

Lab 05

Unit Testing, Classes and Design Patterns

Goals

Test-driven development is *not* about testing. Test-driven development is about *development* (and design), specifically improving the quality and design of code. The resulting unit tests are just an extremely useful by-product.

That's all I'm going to tell you about test-driven development. The rest of this lab will *show* you how it works. In this lab we'll build a very simple tool together. We'll make mistakes, fix them, and change designs in response to what the tests tell us. Along the way, we'll throw in a few refactorings, design patterns, and object-oriented design principles. To make this project fun, we'll do it in Python.

Python is an excellent language for test-driven development because it (usually) does exactly what you want it to without getting in your way. The standard library even comes with everything you need in order to start developing TDD-style. I assume that you're familiar with Python but not necessarily familiar with test-driven development or Python's `unittest` module. You need to know only a little in order to start testing.

Instructions

For this and all future labs, all of your work will be done and submitted through your own student repository on the nisl class server. If you have not done so already, you need to check out a working copy of your personal repository before you can begin this lab.

Use the in-class lab time not only to find the answers to the questions and complete the given tasks, but to practice your skills in using the Python programming language, executing scripts and commands from the command line, and working with Python classes, libraries and the unittest framework. The first portion of the lab is to be completed in class so that you may ask the instructor for help and suggestions for any problems you may be experiencing.

Exercise 5.0: Preliminaries

Check out a working copy of your students repository if you haven't already done so. Your student repository will have a url designation of the form:

<http://nisl.tamu-commerce.edu/repo/csci553/username>

Where you need to substitute your own nisl account username for the last portion above.

In your working copy, create and add to your repository a directory called `lab05` (make sure you name it exactly as shown, make sure you not only create the directory, but you run the appropriate command to add the directory to be under version control). All work for this lab is to be added and checked into the `lab05` directory of your repository.

After creating your `lab05` directory untar and uncompress the file `/home/csci553/classfiles/lab05.tgz` to your new `lab05` directory. Several python scripts and other files are in this archive. Add all of these new files and commit them to your repository.

Exercise 5.1: Python's unittest Module

Since version 2.1, Python's standard library has included a `unittest` module, based on JUnit (by Kent Beck and Erich Gamma), the de facto standard unit test framework for Java developers. Formerly known as PyUnit, it also runs on Python versions prior to 2.1 with a separate download.

Examine the `TestOdd.py` script in `lab05` directory. `TestOdd.py` consists of a single "unit", which in this case is a simple function that should return True if its integer parameter is odd, and False otherwise. You can run the tests by invoking the script:

```
[harry@nisl lab05]$ ./TestOdd.py
.F
=====
FAIL: testTwo (__main__.IsOddTests)
-----
Traceback (most recent call last):
  File "./TestOdd.py", line 16, in testTwo
    self.failIf(IsOdd(2))
AssertionError
-----

Ran 2 tests in 0.001s
FAILED (failures=1)
```

The test is currently failing while executing `testTwo`. Go ahead and fix/implement the `isOdd` function. When you have correctly implemented the function your invocation of the test module should now succeed for all tests.

```
[harry@nisl lab05]$ ./TestOdd.py
```

```
..
```

```
-----
Ran 2 tests in 0.000s
```

```
OK
```

Methods whose names start with the string `test` with one argument (`self`) in classes derived from `unittest.TestCase` are test cases. In our example, `testOne` and `testTwo` are test cases.

Grouping related test cases together, test fixtures are classes that derive from `unittest.TestCase`. In the above example, `IsOddTests` is a test fixture. This is true even though `IsOddTests` derives from a class called `TestCase`, not `TestFixture`. Trust me on this.

Test fixtures can contain `setUp` and `tearDown` methods, which the test runner will call before and after every test case, respectively. Having a `setUp` method is the real justification for fixtures, because it allows us to extract common setup code from multiple test cases into the one `setUp` method.

In Python we typically don't need a `tearDown` method, because we can usually rely on Python's garbage collection facilities to clean up our objects for us. When testing against a database, however, `tearDown` could be useful for closing connections, deleting tables, and so on.

Looking back at our example, the main function defined in the `unittest` module makes it possible to execute the tests in the same manner as executing any other script. This function examines `sys.argv`, making it possible to supply command-line arguments to customize the test output or to run only specific fixtures or cases (use `--help` when you invoke `./TestOdd.py` to see the arguments). The default behavior is to run all test cases in all test fixtures found in the file containing the call to `unittest.main`. Typically, we wouldn't have the tests and the unit being tested in the same file, but it doesn't hurt to start out that way and then extract the code or the tests later.

At this point, once you have your `isOdd` function working for the two tests, add test for negative numbers. You should add two separate tests. `Test -1` (`isOdd` should return `True` for `-1`) and `-2` (`isOdd` should return `False` for `-2`). Does your function still work for negative numbers? If not fix it. Also add a fifth test testing the result of your `isOdd` function on zero (`0`). Zero is usually considered an even number, so your `isOdd` function should return `False` if passed zero as a parameter.

Once your `isOdd` function works for all 5 tests you are complete with part 1. You should commit your additions to the `TestOdd.py` script to the repository before moving to the next part of the lab.

Exercise 5.2: Motivation for the Project

Guess what I have trouble remembering to do:

```
0 0 * * * [ `date +%m` -ne `date -d +4days +%m` ] \
  && mail -s 'Pay the mortgage!' me-and-my-wife@example.org < /dev/null
```

That little puzzle is a line out of my *crontab* that emails me a reminder to pay the rent on the last four days of each month. Pathetic? Probably. It works, though. I haven't been late sending my mortgage payment since I started using it.

As clever as I thought I was for coming up with this, it wasn't practical for everything--especially for events that occur only once. Also, there's no way I could teach my wife enough bash scripting techniques in order to add a reminder to our calendar.

Most people use a good old-fashioned wall calendar for this type of thing. That's not techno-geeky enough for me.

I could use Outlook or Evolution or some productivity application, but that would open up a whole new can of worms. We don't use just one computer. We both use multiple computers and operating systems at home and at work. How could we easily synchronize all of those machines?

It was after realizing that our email is available to us no matter where we were that I hit upon the motivation for my project. The email reminding me to pay the mortgage was with me no matter what machine I'm on because I *always* check my email via IMAP, so my email is accessible from everywhere.

Why not email the upcoming events in my calendar to me just like my reminder to pay the rent? Brilliant, I thought. I know just the tools that can do this, too: the BSD calendar application and the new kid on the block, `pal`.

My wife and I have a private wiki that we use for keeping track of notes. It's great. Despite the fact that my wife's a psychology major and not a geek, she has no trouble using it. I figured we could use the wiki to edit our calendar file. I would write a little cron job to fetch the calendar file--probably using `wget`--from the wiki and pipe that into whatever tool best fit our needs.

Unfortunately, after looking at both `calendar` and `pal`, I discovered that neither was what I was looking for.

The `calendar` file format requires a `<tab>` character between dates and descriptions. Since I wanted to use our personal wiki to edit the calendar file, inserting `<tab>` characters would be an issue (upon hitting `<tab>`, focus jumps out of the text area to the next form control). `calendar` also doesn't support any of the fancy output options that `pal` does.

The `pal` format was much too geeky for even me to want to use, and it didn't support the one really important use case I had so far: setting a reminder for the last day of the month.

Sample Input

My wife and I sat down and came up with something both of us would want to use. Here are some examples:

30 Nov 2007: Dinner with the Saffers.
 August 16: Happy Anniversary!
 Wednesday: Piano lesson.
 Last Thursday: Goody night at book study. Yum.
 -1: Pay the rent!

Unlike the calendar format, a colon separates dates from descriptions. How Pythonic.

Like the calendar format, omitted certain fields are wildcards. The April 10 event happens every year. The Last Thursday event happens on the final Thursday of every month of every year.

The -1 event happens on the last day of every month of every year too. I took this idea from Python's array subscript syntax, where `foo[-1]` selects the last element in the `foo` array. I thought it was a little geeky, but my wife understood it right away.

Our goal is to write a small application that can run from cron to read a file in this format and email my wife and me the events we have scheduled for the next seven days. That shouldn't be too hard, should it?

Exercise 5.3: Getting Started

Being test infected means that we must write this tool by writing all of the unit tests *before* writing the code we expect the tests to exercise.

The first thing to do when starting a new project is to create an empty fixture that fails. Examine the `TestFoo.py` script. This is simply an empty test fixture that always fails. I do this out of habit, just to make sure I have everything typed in correctly and to test that the test runner can find the fixture.

Notice the class named `FOOTests` and its `testFoo` method. At this point we have no idea what we're going to test first. We just want to make sure that everything is ready once things get going.

Let's start out easy and test the first example from above with the full day, month, and year specified for the event. In order to create this test, we need to know *what* to test. Am I testing a class? A function?

This is where we put on our designer hats for a brief moment and try to use our experience and intuition to come up with some piece to the puzzle that will help us reach our goal. It's OK if we make a mistake here; the tests will reveal that right away, before we invest too much in this design. We certainly don't want to draft any documents filled with diagrams. Save those for later, after we have a clue about what will actually work.

For this project, we should probably create objects that can say whether they "match" a given date. These objects will act as a "pattern" for dates. (I'm using regular expressions as a metaphor here.)

Eventually, we'll have to write a parser that will read in a file line by line (similar to the filters you have seen previously) and create these pattern objects, but we'll do that later. These pattern objects are probably an easier place to start.

There might be multiple types of patterns--but we won't think about that now, because we could be wrong. Instead, we'll start coding so we can let it tell us what it wants to become. Modify the `testFoo` method, and change it to look like this:

```
def testMatches(self):
    p = DatePattern(2007, 9, 28)
    d = datetime.date(2007, 9, 28)
    self.failUnless(p.matches(d))
```

Notice that we changed the name of the method from `testFoo` to something more appropriate, because we now have an idea about what to test. We've also invented a class name, `DatePattern`, and a method name, `matches`. (The `datetime` module is part of Python 2.3 and up-- you need to add an import statement at the top of your `TestFoo` file in order to use it. Add the import statement now.)

This test, of course, fails miserably--the `DatePattern` class doesn't even exist yet! You should try running the `TestFoo.py` script now to see what happens. It should fail the test with an (expected) `DatePattern` is not defined exception. But we at least know now the name of the class we need to implement. We also know the name and signature of one of its methods and the signature for its `__init__` method. Here's what we can do with this knowledge:

```
class DatePattern:
    def __init__(self, year, month, day):
        pass
    def matches(self, date):
        return True
```

You should add this class stub in your `TestFoo.py` file, above the `TestFoo` testing framework class). Now the test passes! It's time to move on to the next test. You probably think I'm joking, don't you? I'm not. If your test is successfully passing, go ahead and commit your changes at this point to the repository.

Exercise 5.4: Baby Steps

Test-driven development (or any development/design work in my opinion) is best when you move in the smallest possible increments. You should only be writing code that makes the current failing test case(s) pass. Once the tests pass, you're done writing code. Stop!

The above code is worthless, right? It basically says that *every* pattern matches *every* date. How can I justify spending the time to come up with a "real" implementation? By adding another test. Add the following test under your current test:

```
def testMatchesFalse(self):
    p = DatePattern(2007, 9, 28)
    d = datetime.date(2007, 9, 29)
    self.failIf(p.matches(d))
```

If you run the test script you will see we now have one passing test and one failing test. We could change the matches method to return False in order to make this new test case pass, but that would break the old one! We now have no choice but to implement DatePattern correctly so that both tests can pass. Here's what we can do:

```
class DatePattern:
    def __init__(self, year, month, day):
        self.date = datetime.date(year, month, day)

    def matches(self, date):
        return self.date == date
```

If you run the test script both tests should now pass. Woo-hoo! I'm not happy with the DatePattern class, though. So far, it's nothing more than a simple wrapper around Python's date class. Why are we not just using date instances for the "patterns"?

It might turn out that the DatePattern class is unnecessary, but we're not going to make that decision on our own. Instead, we're going to write another test--one that we *think* will confirm the necessity of the DatePattern class:

```
def testMatchesYearAsWildcard(self):
    p = DatePattern(0, 8, 16)
    d = datetime.date(2007, 8, 16)
    self.failUnless(p.matches(d))
```

You should add the above test to your test script and try and run the tests. Voila! This test fails!

Why am I so happy about a failing test? My reasoning is simple: this *proves* that the current implementation of DatePattern is insufficient. It *can't* be just a simple wrapper around date and therefore can't be just a date.

While typing this test, I had to make a decision about how to represent wildcards. What occurred to me first was to use 0. After all, there's no year 0 (contrary to popular belief), month 0, or day 0. This may not have been the best choice, but we're going to roll with it for now.

It's time to make the new test pass (while making sure not to break the old ones). Modify your DatePattern class to look like the following:

```
class DatePattern:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def matches(self, date):
        return ((self.year and self.year == date.year or True) and
                self.month == date.month and
                self.day == date.day)
```

To be honest, I'm already starting to feel like we'll need to do some refactoring as we add more wildcard functionality to the class, but I want to write a few more tests first.

Let's add a test where the month is a wildcard:

```
def testMatchesYearAndMonthAsWildCards(self):
    p = DatePattern(0, 0, 1)
    d = datetime.date(2007, 10, 1)
    self.failUnless(p.matches(d))
```

At this point your test fixture should fail for the new method unless you fix the matches function of your the DatePattern to handle month wildcards. Go ahead and make the modification to the matches member function to support wildcarded months. If you fix it correctly, then all tests should again pass.

This method is getting uglier every time we touch it--I'm now positive that it will be our first refactoring victim. We now have a test for using wildcards for both years and months. Will we need one for days? A pattern containing nothing but wildcards would match every day. When would that be useful?

At this point I can't think of a reason to support wildcard days, so we won't bother writing a test for it. Because of that, we also won't bother implementing any code to support it in the DatePattern class. Remember, code gets written only when there's a failing test that needs the new code in order to pass. This prevents us from writing code that should not exist in our application, which should help keep it from becoming unnecessarily complex.

Let's move on. We need to support events that occur on a specified day of every week. So we need to first write a test for that case:

```
def testMatchesWeekday(self):
    p = DatePattern(
```

Uh, what now?

At this point, I realized that the DatePattern class might not be what I want to use for this test. Its `__init__` method doesn't accept a weekday. Should I use a different class, or modify the existing one?

We will modify the existing class for now, as that will require the least amount of work. If this turns out to be a bad idea, we can always refactor later.

```
def testMatchesWeekday(self):
    p = DatePattern(0, 0, 0, 2) # 2 is Wednesday
    d = datetime.date(2007, 9, 26) # 2007/9/26 is on a Wednesday
    self.failUnless(p.matches(d))
```

This doesn't pass because DatePattern.`__init__` doesn't accept five arguments (counting self). Modify the `__init__` constructor of your DatePattern class to look like this:

```
def __init__(self, year, month, day, weekday=0):
    self.year = year
    self.month = month
    self.day = day
    self.weekday = weekday
```

Question: before you continue do you understand why we gave weekday a default value? If weekday was a required constructor parameter, rather than optional because of the default, would we have to change our existing tests any to use the constructor?

We gave weekday a default value so that we wouldn't need to update the other test cases. Everything compiles and runs, but the new test case doesn't pass.

The astute reader has probably already realized that I'm now passing in 0 for the day argument. There's the wildcard I didn't think I would need--now I need it!

Here's our new matches method:

```
def matches(self, date):
    return ((self.year and self.year == date.year or True) and
            (self.month and self.month == date.month or True) and
            (self.day and self.day == date.day or True) and
            (self.weekday and self.weekday == date.weekday() or True))
```

You should modify the matches method of the DatePattern class to this new version. Now *all* of the components of a pattern allow for wildcards. How very interesting. What happens now if you run the test suite?

With this new method, testMatchesWeekday passes but testMatchesFalse now fails! What gives?

At this point we have written a significant number of tests and made some interesting discoveries in the course of designing our project. Go ahead and commit your changes at this point before moving on to the next portion of the lab.

Exercise 5.5: Refactoring

I honestly can't tell why testMatchesFalse fails by looking at the code. This is going to call for some simple debugging. Unfortunately, we tried to cram all of the logic for the matches method into one expression (spanning four lines!), so there's no place for us to insert any print statements to help us see which part is failing, and even in a debugger it will be difficult to follow the logic of the large boolean statement in the matches function. It's finally time to do that refactoring we've been needing to do.

The refactoring I want to apply is the Compose Method from Joshua Kerievsky's excellent book, Refactoring to Patterns. We did not get a chance to cover patterns in class, but patterns are a useful design tool for software development. For the Compose Method Pattern, by extracting smaller methods from the current matches method, we can not only make matches clearer but also make it possible to debug whichever part is currently causing us grief.

This is the result:

```
def matches(self, date):
    return (self.yearMatches(date) and
            self.monthMatches(date) and
            self.dayMatches(date) and
            self.weekdayMatches(date))

def yearMatches(self, date):
    if not self.year:
        return True
    return self.year == date.year
```

```
def weekdayMatches(self, date):
    if not self.weekday:
        return True
    return self.weekday == date.weekday()
```

We have (De)composed the original matching task into separate methods for each part. The monthMatches, and dayMatches methods are not shown, but they should be implemented exactly the same as the yearMatches method shown. (The weekdayMatches method uses a method from the datetime library to get the numerical weekday of a date). Modify your matches method and add and create the four matching composition methods for your DatePattern class.

The matches method is now much clearer, don't you agree? It might seem like a ridiculous thing to do, but writing intention-revealing code is much more important than being clever. I was trying to be too clever before and it caused a bug--one that I wouldn't have come across if I had done this from the beginning.

After applying this refactoring and rerunning the tests, I expected to see the testMatchesFalse test still failing, but it's now passing. If you have implemented the code correctly to this point, all 5 of your tests in the test suite should now be passing. Somewhere in the original logic I made an error, and I have no idea where it was--I'll leave finding it as an exercise for the reader. In the meantime, not only do we have simpler code now but it also actually works the way we expect it to. Take that!

Would we have noticed this bug without tests? I have no doubt that we would, but how long would it have been before we realized that this was a problem? With our unit tests, we noticed it immediately, so we knew exactly what to fix.

At this point you should have a working DatePattern class that can match wildcards for years, months, days and days of the week, as well as 5 unit tests of the class. Go ahead and commit your work so far to the repository before moving to the next portion of the lab.

Exercise 5.6: A New Design Emerges

Wildcards essentially work for all of the components we're testing so far. This is good, but I think the next test will cause trouble. It starts out innocently enough:

```
def testMatchesLastWeekday(self):
    p = DatePattern(0, 0, 0, 3)
```

Er, I'm stuck again. In case it's not obvious (and it's not--why didn't Python's datetime module define constants for weekdays?), the 3 represents Thursday.

How do I indicate that I only want to match the *last* Thursday in a month? Do I need to add yet another argument to DatePattern.__init__?

This is where that sneaking suspicion in the back of our head should finally be starting to warrant some closer attention. We might be trying to cram too much functionality into one class. You should be starting to think that the single DatePattern class we have so far is responsible for too much. It's only 30 lines long at this point, but I don't mean its length when I say "too much." What I mean is, conceptually, it does four different things; we even started adding a fifth before we stopped ourself.

What are its four different responsibilities? It has to determine if the year matches. That's one. The month is another, as are the day and the weekday, too. Can we break those responsibilities into four different classes? I'm sure we can, but should we?

Not only that, but for each component of the pattern, we have to check to see if it's a wild card and act appropriately. Maybe we need yet another class to encapsulate that logic.

We can't answer these questions by just thinking about them, so we will explore by writing a few tests:

```
class NewTests(unittest.TestCase):

    def testYearMatches(self):
        yp = YearPattern(2007)
        d = datetime.date(2007, 9, 26)
        self.failUnless(yp.matches(d))
```

Note that at this point we are creating a whole new fixture for a new design and a new set of tests. You should have a file called TestFoo.py that you have been using up to this point. Copy this file to a new file called NewTests.py. Remove all of the DatePattern class code from this new test fixture file, and modify the test class as shown above. If this turns out to be a dead end, we can easily delete this fixture and continue onward with the old. If this ends up being a good idea, we can just as easily delete the old fixture TestFoo.py (and the code it exercised, the DatePattern class).

This new test case, of course, fails. Here's the code required to make it pass:

```
class YearPattern:
    def __init__(self, year):
        pass

    def matches(self, date):
        return True
```

Notice we are starting along a new design path, looking at creating separate classes for matching different portions of a date.. This is pretty boring so far. How about another test?

```
def testYearDoesNotMatch(self):
    yp = YearPattern(2006)
    d = datetime.date(2007, 9, 26)
    self.failIf(yp.matches(d))
```

Of course this new test will fail since our initial implementation of matches always returns true (you should be running the test script each time we add a new test). Now to make the new test pass without breaking the old one:

```
class YearPattern:
    def __init__(self, year):
        self.year = year

    def matches(self, date):
        return self.year == date.year
```

Perfect. What's the point? Before I can show you that, we need to write a few more tests (go ahead and add these to your NewTests.py test fixture now):

```
def testMonthMatches(self):
    mp = MonthPattern(9)
    d = datetime.date(2007, 9, 26)
    self.failUnless(mp.matches(d))

def testMonthDoesNotMatch(self):
    mp = MonthPattern(8)
    d = datetime.date(2007, 9, 26)
    self.failIf(mp.matches(d))

def testDayMatches(self):
    dp = DayPattern(26)
    d = datetime.date(2007, 9, 26)
    self.failUnless(dp.matches(d))

def testDayDoesNotMatch(self):
    dp = DayPattern(25)
    d = datetime.date(2007, 9, 26)
    self.failIf(dp.matches(d))

def testWeekdayMatches(self):
    wp = WeekdayPattern(2) # Wednesday
    d = datetime.date(2007, 9, 26)
    self.failUnless(wp.matches(d))

def testWeekdayDoesNotMatch(self):
    wp = WeekdayPattern(1) # Tuesday
    d = datetime.date(2007, 9, 26)
    self.failIf(wp.matches(d))
```

Here's the code to make these tests pass (you can also put all of these class definitions, for now in your NewTests.py test fixture):

```
class MonthPattern:
    def __init__(self, month):
        self.month = month

    def matches(self, date):
        return self.month == date.month

class DayPattern:
    def __init__(self, day):
        self.day = day

    def matches(self, date):
        return self.day == date.day

class WeekdayPattern:
    def __init__(self, weekday):
        self.weekday = weekday

    def matches(self, date):
        return self.weekday == date.weekday()
```

If you have done things correctly to this point, you should now have a test script with 4 classes and 8 tests, and all 8 tests should pass successfully. Go ahead and add your `NewTests.py` script to the repository (if you haven't already done this) and commit your changes.

Exercise 5.7: Design Patterns

Is it obvious where we're going with this yet? If not, this test should make it clear:

```
def testCompositeMatches(self):
    cp = CompositePattern()
    cp.add(YearPattern(2007))
    cp.add(MonthPattern(9))
    cp.add(DayPattern(26))
    d = datetime.date(2007, 9, 26)
    self.failUnless(cp.matches(d))
```

What we've stumbled across here is an instance of the [Composite Pattern](#). (Interestingly, this is the same name as my class--that wasn't intentional, I promise.) A *composite* is basically an object that contains other objects, where both the composite object and its contained objects all implement the same interface. Using the interface on the composite should invoke the same methods on all of the contained objects without forcing the external client to do so explicitly. Whew, that was a mouthful.

Here, that interface is the `matches` method, which accepts a date instance and returns a bool. Python is a dynamically typed language, so we don't need to define this interface formally (as in a language like Java, which is fine by me).

How do we implement the composite? Like this:

```
class CompositePattern:
    def __init__(self):
        self.patterns = []

    def add(self, pattern):
        self.patterns.append(pattern)

    def matches(self, date):
        for pattern in self.patterns:
            if not pattern.matches(date):
                return False
        return True
```

The composite pattern asks each of its contained patterns if it matches the specified date. If any fail to match, the whole composite pattern fails.

I have to confess that we cheated here. We wrote more code than we needed to pass the test! Sometimes I get ahead of myself. Sorry. It turned out OK this time because all of the tests are passing, but we need to create a test that should *not* match, just to be sure we have everything working correctly:

```
def testCompositeDoesNotMatch(self):
    cp = CompositePattern()
    cp.add(YearPattern(2007))
    cp.add(MonthPattern(9))
    cp.add(DayPattern(28))
    d = datetime.date(2007, 9, 29)
    self.failIf(cp.matches(d))
```

Cool. It passes. At least, if you have implemented this correctly, the 10 tests including the tests of the composite date pattern should now pass.

It might be a little difficult to see this, but the composite contains a `DayPattern` that matches the 28th and I'm matching it against the 29th, which is why I expect the `matches` method to return `False`.

So we can match dates again. Big deal--we were already doing that. What about wild cards?

We'll write a test to match my anniversary with the new classes:

```
def testCompositeWithoutYearMatches(self):
    cp = CompositePattern()
    cp.add(MonthPattern(8))
    cp.add(DayPattern(16))
    d = datetime.date(2007, 8, 16)
    self.failUnless(cp.matches(d))
```

It just works. Why?

There's no `YearPattern` in the composite requiring the passed-in date to match any specific year. Wild cards now work by *not* specifying any pattern for a given component. Remember when I thought we might need a class to do the wild card matching? I was wrong!

At this point you should have a `NewTests.py` python test fixture. Your test fixture contains many small pattern matching classes, and 11 successfully passing unit tests. If your test fixture is working as described, go ahead and commit your changes to the repository at this point.

Exercise 5.8: Cleaning Up

At this point, we should feel really good about the new approach and we will just delete the old tests and code. This is not unusual in a test driven approach to design. Go ahead and remove the `TestFoo.py` file from your repository and commit your changes.

We'll also refactor the tests a bit, and in the process become more familiar with the concept of a test fixture. Did you notice that every one of the new tests contained a duplicate line? I did. It started to bother me, but that's what test fixtures are for:

```
class PatternTests(unittest.TestCase):
    def setUp(self):
        self.d = datetime.date(2007, 9, 26)

    def testYearMatches(self):
        yp = YearPattern(2007)
        self.failUnless(yp.matches(self.d))

    def testYearDoesNotMatch(self):
        yp = YearPattern(2006)
        self.failIf(yp.matches(self.d))
```

I've only shown the first two test cases (in the fixture previously known as `NewTests`) but now all of the test cases refer to the date as `self.d` instead of constructing a local date instance. It's not a huge refactoring, but it makes me feel better. You *do* want us to feel the best we can about our code, don't you? Of course you do.

I did have to change `testCompositeWithoutYearMatches` to use this date instead of my anniversary. As cute as it was to throw that date in there, I decided I'd rather have clean code without duplication than cuteness.

In case you missed that, at this point you need to modify all of your tests to use the `self.d` fixture. We have also changed the name of the unit test to `PatternTests`. We are going to change the name of the file as well, to better reflect its contents. You should do an `svn rename` of your `NewTests.py` to now be called `DatePatterns.py`.

We will also take this opportunity to clean up some more of the code by adding some named constants for weekdays:

```
MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY = range(0, 7)
```

(This will go at the top of your script, after any imports). Now we can use these instead of the hard-coded constants we had in the weekday tests, and also delete the comments we had explaining what the constants represented. Intention-revealing code beats comments any day of the week. Do these changes now as. Do all of your modified tests using the new `setUp` fixture still work?

Where are we now? Before switching gears, we planned to write a test for a pattern that matched the last Thursday of every month (remember). It's time to do that now:

```
def testLastThursdayMatches(self):
    cp = CompositePattern()
    cp.add>LastWeekdayPattern(THURSDAY))
    self.failUnless(cp.matches(self.d))
```

Cool. A new class to implement! The implementation for this class is slightly more complicated than the others:

```
class LastWeekdayPattern:
    def __init__(self, weekday):
        self.weekday = weekday

    def matches(self, date):
        nextWeek = date + datetime.timedelta(7)
        return self.weekday == date.weekday() and nextWeek.month != date.month
```

Oops. It doesn't pass. Why?

The date we're trying to match in the test is a Wednesday, not a Thursday! We need to fix the test (not forgetting to rename it) as well as add a test where we expect the match to fail (which we should have done before implementing matches):

```
def testLastWednesdayMatches(self):
    cp = CompositePattern()
    cp.add>LastWeekdayPattern(WEDNESDAY))
    self.failUnless(cp.matches(self.d))

def testLastWednesdayDoesNotMatch(self):
```

```
cp = CompositePattern()
cp.add>LastWeekdayPattern(WEDNESDAY))
self.failIf(cp.matches(self.d))
```

Rats. The first test passes but the second one fails. The date created in `setUp` is the same for every test case so it will always be a Wednesday, but we need a date that's not on a Wednesday to make this test pass. Rather than creating a new date in this test case (and ignoring the one created in `setUp`), We'll move both of these tests into a new fixture--one specific for testing the `LastWeekdayPattern` class:

```
class LastWeekdayPatternTests(unittest.TestCase):
    def setUp(self):
        self.pattern = LastWeekdayPattern(WEDNESDAY)

    def testLastWednesdayMatches(self):
        lastWedOfSep2007 = datetime.date(2007, 9, 26)
        self.failUnless(self.pattern.matches(lastWedOfSep2007))

    def testLastWednesdayDoesNotMatch(self):
        firstWedOfSep2007 = datetime.date(2007, 9, 5)
        self.failIf(self.pattern.matches(firstWedOfSep2007))
```

Note: this test fixture will still be in the same file, just place it after the original `PatternTests` test fixture. Also don't forget to remove the last Wednesday tests from the original fixture to move into this new fixture. Now they both pass and the tests use everything they create. Nice. If you have correctly followed the exercise to this point, you should see that 13 tests are being run and passed when you run your test suite.

While moving these tests into a new fixture, we also noticed that we created a `CompositePattern` that contained only one pattern. That's kind of pointless, so we stopped doing it.

Should we move the test cases that exercise the various pattern classes into their own fixtures? That's a tendency that many, including me, often have. It's sometimes useful to resist that urge, though. As Dave Astels, author of [Test-Driven Development: A Practical Guide](#), puts it: a fixture is "a way to group tests that need to be set up in exactly the same way." In other words, a fixture is *not* a container for all of the tests for a single class, or at least, it doesn't have to be.

Having said that, I prefer it when all of the test cases in a fixture exercise the same class. In harmony with Dave's definition of fixtures, I just don't require that *all* of the test cases that exercise the same class be in the same fixture. Make sense?

Suppose that I have four test cases for the class `Foo` but half require different `setUp` code than the other half. I'd split those cases up into two fixtures, even though they both exercise the same class. If I then started writing tests for the class `Bar` and discovered that some of its tests could use the same `setUp` code as one of the class `Foo` fixtures, I would *not* just shove those tests into one of the existing fixtures.

Wouldn't that mean I've duplicated the `setUp` code for the two fixtures? Duplication is evil! If I thought the duplication was enough to be a problem, I would [Extract Superclass](#) the duplicated code out of the two fixtures. Yes, you can--and should--refactor your tests, too.

When starting on a new project, I create one fixture with no `setUp` method and add all of my test cases to that one fixture. Eventually, I reach the point where I need to refactor the fixture, and I do it. Remember: do the simplest thing that could possibly work first. Then refactor if necessary.

What, though, is the benefit of ensuring that all of the test cases in a fixture only exercise one class? Well, besides making it more cohesive (and obeying the [Single Responsibility Principle](#)), think about what might happen when you decide a class is no longer necessary. You'll need to delete the tests for that class, too. It's a lot easier to delete a whole test fixture than to look at each test case in a fixture to see if it exercises the class you just deleted.

Think you won't delete classes? Think again. You saw we deleted the `DatePattern` class and all of its tests earlier, didn't you? It wasn't hard. We felt good about it, too.

At this point you should have a test script that contains several date pattern matching classes, and two different testing fixtures. There should be a total of 13 tests among the two fixtures, and when you run your test script all 13 tests should be correctly passing. If you are satisfied that you have things working at this point, you should go ahead and commit your changes to the repository before moving to the next section.

Exercise 5.9: More Patterns

Because it's so easy and fun, we want to add another pattern so we can detect matches to for example the first Wednesday of every month. As usual we will first write our tests for the new pattern:

```
class NthWeekdayPatternTests(unittest.TestCase):
    def setUp(self):
        self.pattern = NthWeekdayPattern(1, WEDNESDAY)

    def testMatches(self):
        firstWedOfSep2007 = datetime.date(2007, 9, 5)
        self.failUnless(self.pattern.matches(firstWedOfSep2007))

    def testNotMatches(self):
        secondWedOfSep2007 = datetime.date(2007, 9, 12)
        self.failIf(self.pattern.matches(secondWedOfSep2007))
```

I don't have an example of this in my use cases as listed at the beginning of this exercise, but it's a feature that both calendar and pal support, so I expected to add it at some point.

Making these tests pass shouldn't be too hard:

```
class NthWeekdayPattern:
    def __init__(self, n, weekday):
        self.n = n
        self.weekday = weekday

    def matches(self, date):
        if self.weekday != date.weekday():
            return False
        n = 1
        while True:
            previousDate = date - datetime.timedelta(7 * n)
            if previousDate.month == date.month:
                n += 1
            else:
                break
        return self.n == n
```

OK, it was harder than I thought. I'm not a huge fan of the way that algorithm looks, but it's OK for now. We really should at least extract it into its own method so we can give it an intention-revealing name (go ahead and make these changes):

```
def matches(self, date):
    if self.weekday != date.weekday():
        return False
    return self.n == self.getWeekdayNumber(date)

def getWeekdayNumber(self, date):
    n = 1
    while True:
        previousDate = date - datetime.timedelta(7 * n)
        if previousDate.month == date.month:
            n += 1
        else:
            break
    return n
```

Better.

The last example I do have at the beginning of this exercise is the "last day of the month" case. That should match days "in reverse." We could modify the existing DayPattern class, but instead we want to add a new pattern class:

```
class LastDayInMonthPatternTests(unittest.TestCase):
    def testMatches(self):
        lastDayInSep2007 = datetime.date(2007, 9, 30)
        pattern = LastDayInMonthPattern()
        self.failUnless(pattern.matches(lastDayInSep2007))
```

While typing in that test, I decided that I couldn't think of a reason to support the second-to-last day in a month, or the third-to-last day, and so on. Can you? I made it easy on ourselves and decided to implement a class called LastDayInMonthPattern. People usually argue that writing tests up front takes too much work, but writing a test first this time actually saved us from writing code I would never use!

The implementation of this new pattern is:

```
class LastDayInMonthPattern:
    def matches(self, date):
        tomorrow = date + datetime.timedelta(1)
        return tomorrow.month != date.month
```

Nice. Although I just realized we're cheating again. Here's how the fixture should have looked (after extracting out the setUp method) before fully implementing the matches method:

```
class LastDayInMonthPatternTests(unittest.TestCase):
    def setUp(self):
        self.pattern = LastDayInMonthPattern()

    def testMatches(self):
        lastDayInSep2007 = datetime.date(2007, 9, 30)
```

```

self.failUnless(self.pattern.matches(lastDayInSep2007))

def testNotMatches(self):
    secondToLastDayInSep2007 = datetime.date(2007, 9, 29)
    self.failIf(self.pattern.matches(secondToLastDayInSep2007))

```

At this point you should have a test script called `DatePatterns.py`. Your test script contains 8 pattern classes and 4 test fixtures that contain a total of 17 tests, that should all be passing. Go ahead and commit your work to the repository.

Conclusion

I feel pretty good about our code right now. That usually means it's time to refactor. Now I really want to do some renaming.

For the last pattern we implemented, we were very explicit about what it did: `LastDayInMonthPattern` only matches the last day in a month and there's no further clarification needed. What about `NthWeekdayPattern` and `LastWeekdayPattern`? I really want to add `InMonth` to the end of both of those class names. Yes, I'm that picky.

I also took this time to rename a few of the test cases and reorder some of the class definitions. I won't bore you with the details, but you can see the final results for yourself in the example solution for the lab (though for extra practice and if you have been following this lab with interest, you could try and do these changes on your own before looking at the solution).

This type of tidying up may seem trivial but it's extremely important. If your code doesn't look clean, you (and others who find themselves working on your code) won't have any incentive to keep it clean. The Pragmatic Programmers call this the [Broken Window Theory](#). If you live with broken windows, don't be surprised when your neighbors start using your lawn as a junk yard.

We have about 60 non-blank lines of code so far, spread across eight classes. That's not too much, but the code is very simple and yet highly flexible. I seriously doubt I would have been able to conceive of this design without writing our tests first.

What's even cooler is that we have about 90 non-blank lines of test code. Yes, we have more test code than we have "real" code. Is that wrong? Absolutely not. That's wonderful! We should feel extremely confident about the quality of the code that we have so far. Is it perfect? I doubt it. When we discover a bug, though, we can add a new test to demonstrate it and fix it so that it never happens again. If we need to perform an optimization, we'll have a suite of tests we can use to verify that we didn't screw anything up while applying the optimization.

What's also interesting to note is the design that emerged from this work. We spent zero time in front of a modeling tool trying to create a design that would both meet our needs today and still be elegant enough to (hopefully) meet all of tomorrow's needs, as well. We didn't intend for this to happen. It just magically happened that way. This isn't rare--this almost always happens when one does test-driven development.

How is this design more flexible than we originally intended? Suppose that we want to create a pattern that matches every Friday the 13th. That wasn't one of our original use cases, and I gave no thought to it while writing the tests. The classes we came up with have no trouble representing that pattern:

```

[harry@nisl lab05]$ python
Python 2.4 (r25:51908, Nov 6 2007, 16:54:01)
[GCC 4.1.2 20070925 (Red Hat 4.1.2-27)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import DatePatterns
>>> fri13 = DatePatterns.CompositePattern()
>>> fri13.add(DatePatterns.WeekdayPattern(DatePatterns.FRIDAY))
>>> fri13.add(DatePatterns.DayPattern(13))
>>> import datetime
>>> jul13 = datetime.date(2007,7,13)
>>> jul13.strftime('%A')
'Friday'
>>> fri13.matches(jul13)
True
>>> sep13 = datetime.date(2007,9,13)
>>> sep13.strftime('%A')
'Thursday'
>>> fri13.matches(sep13)
False
>>>

```

While we're not done with the application yet, we do have a solid foundation to build on. Next, we would need to add some parsing code (probably a filter, as we have done in labs 3 and 4) so that we can read a file containing events in order to construct and use the patterns we implemented above. I'll leave that as an exercise for the student.