Working Effectively With Legacy Code

Last change: 2/26/2003 1:58 PM

Michael Feathers
mfeathers@objectmentor.com

To add:

- FileZilla refactoring example
- Change internal & external to implicit and explicit
- ~~Discussion of in vitro and hermetic harnesses~~
- Extracting what you understand
- Inflection points
- Connection errors
- Legacy management strategy
- Deal with constructor and object lifetime issues more thoroughly
- Cut to the bone on wrapping an API, then nullify the API with mocks and sense through it.
- GUI Testing
- Library trouble
- Modularity makes testing easier
- Strong and weak separation
- Reasoning about effects
- Template specialization for separation
- Need lots of consolidation remarks and bullet lists
- Why we don't like to call something and pass its results to a new thing.
- Why we don't like to use #ifdef
- Mock section
- Sprout pattern
- Add globals encapsulation and Darren's interface for methods pattern (the claw)
- Inheritance is 3D (ST quote)
- Globals make software sticky
- Progressively adding interfaces, one method at a time
- Growing Null objects and sensing objects
- Wrapping close to the bone, or with intention: how to decide which to do could be a section when discussing the claw
- Dependency breakage via primitives
- Separation between test code and production code… showing overriding in subclass to test.
- Talk about evolution cases for each refactoring: parameterize constructor to independent use.
- Writing tests
- Effects… reasoning
- Why no abstract factory
- Refactoring package structure
- Parallelizing rewrites… write test to characterize, test in other language, code to satisfy… look at differences.
- Being a team

Acknowledgements

I’d like to thank everyone who has reviewed the drafts as I’ve put them up for feedback. Thanks to Darren Hobbs, Keith Nicholas, Sven Gorts, “Uncle Bob” Martin, and Martin Lippert.

Introduction

Software is a time bomb. It is easier to write bad code than it is to write good code and it has always been that way. As long as it is easier to write bad code, there will be systems out there that are wild, woolly and difficult to deal with. Developers tend to have a vicious streak of idealism. We get a picture in our minds of the way a system should be, and we use it as a standard for our efforts. Generally, that is a very good thing. Each of us in the field has experience confronting portions of code that are difficult to deal with because no one took the time and effort to make them understandable. The problem is exacerbated because often the people we are complaining about are ourselves. Maybe we fell off the wagon under deadline pressure, we didn't write code as cleanly as we should have because we felt we just had to get it done. Maybe we were having a bad day, distracted, not at our best. The problem is, that day doesn't go away. It stays there in the form of code, mocking our idealism.

In the past few years, refactoring has been a hot topic in the software development community. In 1999, Martin Fowler released his book *Refactoring: Improving the Design of Existing Code*. It provided a comprehensive catalog of refactorings that developers can use to keep their systems nimble and easy to change. To be perfectly frank, I think that Refactoring is most important books in the OO canon. To me there were a couple of important things that set the Refactoring book apart:

1. It described day to day work with existing systems, not just new systems people were developing from scratch.
2. It showed just how malleable the design of a system can be without rewrites.

However, one of the key pieces of advice that Martin Fowler gives in the Refactoring book, to write tests before attempting to refactor, has been very difficult for many people in practice. Many teams I visit have large code bases that were not designed with testing in mind. For a variety of reasons, the tests are difficult to get in place and this puts a strong damper on the work that you can do moving forward.

Without tests, refactoring is pretty scary. Often the safest course you can take is to avoid refactoring entirely, but that safety comes at a terrible price. When you are scared to change existing code, entropy sets in. The way that you add new features is too conservative. You add a little bit of code to an existing method, or a method to an existing class. Maybe you notice that the method is getting a little long or that the name of the class doesn't cover half of what the class does, but you were able to make your change and see that that the app was still running when you checked your changes into version control. Maybe you feel pretty confident because you made only a slight change and couldn't imagine anything that could go wrong. The downside is that your methods and classes keep growing. Over time they become harder to understand and you feel less and less confident in the changes you are making. Bugs slip in. It takes longer and

longer to make modifications to the system. People start to think seriously about a rewrite.

Unfortunately, many teams have mature systems made up of bloated classes containing bloated methods that are nearly impossible to test. If they are nearly impossible to test, they are nearly impossible to refactor. The developers know it is a problem, and while they want to refactor they need help getting started. That is what this book is about. It is a primer for getting legacy code bases under control. How do you start, what do you do when you hit roadblocks? We'll deal with all of those issues. But first, let's talk about legacy code.

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 really saying immediately, and the thought of it hit me hard. True, there is an emotionally neutral definition of "legacy code": code from the past, maintained because it works. But, for people who deal with old code day in and day out "legacy code" is a Pandora's Box. "Legacy code" means 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 a lack of comprehensive 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 mistakenly broke behavior you need to preserve? It would be pretty easy, wouldn't it? Unfortunately, on most projects when we edit code we have very little feedback about whether the edits we made changed the code in a good way or a bad way.

Few developers work completely in the dark, some have a set of manual actions they perform against the system each time they recompile just to see if they changed behavior the way they expected, but what about all of the other behavior of the system? Unless it is being tested also, there is a possibility that you are introducing subtle bugs. In the back of our minds we know this, so what do we do? We become more cautious, we try to reason about whether a change we are making could cause anything else to change, and we hesitate in the face of routine refactorings. A method gets a little long, should we extract a method? We could, but if we are scared that we may break the code, we'll avoid it. That avoidance is the slippery slope that makes people use "legacy code" as a pejorative term.

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 often 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 tests, you don't really know whether you are making them 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" without affecting anything else. When you can do that, you can focus on the work that you need to do, get real feedback and confront ramifications of your work immediately rather than hearing about them from irate users months later.

What happens if you don't have the tests that you need? Well, if you are really smart you can get away with making changes without tests. But don't get too excited. I'm continually astounded by the fact that "smart" doesn't cut it. Very smart people do make

mistakes because it very hard to reason about effects in software. Worse than that, if you rely on reasoning rather than tests, chances are you are being very conservative. The old adage goes: "if it ain't broke, don't fix it." Unfortunately, that philosophy leads to really messy systems. If you don't add a feature in the most understandable way because you want to avoid changing the existing system, you've just made the system a bit more chaotic. It turns out that avoiding tests by being smart can be stupid.

In my opinion, the most difficult problem in software development is *understanding*. Do we really know what a particular piece of code is doing? If we don't know what each piece of code is doing, how will we have any idea about how long it takes to make a change to the system? In the worst case, no one can provide an estimate within an order of magnitude because understanding what needs to change is akin to an archeological dig. The cost of making the change is negligible next to the cost of figuring what has to change. You might as well do the work while you are getting information for an estimate. Many systems go on like this for years.

We can try to build up understanding by reasoning about software, but that isn't very efficient. We have to re-reason about it whenever we want to understand it again. Tests tell us immediately what at piece of software does. Armed with that knowledge, we end up with grounded understanding of our system. Without that understanding, there isn't any decent way of knowing how long it will take to make a change; we just know that it is taking progressively longer.

Getting a System under Test

Okay. You realize that your system has a problem. You know that you have to do something to keep things from getting worse and you are ready to start. You have a couple million lines of source code that your team has feverishly developed over the past ten years. The product is out in the market and doing well. There are some quality issues, but you have a QA department. They run tests before each release, but you get bug reports from the field and they are consuming more and more of your development time. It feels like there is a rock on your shoulders. When you look at your teammates, they look crushed too. You can read their faces as they walk in each morning: "back to the salt mines."

You'd like to be able to add tests to your code as you work, but that means refactoring. You've been browsing the web and you've come across all sorts of material about refactoring. The resounding theme is "don't refactor unless you have tests. If you don't have tests, you're in trouble!" You look around the room and you say to yourself, "how can we get out of this chicken and egg situation?"

One thing that is important to realize is that everything does not have to happen at once. You can get your system under test incrementally. Truth be told, legacy code is not a problem until you have to change it. Having several megabytes of tawdry source code with few tests is not a very comfortable position, but if the system is running and deployed, you do have a product; an asset that you can work with. Working with it is the difficult part.

Characterization Tests

If you look through any book on testing, you'll find a great deal written about designing tests that verify correct behavior. Correctness is very important, but when it comes to being able to work with a system it is equally important to just know what it currently does. In fact, knowing what the system currently does, in detail, is a good first step towards knowing whether it is correct. The kinds of tests we need are tests which characterize the current behavior of the system. When we have tests that capture the behavior of the system, we can use them as an invariant for development. If we are refactoring, the tests should continue to pass at each step. If we are adding new behavior they should all pass also. If we are changing existing behavior, some may fail but we can examine the failures and decide whether we are removing and adding the right behaviors.

Let's take a look at an example. Suppose we have a small program that plays Tic-Tac-Toe, but it doesn't have any tests. By default, the human player gets to go first and the program responds with its move. The player and the program trade moves until one wins or the game ends in a draw.

Here is the transcript of a game (the squares are numbered from 1-9 starting from the upper left hand corner and moving across and downwards):

Here is a typical game:

```
Player : 5
Program: 1
Player : 4
Program: 7
Player : 8
Program: 3
Player : 2
Program: 6
Player : 9
Program: "draw"
```

In a little testing framework, we could code this up as a script. Every time we pass a number to the program, we can record the result and set it up as the expected value for a test. If our program has a decent class interface class we could start writing tests like this:

```
TicTacToeGame game = new TicTacToeGame();

game.acceptMove("5");
assertEquals("", game.getLastResponse());
```


In those lines we are giving the game our move and then saying that the game should not have any response at all, an empty string. We know better, but when the test fails, we'll be able to see what the actual value is. In this case it is "1"

Now we can modify the test to get it to pass:

```
TicTacToeGame game = new TicTacToeGame();

game.acceptMove("5");
assertEquals("1", game.getLastResponse());
```

We can do this over and over again, building up scenarios that characterize the behavior of the game class.

Now what happens if we go through a series of these scenarios and we end up finding a bug? Take a look at this scenario:

```
TicTacToeGame game = new TicTacToeGame();

game.acceptMove("5");
assertEquals("1", game.getLastResponse());

game.acceptMove("3");
assertEquals("2", game.getLastResponse());

game.acceptMove("7");
assertEquals("3", game.getLastResponse());

game.acceptMove("9");
assertEquals("I won!", game.getLastResponse());
```

The program decided it'll just ignore your win, place its piece over yours and declare victory. It's either a bug or arrogance, but it makes me glad I'm a programmer because I can deal with arrogant programs in ways non-programmers can't.

We have a choice as we are adding tests for the program. We can stop right now and fix the bug we've discovered, or we can mark this test as a bug and carry on writing tests. In general, I like to set out with an idea of what behavior I am going to characterize and carry it through to completion. If I mark something as a bug and get back to it later, I have the advantage of all of the other tests that I've written. They will show me whether I am changing any other behavior when I fix the bug. The same is true for behavior that I

don't recognize as a bug. If someone reports a bug to me later I can see if there is a test that shows the behavior, if there is, I can change its expected value to make it fail and then use the other tests to bolster my work as I make it pass.

The important thing about characterizing behavior with tests is that you are building up a way of sensing when the behavior of your system has changed. When you are able to sense behavioral change, you have a better chance of controlling it.

Getting Tests in Place

Smoke Tests

The hardest part of getting tests in place is getting access to the things you want to test. In large legacy systems this is non-trivial. Often you have to refactor to be able to test, but refactoring without tests can break things. So, what do you do?

I like to start establishing a key invariant at the system boundary. Look for the key thing that the system does; the basest case. Are you developing a payroll system? Okay, find a dataset of employees with odd payment characteristics. Run them through the system and record the results. Are you making a GUI file comparison program? If so, do you have a screen scraping tool that you can use to capture comparison results for a few files? Are you working on code for a website? If so write a little program that pokes it and captures HTML from it.

In many cases you can get some tests in place without touching your system's source code. In other cases, you just have to go in there and modify the source code. This is particularly true in GUI applications that do not have any programmatic interface to drive them or retrieve their results. Worse, the GUI applications that really merit the legacy "badge of honor" have their business logic so deeply encrusted in the presentation code that developing a programmatic interface is quite a bit of work. Regardless, getting a key invariant in place is very important. Steve McConnell calls this sort of test a "smoke test": does the system "smoke" when it runs? For us, it is nice to have some confidence that as we work with the code, we are at least preserving its key behavior. Moreover, it seems that the hardest system level test to get in place for any project is the first one. Once you have it, it is far easier to add tests because you've already done some setup and cleared a path towards sensing the system's behavior.

Let's take a look at a few examples of getting Smoke tests in place.

<FileZilla,HTML bridge, etc>

Tools of the Trade

Getting tests in place is hard work. You have to create objects, manipulate them and record the results. Unfortunately, the way that classes are often knit together in real applications makes it hard to pull them out for testing. You have the dependencies among the classes to deal with along with build dependencies.

Build dependencies can be particularly problematic. In some large projects, there is enough componentization to be able to build and test small parcels of an application independently. In other projects, the amount of work that you have to do to create separate projects for testing is substantial. Often you have to make a decision about how you are going to get tests in place on a project.

In the field, I distinguish between two types of test harnesses: *in vitro* and *in vivo*.

In vitro Harness

An in vitro harness is the typical external test harness that people often use for unit testing. When you start to write tests you set up a little project, create a test case class, and attempt to create the objects you want to bring under test. Often you have to set up dependencies to particular jar files, DLLs, or assemblies in order to get your tests to compile and execute. In the process of working in vitro, you often have to break a wide variety of dependencies to build the code that you want under test and leave out the code that you don't need.

One advantage of in vitro harnesses is that they force you to confront the build process: are you building things you don't need for this particular test? If so, you are better off removing what you don't need. If you don't, your edit/compile/link cycle will be longer than it needs to be.

It is easy to underestimate the power of short edit/compile/link cycles. There is as substantial a difference between cycles of 1 and 10 seconds as there is between 10 seconds and 10 minutes. In addition to the obvious loss of productive time, waiting causes mental fatigue, which slows down work further. Getting in vitro harnesses in place is a good opportunity to break dependencies and create areas of code which have very small build times.

In vivo Harness

An in vivo harness is a test harness that "hijacks" the build process of your project and makes tests available from the user interface of your application. In vivo harnesses are easy to add to a project. You create test classes in your project, link in the test harness and provide some mechanism for running them from the application. In desktop applications, you can provide some hotkey like Ctrl-F12 which brings up the UI for your

test harness and lets you run all of the tests that are part of your application. In a web based application, you can provide a separate URL which runs all of the unit tests automatically.

The key advantage of in vivo harnesses is the fact that you can get tests in place with minimal modification to your existing build and project. The disadvantage is that installing an in vivo harness does nothing to break down dependencies in projects. While writing tests is easy, you generally have to wait for a complete build before you are able to run them.

Software Plumbing

At the time of this writing, software development has been around for at least fifty years. For most of that time, it has been searching for its own identity. The first programmers were hardware developers and they brought a very pragmatic focus to programming. In a parallel path, people in the academic community laid the foundations of computer science and approached programming from a very mathematical perspective. From the beginning, many people have bemoaned the fact that software development is not an engineering discipline. Their voices seem to get louder at times and then recede in regular cycles. All along we are left with the same questions: what is programming? Is it math, a form of writing, or something like manufacturing? I don't think we'll ever have a satisfying answer, but I do know that one thing that helps me when I program is to think about the work in very physical terms, to consider what software is made of.

The Material of Software

We know that software is ultimately all zeros and ones, and our grandfathers and grandmothers noticed that zeros and ones are hard to deal with so they started to make high level constructs make programming easier. In the beginning there were assemblers which allowed little mnemonics to be used to represent machine instruction numbers. At that point software was a long list of commands with scattered jumps from one place to another, a step above zeros and ones but still very hard to manage.

As time went on people noticed that these lists of commands could be reused and the subroutine was born. Still, however, software development was really an issue of list management, creating these lists of commands and reusing them where possible. Programmers had to structure their lists to make them easy to understand while visualizing how instructions worked on registers and stacks in the hardware.

The next big leap forward came with the advent of high level languages. Programmers were removed even further from the machine. In typical programming they didn't have to think about registers at all, they had variables, symbolic names for storage, and the notion of a "stack" was built into their languages.

At this point, the "material" of software had a very interesting quality. When you called a function, the function you went to was precisely the one you called. The compiler and linker hard coded an address that your code would jump to and there was no easy way to substitute another function short of just changing the code. In mechanical terms it was as if the caller and the callee were glued together. You can build very large systems with this sort of material, but if you want to mix and match pieces of programs, it is rather tough going.

When object orientation came along the material of software changed quite a bit. Instead of making functions and calling them directly, developers could write little pieces of software and pretend that they were "things" that do work by sending messages to each other. This kind of physical reasoning can be very handy. You can think primarily about the things you want to have in your application and how they interconnect rather than tying yourself to a particular algorithm.

When object-orientation started to enter the mainstream, people were very excited about the possibilities for reuse. If you could make "things" and plug them together then you could make reusable things that could be used in many contexts. It turns out that reuse is very hard, but there is one key thing that objects enabled that really set them apart from procedure and functions. When you call a function in an OO language, the function that you execute isn't hard coded. There is a little "wedge" in OO that decouples software. Let's take a look at an example. If we have a little C++ function like this:

```
double distance(point a, point b)
{
        return sqrt(pow(a.x-b.x,2.0)+pow(a.y-b.y,2.0));
}
```

it is pretty clear that the function is called in the following code:

```
void printDistances(tree *root)
{
        if (root->left == 0 || root->right == 0)
                return;
        printf("distance to left is %lf\n", distance(root->location, root->left->location));
        printf("distance to right is %lf\n", distance(root->location, root->right->location));

}
```

but imagine that our calling code looked like this:

```
void printDistances(tree *root)
{
        if (root->left == 0 || root->right == 0)
                return;
        printf("distance to left is %lf\n",
                root->location->distanceFrom(root->left->location));
        printf("distance to right is %lf\n",
                root->location->distanceFrom(root->right->location));

}
```

By adding that distanceFrom function, we've added a little wedge to the code. If we change the class of the location variable, we can execute different code for distanceFrom.

For instance, trees held onto cartesian points, the code for distanceFrom might look like this:

```
class cartesian_point : public point
{
private:
		double x,y;
public:
	virtual double distanceFrom(cartesian_point other)
	{
		return sqrt(pow(x-other.x,2.0)+pow(y-other.y,2.0));
	}
	…
}
```

If they held onto polar points, the code might look like this:

```
class polar_point : public point
{
private:
		double rho,theta;
public:
	virtual double distanceFrom(polar_point other)
	{
		return sqrt(
			pow(rho,2.0)+ pow(other.rho,2.0)
				– 2.0*rho*other.rho*(other.theta-theta));
	}
	…
}
```

If both cartesian_point and polar_point inherited from a common class with a virtual distanceFrom method you could interchange point types pretty easily without having to make modifications to existing functions like printDistances. That little "wedge" between the call and what is actually executed makes OO code quite a bit more flexible than procedural code.

"A wedge is a place in software where you can substitute one component for another"

Why Is Flexible Material Important?

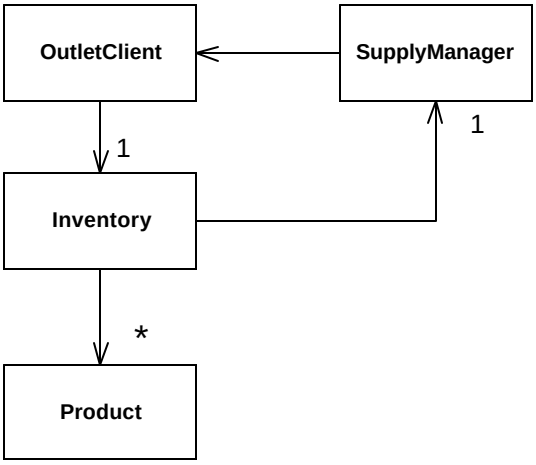
The flexibility you get with objects can seem like overkill in some development situations. After all, if you are just writing a little script to do some text processing, you make calls to existing functions, and you don't care whether your calls are hard coded or not. However, when it comes to modifying code, little wedges make a big difference.

Sensing and Separating

Ideally, you shouldn't have to do anything special to the classes of a system to start working with it effectively. In an ideal system, you can add an in vivo harness and start to add tests for any new piece of work that you are doing. If you have a question about how something works, you create objects, write characterization tests for them, and then move along to other things. If it was this easy, there would be no need to really write about any of this, unfortunately, it is often hard. Dependencies among classes can make it very difficult to get particular clusters of objects under test. You want to create an object of one class and ask it questions, but to create it, you need objects of another class and those objects need objects of another class, and so on, and so on.. Eventually, you end up with nearly the whole system in a harness and one whopper of a setup method. In some languages, this isn't a very big deal. In others, most notably C++, the link time for your system alone can make rapid turnaround nearly impossible if you don't break dependencies to separate out parts of your system.

In systems that weren't developed using Test Driven Development, you often have to break dependencies to get classes into a test harness, but that isn't the only reason to break dependencies. Sometimes the class that you want to test has effects on other classes that you need to know about. Sometimes you can sense the effects through the interface of the other class. At other times, you can't so the only choice you have is to impersonate the other class so that you can sense the effects directly.

Let's take a look at an example that highlights these issues.



Here is a model for a simple point of sale system. In this system, OutletClient is a monolithic class, it instantiates a variety of objects from vendor supplied libraries and it

would be particularly difficult to pull into a test harness. One of the things that an outlet client does is update the inventory when a sale is made. The inventory notifies the supply manager whenever it has changed and if the supply manager finds that it needs to send more products out to an outlet, it sends notification to the outlet about a pending shipment of products.

What we'd like to do is make some changes to SupplyManager, Inventory, and some of the other classes they collaborate with, but it would be pretty painful to bring OutletClient into a testing harness. Unfortunately, OutletClient is really a focal point in this design. How can we write characterization tests for SupplyManager? The tough part is that OutletClient is the only class which knows about some of the actions that SupplyManager takes. If we can create an inventory independently of the outlet client, we can poke through its interface and exercise the supply manager independently. On the other hand, we may just be interested in characterizing behavior that can be triggered through the public methods of outlet client. In this app, outlet client is the only client of inventory. Since that is the case, we can be sure that Inventory will not be used in any way other than how it is used as side effect of calls to outlet client. That cuts down the scope of the behavior that has to be characterized.

We have both sensing and separation issues. We want to separate out the portions of outlet client which depend upon things that would be problematic to put into a harness, but we also want to know things that only outlet client would know.

In the next couple of sections, I'll be going over several patterns that can be used to break dependencies for sensing or for separation. But first let's talk about various types of dependencies and a couple of features of implementation inheritance that can help us do our work.

Sensing Change

Internal and External Dependencies

The techniques that you use to break dependencies in OO software really depend upon the types of dependency you have and what your goals are.

By far, the easiest dependencies to deal with are what I call external dependencies, dependencies on things that pass into a class from the outside world. In short, if an object came into your class as a method or constructor parameter, it is an external dependency. External dependencies are pretty easy to deal with because you have control over what you pass to a class. You can pass in the original collaborator, but if you need to sense or separate, you can pass in an object which acts like a collaborator, but has fewer dependencies, or records calls that are made to it. Being able to decide what kind of object to pass in is a key advantage.

Internal dependencies, on the other hand, are much harder to deal with. An internal dependency is the opposite of an external dependency. It is a dependency on something that did not pass into the object from outside world. If you create new objects in a class, that class has an internal dependency on the class it uses to create those objects. If your class calls a singleton or some global variable, that is an internal dependency as well. If your are using a language like C++ which allows free functions, functions that are not part of any class, calls to those functions are internal dependencies. Breaking internal dependencies often requires a bit of tricky work.

Inheritance as a Tool

Implementation inheritance is one of the most contentious features in object oriented development. Novices often don't know how to use it correctly so they avoid it entirely. Many experienced people have used it too often in the past and been burned so severely that they avoid it whenever they can. My own view is that implementation inheritance is a powerful tool, but I don't go out of my way to use it. Most often implementation inheritance shows up in my work when I notice duplication across several classes that could be eliminated if I created a superclass for them and moved the duplicated methods into that superclass. When it comes to legacy code, however, I use implementation inheritance very aggressively because it is a good way of being able to sense and separate.

Let's take a look at a method in a little application:

```
class MessageForwarder
{
   private Message createForwardMessage(Session session, Message message)
                    throws MessagingException, IOException {
        MimeMessage forward = new MimeMessage (session);
        forward.setFrom (getFromAddress (message));
        forward.setReplyTo (new Address [] { new InternetAddress (listAddress) });
        forward.addRecipients (Message.RecipientType.TO, listAddress);
        forward.addRecipients (Message.RecipientType.BCC, getMailListAddresses ());
        forward.setSubject (transformedSubject (message.getSubject ()));
        forward.setSentDate (message.getSentDate ());
        forward.addHeader (LOOP_HEADER, listAddress);
        buildForwardContent(message, forward);
        return forward;
    }
    …
}
```

MessageForwarder has a quite a few methods that aren't shown here. One of the public methods calls this method, createForwardMessage, to build up a new message. If we wanted to separate out the dependency on the MimeMessage class, we have a couple of choices. We can modify createForwardMessage so that it accepts an additional parameter, a message it will fill up and then return:


```
class MessageForwarder:
{
   private Message createForwardMessage(
                                        Session session,
                                        Message message,
                                        Message forward)
                    throws MessagingException, IOException {
       forward.setFrom (getFromAddress (message));
       forward.setReplyTo (new Address [] { new InternetAddress (listAddress) });
       forward.addRecipients (Message.RecipientType.TO, listAddress);
       forward.addRecipients (Message.RecipientType.BCC, getMailListAddresses ());
       forward.setSubject (transformedSubject (message.getSubject ()));
       forward.setSentDate (message.getSentDate ());
       forward.addHeader (LOOP_HEADER, listAddress);
       buildForwardContent(message, forward);
       return forward;
   }
   …
}
```

The only problem is that the method buildForwardContent expects forward to be a MimeMessage, so we haven't really separated out MimeMessage. However, if we really want to avoid a dependency on MimeMessage what we could do is make createForwardMessage protected and override it in a new subclass that we make just for testing:

```
class TestMessageForwarder extends MessageForwarder
{
   protected Message createForwardMessage(Session session, Message message) {
       ....
       return forward;
   }
   …
}
```

In this new subclass we can do whatever we need to do to get the separation or the sensing that we need. In production code, we instantiate MessageForwarders, in tests we instantiate TestMessageForwarders. We were able to get separation with minimal modification of the production code. All we did was change the scope of a method from private to protected.

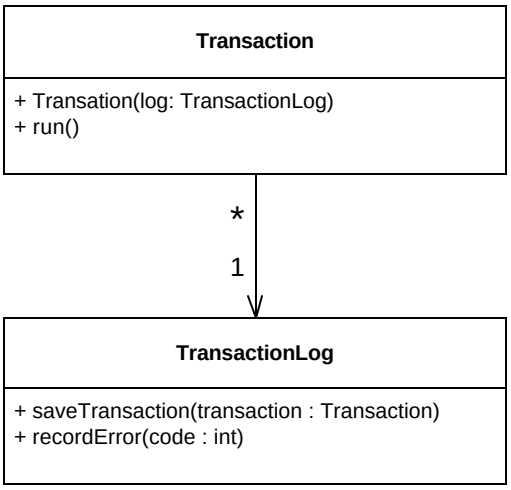
In general, the factoring that you have in a class determines how well you can use inheritance to separate out dependencies. Sometimes you have dependency you want to get rid of isolated in a small method. At other times, you have to override a larger method in order to separate out a dependency. Subclassing to test is a powerful

technique, but you have to be careful. In the example above, I can return a empty message without a subject, from address, etc but that would only make sense if I was, say, testing the fact that I can get a message from one place in the software to another, but don't care what the actual content and addressing are.

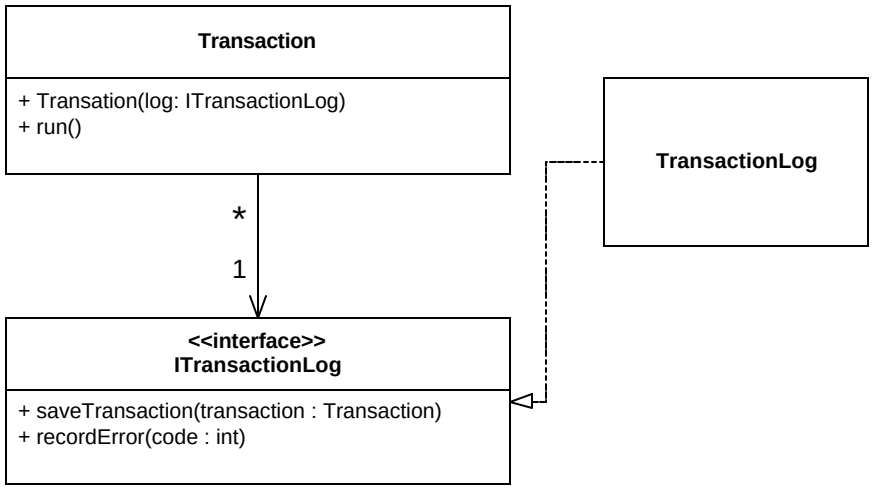
Several of the patterns that I describe in the following sections will use implementation inheritance to achieve sensing or separation. As a short-hand, I'll call this technique, Subclass to Test.

Breaking External Dependencies

External dependencies are very easy to break. Let's take a look at an example.



In this model we have two classes, Transaction and TransactionLog. Whenever a transaction is created, we give it a transaction log to write to. The fact that we are the ones who are creating the log and passing it to the transaction gives us a great deal of control. The easiest thing to do is to apply one of the safest refactorings in Martin Fowler's book: Extract Interface. Once we've done that, we can create objects which appear to be transaction logs and pass them to the transactions.



In Java or C++, extracting interfaces is very simple. You create an empty interface make the target class implement it. Then you change the using class to that it accepts objects of the interface type not the target class. Once you have done that, attempt to compile the system. Each time the compiler tells you that you are missing a method on the interface add it to the interface. If the object is held by other references in the using class, you may have to change them to the interface type also.

Extract Interface is a powerful way of getting sensing or separation. You can use it the same way regardless of whether the dependency you want to break comes through as a constructor argument or an ordinary method argument.

Breaking Internal Dependencies

What are internal dependencies and what would the world look like if we didn't have them? Internal dependencies are dependencies on anything that does not pass through the interface of a class. In this category, we have global variables, singletons, global constants, free functions and even the classes of objects we create in our class.

In general encapsulation is a great idea, but at times it gets in the way of parameterization. You have a class which behaves nicely but there is something that it does that you want to alter. You discover that what you want to alter is actually behavior handled by a collaborator of your class, but you can't find any way to change it without altering the source because the plug point, the place where you would create and use the other object, is hidden away in your class. Fortunately, getting rid of these dependencies is usually pretty easy. Depending upon your programming language, you can often do it non-invasively.

Here is a list of the different kinds of internal dependencies you are liable to run into:

- Global Variables/Singletons
- Free Functions
- Object Creation in Constructors
- Object Creation in Methods

The following sections contain patterns that can be used to break these sorts of dependencies.

Global References

Introduce Static Setter

Maybe I am a purist, but I don't like global mutable data at all. When I visit clients, it is usually the most apparent hurdle to getting portions of their system out to test. You want to pull out a set of classes into an in vitro harness, but you discover that some of them need to be set up in particular states to be used at all. When you have your harness set up you have to run down the list of globals to make sure that each one has the state you need for the condition you want to test. Quantum physicists didn't discover "spooky action at a distance", in software, we've had it for years.

All griping about globals aside, many systems have them. In some systems, they are very direct and unself-conscious; someone just declared a variable someplace. In others they are dressed up as singletons with strict adherence to the singleton pattern. In any case, getting a mock in place for sensing is very straightforward. If the variable is an unabashed global, sitting outside a class or plainly out in the open as a public static variable, you can just replace the object. If the reference is const or final, you may have to remove that protection. Leave a comment in the code saying that you are doing it for test and that people shouldn't take advantage of the access in production code.

Replacing singletons is just a little more work. Add a static setter to the singleton to replace the instance, and then make the constructor protected. You can then subclass the singleton, create a fresh object and pass it to the setter.

You may be left a little queasy by the idea that you are removing access protection when you use static setter, but remember that the purpose of access protection is to prevent errors. We are putting in tests to prevent errors also. It just turns out that in this case we need the stronger tool.

Here is an example of Introduce Static Setter in C++:

```
void MessageRouter::route(Message *message) {
        …
        Dispatcher *dispatcher = ExternalRouter::instance()->getDispatcher();
        if (dispatcher != NULL)
                dispatcher->sendMessage(message);

}
```

In the MessageRouter class we use a singleton in a couple of places to get a dispatcher. The ExternalRouter class is a typical singleton. It uses a static method named "instance" to provide access to the one and only instance of ExternalRouter. The ExternalRouter class has a getter for a dispatcher. We can replace the dispatcher with another one by replacing the external router that serves it.

This is what the ExternalRouter class looks like before we introduce the static setter:

```
class ExternalRouter
{
private:
   static ExternalRouter *_instance;

public:
   static ExternalRouter *instance();

   … // other methods
};

ExternalRouter *ExternalRouter::_instance = 0;

ExternalRouter *ExternalRouter::instance()
{
    if (_instance == 0)
       _instance = new ExternalRouter;
    return _instance;
}
```

Notice that the router is created on the first call to the instance method. To substitute in another router we have to change what instance returns. The first step is to introduce a new method to replace the instance.

```
void ExternalRouter::setInstance(ExternalRouter *newInstance)
{
    delete _instance;
    _instance = newInstance;
}
```

Of course, this assumes that we are able to create a new instance. When people use the singleton pattern they often make the constructor of the class private to prevent people from creating more than one instance. If you make the constructor protected, you can subclass the singleton to sense or separate and pass the new instance to the setInstance method. In the example above, we'd make a subclass of ExternalRouter named TestingExternalRouter and override the getDispatcher method so that it returns the dispatcher we want, possibly a null dispatcher.

```
class TestingExternalRouter : public ExternalRouter
{
public:
   virtual void Dispatcher *getDispatcher() const {
      return new NullDispatcher;
   }
};
```


This may look like a rather roundabout way of substituting in a new dispatcher. We end up creating a new ExternalRouter just to substitute dispatchers. There are some shortcuts we can take, but they have different tradeoffs. Another thing that we can do is add a Boolean flag to ExternalRouter and let it return a different dispatcher when the flag is set. In C++ or C# we can use conditional compilation to select dispatchers also. These techniques can work well, but they are invasive and they can get unwieldy if you use them throughout an application. In general, I like to keep separation between production and test code.

Using a setter method and a protected constructor on a singleton is mildly invasive, but it does help you get tests in place. Could people misuse the public constructor and make more than one singleton in the production system? Yes, but in my opinion, if it is important to only have one instance of an object in a system the best way to handle it is to make sure everyone on the team understands that constraint. Creating multiple instances of something you should only have one of is more of a team communication problem not a design problem, and it rarely comes up. Many users of the singleton pattern are usually more interested in creating global variables than they are in controlling the number of instances.

In the example above, we replaced a singleton with a static setter. The singleton was an object that served up another object, a dispatcher. Occasionally, you see global factories in systems. Rather than holding onto an instance they serve up fresh objects every time you call one of their static methods. Substituting in another object to return is kind of tricky, but often you can do it by having the factory delegate to another factory. Let's take a look at an example:

```
class WidgetFactory
{
        static Widget makeFlyingWidget() {
           return new AirplaneWidget();
        }
}
```

WidgetFactory is a straightforward global factory. As it stands, it doesn't allow us to replace the widgets it serves under test, but we can alter it so that it can.

```
interface IWidgetServer
{
        Widget makeWidget();
}

class WidgetFactory
{
        static Widget makeFlyingWidget() {
           return server.makeWidget();
```



```
        }
        static setServer(IWidgetServer aServer) {
                server = aServer;
        }
        static IWidgetServer server = new IWidgetServer() {
                public Widget makeWidget() {
                        return new AirplaneWidget();
                }
        };
}
```

In a test, we can do this:

```
protected void setUp() {
        WidgetFactory.setServer(new IWidgetServer() {
                public Widget makeWidget() {
                        return new NullWidget();
                }
        });
}
```

But, it is important to remember in any of these static setter patterns, you are modifying state that is available to all tests. You can use the tearDown method in xUnit to put things back into some known state before the execution of the rest of your tests. In general, I do that only in cases where using the wrong state in the next test could be misleading. If I am substituting in a mock MailSender in all of my tests, putting in another doesn't make much sense. On the other hand, if I have global that keeps state that affects the results of the system, often I do the same thing in the setUp and tearDown methods to make sure that I've left things in a clean state:

```
protected void setUp() {
        Node::count = 0;
        …
}

protected void tearDown() {
        Node::count = 0;
}
```

At this point, I'm imagining you with my mind's eye. You are sitting there disgusted at the carnage that I am wreaking on the system just to be able to get some tests in place. And you are right; these patterns can uglify a system considerably. Surgery is never pretty, particularly at the beginning. What can you do to get the system back on the straight and narrow?

One thing to consider is parameter passing. Take a look at the classes which need access to your global and consider whether you can give them a common superclass. If you can,

you can pass the global to them on creation, and slowly move away from having globals at all. Often people are scared that every class in the system will require some global or another. Often you'll be surprised. I once worked on an embedded system which encapsulated memory management as a class, passing it to whoever needed it. Over time there was a clean separation between classes which needed the class and classes which didn't.

Link-time Polymorphism

OO gives us wonderful opportunities to substitute one object for another. If two classes implement the same interface or have the same superclass, you can substitute one for another pretty easily. Unfortunately, people working in procedural languages like C don't have that option. If you have a function like:

```
void account_deposit(int amount);
```

there is no way to substitute one function for another at compile time short of using the preprocessor. People who use the preprocessor to make macros that look exactly like function calls are often summarily executed on teams.

Are there other alternatives? Yes, you can use link-time polymorphism to replace one function with another. To do this, create a dummy library that has functions with the same signatures as the functions you want to mock out. If you are sensing, you'll need to set up some mechanism for saving notifications and querying them. You can use files, global variables, anything that would be convenient under test.

Here is an example:

```
void account_deposit(Currency *amount)
{
        struct Activity *activity = (struct Activity *)malloc(sizeof (struct Activity));

        activity->type = ACT_DEPOSIT;
        activity->amount = amount;
        append(g_accountActivities,activity);
}
```

You can do link-time polymorphism with classes but it is really more trouble than it is worth. In C++ systems that make calls to external C libraries, however, it can be very useful. The most useful cases are libraries that are pure data sinks. Things like graphics libraries are particularly useful to mock out with link-time polymorphism. You can make your link times shorter and not have to watch your display flicker wildly under test.

<need a section on mock objects>

Encapsulate References

When you are working with references to globals, you essentially have three choices. You can try to make the global act differently under test, you can link to another global, or you can put a wrapper on the global so that you can decouple things further. The last option is called Encapsulate References. Often it is a very good choice. Here is an example:

```
bool AGG230_activeframe[AGG230_SIZE];
bool AGG230_suspendedframe[AGG230_SIZE];

void AGGController::suspend_frame()
{
        frame_copy(AGG230_suspendedframe,
                AGG230_activeframe);
        clear(AGG230_activeframe);
        flush_frame_buffer();
}
```

In this example, we have some code that does some work with a few global arrays. The suspend_frame method needs to access the frames. At first glance it looks like we can take the frames and make them members of the AGGController class, but there are some other classes (not shown above) that use the frames. What can we do?

One immediate thought is that we can pass them as parameters to the suspend_frame method using Parameterize Method, but once we do that we'd have to parameterize the methods we're calling here: frame_copy and clear. Worse, if we take a look at what happens in flush_frame_buffer, we see that AGG230_activeframe is used directly in that method.

We could pass both the active and suspended frames into the AGGController class, but when we look around the code it seems that whenever we use one, we are using the other.

The best way to handle this situation is to look at the data that we need to use in the class, the active and suspended frame, and think of whether we can come up with a good name for a smart class that would hold them. Sometimes this is a little tricky. You have to think forward and consider what sort of methods you'll eventually have on that class. Chances are, they may be methods or functions that already exist but use the data. In other cases, you will eventually extract new methods from methods that use the data.

In the example above, we'd expect that over time, the *frame_copy* and *clear* methods might move to the new class that we are going to create. We could just make a Frame class and move on, but then we'd have two instances of frame that we'd pass into the AGGController. Is there work that is common to the suspended and the active frame? It looks like there is in this case. The suspend_frame function on AGGController could

probably move to a new class that had both the suspended_frame and the active_frame. What could we call that new class? We could just call it Frame and say that each frame has an active buffer and a suspended buffer. This requires us to change our understanding of the system a bit, but what we will get in exchange is a smarter class that hides more detail.

Here's how we can do it, step by step.

First, we create a class that looks like this:

```
class Frame
{
public:
        const int AGG230_SIZE;
        bool AGG230_activeframe[AGG230_SIZE];
        bool AGG230_suspendedframe[AGG230_SIZE];
};
```

We've left the names of the data the same intentionally just to make the next step easier. Next, we declare a global instance of the Frame class:

```
Frame  frameForAGG230;
```

Next we comment out the original declarations of the data and attempt to build:

```
// bool AGG230_activeframe[AGG230_SIZE];
// bool AGG230_suspendedframe[AGG230_SIZE];
```

At this point, we'll get all sorts of compile errors telling us that AGG_activeframe and AGG230_suspendedframe don't exist, threatening us with terrible consequences, and if the build system is sufficiently petulant it will round things off with an attempt at linking leaving us with about ten pages of unresolved link errors. We could get upset, but we expected all of that to happen, didn't we?

To get past all of those errors, we can stop at each one and place "frameForAGG230." in front of the reference that is causing trouble.

```
void AGGController::suspend_frame()
{
        frame_copy(frameForAGG230.AGG230_suspendedframe,
                frameForAGG230.AGG230_activeframe);
        clear(frameForAGG20.AGG230_activeframe);
        flush_frame_buffer();
}
```

When we are done doing that, we will have ugly code, but it will all compile and work correctly, so it is a complete refactoring. Now that we have it done, we can pass a Frame

object through the constructor of the AGGController class and get the separation we need to move forward.

Why did we do it this way? After all, we spent some time thinking about what to call the new class and what sorts of methods to place on it. We could have started marching along, creating a mock Frame object that we could delegate to in AGGController and other classes and move all of the logic for those methods onto a real Frame class. We could do that but it is a lot to attempt all at once. Worse, when you don't have tests in place and you are trying to do the minimal work you need to get tests in place, it is best to leave logic alone as much as possible. Try not to move it, try to get separation by putting in wedges that allow you to call one method instead of another or access one piece of data rather than another. Later, when you have more tests in place, you can move behavior from one class to another with impunity.

Once we've passed the frame into the AGGController, we can do a little renaming to make things a little clearer. Here is our ending state for this refactoring:

```
class Frame
{
public:
        const int BUFFER_SIZE;
        bool activebuffer[BUFFER_SIZE];
        bool suspendedbuffer[BUFFER_SIZE];
};

Frame  frameForAGG230;

void AGGController::suspend_frame()
{
        frame_copy(frame.suspendedbuffer,
                frame.activebuffer);
        clear(frame.activeframe);
        flush_frame_buffer();
}
```

It may not seem like much of an improvement, but it is an extremely valuable first step. Once you've moved the data to a class, you have separation and you are poised to make the code much better over time.

In the example above, I showed how to do Encapsulate References with global data. You can do the same thing with free functions in C++ programs. Often when you are working with some C API you will have calls to global functions scattered throughout an area of code that you want to work with. The wedge that you have is linking. You can use Link-time Polymorphism to get separation, but you can end up with better structured code if you using Encapsulate References to build another wedge. Here is an example:

In a piece of code that we want to put under test, there are calls to two functions: Option GetOption(string optionName) and void setOption(string name, Option option). They are just free functions, not attached to any class, but they are used prolifically in code like this:

```
void ColumnModel::update()
{
        alignRows();
        Option resizeWidth = GetOpen("ResizeWidth");
        If (resizeWidth.isTrue())
                resize();
        else
                resizeToDefault();
}
```

In a case like this, we could look at some of the old standbys, techniques like Parameterize Method and Introduce Deferred Getter but if the calls are across multiple methods and multiple classes, it would be cleaner to use Encapsulate References. To do this, create a new class like this:

```
class  OptionSource
{
public:
        virtual ~OptionSource() = 0;
        virtual Option GetOption(string optionName) = 0;
        virtual void SetOption(string optionName, Option newOption) = 0;
};
```

and make abstract methods for each of the free functions that you need in an area of code. Next, subclass to make a mock for the class. In this case, we could have a map in the mock which allows us to hold onto a set of options will be used during tests. We could provide an add method to the mock, or just a constructor that accepts a map, whatever is convenient for the tests. When we have the mock, we can Create the real option source:

```
class  ProductionOptionSource
{
public:
        Option GetOption(string optionName);
        void SetOption(string optionName, Option newOption) ;
};

Option ProductionOptionSource::GetOption(string optionName)
{
        ::GetOption(optionName);
}

void ProductionOptionSource::SetOption(string optionName, Option newOption)
{
        ::SetOption(optionName, newOption);
```



```
}
```

This refactoring turned out very well. When we introduced the wedge, we ended up just doing a simple delegation to the API function. Now that we've done that, we can parameterize the class to accept an OptionSource object so that we can use a mock one under test and the real one in production.

<need lessons learned soundbites throughout the text, and a hitlist>

Object Creation in Constructor

Parameterize Constructor

If you are creating an object in a constructor, the easiest way to plug in another object is to just accept it as a constructor parameter. Here is an example.

We start with this:

```
public class MailChecker
{
   public MailChecker (int checkPeriodSeconds) {
      this.receiver = new MailReceiver();
      this.checkPeriodSeconds = checkPeriodSeconds;
   }
   ...
}
```

And introduce a new parameter like this:

```
public class MailChecker
{
   public MailChecker (MailReceiver reciever, int checkPeriodSeconds) {
      this.receiver = receiver;
      this.checkPeriodSeconds = checkPeriodSeconds;
   }
   ...
}
```

One reason why people don't often think of this refactoring is because they assume that it has to force all clients to pass an additional argument. However, you can write a constructor which keeps the original signature around:

```
public class MailChecker
{
   public MailChecker (MailReceiver reciever, int checkPeriodSeconds) {
      this.receiver = receiver;
      this.checkPeriodSeconds = checkPeriodSeconds;
   }

   public MailChecker (int checkPeriodSeconds) {
         this(new MailReceiver(), checkPeriodSeconds);
   }

}
```


Parameterize Constructor is great for substituting sensing objects, but poor for separation because the class keeps a dependency on the original class. To get separation with this refactoring, you can use Extract Interface on the class of the argument.

When to use it

Use Parameterize Constructor when you want to substitute a sensing object and the object you want to replace is created in a simple way in the constructor.

Alternatives

In cases where the internal object you want to replace has multiple arguments Parameterize Constructor can be unwieldy. Sometimes it forces the test fixture class to do setup that duplicates work in the original constructor. In those cases, Introduce Deferred Factory Method is often a better choice.

Introduce Deferred Factory Method

In the section on inheritance, we talked about Subclass to Test. It turns out that subclassing can often be a powerful way of moving towards sensing or separation during testing.

Let's look at an example:

```
public class WorkflowEngine
{
   public WorkflowEngine () {
         Reader reader = new ModelReader(AppConfig.getDryConfiguration());
         Persister persister = new XMLStore(AppConfiguration.getDryConfiguration());
         this.tm = new TransactionManager(reader,persister);
         ...
   }
   ...
}
```

WorkflowEngine creates a TransactionManager in its constructor. If the creation was someplace else, we could introduce some separation. One of the options we have is to introduce a factory method in the class:

```
public class WorkflowEngine
{
   public WorkflowEngine () {
         this.tm = makeTransactionManager();
         ...
   }

   protected TransactionManager makeTransactionManager() {
         Reader reader = new ModelReader(AppConfiguration.getDryConfiguration());
         Persister persister = new XMLStore(AppConfiguration.getDryConfiguration());
         return new TransactionManager(reader,persister);
   }
   ...
}
```

When we have that factory method, we can Subclass to Test to plug in another object:


```
public class TestWorkflowEngine extends WorkflowEngine
{
   protected TransactionManager makeTransactionManager() {
         return new NullTransactionManager();
   }
}
```

Introduce Deferred Factory Method is a great way to introduce separation from classes with complicated setup. If, in testing, we don't care about the results of any interactions with the object we are separating out, it is an ideal choice. If we are after sensing rather than separation, Introduce Deferred Getter is a better choice.

When to use it

Use Introduce Deferred Factory Method when you want to introduce separation and Parameterize Constructor would force too much setup work on your test fixture or duplicate code already in one of the constructors of the class.

You can also use it in conjunction with Extract Interface when you want strong separation. To do this, extract an interface for the class you want to separate out and use it as the return type of the factory method. Make the factory method abstract and introduce two subclasses: one which overrides the factory method for testing and one which overrides the factory method to produce the object you need in production code. The latter class will be the class that you need to instantiate in your production code.

Alternatives

Introduce Deferred Factory Method can be overkill, and it only works in languages which allow calls in constructors to resolve to methods in subclasses. In C++, for instance, if you call a virtual method in a constructor it will resolve to the method definition in the class, not any of the overrides in derived classes. In C++, Supercede Instance Variable and Introduce Deferred Getter are often better alternatives.

Introduce Deferred Factory Method is best when introducing separation. When you want to sense, you need a pattern which allows you to hold onto the sensing object. Introduce Deferred Getter is often better for this.

Introduce Deferred Getter

If you are creating an object in a constructor and you want to introduce another object for sensing, you have a couple of options. You can use Parameterize Constructor, but that has the nasty effect of complicating the class you are working on. In addition, it isn't very useful when the creation of the object in the constructor is very complicated. When you need to sense, Introduce Deferred Getter is a more powerful alternative.

The gist of this refactoring is to introduce a getter for the instance variable you'd like to replace with a sensing object. Then refactor to use the getter every place in the class. You can then Subclass to Test and override the getter with a new getter in the subclass.

In this example, we create a transaction manager in a constructor. We'd like to set things up so that the class can use this transaction manager in production and a sensing one under test.

```
public class WorkflowEngine
{
   public WorkflowEngine () {
         Reader reader = new ModelReader(AppConfig.getDryConfiguration());
         Persister persister = new XMLStore(AppConfiguration.getDryConfiguration());
         this.tm = new TransactionManager(reader,persister);
         ...
   }

   private Transaction getTransaction() {
         return tm.getNextTransaction();
   }
   ...

}
```

The first thing we do is introduce a lazy getter, which creates the transaction manager on first call. Then we replace all uses of the variable with calls to the getter:

```
public class WorkflowEngine
{
   public WorkflowEngine () {
         Reader reader = new ModelReader(AppConfig.getDryConfiguration());
         Persister persister = new XMLStore(AppConfiguration.getDryConfiguration());
         this.tm = new TransactionManager(reader,persister);
         ...
   }
```



```
    protected TransactionManager getTransactionManager() {
        if (tm == null) {
                Reader reader = new ModelReader(AppConfig.getDryConfiguration());
                Persister persister
                        = new XMLStore(AppConfiguration.getDryConfiguration());
                tm = new TransactionManager(reader,persister);
        }
        return tm;

    }

    private Transaction getTransaction() {
        return getTransactionManager().getNextTransaction();
    }
    ...

}
```

When we have that getter, we can Subclass to Test to plug in another object:

```
public class TestWorkflowEngine extends WorkflowEngine
{
    public TestWorkflowEngine() {
       transactionManager = new MockTransactionManager();
    }

    protected TransactionManager getTransactionManager() {
        return transactionManager;
    }

    MockTransactionManager transactionManager;
}
```

In a test, we can easily access the mock transaction manager if we need to:

```
public class WorkflowEngineTest extends TestCase
{
        public void testRunSimple() {
                TestWorkflowEngine  engine = new TestWorkflowEngine();
                engine.run();
                assertEquals(0,engine.transactionManager.getTransactionCount());
        }
        ...
}
```


Even though we have a getter for transaction manager, I have no problem with leaving variables public on test classes and using them directly. Ease of use trumps object niceties in non-production code.

When to use it

Use Introduce Deferred Getter when you need to replace an object for sensing or when you want shallow separation in languages like C++ which disallow virtual calls in constructors.

Alternatives

Introduce Deferred Getter is a "big gun." Consider Parameterize Constructor whenever possible.

Supercede Instance Variable

One of the big downsides to the other *Object Creation in Constructor* patterns is that they rely on the ability to override calls to methods in constructors. In Java, you can override a method called in a constructor and everything works fine. However, in C++, the call will be to the method you define in the class that has the constructor, not the overridden method in the subclass.

If you are working in C++, you can use another technique to get a sensing object in place. This technique is particularly useful when you have non-trivial creation.

```
BlendingPen::BlendingPen()
{
   setName("BlendingPen");

   m_param = ParameterFactory::createParameter("cm", "Fade", "Aspect Alter");
   m_param->addChoice("blend");
   m_param->addChoice("add");
   m_param->addChoice("filter");

   setParamByName("cm", "blend");
}
```

In this case, a constructor is creating a parameter through a factory. We could use Introduce Static Setter get some control over the next object the factory returns, but that is pretty invasive. If we don't mind adding an extra method to the class we can supercede the parameter we created in the constructor:

```
void BlendingPen::supercedeParameter(Parameter *newParameter)
{
   delete m_param;
   m_param = newParameter;
}
```

In tests, we can create pens as we need them, and call supercedeParameter when we need to put in a sensing object.

On the surface, Supercede Instance Variable looks like a poor way of getting a sensing object in place, but in C++, when Parameterize Constructor is too awkward because of tangled logic in the constructor, Supercede Instance Variable can be the best choice. In languages which allow virtual calls in constructors, a deferred factory method or a deferred getter is a better choice.

There is one nice thing about using the word "supercede" as the method prefix, it is kind of fancy and uncommon. If you ever get concerned about whether people are using the superceding methods in production code, you can do a quick search to make sure they aren't.

When to use it

Use Supercede Instance Variable when you want to sense or do shallow separation, but creation logic is too tangled for Parameterize Constructor.

Alternatives

One downside to Supercede Instance Variable is the fact that people can call it in production code to replace objects on whim. If you want a cleaner way of replacing objects, consider Introduce Deferred Factory Method or Introduce Deferred Getter.

Object Creation in Method

In a previous section I showed you a set of refactorings you can use to when objects created in a constructor get in the way of sensing and separation. In some ways, breaking dependencies on these objects is easier because whatever you do is going to affect the state of the object from cradle to grave. When you create an object in a method, the refactorings that use Subclass to Test tend to be pretty messy because you have to deal with object lifetime issues. You can use Introduce Deferred Getter or Introduce Deferred Factory Method in simple cases, but the former essentially makes a local variable look like part of an object's state. The latter works well, but to keep your code clean it is good to start to use the factory every place in the class. By far, the best alternative when trying to separate or sense is to use Parameterize Method.

Parameterize Method

You have a method that creates an object internally and you'd like to replace the object to separate or sense. The easiest way to do this is to pass the object from the outside. Here is an example.

```
void TestCase::run() {
   delete m_result;
   m_result = new TestResult;
   try {
      setUp();
      runTest(m_result);
   }
   catch (Exception& e) {
      result->addFailure(e,this);
   }
   teardown();
}
```

Here we have a method that creates a result object whenever it is called. If we want to sense or separate, we can add the created object to the constructor.

```
void TestCase::run(TestResult *result) {
   delete m_result;
   m_result = result;
   try {
      setUp();
      runTest(m_result);
   }
   catch (Exception& e) {
      result->addFailure(e,this);
   }
   teardown();
}
```

Then we can write a little forwarding method that keeps the original signature intact:

```
void TestCase::run() {
   run(new TestResult);
}
```


When to use it

Alternatives

Introduce Deferred Factory Method

Wait a minute! I already discussed this one under Object *Creation in Constructor*. Essentially, it is the same refactoring. The only caveat is that I like to use the factory uniformly throughout the class once it is introduced.

When to use it

Alternatives

