Working Effectively with Legacy Code

Notes By: Rachel Burke

Part I:
The Mechanics of Change

Chapter 1

Changing Software

- Four Reasons to Change Software
- Risky Change

Four Reasons to Change Software

For simplicity's sake, let's look at four primary reasons to change software.

1. Adding a feature
2. Fixing a bug
3. Improving the design
4. Optimizing resource usage

Adding Features and Fixing Bugs

- Does the client want a new feature or a bug fix?
  - Usually depends on the point of view
- Bug fixes and features usually tracked separately
- Edit Code ~ Edit Behavior
- Add Code ~ Add Behavior
  - Sometimes adding code does nothing at all

Behavior is the most important thing about software. It is what users depend on. Users like it when we add behavior (provided it is what they really wanted), but if we change or remove behavior they depend on (introduce bugs), they stop trusting us.

Improving Design

- Different kind of software change
- Goal of keeping behavior intact while modifying structure
- ***Refactoring*** - the act of improving design without changing its behavior
- Write tests to ensure the code does not change
- Differs from general cleanup because we aren't just doing low-risk things such as reformatting source code, or invasive and risky things such as rewriting chunks of it
- Refactoring is a series of small structural modifications, supported by tests to make the code easier to change.
- No functional changes when you refactor

Optimization

- “We’re going to keep functionality exactly the same when we make changes, but we are going to change something else”
- Refactoring = program structure
- Optimization = improve resources
  - Time
  - Memory

Putting It All Together

	Adding a Feature	Fixing a Bug	Refactoring	Optimizing
Structure	Changes	Changes	Changes	
New Functionality	Changes			
Functionality		Changes		
Resource Usage				Changes

Preserving existing behavior is one of the largest challenges in software development. Even when we are changing primary features, we often have very large areas of behavior that we have to preserve.

Risky Change

To mitigate risk, we have to ask three questions:

1. What changes do we have to make?

2. How will we know that we've done them correctly?

3. How will we know that we haven't broken anything?

You do not minimize problems by avoiding them.

Avoiding change causes fear!

Chapter 2

Working with Feedback

- What is Unit Testing?
- Higher-Level Testing
- Test Coverings
- The Legacy Code Change Algorithm

Working With Feedback

- Edit and Pray → Industry Standard of working with care
- Cover and Modify → Work with a safety net when changing software
  - Using testing in an attempt to show correctness
  - Acts like a vise around code that detects if major behavioral changes are made
- Regression Testing - application-level style testing
- Unit Testing - an important components in legacy code work

When we have tests that detect change, it is like having a vise around our code. The behavior of the code is fixed in place. When we make changes, we can know that we are changing only one piece of behavior at a time. In short, we're in control of our work.

What Is Unit Testing?

- Unit Testing - testing in isolation...
  - Run Fast - 1/10th of a second per test is too slow
  - Localize problems
- ***Error localization*** -As tests get further from what they test, it is harder to
- determine what a test failure means
- ***Execution Time*** - Larger tests take longer to execute
- ***Coverage*** - Observing the connection between a piece of code and the values that exercise it
- A test is not a unit test if it talks to a database, communicates across a network, touches a filesystem, or does special things to the environment to run it.

Higher-Level Testing

- Higher-Level Tests - tests that cover scenarios and interactions in an application
- Higher-level tests can be used to pin down behavior for a set of classes at a time
- Helps enable the writing of tests for individual classes more easily

Test Coverings

- Dependency is one of the most critical problems in software development. Much legacy code work involves breaking dependencies so that change can be easier.
- The Legacy Code Dilemma
  - When we change code, we should have tests in place. To put tests in place, we often have to change code.
- When you break dependencies in legacy code, you often have to suspend your sense of aesthetics a bit. Some dependencies break cleanly; others end up looking less than ideal from a design point of view.

The Legacy Code Change Algorithm

1. Identify change points.
2. Find test points.
3. Break dependencies.
4. Write tests.
5. Make changes and refactor.

Chapter 3

Sensing and Separation

- Faking Collaborators

Sensing and Separation

- Two reasons to break dependencies:
  - 1. ***Sensing*** - We break dependencies to sense when we can't access values our code computes.
  - ***Separation*** -We break dependencies to separate when we can't even get a piece of code into a test harness to run.
- There are many ways to separate software, but only one dominant technique for sensing.

Faking Collaborators: Fake Objects

- **Fake Object** - an object impersonates some collaborator of your classes being tested
- We can write tests against our code to see what the code does against the fake object.
- When we write tests for individual units, we end up with small, well-understood pieces. This can make it easier to reason about our code.
- When we write tests we have to divide and conquer.

The Two Sides of a Fake Object

- There exists two sides to a fake object
  - The code that the test cares about and sees
  - The code that only the object sees

Mock Objects

- An advanced version of a fake object
- Perform assertions internally
- Not available in all object frameworks

Chapter 4

The Seam Model

- A Huge Sheet of Text
- Seams
- Seam Types

A Huge Sheet of Text

- Correctness vs. Modularity and OOD
- A little change can change a whole document
- Reuse code
- Independent and Dependent software pieces

Seams

- A **seam** is a place where you can alter behavior in your program without editing in that place.
- ***Object Seam*** - writing tests for OOD code without any nasty side effects
- Biggest challenges of legacy code - breaking dependencies
- The seam view of software helps us see opportunities that exist in the code base.

Seam Types

- Preprocessing Seams
  - Preprocessors give more seams
  - We can verify more functions carry the right parameters
    - Header files give seams that we can use to replace text before it is compiles
- **Enabling Point** - a place where you can make the descison to use one behavior or another
- Link Seams
  - Linkers combine intermediate representations of the code and their calls to other files
  - Separation is a big reason to link a seam
- Object Seams
  - Most useful seams in OOP Languages
  - Not all method calls are seams

Chapter 5

Tools

- Automated Refactoring Tools
- Mock Objects
- Unit-Testing Harness
- General Test Harness

Automated Refactoring Tools

- **Refactoring** - a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its existing behavior
- Implement tests around code before refactoring to ensure you maintain the behavior of the code

Unit-Testing Harness

- xUnit
  - It lets programmers write tests in the language they are developing in.
  - All tests run in isolation.
  - Tests can be grouped into suites so that they can be run and rerun on demand.
- Ensure separateness of objects between methods
- ProgramsL JUnit, CppUnitLite, NUnit, etc.

General Test Harness

- Framework for Integrated Tests
  - FIT is a concise and elegant testing framework that was developed by Ward
  - Cunningham.
  - The idea behind FIT is simple and powerful.
  - If you can write documents about your system and embed tables within them that describe inputs and outputs for your system, and if those documents can be saved as HTML, the FIT framework can run them as tests.
  - Fosters connection between software and people
- Fitnesse
  - FIR hosted in a wiki
  - Supports hierarchical web pages that define fit tests

Part II:
Changing Software

Chapter 6

I Don’t Have Much Time and I Have to Change It

- Sprout Method
- Sprout Class
- Wrap Method
- Wrap Class
- Summary

I Don’t Have Much Time and I Have to Change It

- Is the time it takes to fix the code worth it?
  - Change Time
  - Test Time
  - Debug Time
  - Time for Users Saved
- Make changes to code and always have a test to cover the change
- Do not give in to the temptation to write hacky code

Code is your house, and you have to live in it.

Sprout Method

1. Identify where you need to make your code change.
2. If the change can be formulated as a single sequence of statements in one place in a method, write down a call for a new method that will do the work involved and then comment it out.
3. Determine what local variables you need from the source method, and make them arguments to the call.
4. Determine whether the sprouted method will need to return values to source method. If so, change the call so that its return value is assigned to a variable.
5. Develop the sprout method using test-driven development.
6. Remove the comment in the source method to enable the call.

Advantages and Disadvantages

- Key Advantage: You can move forward with your work with more confidence than you could if you were making invasive changes
- Key Disadvantage: Conceptual Complexity
  - You gut the abstractions and do the bulk of work in other classes

Wrap Method

- First Method
  1. Identify a method you need to change.
  2. If the change can be formulated as a single sequence of statements in one place, rename the method and then create a new method with the same name and signature as the old method.
  3. Place a call to the old method in the new method.
  4. Develop a method for the new feature, test first, and call it from the new method.

Wrap Method

- Second Method
  1. Identify a method you need to change.
  2. If the change can be formulated as a single sequence of statements in one place, develop a new method for it using test-driven development.
  3. Create another method that calls the new method and the old method.

Advantages and Disadvantages

- Advantages:
  - Good way of getting new, tested functionality into an application when we cannot easily write tests for the code
  - Does not increase the size of existing methods
  - Explicitly makes the new functionality independent of the existing functionality
- Disadvantages:
  - Leads to poor names
  - Confusing

Wrap Class

- Class-level companion to Wrap Method
- If we need to add behavior to a system we can add it to another class
- Decorator Pattern - allows you to build complex behaviors by composing objects at runtime
    - Create objects of a class that wraps around another class and pass them around
    - Create an abstract class that defines the set of operations you need to support
    - Create a subclass that inherits from the abstrat class and accepts adn instance of the class in its constructor and provides a body for each method

Wrap Class

1. Identify a method where you need to make a change.

2. If the change can be formulated as a single sequence of statements in one place, create a class that accepts the class you are going to wrap as a constructor argument.

3. Create a method on that class, using test-driven development that does the new work. Write another method that calls the new method and the old method on the wrapped class.

4. Instantiate the wrapper class in your code in the place where you need to enable the new behavior.

Wrap Class

Instances to use:

1. The behavior that I want to add is completely independent, and I don’t want to pollute the existing class with behavior that is low level or unrelated.

2. The class has grown so large that I really can’t stand to make it worse. In a case like this, I wrap just to put a stake in the ground and provide a roadmap for later changes.

Chapter 7

It Takes Forever to Make a Change

- Understanding
- Lag Time
- Breaking Dependencies
- Summary

Understanding

- Key Difference Between Well-Maintained and Legacy Systems:
  - Well-Maintained System - Time to figure out change is hard, but making change is easy
  - Legacy System - Time to figure out change is hard and making change is hard

Lag Time

- **Lag Time** - the amount of time that passes between a change that you make and the moment that you get real feedback about that change
- Make Changes, Start Build, Find Out Later
- You should be able to compile every class or module in your system separately from the others and in its own test harness

Breaking Dependencies

- We can break dependencies!
- Easy Cases of Instantiating Classes for Tests:
  - importing or declaring classes we depend upon
  - Low Execution cost
  - Well-Maintained Systems
- Hard Cases of Instantiating Classes for Tests:
  - Legacy systems are all tied up
  - Huge and interdependent classes
  - Try Cutting the code to make it better!

The Dependency Inversion Principle

It is better to depend on interfaces or abstract classes than it is to depend on concrete classes. When you depend on less volatile things, you minimize the chance that particular changes will trigger massive recompilation.

Build Dependencies

- Determine what dependencies will get in the way of building more quickly
- Extract interfaces for the classes in your cluster that are used by classes outside the cluster
- As we break dependencies and section off classes into new packages or libraries, the overall cost of a rebuild of the entire system grows, but the average time for a build can decrease.
- When you introduce more interfaces and packages into your design to break dependencies, the amount of time it takes to rebuild the entire system goes up slightly. There are more files to compile. But the average time for a make, a build based on what needs to be recompiled, can go down dramatically.

Chapter 8

How Do I Add a Feature?

- Test-Driven Development (TDD)
- Programming by Difference
- Summary

Test-Driven Development (TDD)

Test-driven development uses a little algorithm that goes like this:

1. Write a failing test case.
2. Get it to compile.
3. Make it pass.
4. Remove duplication.
5. Repeat.

TDD and Legacy Code

One of the most valuable things about TDD is that it lets us concentrate on one thing at a time. We are either writing code or refactoring; we are never doing both at once.

That separation is particularly valuable in legacy code because it lets us write new code independently of new code.

After we have written some new code, we can refactor to remove any duplication between it and the old code.

Test-Driven Development in Legacy Code

0. Get the class you want to change under test.

1. Write a failing test case.

2. Get it to compile.

3. Make it pass. (Try not to change existing code as you do this.)

4. Remove duplication.

5. Repeat.

Programming by Difference

- TDD is not tied to OO
- Programming by Difference
  - Can cause programs to degrade rapidly if not used carefully
  - Takes advantage of inheritance
- Programming by Difference is a useful technique. It allows us to make changes quickly, and we can use tests to move to a cleaner design. But to do it well, we have to look out for a couple of “gotchas.” One of them is Liskov substitution principle (LSP) violation.

Liskov Substitution Principle

Objects of subclasses should be substitutable for objects of their superclasses throughout our code. If they aren’t we could have silent errors in our code.

Programming by Difference

1. Whenever possible, avoid overriding concrete methods.

2.If you do, see if you can call the method you are overriding in the overriding method.

- Normalized Hierarchy - no class has more than one implementation of a method
- Programming by Difference lets us introduce variations quickly in systems. When we do, we can use our tests to pin down the new behavior and move to more appropriate structures when we need to. Tests can make the move very rapid.

Chapter 9

I Can’t Get This Class into a Test Harness

- The Case of the Irritating Parameter
- The Case of the Hidden Dependency
- The Case of the Construction Blob
- The Case of the Irritating Global Dependency
- The Case of the Horrible Include Dependencies
- The Case of the Onion Parameter
- The case of teh Aliased Parameter

I Can’t Get This Class into a Test Harness

Four Most Commonly Encountered Problems:

1. Objects of the class can’t be created easily.

2. The test harness won’t easily build with the class in it.

3. The constructor we need to use has bad side effects.

4. Significant work happens in the constructor, and we need to sense it.

The Case of the Irritating Parameter

- The best way to see if you will have trouble instantiating a class in a test harness is to just try to do it. Write a test case and attempt to create an object in it. The compiler will tell you what you need to make it really work.
- Common Irritating Parameter - a connection to some outside source

Test Code vs. Production Code

- Test code doesn’t have to live up to the same standards as production code.
- Make variables public if needed
- Test code should be clean. It should be easy to understand and change.

Pass Null

- When you are writing tests and an object requires a parameter that is hard to construct, consider just passing null instead.
- If the parameter is used in the course of your test execution, the code will throw an exception and the test harness will catch the exception. I
- If you need behavior that really requires an object, you can construct it and pass it as a parameter at that point.
- It works well in Java and C# and in just about every language that throws an exception when null references are used at runtime.

Null Object Pattern

- The *Null Object Pattern* is a way of avoiding the use of null in programs.
- What if an employee has no ID?
- Able to shield clients from explicit error checking
- Null objects are useful specifically when a client doesn't have to care whether an operation is successful.

The Case of the Hidden Dependency

- Hidden Dependency - a resource that is not able to be accessed nicely in a test harness
- Parameterize Constructor - externalize a dependency that we have in a constructor by passing it into the constructor
  - Convenient way to externalize constructor dependencies
  - Not all clients of the class will have to be changed to pass ne new parameter

The Case of the Construction Blob

- Constructors tha construct a large number of objects internally or access a large number of globals can make for a large parameter list
- Extract and Override Factortoring Method can be used on code in a constructor if we have a refactoring tool to safely extract methods
- Supersede Instance Variable can be used to write a setter on the class and sway in another instance of another constructed object
    - Enables us to use the Extract Interface or Extract Implementer
    - Usually can be a technique to avoid

The Case of the Irritating Global Dependency

- Old-style reuse of code happens when we find some class or set of classes we want to use in our application and we just do it.
- We must test the other parts of these included classes and their usages.
- Singleton Design Pattern - limiting the sole instances of singletons that exist in an application, used to make global variables
- Global Variables are bad because they make code opaque
- All tests must be independent of other tests
- Can be resolved by Introducing a Static Setter, Creating a set up and tear down to refresh singletons

The Case of the Irritating Global Dependency

- Why do we want only one instance of a class in a system?
  1. We are modeling the real world, and there is only one of these things in the real world.
  2. If two of these things are created, we could have a serious problem. in the hardware control domain.
  3. If someone creates two of these things, we'll be using too many resources.

The Case of the Irritating Global Dependency

- Introduce Static Setter - technique we can use to get tests in place despite extensive global dependencies
  - Use Parameterize Method and Parameterized Constructor
- Global variables are usually not globally used

The Case of the Horrible Include Dependencies

- One part of C++'s C legacy that is especially problematic is its way of letting one part of a program know about another part.
- The compiler looks for that class and checks to see if it has been compiled already. If it hasn't, it compiles it. If it has been compiled, the compiler reads a brief snippet of information from the compiled file, getting only as much information as it needs to make sure that all of the methods the original class needs are on that class.
- C++ compilers generally don't have this optimization. In C++, if a class needs to know about another class, the declaration of the class (in another file) is textually included in the file that needs to use it. This can be a much slower process.

The Case of the Horrible Include Dependencies

- We only want to include files that we need
- We can get some reuse from writing a different program for tests using fakes for the fakes that we create this way.
- Put definitions of classes in a separate include file that can be used across a set of tests

The Case of the Onion Parameter

- Objects that require objects that require objects that require objects...
- Solutions:
  - Pass Null
  - Extract Interface or Extract Implementer
- In any language where we can create interfaces or classes that act like interfaces, we can systematically use them to break dependencies.

The Case of the Aliased Parameter

- The case where it is hard to make the object because it has horrible dependencies
- Interfaces are yucky to work with in this instance
- Extract Interface is one way of breaking a dependency parameter
  - Butally severing the connection to a class
- Use the Subclass and Override Method
  - Make class that supplies methods and override existing methods

Chapter 10

I Can’t Run This Method in a Test Harness

- The Case of the Hidden Method
- The Case of the “Helpful” Language Feature
- The Case of the Undetectible SIde Effect

I Can’t Run This Method in a Test Harness

Problems we can run into when writing tests for methods:

- The method might not be accessible to the test. It could be private or have some other accessibility problem.
- • It might be hard to call the method because it is hard to construct the parameters we need to call it.
- The method might have bad side effects (modifying a database, launching cruise missile, and so on), so it is impossible to run in a test harness.
- We might need to sense through some object that the method uses.

The Case of the Hidden Method

- We need to make a change to a method but it is private
- Can we test though a public method?
  - Yes - do it
- How do we write a test for a private method?
  - Make it public
  - Change necessary related things ro protected
- Issues
  - The method is just a utility; it isn't something clients would care about.
  - If clients use the method, they could adversely affect results from other methods on the class.
- Good Design is testable, and design that is not testable is bad.

Subverting Access Protection

In many OO languages newer than C++, we can use reflection and special permissions to access private variables at runtime. It is very helpful when we want to break dependencies, but I don't like to keep tests that access private variables around in projects. That sort of subterfuge really prevents a team from noticing just how bad the code is getting. It might sound kind of sadistic, but the pain that we feel working in a legacy code base can be an incredible impetus to change. We can take the sneaky way out, but unless we deal with the root causes, overly responsible classes and tangled dependencies, we are just delaying the bill. When everyone discovers just how bad the code has gotten, the costs to make it better will have gotten too ridiculous.

The Case of the “Helpful” Language Feature

- Language designers try to make lives easier
  - Hard job because they must balance teh ease of programming against security concerns and safety
- Sealed classes cause problems for testing and refactoring code
- Use the Adapt Parameter technique in this instance
- Learn to Lean on the Compiler
- Keywords sealed and final make it hard for us to see and alter code from shared files and libraries

The Case of the Undetectable Side Effect

- An object we make does not communicate with another object when it is supposed to
- How do we sense what the code does?
  - Check the names of commands
  - Separate Dependencies
  - Extract methods for the code that access the other things

Command/Query Separation

Command/Query Separation is a design principle first described by Bertrand Meyer. Simply put, it is this: A method should be a command or a query, but not both. A command is a method that can modify the state of the object but that doesn’t return a value. A query is a method that returns a value but that does not modify the object.

Chapter 11

I Need to Make a Change.
What Methods Should I Test?

- Reasoning About Effects
- Reasoning Forward
- Effect Propagation
- Tools for Effect Reasoning
- Learning from Effect Analysis
- Simplifying Effect Sketches

Reasoning About Effects

- For every functional change in software, there is some associated chain of effects.
- There does not exist an IDE that shows the relationships between pieces of code...
- Male a list of all the things that can be changed after an object is created that would affect results returned by any of its methods
  - Adding additional elements to declaration lists
  - Altering an object in the declaration list, replacing elements, affecting same methods
- Draw Effect Sketches to visualize the problem
  - Simple Effect Sketch = Well Structured Code

Reasoning Forward

- Reason Effects → deduce set of objects that affect values at a particular point in code
- Writing Characterization Tests → look at a set of objects and figure out what will change downstream if they stop working
- Draw more effects sketches
    - When you are sketching effects, make sure that you have found all of the clients of the class you are examining. If your class has a superclass or subclasses, there might be other clients that you haven't considered
- Always determine the effects of changes before picking and choosing how to write tests

Effect Propagation

- Trace effects from point to point
  - Find where values get changed
- Understand what is pass by value and pass by reference
  - Differs by language
- Note values that are constant... are they declared const (if able)?
- Effects propagate in code in three basic ways:

1. Return values that are used by a caller

2. Modification of objects passed as parameters that are used later

3. Modification of static or global data that is used later

Looking for Effects

1. Identify a method that will change.
2. If the method has a return value, look at its callers.
3. See if the method modifies any values. If it does, look at the methods that use those values, and the methods that use those methods.
4. Make sure you look for superclasses and subclasses that might be users of these instance variables and methods also.
5. Look at parameters to the methods. See if they or any objects that their methods return are used by the code that you want to change.
6. Look for global variables and static data that is modified in any of the methods you've identified.

Tools for Effect Reasoning

- Most important tool is our programming language
  - Everyone has firefalls to prevent propagation
- Firewalls in Programming Languages:
  - Private, Public, Protected
  - Const
  - Mutable

KNOW YOUR LANGUAGE

Learning from Effect Analysis

- Effect Analysis allows you to eventaually feel comfy in your code
- You can find rules embedded in your code base
- Programming gets easier the more we narrow effects in a program
- Restricting effects makes tests easier with less hurdles to jump through

Simplifying Effect Sketches

- Try to use functions internally
- Removing duplication helps develop effect sketches with a smaller set of endpoints
  - Results in easier testing decisions
- Effects and Encapsulation
  - Breaking encapsulation occurs
  - Encapsulation is important because it helps us reason about code
  - Encapsulation creates fewer paths to follow
  - Breaking encapsulation can make reasoning about code harder, but it can be easier if we end up with good explanatory tests afterwards

Chapter 12

I Need to Make Many Changes in One Area

- Interception Points
- Judging Design with Pinch Points

Do I Have to Break Dependencies for All the Classes Involved?

- It can pay to test “one level back”, or find a place where we can write tests for several changes at once
- Higher-level tests are more prefered than finely grained tests at each class because change is harder when lots of little tests are written against an interface that has to change.
- While higher-level tests are an important tool, they shouldn’t be a substitute for unit tests. Instead, they should be a first step toward getting unit tests in place.

Interception Points

- An *interception point* is simply a pointin a program where the detection of effects of a particular change are found
- They can be hard to find in some programs
- If you have an application whose pieces are glued together without too many natural seams, finding a decent *interception point* can be a big deal and requires effect reasoning and dependency breaking.
- The best way to start:
  - Identify places where you need to make changes
  - Start tracing the effects outward from those change point
  - At each place where you detect effects, you have an *inception point*
  - You must determine the best *inception point*

The Simple Case

- It is a good idea to pick interception points that are very close to your
- change points
- Reasons Why:
  - Safety: Every step between a change point and an interception point is like a step in a logical argument.
  - More distant interception points are worse because it is often harder to set up tests at them.

Higher-Level Interception Points

- The best interception point we can have for a change is a public method on the class we are changing
- The interception points are easy to fund and easy to use, but sometimes are not the best choice.
- We can use higher-level interception points to characterize areas of code
- *Pinch Point* - a narrowing in an effect sketch, or a place where it is possible to write tests to cover a wide set of changes
- Pinch points are determined by change points
- Pinch points can be found by looking for common usage across an effect sketch

Judging Design with Pinch Points

- Pinch points have other uses including:
  - Finding natural encapsulation boundaries
  - They are narrow funnels for all of the effects of a large piece of code
  - It is easy to notice how responsibilities can be allocated across classes to give better encapsulation
- Use effect sketches to find hidden classes
  - Look for natural encapsulation boundaries to develop classes
- Writing tests at pinch points is an idea way to start invasive work in a program
  - You can write characterization tests and then make changes with impunity
  - Be careful - you can trap yourself in an “oasis”

Pinch Point Traps

- How to get into trouble:
  - Let unit tests slowly grow into mini-integration tests
    - These will be massive and take forever to run
    - BREAK THESE DOWN OR BREAK YOUR CLASS DOWN
    - Write unit tests for no code and make the testing independent of other classes
- Use fakes to help address many of these issues
- You can delete tests at pinch points over time and let the tests for each class support your development work

Chapter 13

I Need to Make a Change, but I Dont Know What Tests to Write

- Characterization Tests
- Characterizing Classes
- Targeted Testing
- A Heuristic for Writing Characterization Tests

Characterization Tests

Characterization Test - tests that characterizes the behavior of code

Algorithm for writing Characterization Tests

1. Use a piece of code in a test harness.

2. Write an assertion that you know will fail.

3. Let the failure tell you what the behavior is.

4. Change the test so that it expects the behavior that the code produces.

5. Repeat.

Characterization Tests

Characterization tests record the actual behavior of a piece of code. If we find something unexpected when we write them, it pays to get some clarification. It could be a bug. That doesn't mean that we don't include the test in our test suite; instead, we should mark it as suspicious and find out what the effect would be of fixing it.

This is not black box testing.

The Method Use Rule

Before you use a method in a legacy system, check to see if there are tests for it.

If there aren’t, write them.

When you do this consistently, you use tests as a medium of communication.

The act of making a class testable in itself tends to increase code quality.

Characterizing Classes

How to figure out what to test:

1. Look for tangled pieces of logic. Introduce a sensing variable to characterize logic and execute particular areas of the code.
2. As you discover the responsibilities of a class or method, stop to make a list of the things that can go wrong. Formulate tests that trigger them.
3. Think about the inputs you are supplying under test.
4. 4. Should any conditions be true at all times during the lifetime of the class? Attempt to write tests to verify invariants. Refactor to discover these conditions. If you do, the refactorings often lead to new insight about how the code should be.

When You Find Bugs

- If the system has not been deployed, fix the bug
- If the system has been deployed, examine the possibility that someone is depending on the behavior even though it is a bug.
- My bias is toward fixing bugs as soon as they are found. When behavior is clearly in error, it should be fixed. If you suspect that some behavior is wrong, mark it in the test code as suspicious and then escalate it. Find out as quickly as you can whether it is a bug and how best to deal with it.

Targeted Testing

- Look at the things we want to change and determine if tests cover them
- See if there is any other way that the test could pass, aside from executing that branch. Use a sensing variable or the debugger to find out whether the test is hitting it.
- One important thing to figure out when you are characterizing branches.
- When refactoring, check:
  - Does the behavior exist after the refactoring, and is it connected correctly?
- The most valuable characterization tests exercise a specific path and exercise each conversion along the path.

A Heuristic for Writing Characterization Tests

1. Write tests for the area where you will make your changes. Write as many cases as you feel you need to understand the behavior of the code.

2. After doing this, take a look at the specific things you are going to change, and attempt to write tests for those.

3. If you are attempting to extract or move some functionality, write tests that verify the existence and connection of those behaviors on a case-by-case basis. Verify that you are exercising the code that you are going to move and that it is connected properly. Exercise conversions.

Chapter 14

Dependencies on Libraries Are Killing Me

Dependencies on Libraries Are Killing Me

- Code reuse helps development
- Do not become over-reliant on a library
- Avoid littering direct calls to library classes in your code. You might think that you'll never change them, but that can become a self-fulfilling prophecy.
- Library designers who use language features to enforce design constraints are often making a mistake. They forget that good code runs in production and test environments. Constraints for the former can make working in the latter nearly impossible.

Dependencies on Libraries Are Killing Me

- A fundamental tension exists between language features that try to enforce good design and things you have to do to test code
  - Once dilemma - if the library assumes there is going to be only one instance of a class in a system it can make the use of fake objects difficult
  - There may not be any way to introduce a static setter
  - Dependency breaking techniques can be impossible
- best of both worlds.
- Sometimes using a coding convention is just as good as using a restrictive language feature. Think about what your tests need.

Chapter 15

My Application is All API Calls

My Application is All API Calls

- Do not assume you do not need tests
- Changes can still be uncertain and you have to maintain it even though you did not write all of it
- Systems that are littered with library calls are harder to deal with than homegrown systems, in many respects.
    - It is often hard to see how to make the structure better because all you can see are the API calls. Anything that would've been a hint at a design just isn't there. T
    - API-intensive systems are difficult is that we don't own the API. If we did, we could rename interfaces, classes, and methods to make things clearer for us, or add methods to classes to make them available to different parts of the code.

My Application is All API Calls

- How to structure code better -> Identify the computational core of code
  - Skin and Wrap API
    - Can remove dependencies on API Code
    - Wrappers can allow us to use fakes in testing
  - Responsibility-Based Extraction
- Trade-offs between API Wrapping and Responsibility Based Extraction:
  - Skin and Wrap
    - API is small
    - You want to completely separate out dependencies on a third-party library
    - You do not have tests and cannot write them because of the API
  - Responsibility-Based Extraction
    - The API is complicated
    - You have a tool that provides a safe extract method support

Chapter 16

I Don’t Understand the Code Well Enough to Change It

- Notes/Sketching
- Listing Markup
- Scratch Refactoring
- Delete Unused Code

Notes/Sketching

- Start drawing pictures and making notes when reading through code gets confusing
- Write down the names of things that seem important
- Can help you explain to others the way you believe things are occurring and identify the relationships in your code
- You may be able to identify the structure of your code, or see that it has no structure

Listing Markup

- **Listing Markup** is useful to use with very long methods
- Use depends on what you want to understand
- Separating Responsibilities
  - Use a marker to group things - color coding
- Understanding Method Structure
  - Line up blocks by drawing lines from the beginning to end of blocks
- Extract Methods
  - Circle code you want to extract and identify coupling
- Understanding the Effects of a Change
  - Mark lines of code you want to change and mark all the variables that are affected
  - This helps identify what all needs to be covered by a test

Scratch Refactoring

- Best technique for learning about code is refactoring
- Be careful if you do not have tests
- Scratch refactoring - checking out code and play with it, but do not check it in. Throw it away when you are done
- You can learn alot about the code and see the essentials and really learn how code works
- Be cautious and do be aware of what the system is trying to do
- False views of the system can cause problems when you try to make changes to the real system

Delete Unused Code

- If you find code that is not used, delete it.
- Version controls exist for a reason
- If you need it later, you can go back

Chapter 17

My Application Has No Structure

- Telling the Story of the System
- Naked CRC
- Conversation Scrutiny

My Application Has No Structure

- Long-lived applications tend to sprawl
- There is no remedy for fixing this immediately
- What gets in the way of system architecture awareness:
  - The system can be so complex that it takes a long time to get the big picture.
  - The system can be so complex that there is no big picture.
  - The team is in a very reactive mode, dealing with emergency after emergency so much that they lose sight of the big picture.
- Architecture is too important to be left exclusively to a few people
- Everyone must be working off the same set of ideas and uphold the architecture

Telling the Story of the System

- Get some people and ask “What is the Architecture of the System?”
- Everyone explains using a few concepts pretending that the others know nothing about the system
- This makes the architecture easier to understand and everyone can develop an idea of what is the best and most ideal architecture for the system for everyone to follow

Naked CRC

- We think about code differently today than when it was written in the early days of programming
- CRC stands for Class, Responsibility, and Collaboration
- You mark up each card with a class name, its responsibilities, and a list of its collaborators
- Use cards and describe connections between cards
- There are just two guidelines in Naked CRC:

  1. Cards represent instances, not classes.

  2. Overlap cards to show a collection of them.

Conversation Scrutiny

- Listen to conversations about your design
- Software has to satisfy stronger constraints than just being easy to talk
- about, but if there isn't a strong overlap between conversation and code, it's important to ask why.
- The answer is usually a mixture of two things:
  - The code hasn't been allowed to adapt to the team's understanding, or the team needs to understand it differently.
    - In any case, being very tuned to the concepts people naturally use to describe the design is powerful. When people talk about design, they are trying to make other people understand them. Put some of that understanding in the code.
- Bloating is the worst thing you can do to a legacy system

Chapter 18

My Test Code Is in the Way

- Class Naming Conventions
- Test Location

Class Naming Conventions

- Make the unit test class name a variation of the class name
- Make a convention and keep with it
- Use fake classes and have a prefix of fake or some other convention you will stick with
- Subclasses are similarly handled

Test Location

- Placing code in the same directory is a easy way to structure a project
- Be aware of the size of deployment if on a commercial system
- An alternative is to keep the production code and the test code in the same location but to use scripts or build settings to remove the test code from the deployment.
- Have good reasons for separating test and production code

Chapter 19

My Project Is Not Object Oriented. How Do I Make Safe Changes?

- An Easy Case
- A Hard Case
- Adding New Behavior
- Taking Advantage of Object Orientation
- It’s All Object Oriented

My Project Is Not Object Oriented. How Do I Make Safe Changes?

- Some languages are not object oriented and there are many other ways of writing code
- Extremely hard to test - little you can do

Best Strategy: Get a lot of code under a test before doing anything else and then use the test to get feedback while developing.

An Easy Case

- Procedural code is not always a problem
- Some functions are easily testable, as we can make objects and vairables and pass them into a function to test them
- Other functions are harder to deal with
    - Ex. Functions that does some I/O or includes a vendor's library

A Hard Case

- Code that works with a third-party system makes functions hard to work with
- Can use link seams in some cases
- Can make a library that contains fakes, functions with the same names as the original functions that do not do what they are supposed to
- Sometimes link seams are better than fake libraries - depends on situation
    - Fake libraries are good when they can be used frequently

Adding New Behavior

- In procedural legacy code, it pays to bias toward introducing new functions rather than adding code to old ones because we can write tests for the ones we write
- Must avoid introducing dependency traps in procedural code
  - Can be done using test-driven development (works in OO and procedural code)
- Put all of the logic into one set of functions so we can keep them free of problematic dependencies
- Take advantage of function pointers if you can use them to point to different seams

Taking Advantage of Object Orientation

- In OO we have object seams that have nice properties
  - Easy to notice
  - Can be used to break code down into smaller, more understandable pieces
  - Provide flexibility
- You can make some procedural languages move toward being object oriented
- First step is to Encapsulate Global References to get the changing pieces under a test
- Can use it to get out of bad dependency situations
- Use Preserve Signatures to minimize chances of errors

It's All Object Oriented

- Procedural programs are all object oriented, but many contain only one object
- When procedural languages have object-oriented extensions, they allow us to move in an an OO direction
- Group related functions in classes and extract methods to break apart tangled responsibilities
- Pay attention to the seams procedural languages present and use them to get more tests in place and better designed code

Chapter 20

This Class Is Too Big and I Don't Want It to Get Any Bigger

- Seeing Responsibilities
- Other Techniques
- Moving Forward
- After Extract Class

Single Responsibility Principle

- The key remedy for big classes is refactoring. It helps break down classes into sets of smaller classes.

  Single Responsibility Principle:

  Every class should have a single responsibility: It should have a single purpose in the system, and there should be only one reason to change it.

Seeing Responsibilities

- Heuristic #1: Group Methods
  - Look for similar method names. Write down all of the methods on a class, along with their access types (public, private, and so on), and try to find ones that seem to go together.
- Heuristic #2: Look at Hidden Methods
  - Pay attention to private and protected methods. If a class has many of them, it often indicates that there is another class in the class dying to get out.
- Heuristic #3: Look for Decisions That Can Change
  - Look for decisions—not decisions that you are making in the code, but decisions that you've already made. Is there some way of doing something (talking to a database, talking to another set of objects, and so on) that seems hard-coded? Can you imagine it changing?

Seeing Responsibilities

- Heuristic #4: Look for Internal Relationships
  - Look for relationships between instance variables and methods. Are certain instance variables used by some methods and not others?
- Create Feature Sketches
  - Find lumps of things in code you consider to be a feature
  - Make a sketch of them and draw their relationshios to other things in the class
  - They are great for showing the internal structure of classes
- Heuristic #5: Look for the Primary Responsibility
  - Try to describe the responsibility of the class in a single sentence.
    - Single Responsibility Principle
      - Violated at interface level
      - Violated at implementation level

Interface Segregation Principle

When a class is large, rarely do all of its clients use all of its methods. Often we can see different groupings of methods that particular clients use. If we create an interface for each of these groupings and have the large class implement those interfaces, each client can see the big class through that particular interface. This helps us hide information and also decreases dependency in the system. The clients no longer have to recompile whenever the large class does.

Seeing Responsibilities

- Heuristic #6: When All Else Fails, Do Some Scratch Refactoring
  - If you are having a lot of trouble seeing responsibilities in a class, do some scratch refactoring.
- Heuristic #7: Focus on the Current Work
  - Pay attention to what you have to do right now. If you are providing a different way of doing anything, you might have identified a responsibility that you should extract and then allow substitution for.

Other Techniques

- Heuristics for identifying responsibilities can help find new abstractions in old classes
- Read more books about design patterns
- Read code

Moving Forward

Strategy

- Refactoring - All at once
- Breaking down big classes by identify responsibilities - As needed

Tactics

- Identify a responsibility and what it needs
- Extract all of the necessary bodies into a new class
- Clean up code and fix any other instances

After Extract Class

- Do not get over ambitious
- Extracting classes from a big class is a good first step
- Remember the positive direction you want to move in

Chapter 21

I’m Changing the Same Code All Over the Place

- First Steps

First Steps

- Get a grasp on scope
- Make a function
- Try a command pattern - make a parent superclass
- Do not use abbreviations
- Orthogonality means independence
  - Removing duplication creates this - only one place you need to go when you need to make a change
- Open/Closed Principle
  - Open for extension but closed for modification

Chapter 22

I Need to Change a Monster Method and I Can’t Write Tests for It

- Varieties of Monsters
- Tackling Monsters with Automated Refactoring Support
- The Manual Refactoring Challenge
- Strategy

Varieties of Monsters

- Monster methods are methods that are so long and complex that you do not feel comfortable touching it
- Bulleted methods - ones with little to no indentation
- Snarled methods - ones that have a single large indented section
- Mixed - a bit of bulleted and a bit snarled

Tackling Monsters with Automated Refactoring Support

When doing automated refactoring without tests, use the tool exclusively. After a series of automated refactorings, you can often get tests in place that you can use to verify any manual edits that you make.

When you do your extractions, these should be your key goals:

1. To separate logic from awkward dependencies

2. To introduce seams that make it easier to get tests in place for more refactoring

The Manual Refactoring Challenge

Possible ways to create errors:

1. We can forget to pass a variable into the extracted method. Often the compiler tells us about the missing variable (unless it has the same name as an instance variable), but we could just think that it needs to be a local variable and declare it in the new method.
2. We could give the extracted method a name that hides or overrides a method with the same name in a base class.
3. We could make a mistake when we pass in parameters or assign return values. We could do something really silly, such as return the wrong value. More subtly, we could return or accept the wrong types in the new method.

The Manual Refactoring Challenge

- Introduce a sensing variable
  - Use a variable to help you refactor and get rid of it when you are done
- Extract what you know
  - Get rid of what you can confidently and then make tests and move what your are not confident in
  - Analyze coupling
- Gleaning dependencies
  - Write tests for the logic you need to preserve, extract things that the tests do not cover
  - Helps preserve the important behavior
- Break out a method object
  - Create a class whose only responsibility is to do the work of the monster method

Strategy

- Skeletonize Methods
  - Extract things separately instead of together
  - Skeletonize when the control structure will need to be refactored
- Find Sequences
  - Extract the condition and the body together
  - Need to identify an overarching sequence to make the code clearer
- Extract ot the Current Class First
  - Do not extract things to another class, even if the name bothers you
- Extract Small Pieces
  - It is easier to understand things in large chunks in order to have to understand something major
- Be Prepared to Redo Extractions
  - Do not fear undoing and redoing work

Chapter 23

How Do I Know That I’m Not Breaking Anything?

- Hyperaware Editing
- Single-Goal Editing
- Preserve Signatures
- Lean on the Computer

Hyperaware Editing

- Implement test driven development
- Tests foster hyperaware editing
- Pair programming does too

Single-Goal Editing

- “Programming is the art of doing one things at a time”
- Challenge your partner by asking them what they are doing and ensuring that they are only working toward one goal

Preserve Signatures

- Complete the same “automated” process or series of steps over and over again when refactoring
- Make new method declarations
- Make a set of instance methods for all the arguments to a method

Lean on the Compiler

- The primary purpose of a compiler is to translate source code into some other form
- In some statically typed languages you can take advantage of its type checking and use it to identify changes you need to make
- The key thing about this technique is that you are letting the compiler guide you toward the changes you need to make.
- Lean on the Compiler involves two steps:

  1. Altering a declaration to cause compile errors

  2. Navigating to those errors and making changes.

Pair Programming

- Increases quality and spreads knowledge around a team
- Great to do when using dependency-breaking techniques
- A second set of eyes always helps

Chapter 24

We Feel Overwhelmed. It Isn’t Going to Get Any Better

We Feel Overwhelmed. It Isn't Going to Get Any Better

- Creating a new system is not always the best thing
  - Separating a team and making one work on the old and the other on the new places stress on both teams
- The grass is not greener in the "green-field" development
- Key to Thriving: What Motivates You
  - Connecting to community
  - Program for fun
  - Find a way to get control of your code
    - Little successes

Part III:
Dependency-Breaking Techniques

Chapter 25

Dependency-Breaking Techniques

- Adapt Parameter
- Break Out Method Object
- Definition Completion
- Encapsulate Global References
- Expose Static Method
- Extract and Override Call
- Extract and Override Factory Method
- Extract and Override Getter
- Extract Implementer
- Extract Interface
- Introduce Instance Delegator
- Introduce Static Setter
- Link Substitution
- Parameterize Constructor
- Parameterize Method
- Primitive Parameter
- Pull Up Feature
- Push Down Dependency
- Replace Function with Function Pointer
- Replace Global Reference with Getter
- Subclass and Override Method
- SuperSede Instance Variable
- Template Redefinition
- Text Redefinition

Adapt Parameter

Use Adapt Parameter when you cannot use Extract Interface on a parameter's class or when a parameter is difficult to fake.

Move toward interfaces that communicate responsibilities rather than implementation details to make code easier to read and easier to maintain.

Adapt Parameter is one case in which we do not have to Preserve Signatures.

Once you have tests in place, you can make changes more confidently.

Adapt Parameter

To use Adapt Parameter, perform the following steps:

1. Create the new interface that you will use in the method. Make it as simple and communicative as possible, but try not to create an interface that will require more than trivial changes in the method.
2. Create a production implementer for the new interface.
3. Create a fake implementer for the interface.
4. Write a simple test case, passing the fake to the method.
5. Make the changes you need to in the method to use the new parameter.
6. Run your test to verify that you are able to test the method using the fake.

Break Out Method Object

Break Out Method Object has several variations. In the simplest case, the original method doesn't use any instance variables or methods from the original class. We don't need to pass it a reference to the original class.

In other cases, the method only uses data from the original class. At times, it makes sense to put this data into a new data-holding class and pass it as an argument to the method object.

Break Out Method Object

You can use these steps to do Break out Method Object safely without tests:

1. Create a class that will house the method code.
2. Create a constructor for the class and Preserve Signatures to give it an exact copy of the arguments used by the method. If the method uses an instance data or methods from the original class, add a reference to the original class as the first argument to the constructor.
3. For each argument in the constructor, declare an instance variable and give it exactly the same type as the variable. Preserve Signatures by copying all the arguments directly into the class and formatting them as instance variable declarations. Assign all of the arguments to the instance variables in the constructor.

Break Out Method Object

4. Create an empty execution method on the new class. Often this method is called run(). We used the name draw in the example.
5. Copy the body of the old method into the execution method and compile to Lean on the Compile.
6. The error messages from the compiler should indicate where the method is still using methods or variables from the old class. In each of these cases, do what it takes to get the method to compile.
7. After the new class compiles, go back to the original method and change it so it creates an instance of the new class and delegates its work to it.
8. If needed, use Extract Interface (362) to break the dependency on the original class.

Definition Completion

To use Definition Completion in C++, follow these steps:

1. Identify a class with definitions you'd like to replace.

2. Verify that the method definitions are in a source file, not a header.

3. Include the header in the test source file of the class you are testing.

4. Verify that the source files for the class are not part of the build.

5. Build to find missing methods.

6. Add method definitions to the test source file until you have a complete build.

Encapsulate Global References

- If several globals are always used or are modified near each other, they belong in the same class.
- When naming a class, think about the methods that will eventually reside on it. The name should be good, but it doesn't have to be perfect. Remember that you can always rename the class later.
- The class name that you find might already be in use. If so, consider whether you can rename whatever is using that name.
- Referencing a member of a class rather than a simple global is only the first step. Afterward, consider whether you should use Introduce Static Setter, or parameterize the code using Parameterize Constructor (379) or Parameterize Method.

Encapsulate Global References

- When you use Encapsulate Global References, start with data or small methods. More substantial methods can be moved to the new class when more tests are in place.
- To encapsulate references to free functions, make an interface class with fake and production subclasses. Each of the functions in the production code should do nothing more than delegate to a global function.

Encapsulate Global References

To Encapsulate Global References, follow these steps:

1. Identify the globals that you want to encapsulate.
2. Create a class that you want to reference them from.
3. Copy the globals into the class. If some of them are variables, handle their initialization in the class.
4. Comment out the original declarations of the globals.
5. Declare a global instance of the new class.
6. Lean on the Compiler to find the unresolved references to the old globals.

Encapsulate Global References

7. Precede each unresolved reference with the name of the global instance of the new class.
8. In places where you want to use fakes, use Introduce Static Setter, Parameterize Constructor, Parameterize Method, or Replace Global Reference with Getter.

Expose Static Method

- When you are breaking dependencies without tests, Preserve Signatures of methods whenever possible. If you cut/copy and paste whole method signatures, you have less of a chance of introducing errors.
- Static methods and data really do act as if they are part of a different class. Static data lives for the life of a program, not the life of an instance, and statics are accessible without an instance.
- The static portions of a class can be seen as a “staging area” for things that don’t quite belong to the class. If you see a method that doesn’t use any instance data, it is a good idea to make it static to make it noticeable until you figure out what class it really belongs on.

Expose Static Method

To Expose Static Method, follow these steps:

1. Write a test that accesses the method that you want to expose as a public static method of the class.
2. Extract the body of the method to a static method. Remember to Preserve Signatures. You'll have to use a different name for the method. Often you can use the names of parameters to help you come up with a new method name.
3. Compile.
4. If there are errors related to accessing instance data or methods, take a look at those features and see if they can be made static also. If they can, make them static so that the system will compile.

Extract and Override Call

To Extract and Override Call, follow these steps:

1. Identify the call that you want to extract. Find the declaration of its method. Copy its method signature so that you can Preserve Signatures (312).

2. Create a new method on the current class. Give it the signature you've copied.

3. Copy the call to the new method and replace the call with a call to the new method.

Extract and Override Factory Method

To Extract and Override Factory Method, follow these steps:

1. Identify an object creation in a constructor.

2. Extract all of the work involved in the creation into a factory method.

3. Create a testing subclass and override the factory method in it to avoid dependencies on problematic types under test.

Extract and Override Getter

- A lazy getter is a method that looks like a normal getter to all of its callers.
  - Key difference is that lazy getters create the object they are supposed to return the first time they are called.
  - They contain logic
  - Used int the SIngleton Design Pattern
- When you use Extract and Override Getter, you have to be very conscious of object lifetime issues, particularly in a non-garbage-collected language such as C++. Make sure that you delete the testing instance in a way that is consistent with how the code deletes the production instance.

Extract and Override Getter

To Extract and Override Getter, follow these steps:

1. Identify the object you need a getter for.
2. Extract all of the logic needed to create the object into a getter.
3. Replace all uses of the object with calls to the getter, and initialize the reference that holds the object to null in all constructors.
4. Add the first-time logic to the getter so that the object is constructed and assigned to the reference whenever the reference is null.
5. Subclass the class and override the getter to provide an alternative object for testing.

Extract Implementer

- Naming is a key part of design. If you choose good names, you reinforce understanding in a system and make it easier to work with. If you choose poor names, you undermine understanding and make life hellish for the programmers who follow you.’
- Naming is the hardest part of this handy technique
  - Make up something foolish
  - Look at methods and come up with some name based on them

Extract Implementer

To Extract Implementer, follow these steps:

1. Make a copy of the source class's declaration. Give it a different name. It's useful to have a naming convention for classes you've extracted.
2. Turn the source class into an interface by deleting all non-public methods and all variables.
3. Make all of the remaining public methods abstract. If you are working in C++, make sure that none of the methods that you make abstract are overridden by non-virtual methods.

Extract Implementer

4. Examine all imports or file inclusions in the interface file, and see if they are necessary. Often you can remove many of them. You can Lean on the Compiler to detect these. Just delete each in turn, and recompile to see if it is needed.
5. Make your production class implement the new interface.
6. Compile the production class to make sure that all method signatures in the interface are implemented.
7. Compile the rest of the system to find all of the places where instances of the source class were created. Replace these with creations of the new production class.
8. Recompile and test.

Extract Interface

- One of the safest dependency-breaking techniques
- Compiler tells you immediately if there is a bug
- Three ways of doing Extract Interface:
  - Use automated refactoring support
  - Extract a method incrementally
  - Cut/Copy/Paste several methods from a class at once and place their declarations in and interface.
- When you extract an interface, you don't have to extract all of the public methods on the class you are extracting from. Lean on the Compiler to find the ones that are being used.

Extract Interface

To Extract Interface, follow these steps:

1. Create a new interface with the name you'd like to use. Don't add any methods to it yet.
2. Make the class that you are extracting from implement the interface. This can't break anything because the interface doesn't have any methods. But it is good to compile and run your test just to verify that.
3. Change the place where you want to use the object so that it uses the interface rather than the original class.
4. Compile the system and introduce a new method declaration on the interface for each method use that the compiler reports as an error.

Introduce Instance Delegator

To Introduce Instance Delegator, follow these steps:

1. Identify a static method that is problematic to use in a test.

2. Create an instance method for the method on the class. Remember to Preserve Signatures. Make the instance method delegate to the static method.

3. Find places where the static methods are used in the class you have under test. Use Parameterize Method or another dependency-breaking technique to supply an instance to the location where the static method call was made.

The Singleton Design Pattern

The Singleton Design Pattern is a pattern that many people use to make sure that there can only be one instance of a particular class in a program. There are three properties that most singletons share:

1. The constructors of a singleton class are usually made private.

2. A static member of the class holds the only instance of the class that will ever be created in the program.

3. A static method is used to provide access to the instance. Usually this method is named instance.

Introduce Static Setter

- One alternative to decreasing constructor protection and subclassing is to use Extract Interface on the singleton class and supply a setter that accepts an object with that interface.
- The downside of this is that you have to change the type of the reference you use to hold the singleton in the class and the type of the return value of the instance method.
- These changes can be quite involved, and they don’t really move us to a better state.
- The ultimate “better state” is to reduce global references to the singleton to the point that it can just become a normal class.

Introduce Static Setter

To Introduce Static Setter, follow these steps:

1. Decrease the protection of the constructor so that you can make a fakeby subclassing the singleton.

2. Add a static setter to the singleton class. The setter should accept a reference to the singleton class. Make sure that the setter destroys the singleton instance properly before setting the new object.

3. If you need access to private or protected methods in the singleton to set it up properly for testing, consider subclassing it or extracting an interface and making the singleton hold its instance as reference whose type is the type of the interface.

Link Substitution

To use Link Substitution, follow these steps:

1. Identify the functions or classes that you want to fake.

2. Produce alternative definitions for them.

3. Adjust your build so that the alternative definitions are included ratherthan the production versions.

Parameterize Constructor

To Parameterize Constructor, follow these steps:

1. Identify the constructor to parameterize and make a copy of it.
2. Add a parameter to the constructor for the object whose creation you are going to replace. Remove the object creation and add an assignment from the parameter to the instance variable for the object.
3. If you can call a constructor from a constructor in your language, remove the body of the old constructor and replace it with a call to the old constructor. Add a new expression to the call of the new constructor in the old constructor. If you can’t call a constructor from another constructor in your language, you may have to extract any duplication among the constructors to a new method.

Parameterize Method

To Parameterize Method, follow these steps:

1. Identify the method that you want to replace and make a copy of it.

2. Add a parameter to the method for the object whose creation you are going to replace. Remove the object creation and add an assignment .from the parameter to the variable that holds the object.

3. Delete the body of the copied method and make a call to the parameterized method, using the object creation expression for the original object.

Primitive Parameter

Primitivize Parameter leaves code in a rather poor state. Overall, it is better to add the new code to the original class or to use Sprout Class to build up some new abstractions that can serve as a base for further work. The only time I use Primitivize Parameter is when I feel confident that I will take the time to bring the class under test later. At that point, the function can be folded into the class as a real method.

Primitive Parameter

To Primitivize Parameter, follow these steps:

1. Develop a free function that does the work you would need to do on the class. In the process, develop an intermediate representation that you can use to do the work.

2. Add a function to the class that builds up the representation and delegates it to the new function.

Pull Up Feature

To do Pull Up Feature, follow these steps:

1. Identify the methods that you want to pull up.

2. Create an abstract superclass for the class that contains the methods.

3. Copy the methods to the superclass and compile.

4. Copy each missing reference that the compiler alerts you about to the new superclass. Remember to Preserve Signatures to reduce the chance of errors.

5. When both classes compile successfully, create a subclass for the abstract class and add whatever methods you need to be able to set it up in your tests.

Push Down Dependency

To Push Down Dependency, follow these steps:

1. Attempt to build the class that has dependency problems in your test harness.

2. Identify which dependencies create problems in the build.

3. Create a new subclass with a name that communicates the specific environment of those dependencies.

Push Down Dependency

4. Copy the instance variables and methods that contain the bad dependencies into the new subclass, taking care to preserve signatures. Make methods protected and abstract in your original class, and make your original class abstract.

5. Create a testing subclass and change your test so that you attempt to instantiate it.

6. Build your tests to verify that you can instantiate the new class.

Replace Function with Function Pointer

Replace Function with Function Pointer is a good way to break dependencies. One of the nice things about it is that it happens completely at compile time, so it has minimal impact on your build system. However, if you are using this technique in C, consider upgrading to C++ so that you can take advantage of all of the other seams that C++ provides you. At the time of this writing, many C compilers offer switches to allow you to do mixed C and C++ compilation. When you use this feature, you can migrate your C project to C++ slowly, taking only the files that you care to break dependencies in first.

Replace Function with Function Pointer

To use Replace Function with Function Pointer, do the following:

1. Find the declarations of the functions you want to replace.
2. Create function pointers with the same names before each function declaration.
3. Rename the original function declarations so that their names are not the same as the function pointers you've just declared.
4. Initialize the pointers to the addresses of the old functions in a C file.
5. Run a build to find the bodies of the old functions. Rename them to the new function names.

Replace Global Reference with Getter

To Replace Global Reference with Getter, do the following:

1. Identify the global reference that you want to replace.

2. Write a getter for the global reference. Make sure that the access protection of the method is loose enough for you to be able to override the getter in a subclass.

3. Replace references to the global with calls to the getter.

4. Create a testing subclass and override the getter.

Subclass and Override Method

To Subclass and Override Method, do the following:

1. Identify the dependencies that you want to separate or the place where you want to sense. Try to find the smallest set of methods that you can override to achieve your goals.

2. Make each method overridable. The way to do this varies among programming languages. In C++, the methods have to be made virtual if they aren't already. In Java, the methods need to be made non-final. In many .NET languages, you explicitly have to make the method overridable also.

Subclass and Override Method

3. If your language requires it, adjust the visibility of the methods that you will override to so that they can be overridden in a subclass. In Java and C#, methods must at least have protected visibility to be overridden in subclasses. In C++, methods can remain private and still be overridden in subclasses.

4. Create a subclass that overrides the methods. Verify that you are able to build it in your test harness.

Supersede Instance Variable

Generally, it is poor practice to provide setters that change the base objects that an object uses. Those setters allow clients to drastically change the behavior of an object during its lifetime. When someone can make those changes, you have to know the history of that object to understand what happens when you call one of its methods. When you don’t have setters, code is easier to understand.

Supersede Instance Variable

To Supersede Instance Variable, follow these steps:

1. Identify the instance variable that you want to supersede.

2. Create a method named supersedeXXX, where XXX is the name of the variable you want to supersede.

3. In the method, write whatever code needed to destroy the previous instance of the variable and set it to the new value. If the variable is a reference, verify there aren't other references in the class to the object it points to. If there are, additional work can be done in the superceding method to make sure that replacing the object is safe and has the right effect.

Template Redefinition

Template Redefinition in C++ has one primary disadvantage. Code that was in implementation files moves to headers when you templatize it. This can increase the dependencies in systems. Users of the template then are forced to recompile whenever the template code is changed.

Template Redefinition

Here is a description of how to do Template Redefinition in C++. The steps might be different in other languages that support generics, but this description gives a flavor of the technique:

1. Identify the features to replace in the class you need to test.
2. Turn the class into a template, parameterizing it by the variables needing replaced and copying the method bodies up into the header.
3. Give the template another name.
4. Add a typedef statement after the template definition, defining the template with its original arguments using the original class name.
5. In the test file, include the template definition and instantiate the template on new types that will replace the ones needing replaced.

Text Redefinition

To use Text Redefinition in Ruby, follow these steps:

1. Identify a class with definitions that you want to replace.

2. Add a require clause with the name of the module that contains that class to the top of the test source file.

3. Provide alternative definitions at the top of the test source file for each method that you want to replace.