Working Effectively With Legacy Code

Michael Feathers
Object Mentor, Inc.
mfeathers@objectmentor.com

Last Change: April 9, 2002

Over the past fifteen years, much has been written about object oriented design and team development processes. The unfortunate thing is that much design and process advice assumes that your project is a blank page. In actuality, greenfield projects are noticably rare. Most projects carry some amount of legacy code. In many new development efforts, the amount of legacy code will overwhelm the amount of new code by factors of 100 to 1, or 1000 to 1. Needless to say, you can't work very fast with a legacy code base. Work is often much slower, but you can speed it up if you establish a strategy to deal with your existing code and mitigate risk as new development goes forward. In this paper, I'll outline a strategy that can be used to work with legacy code. But first let's establish some background.

What is Legacy Code?

A few years ago, I asked a friend how his new client was doing. He said "they're writing legacy code." I knew what he was saying immediately, and the idea hit me like a ton of bricks. After all, there is an emotionally neutral definition of "legacy code." Legacy code is code from the past, maintained because it works. But, for people who deal with it day in and day out "legacy code" is a pandora's box: sleepless nights and anxious days poring through bad structure, code that works in some incomprehensible way, days adding features with no way of estimating how long it will take. The age of the code has nothing to do with it. People are writing legacy code right now, maybe on your project.

The main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests. We can get a sense of this with a little thought experiment: how easy would it be to modify your code base if it could bite back, if it could tell you when you made a mistake? It would be pretty easy, wouldn't it? Most of the fear involved in making changes to large code bases is fear of introducing subtle bugs; fear of changing things inadvertently. With tests, you can make things better with impunity. To me, the difference is so critical, it overwhelms any other distinction. With tests, you can make things better. Without them, you just don’t know whether things are getting better or worse.

The key to working effectively with legacy code is getting it to a place where it is possible to know that you are making changes *one at a time.* When you can do that, you can nibble away at uncertainty incrementally. The tests that you use to do this are a bit different from traditional tests. I like to call them test coverings.

Test Coverings

A "test covering" is a set of tests used to introduce an invariant on a code base. These tests are a bit different from the ones that are most often talked about in Extreme Programming. For one thing, they tend to cover the behavior of a set of classes rather than just a class and its immediate collaborators. For another, they tend to cover some small area of a system just well enough to provide some "invariant" that lets us know when we've changed the behavior of our system. The key thing is that correct behavior is defined by what the set of classes did yesterday, not by any external standard of correctness.

In a design driven from the beginning using tests, the tests serve a couple of purposes. They seed the design, they record the intentions of the designers, and they act as a large invariant on the code. In legacy code, it would be great to have tests which do all three for us, but nothing is free. We can produce the most value working backwards: build the invariant first, then refactor to make the code clear. If we discover that the results from our test covering are not what the system should calculate, we can deal with that as a separate issue. The primary goal is to get that invariant before refactoring or adding new behavior.

Your approach to test coverings can vary quite a bit depending upon whether your system is in production or not. Systems that are already deployed often require far more diligent covering because the cost of errors is significantly higher. Typically, there are users who've grown to depend upon the current behavior of the system. On the other hand, if a system has never been deployed and there is quite a bit of legacy code, i.e, code without tests, you can often refactor with relative impunity, bringing things under test as you go. It is quite literally the case that no one "knows" whether the code is correct or not. The number of bugs that you introduce by doing some initial refactoring without a strong test covering, may be marginal compared to the number of bugs that you'll discover as you bring code under test. Essentially, you have to make a judgement call about how much risk you are willing to assume as you bring the system under test.

Now, let's talk about how to move forward.

Legacy Management Strategy

By itself, legacy code doesn't hurt anything. As long as it works, it only becomes painful when you have to make modifications. Fortunately, the first steps you have to take to work effectively with legacy code also make it easier to clean things up.

The general legacy management strategy is:

1. Identify change points
2. Find an inflection point
3. Cover the inflection point
    a. Break external dependencies
    b. Break internal dependencies
    c. Write tests
4. Make changes
5. Refactor the covered code.

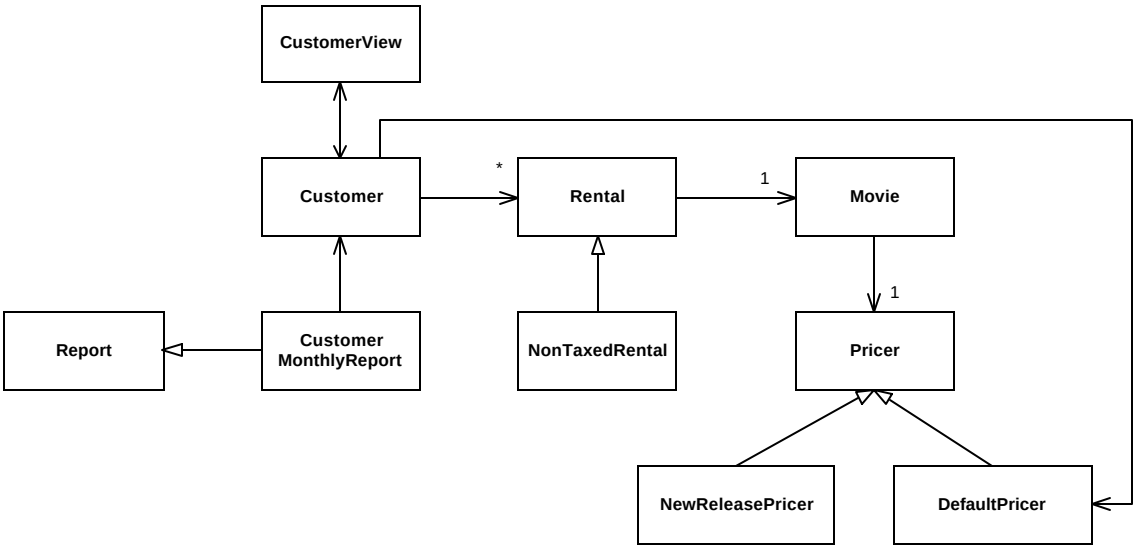
Identify Change Points

When you have to make a change to legacy code, the first step is to figure where the changes will need to be made. There really isn't much to say about this except that the amount of work involved varies with the degree of sickness in the code. Some changes may require a lot of work in different areas of the system. However, if there are multiple ways of making the changes, and you do not yet have test coverings in place, there are dividends for choosing the way which requires the fewest changes. In other words, the "right" way to add a change may not be the best initial choice. Why would this be? For the most part it is because bringing large areas of code under coverage can take an incredible amount of time. Fortunately, once you bring an area of code under test, it is easier to deal with the next time. Over time, the “islands” of coverage that you create in your code base will merge and you can refactor the classes in them to consolidate further.

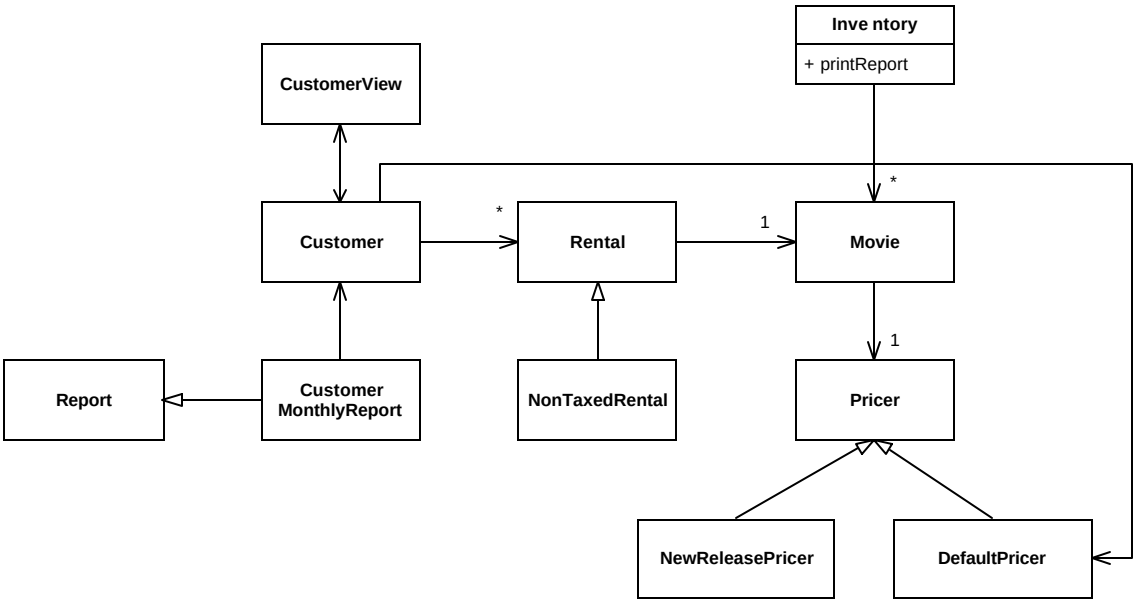
Find an Inflection Point

After you’ve found the places in the code that you need to change, the next thing you need to do is find an inflection point. An inflection point is a narrow interface to a set of classes. If anyone changes any of the classes behind an inflection point, the change is either detectable at the inflection point, or inconsequential in the application.

Let’s use a variation of the video store example that Martin Fowler developed in his book Refactoring [1]. We would like to make changes to the way that videos are priced. Customer is an inflection point for Rental, Movie, Pricer, and all of their subclasses. It is an inflection point, because there is no way that any behavioral change in Rental, Pricer, and Movie or any of their subclasses can change the behavior of the application without going through it. More specifically, the rest of the system receives state changes through methods on Customer. If we have comprehensive enough tests at the Customer level, we can be reasonably certain that we are not changing other things when we refactor or add things behind the inflection point.



Let's imagine another scenario. Suppose that we introduce a class named Inventory which maintains a collection of all the movies in the system and prints a report.



Once we've done that, Customer is still an inflection point for Rental, but not for Pricers or Movies. Any change that we make to Rental must go through Customer to be effective in the system. But, if we start to make changes to Pricer, those changes could be propagated to the rest of the system via Inventory as well as Customer.

Inflection points are not solely determined by physical dependency, but rather by the way that effects are propagated at run time in software. To see this, imagine that Customer did not have any methods which returned values. The dependency picture above would be the same even though CustomerMonthlyReport and CustomerView would have to get information about the customer in some other way.

When you try to find inflection points, move outward from the places you are going to change the software. Look for a narrow interface. It could be one class or several. As well, it is important not to depend on preexisting UML diagrams as you do this. Diagrams rarely show all of the users of particular classes.

Cover Inflection Point

Covering an inflection point involves writing a tests for it. The hard part of this is getting your legacy code to compile in a test harness. You often have to break dependencies.

In the case of the little example system above, we'd like to get a Customer object in a harness so that we can write some tests which cover its functionality. The easiest thing is to just try to create a new instance of the class. At that point, we'll discover what we need to provide it to get it to work properly. There are two types of dependencies which we will run into immediately: external dependencies and internal dependencies. External dependencies are objects which we have to provide to setup the object we are creating. Often they are constructor parameters, or objects which we have to set at the object's interface. In the case of Customer, it seems that we will have to provide a CustomerView object when we create it, but do we really?

Breaking External Dependencies

Objects talk to other objects to get work done. This can be pretty distressing if you want to separate out a cluster of objects to make a test covering. Fortunately, you can easily sever the connection between any two objects. This is known as *dependency inversion*, and here is how you do it:

```
class CustomerView
{
        private Customer _customer;
        public void setCustomer (Customer customer) {
                _customer = customer;
        }

        public void update() {
           nameWidget.setText(_customer.getName());
        }

        ...
}
```



```
class Customer
{
        public Customer(CustomerView view) {
                _view = view;
                _view.setCustomer(this);
        }
        ...
}
```

As we can see, Customer objects have a direct dependency on CustomerViews. We need a customer view whenever we create a customer. Actually, the situation is far worse than that. We need a customer view and we also need everything that a customer view needs, transitively. If customer view has a non-trivial setup, we could spend hours hunting through chains of references, finding more thing to create just so that our tests will be able to run.

The problem is that Customer depends on a concrete class, worse it is one we don't even care about when testing. For our test covering, we are just interested in providing values to customer objects and asking for results through the customer interface.

We can break the dependency by changing CustomerView into an interface and renaming the original class:

```
interface CustomerView
{
        void setCustomer(Customer customer);
        void update();
}


class StandardCustomerView implements CustomerView
{
        private Customer _customer;
        public void setCustomer (Customer customer) {
                _customer = customer;
        }

        public void update() {
           nameWidget.setText(_customer.getName());
           ...
        }

        ...
}
```



```
class Customer
{
        public Customer(CustomerView view) {
                _view = view;
                _view.setCustomer(this);
        }
        ...
}
```

Now, we can create a customer object in our test without much trouble:

```
Customer customer = new Customer(new CustomerView () {
        public void setCustomer(Customer customer) {}
        public void update() {}
});
```

The bodies of the methods are empty because we just don't care what happens to the view as we test through a Customer object.

Breaking Internal Dependencies

Internal dependencies are a little trickier to deal with. When the class we want to cover creates its own objects internally, sometimes the best thing that you can do is subclass to override the creations.

Imagine that customer creates an archiver that records actions taken against its interface. We want to put Customer under test, but we don't want to have a dependency on the Archiver class. Archiving is a real performance hit and our tests would run abysmally slow if we did it. Unfortunately, Customers create archivers in the constructor:

```
class Customer
{
        private Archiver _archiver;
        public Customer(CustomerView view) {
                ...
                archiver  = new FileArchiver(customerPersistenceName);
                ...
        }

}
```


One way around this is to extract the creation of the archiver. We can create a method named makeArchiver and add it to the customer class:

```
class Customer
{
        private Archiver archiver;

        public Customer(CustomerView view) {
                ...
                archiver = makeArchiver();
                ...
        }

        protected Archiver makeArchiver() {
                return new FileArchiver(customerPersistenceName);
        }

}
```

Now, to create a customer in our test, we can do this:

```
class TestingCustomer extends Customer
{
        protected Archiver makeArchiver() {
                return new NullArchiver();
        }
}

Customer customer = new TestingCustomer(new CustomerView () {
        public void setCustomer(Customer customer) {}
        public void update() {}
});
```

Here we've introduced a new class TestingCustomer which just overrides the makeArchiver method. When we create a TestingCustomer, we can be sure that we having something which behaves like the Customer in our tests, but with one tiny difference: it doesn't use a real archiver. We can use it to test everything past the inflection point.

The makeArchiver method in TestingCustomer is a bit special too. It creates a NullArchiver. All NullArchiver is, is a class which implements the Archiver interface

and provides bodies that don't actually do anything. We could create a NullCustomerView class also so that we don't have to create an anonymous inner class when we test:

```
Customer customer = new TestingCustomer(new NullCustomerView ());
```

Generally, I create Null classes whenever I end up duplicating code in anonymous inner classes. They can be very handy as you move forward in testing. If you need to tailor a few methods for use in a particular test, you can override only those methods, assured that the rest do nothing.

If you are tempted to use this strategy in C++, be leery of the fact that when you call a virtual function in a base class constructor, the method that is executed is the one in the base class, not the one you've overridden in the derived class. To get around this in C++, you either have to use two stage initialization or lazy-initializaton. In two-stage initialization, you add an *initialize ()* method to Customer and make sure that all clients of Customer call *initialize ()*. Then you move the creation of archiver to the initialize method and extract the makeArchiver method. In lazy-initialization, you write a getter for the archiver field and make sure that all accesses of the archiver go through the getter. Then you use an internal test to determine whether it is time to create the archiver.

```
class Customer
{
        protected:
                Archiver *archiver;
                Archiver *getArchiver() {
                        if (archiver == 0)
                                archiver = new FileArchiver(customerPersistenceName);
                        return archiver;
                }
        public:
                Archiver() : archiver(0) { … }
                …
};

class TestingCustomer : public Customer
{
        protected:
                Archiver *getArchiver() {
                        if (archiver == 0) archiver = new NullArchiver;
                        return archiver;
                }

};
```


Subclassing is a great strategy when you need to override the creation of objects, but what do you do when you have internal dependencies on things that you haven't created? These dependencies are affectionately known as *global variables*. From a testing perspective, it isn't the fact that these variables are global that is a problem as much as the fact that they are *variables*. Let's think about this in terms of *effects*.

When we were discovering inflection points, we were trying to find narrow parts of the design which channel all of the effects of set of classes. When we put the classes for that inflection point in a test harness, we have to faithfully set up all of things those classes depend upon. Why? Well, it isn't just a matter of physical dependencies. The things we depend upon must *act* as they would in production code. The problem with global variables is that prior to any test run, you must provide them with a good known initial state. Often that requires a lot of tedious work. If you forget to set up particular variables, you can easily have tests which bleed state from one execution to another. You may not be testing what you think you are testing.

In OO systems, global variables often show up as instances of the singleton pattern or just as static data in classes. Not all singletons are on the same footing as global variables. In particular, singletons which do not affect the functional behavior of an application (caches, factories), can behave well as internal dependencies.

Writing Tests

Once we are able to put the objects of an inflection point in a test, we have to go through the process of placing some sort of an invariant on the code guarded by the inflection point.

Remember our key assumption here: code changes behind the inflection point can not have effect in the system without passing through the inflection point. If that is true, then we can start to write tests for using the interfaces of the inflection point. In the video store example, we would write tests against the Customer interface.

How do you find good covering tests? If the interface is narrow enough, you can start with boundary values to see how the subsystem behaves at the edges. Remember that correctness, at this point in time, is just how the system behaves currently. Another thing that you can do is take the code behind the inflection point and start to change it to see how values change at the interface. Often this can give you ideas for additional tests. Needless to say, when you do this you should always be very clear about where the golden, ostensibly correct copy of the code is.

One thing that I haven't done, but I'm eager to try, is automated test generation. Since the goal is simply to characterize the existing behavior of the code and sense when changes invalidate it, a set of scripts can be used generate sample data. While the cases

that you generate that way may not be the ones you would like to keep in the system over its lifetime, they can be used to keep an invariant.

Make Changes

This is programming, pure and simple. Once you have test coverings over the change areas, you can incrementally make your changes. Take care to run your tests often. Try writing tests first, for the changes that you make. After you make changes, add additional tests to bolster your work.

Refactor Covered Code

Once you have a test covering in place, you have a wonderful opportunity to clean up your design. At times, it can be hard to figure out how to take advantage of it; particularly if the code is very poorly structured. The hallmarks of this sort of code are: large classes with extremely large methods.

The important thing to realize is that getting the covering in place is the hard work. You can refactor on demand as you make changes. However, I've discovered that the refactoring that I do on legacy code tends to follow a little pattern. If I want to start to clarify some covered code, I first start by doing the *extract method* [1] refactoring over and over again. Then, I start to pay attention the pattern of usage in the class. Are there groups of methods which *use* the other methods and data in the class? If there are, I may have found a place where the class can be split. At that point, I use the *extract class* [1] refactoring. Being very sensitive to these patterns of *use* in class can make a big difference.

As you refactor, remember to keep writing tests. Even though test coverings can remain as part of your test suite, it can be easy to forget they are there, or worse, believe that they test things that they do not. The best thing that you can do as you refactor is get into the habit of refactoring into tests. Refactoring into tests is a little like test first design. In the case of *extract method*, you look at a large method, imagine a portion you'd like to extract, write a test for it, then you extract the method to make the test pass. When you refactor into a test, you are forcing yourself to consider all of the decoupling that you need to do to write good tests. Once you have the extraction, you can cover it further by writing more tests.

Conclusion

The strategy that I've outlined works for a wide variety of changes, however there are some caveats. Sometimes the only decent inflection point that you can find for a set of classes is the system boundary. In some applications, the system boundary can be pretty wide: it encompasses the GUI, calls to other external libraries, the database, etc. In those cases, the best way to get an invariant is to start writing what Steve McConnell calls "smoke tests" against the rest of the system [2].

One promising avenue for Java programmers is the development of AspectJ [3]. With AspectJ, you can write code that will intercept calls in an existing application without modification. You can use it to log results. Then, when you refactor, you can run again and see if the new set of results differs from the old one. If it does, you've modified behavior. It's time to roll back and start over again.

Another issue that has come up as I've explained inflection points is the question of how prevalent they are in legacy code Paradoxically, I've found that they are often easier to find in poorly structured code than in well-structured code. In poor code, there isn't a high degree of internal reuse so it is often pretty easy to find somewhat localized class usage. When you start refactoring code and eliminating duplication, infection points can go away. It may seem that we are working at cross purposes, but once unit tests are in place, scaffolding each class, inflection points don't really do much for us. It is odd, but it is kind of nice that there is *one* quality of poor design that helps us when we want to improve it!

[1] Fowler, Martin *Refactoring: Improving the Design of Existing Code*, Addison-Wesley 1999

[2] McConnell, Steve *Rapid Development: Taming Wild Software Development Schedules*, Microsoft Press 1996

[3] AspectJ website: www.aspectj.org

