
        context.setState('stopped');
    }
};

var Stopped = function(name, audioLib) {
    this.name = name;
    this.audioLib = audioLib;
};
Stopped.prototype = new IState();
Stopped.prototype = {
    play: function(context) {
        this.audioLib.play();
        context.setState('playing');
    },
    pause: function(context) {
        console.log("Can't pause when stopped...");
    },
    stop: function(context) {
        console.log("Already stopped... nothing to do");
    }
};

module.exports.IState = IState;
module.exports.AudioPlayer = AudioPlayer;
module.exports.AudioLib = AudioLib;
module.exports.Playing = Playing;
module.exports.Paused = Paused;
module.exports.Stopped = Stopped;
```

AudioLib is an imaginary library for doing the low-level audio work. As can be seen from the above code, our State's implement all of the possible actions we've accounted for.

Essentially, one of three things can happen for a given action method: 1) the corresonding action is carried out 2) same as 1 but a transition to a new state is initiated 3) the method "no ops" (does nothing). In our system only 2 and 3 ever happen, but a more complicated system will have all three.

Since these action handlers are just methods, the implementer is free to put in whatever logic is required. This makes the State pattern particularly flexible (but also prone to too much complexity if a disciplined approach is not used).

As there's always some flexibility in how we use a design pattern, we could have chosen to use an abstract intermediary class creating empty "no op" methods there, so that ConcreteState implementations only had to implement methods they care about. We'll opt out of digressing in to how that might be implemented, but bring it up in case you're bothered by the "no ops" above.

If you compare the above implementation to our Strategy one, you may be wondering if we've "lost out" since we now only have the one AudioLib (recall in our Strategy example we had HTML5, Flash, etc., audio implementations); with a bit of ingenuity we couldn certainly pass in various types of audio libraries at run-time if we wanted. We could even mix in Strategy and State (as design patterns are just guidelines to help us tackle problems. Using such *compound patterns* is quite common in practice.).

Other Use Cases

- **Sales Order:** One classic example is a sales order that has discrete states like "New order", "Dispatched", "Shipped", "Cancelled", "Billed", and so on.
- **Transactions:** Imagine a transaction that can be in the states: "not_in_transaction", "in_transaction", "commited", "rolled_back", etc.
- **Graphics Tool:** A simple paint tool might be able take on states like: "blur", "erase", "smudge", "line", "arrow", etc., and draw accordingly.
- **Intern to employee:** Perhaps a company has states for their employees straight out of school. To become a "full blown employee" from intern that have to go through: "gopher", "intern", "skilled_helper", "appreciated_apprentice", etc. Perhaps the salary the intern is paid changes based on these levels. Further, perhaps, they must go through each level in sequence (you can't jump from "gopher" to "appreciated_apprentice" for example).
- **Vending machine:** Another classic example with states like "waiting", "coin_in", "beverage_sent", "coin_returned", etc.

Exercises

Implement one or more of the above example use cases.

Benefits

- Replaces long if or switch statements with encapsulated classes representing each state (DRY)

- Gets rid of duplicate branching (related to first bullet point)
- The encapsulated classes increase cohesion in the system
- uses composition so we're programming to an interface
- Since the Context simply delegates to State classes more can be added as needed as long as they conform to the State interface

Issues

- Increases complexity and might cause State class explosion
- Since States are generally in charge of initiating transitions, they become coupled to the state(s) they transition to. However, it could be argued that this structured program flow is really more of a benefit than a burden

Summary

The state pattern can be a helpful way to model problems that involve discrete states and actions that would typically require duplicate conditional logic throughout a code base. It's a good example of using compositional delegation and programming to an interface. A well implemented state design can increase understandability since the states, actions, and transitions all relate very closely to the domain being modelled.

Composite Pattern

Overview

The Composite pattern allows us to:

> Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.—Gang of Four

Before we can understand how Composite pattern works, we need to understand some terms:

- *Leaf*: A leaf is an individual component that has no children
- *Composite* A composite (also known as a *node* in tree terminology) is a component which may, in turn, contain other components (either leaf or composite).

The structure we're describing is, essentially, an inverted tree.

Using the Composite pattern, our leaf and composite components share one or more common operations—removing the need for client code to have to distinguish between the two—thus reducing the complexity of our code.

For example, imagine an online store that sells individual computer components, but also lets its customers assemble custom computers by arranging combinations of these components (they can choose their motherboard, cpu, power supply, etc.). When a customer proceeds to checkout, they might have any combination of individual parts, assembled computers, or both! How does the system get the price of each item given that the computers are really *composites* of individual components?

Using the composite pattern, the shopping cart system would be able to call *getPrice* on either an invidual component or an assembled computer. This is because the pattern defines a shared *Component* interface that both the leaf and composite implement (we'll see later that this does come at a price).

Implementation

At the core of this pattern is the ability for clients to treat individual items and aggregates the same. In our example, we want to be able to get the prices for individual computer parts, or, fully assembled computers. In the diagram below, you'll notice that both the Leaf and the Composite components share the *operation* method—in our shopping cart example, this would map to the *getPrice* method:

The following is an implementation solving the issue of totalling the prices for both computer parts and assembled computers as described earlier. This code is in JavaScript and uses the mocha.js testing library. If you'd like to run this code, you will need to install node.js, npm, and mocha.js. First let's look at the tests:

```
// Install node.js, npm, and then mocha:
// $ npm install -g mocha
// $ mocha --ignore-leaks
var assert, Component, Computer, ComputerPart;

assert = require('assert');
Component = require("../composite.js").IComponent;
Computer = require("../composite.js").Computer;
ComputerPart = require("../composite.js").ComputerPart;

describe("Composite tests", function() {

    describe("Shared methods", function() {
```

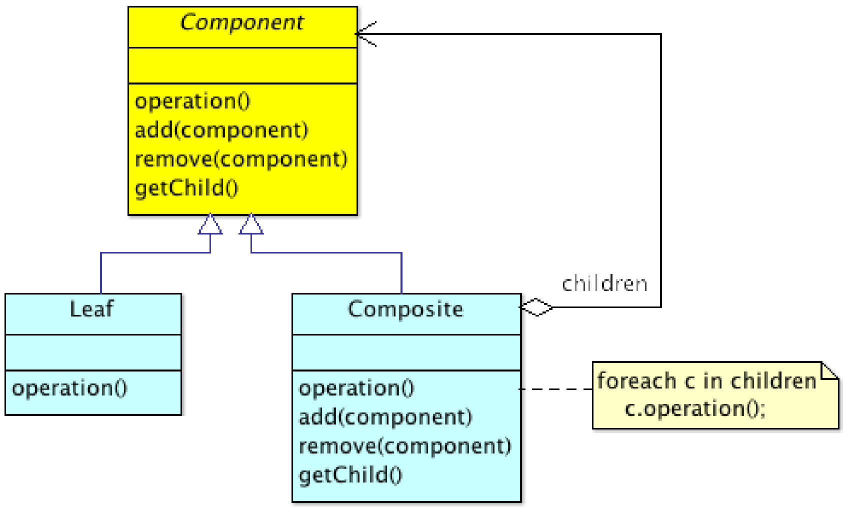

Figure 14:

```
it("should get price of a computer part",
    function() {
    var computerPart,
        expected = 99;
    computerPart = new ComputerPart(expected);
    assert.equal(expected, computerPart.getPrice());
});
it("should get price of a part", function() {
    var part = new ComputerPart(10);
    assert.equal(10, part.getPrice());
});
it("should be able to get price of a computer",
    function() {
    var computer, expected;
    expected = 1234;
    computer = new Computer(expected);
    assert.equal(expected, computer.getPrice());
});
it("should add parts and get total price",
    function() {
    var computer, computerPart, computerPart2;
    computerPart = new ComputerPart(10);
    computerPart2 = new ComputerPart(10);
    computer = new Computer(10);
```

```
            computer.add(computerPart);
            computer.add(computerPart2);
            assert.equal(30, computer.getPrice());
        });
        it("should nest composite and leafs",
            function() {
            var c1, c2, c3, p1, p2, p3;
            p1 = new ComputerPart(1);
            p2 = new ComputerPart(1);
            p3 = new ComputerPart(1);
            c1 = new Computer(1);
            c2 = new Computer(1);
            c3 = new Computer(1);
            c3.add(p3);
            c2.add(p2);
            c2.add(c3);
            c1.add(p1);
            c1.add(c2);
            assert.equal(6, c1.getPrice());
        });
        it("should remove parts from computers",
            function() {
            var computer, computerPart, computerPart2;
            computerPart = new ComputerPart(10);
            computerPart2 = new ComputerPart(10);
            computer = new Computer(10);
            computer.add(computerPart);
            computer.add(computerPart2);
            computer.remove(computerPart);
            assert.equal(20, computer.getPrice());
        });
        it("should not throw Error for no op methods",
            function() {
            var part = new ComputerPart(10);
            assert.equal(part.add(null), undefined);
            assert.equal(part.remove(null), undefined);
        });
    });
});
```

These tests show that we can, in fact, get the total price of recursively nested components (both leafs and composites). The very last test shows that we've opted to "no op" for the methods that aren't appropriate for the Leaf component. These are *add* and *remove*, operations really meant for dealing with the nested composites (which doesn't make sense for Leaf). This is an unfortunate aspect of Composite pattern—the Leaf component dredges up inappropriate methods from the Component interface it implements (we'll discuss this more later in the chapter).

Let's look at the implementation. Note that since JavaScript doesn't offer interfaces (at the language level), we define "mandatory methods" on the prototype as a way to enfore they will be overriden.

```
var IComponent = function() {
};
IComponent.prototype.getPrice = function() {
    throw new Error("Method must be implemented.");
};
IComponent.prototype.add = function(component) {
    throw new Error("Method must be implemented.");
};
IComponent.prototype.remove = function(component) {
    throw new Error("Method must be implemented.");
};

var Computer = function(price) {
    this.price = price;
    this.components = [];
};
Computer.prototype = new IComponent();
Computer.prototype.getPrice = function() {
    var i,
        len=this.components.length,
        total = this.price;

    if (len) {
        for(i=0; i<len; i++) {
            total += this.components[i].getPrice();
        }
    }
    return total;
};
Computer.prototype.add = function(component) {
    this.components.push(component);
};
Computer.prototype.remove = function(componentToRemove) {
    var i,
        len = this.components.length,
        updatedArray = [];

    for (i = 0; i < len; i++) {
        // Pushes all but the one being removed
        if(componentToRemove !== this.components[i]) {
            updatedArray.push(this.components[i]);
        }
    }
    this.components = updatedArray;
};

var ComputerPart = function(price) {
    this.price = price;
};
ComputerPart.prototype = new IComponent();
ComputerPart.prototype.getPrice = function() {
    return this.price;
};
// No ops
ComputerPart.prototype.add = function(component) {};
ComputerPart.prototype.remove = function(component) {};


module.exports.IComponent = IComponent;
module.exports.Computer = Computer;
module.exports.ComputerPart = ComputerPart;
```

Issues

Two S.O.L.I.D. principles are violated by this pattern:

- The interface has two responsibilities, those related to leafs, and those related to composites. Thus we can say that it violates the Single Responsibility Principle.
- Since Leaf components will need to implement *add* and *remove*, they will, in doing so, break the Liskov Substitution Principle (LSP) (one is quite *surprised* to see an *add* or *remove* method in a Leaf!).

As you recall, we simply chose to "no op" in the Leaf implementation for these methods. The drawback to this approach is that there's a chance we're masking buggy client code.

There are some other alternatives (all with their own tradeoffs):

- Only define the composite related methods in the Composite class

This has the drawback that client code now has to do run-time checks to determine if it's dealing with a Leaf or a Composite component. Thus we can no longer treat all components uniformly (one of the main supposed benefits!)

- Throw exceptions in composite methods if called on the Leaf

This trades robustness (the ability of a system to cope with errors, or, not crash), for type safety, ensuring that buggy client code isn't masked.

You may also need to deal with some of the following challenges:

- Enforcing the restriction of certain components from composites
- Ordering nested components
- Performance issues due to deeply nested composites

Summary

As we have seen, there are some design tradeoffs to take in to account when deciding whether or not to implement the Composite pattern. It favors *transparency* (being able to treat all components uniformly) over *safety*; and so you will be able to treat leaf and composite components operations the same. However, you will be violating SRP and LSP to some degree. As tradeoffs are a reality, you will have to use your judgement to decide if the benefits outweigh the costs.