# Working Effectively with Legacy Code Book Report

NetID: jasonh11
Author: Michael Feathers
Edition: 1st Edition, Addison-Wesley, 2004

## Table of Contents



## Part I: The Mechanics of Change

### Chapter 1: Changing Software

*Chapter 1 Summary (includes Foreword, Preface, and Introduction)*

Legacy code, though different from what most programmers think of as poorly designed and written code, is more specifically defined in this book as code without tests. While this definition does not provide a straightforward understanding of what legacy code is like, it offers a direction for creating a maintainable and improvable codebase. Nevertheless, having tests and good design still does not prevent code from becoming legacy. The requirements of a program will always change, and with them, so will its original structure.

There exists four primary reasons to change software:

(1, 2) Adding a feature and fixing a bug

> There is always some overlap between adding a feature and fixing a bug, despite them sounding completely different. A key distinction between the two is whether we are adding a behavior or modifying/removing one. From a programmer's perspective, this translates to whether code is only added or also changed. However, this can still lead to some confusion, such as when adding a new button seems like a feature but also alters the original page's behavior by introducing the button.

(3) Improving the design

> In recent years, code refactoring has gained popularity and has become synonymous with improving software design. With the support of tests at different levels, the goal is to enhance

maintainability while preserving all original behaviors. In the past, refactoring was less common due to the risk of unintentionally altering behavior or introducing bugs.

(4) Optimizing resource usage

Similar to refactoring, optimization also aims to preserve all original behaviors. However, instead of focusing on maintainability, its goal is to reduce time or memory usage.

| | **Adding a Feature** | **Fixing a Bug** | **Refactoring** | **Optimizing** |
|---|---|---|---|---|
| Structure | Changes | Changes | Changes | — |
| New Functionality | Changes | — | — | — |
| Functionality | — | Changes | — | — |
| Resource Usage | — | — | — | Changes |

Despite structural modifications in three of the four reasons for changing software, the amount of behavior that changes is relatively small compared to what remains the same. While ensuring changes meet new requirements, the greatest challenge in software development is preserving existing functionality and identifying which behaviors are at risk. This leads to three critical questions: (1) What changes need to be made? (2) Are the changes implemented correctly? (3) Did the changes break anything? Since the latter two are particularly difficult to answer, developers often approach the first by minimizing large changes, such as creating new methods or classes. However, this can eventually lead to maintenance issues and a reluctance to modify the code at all.

*Chapter 1 Experience*

In Unity game development, the engine provides developers with immense freedom: How should the character's movement and collision be detected? By whom? How should the game state be controlled? Which unit(s) should be responsible for spawning new mobs? The list goes on. Even with thoughtful design considerations, the codebase can quickly turn into legacy code.

When introducing a new feature, developers may easily modify existing structures to fit the new requirements. However, this approach becomes unsustainable as the structure grows more difficult to change while preserving all original behaviors. At this point, the next stage of legacy code emerges—features being directly implemented within existing files. This, in turn, leads to hesitation in development, as developers struggle to ensure that new features are correctly implemented without unintended side effects.

To address these challenges, multiple team members may step in to help by distributing the workload. However, as stated at the end of Chapter One: “Maybe we can hire more people so that there is enough time for everyone to sit and analyze … Surely more time and scrutiny will make change safer. Or will it? After all of that scrutiny, will anyone know that they’ve gotten it right?”

## Chapter 2: Working with Feedback

*Chapter 2 Summary*

System changes are usually made in two ways:

(1) Edit and Pray:

> This is the most common approach taken in most companies and is considered the industry standard. In this method, developers try to understand the code well enough before making modifications, followed by the testing team examining the updated code. While this may seem like a careful process, the codebase remains vulnerable because testing is designed after the modifications. As a result, tests often serve more to confirm correctness than to actively try to "break" the system.

(2) Cover and Modify:

> This approach creates tests before any modifications, establishing a protective layer that prevents potential negative effects from spreading through the codebase. Compared to Edit and Pray, the testing cycle here is much shorter since developers don't need to wait for feedback from the testing team.

This method, known as regression testing, sounds ideal but is difficult to implement in real-world scenarios. Since regression tests are created prior to development, they generally focus on application-level or high-level functionalities. This makes it time-consuming to run. If developers make additional changes during this time, it becomes challenging to trace which modification caused a test failure. In contrast, unit tests, which isolate individual components, provide faster feedback and can pinpoint specific issues. A clear definition of a unit test emerges: it should run quickly and isolate errors. Therefore, a test is not considered a unit test if it (1) talks to a database, (2) communicates over a network, (3) interacts with the file system, or (4) requires a specific environment to run. While these types of tests are important, they should be kept separate from true unit tests to preserve the core purpose of unit testing.

When creating unit tests, developers often need to break dependencies between classes—for example, when a class requires a database connection or servlet. Breaking these dependencies can introduce potential issues, like adding unnecessary code to the production environment, but it allows for unit tests that can catch more severe bugs. Writing unit tests for legacy code can be particularly challenging at first, but as test coverage increases, the codebase gradually becomes fully test-covered. This process follows the Legacy Code Change Algorithm, which will be explored in later chapters:

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

*Chapter 2 Experience*

Creating unit tests in legacy code feels like a blend of Edit and Pray and Cover and Modify. The first few steps in the Legacy Code Change Algorithm rely heavily on understanding the existing codebase, and breaking dependencies fundamentally resembles Edit and Pray. However, the book will delve deeper into these concepts in later chapters, potentially offering solutions for each.

One particularly interesting aspect of the algorithm is the "break dependencies" step. This introduces new opportunities for flaws due to the complexity of interconnected classes. Striking the right balance between breaking dependencies and maintaining clean code can be difficult to achieve. This will undoubtedly be a primary focus in the following chapters.

When working on course projects or machine problems, the auto-grader functions similarly to the Cover and Modify approach. Tests are written independently by the staff before student submissions, providing a pre-existing safety net. Well-designed auto-graders can even pinpoint errors down to the exact function and scenario, giving students precise feedback. With this information, students can quickly identify and fix bugs before submitting their work.

## Chapter 3: Sensing and Separation

*Chapter 3 Summary*

In an ideal system, when starting on development of a project there is not much prior planning that is required. Developers can just dive into development and start building features; however, the larger the system is the more impossible. Classes will depend on each other, such that most objects could not be isolated out alone. This makes testing fairly difficult. When creating unit tests, developers often need to break dependencies in order to create individual test harnesses especially when testing is not implemented concurrently with development.

The two main reasons to break dependency are sensing and separation. In sensing, developers can not access the values the code requires. In separation, developers can not get a piece of code into the test harness. The book continues to use an example of the NetworkBridge class where the constructor takes an array of EndPoint objects and a formRouting method. This particular class faces both sensing in unable to sense the effect of the calls to the method, and separation in it unable to be tested separately from the rest of the application (without support from hardware endpoints).

Fake objects are used to replace the dependency objects. The content of fake objects could be confusing in that some methods are only used by certain classes, causing them to have multiple faces. For non object oriented languages, an alternative function that records values in global data structure can be used for tests. Another technique similar to faking an object is mocking it. These mocking objects contain setExpectation and verify methods for the test to call and confirm the correct flow is executed.

*Chapter 3 Experience*

In a previous internship project, I have worked on the implementation of tests for the code I have completed. It utilized MockK for Kotlin. Despite similar to the Mock Objects section of this chapter, it is more of a fake object idea. Therefore the difference between languages could really confuse developers. In my particular project, mock objects are used almost for all classes due to its simplicity in setting up the testing environment. While not receiving 100% coverage, it was able to cover over 98% lines I have contributed.

In the CS427 group project milestone 5, we have used a combination of both approaches. For all scenarios, the team has created a fake activity without prior content and intent information to set up the environment for testing, resolving the separation issue. In particular cases, fake users are created to provide information needed by the program, resolving the sensing issue.

Overall problems with sensing and separation could add significant difficulties to test developers. However, working between the dependencies could greatly improve the development team's understanding of their program and potentially reveal cases that the design misses out on. Therefore, aside from the extra layer of safety the tests provide, it also helps to identify bugs before launching into production.

## Chapter 4: The Seam Model

*Chapter 4 Summary*

When shifting to the testing phase of development, programmers mostly find their code difficult to test. Thus, the workaround of development of tests while working on the actual code or implementation of a "design for testability" mindset have been adopted. However, the latter usually does not work. Just as in academic courses where programs are written to only be functional, professional code as well experience similar difficulties in having an enormous amount of dependencies.

When creating tests for specific methods, testers will try to first isolate the lines where the program communicates with external code. Seam as defined in the book is "a place where you can alter behavior in your program without editing in that place." Say wanting to test class A that has a line that calls to B.b(), a subclass for A can be created with an override of b() in replacement to B.b().

In response to various programming languages, there exists many types and use cases of seam. One example is in C where one could use macros (#define) to create seams which could potentially replace actions like database updates. Since seam should be in place, these macros could be factored to only be enabled when an enabling point like TESTING is defined.

Another variation is link seam, in programming languages like Java, a file could import `fit.Parse` and `fit.Fixture`. For which, a classpath environment variable can be used to alter where the program looks for import. Thus, a separate `fit.Parse` and `fit.Fixture` can be created in a separate folder just for testing.

One other mentioned seam is object seam. In object-oriented languages, methods can be called without the specification of the object. For example, a.spell() can be referencing A.spell(), Alphabet.spell(), or even B.spell(). Therefore, when a method takes in a parameter like `A a`. Seam can be applied with use of subclass and overrides. While this can be confusing particularly being something as well in production, it is as well an application of seam.

*Chapter 4 Experience*

Seam could be very powerful when working with legacy code where the application is very difficult to separate. Object seems to be the best choice in object-oriented programming languages that exploits only the language itself. This technique as well enables tests while providing an invisible boundary between the two. While link seam also provides similar functionality, it lacks application as it can only be used on more recent languages like Java where legacy code may not exist as much that requires the implementation of the seam model.

Milestone 5 of the CS427 group project also faces the challenge of separating classes from each other just as in any mobile application. An application of the seam model could be justifiable. For example, using object seam for API calls. As API keys are generally not required under the testing environment, a subclass of the LLMService could be created with a mocking api call. In an earlier milestone where the feature of different themes for different users is implemented, an approach with similar ideology to link seam is applied as the corresponding XML and CSS are applied from an enabling point.

## Chapter 5: Tools

*Chapter 5 Summary*

This chapter introduces some tools that could come in handy when creating tests for legacy applications. Refactoring which takes a long time if done manually has been transferred to be completed automatically. In the 1990s, Bill Opdyke shared the idea of automated refactoring that later inspired

others like John Brany and Don Roberts at the University of Illinois to develop the Smalltalk refactoring browser, being the state-of-the-art for a long time. By definition refactoring is "(n.). A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its existing behavior."

Choosing the right tool for refactoring with care can save a significant amount of time. Each company, team, and developer has a different purpose of refactoring their system. Thus, one could run small sanity checks to assure the outcome while assessing the tool.

(1) During refactoring, classes will be temporarily removed, and mock objects could serve as a great resource for substitution. Free libraries like www.mockobjects.com can then be used.
(2) Unit testing harness as mentioned in the previous chapter can be valuable to assure existing behaviors are maintained. Frameworks like xUnit can be implemented for simple and focused tests.
(3) JUnit tests can also bring value. This tool allows each test to be run independently from one another, further emphasizing isolation. Through overriding functions like setUp() and tearDown(), the test can set up the desired space and correctly adjust the flow between unit tests.
(4) CppUnitLite was created to overcome C++'s lack of reflection, which made CppUnit verbose and tedious. CppUnitLite allows writing tests using a simple macro-based approach. This avoids writing suite functions manually, making tests easier to write and maintain.
(5) NUnit is a .NET framework similar to JUnit. It uses attributes to mark test classes and methods. This makes it simple to identify and run tests using .NET's reflection.

Beyond unit testing, FIT and Fitnesse handle larger integrated tests. FIT runs tests written in HTML tables and colors cells to show pass/fail results, making specs and tests one and the same. Fitnesse extends FIT into a wiki, supporting collaboration and easy access to live, executable requirements.

*Chapter 5 Experience*

Refactoring could be the result of an unplanned project, and this process is very time consuming, requiring a deep understanding of the project and difficult to be done by multiple members. Refactoring tools as mentioned in this chapter could save a lot of time and put the team back to feature development sooner.

Many Java refactoring tools are already available for consumers, most of which have already been integrated into IDEs. Delphi and C++ also have dependable refactoring tools. In comparison to the time the book is written, C# has also received many tools. With the advancements in AI, refactoring has become simpler than ever, together with others like Copilot; however, its reliability in comparison to the comparingly manual tools is still a question out in the unknown.

## Part II: Changing Software

### Chapter 6: I Don't Have Much Time and I Have to Change It

*Chapter 6 Summary*

Creating a well rounded test with high coverage is going to take a lot of time. In many cases, developers may spend multiple hours breaking dependencies and implementing tests and just a few minutes after which to complete the feature. They will likely reflect that the tests are a waste of time. However, the impact of test harness is long term.

When returning to a piece of code a few years later, one could safely make the changes without worrying about breaking something up due to unfamiliarity. In other cases where team work on iterations of the codebase frequently, they will notice how they are saving a few minutes each feature that adds up and boosts efficiency. The new features will also be easier to build for the increase in quality of the code.

Nevertheless, the time may just now allow the implementation of the test. A developer could not understand the code well enough to change it. In some cases, there is just not enough time for the developer to write tests due to the deadline. They may think that they will return to it tomorrow, but this never happens as there are new tasks to complete.

One of the methods to add new features without breaking original dependencies is the sprout method. This method could be used whenever the code to be added exists in a place where tests are not available. The developer should (1) identify the location where the change is to be made. (2) If the change can be completed in a single sequence of statements, the change should be made in a new method with a call to it from the original code. (3) Determine the variables required, and make them the parameters. (4) Develop the sprout method using test-driven development. (5) Set up the call and the required process for the returned value. The sprout method excels in separating the new code away from the legacy code for which can be wrapped with tests immediately. However, this does mean the original source code would still be in limbo, and as the number of sprouts increases development would as well get harder.

One other method is to use the sprout class. Similar to the sprout method, this creates a new class in parallel to the original. The same steps can be followed to complete a sprout class. In comparison to the sprout method, the sprout class approach allows modification without invading the original class. Aside from the advantages sprout method has, this also allows the developer to later complete the refactoring of the original class and easily merge the two together. However, it also means more complexity on the conceptual level. When a sprout class is introduced, the developers face a new concept for perhaps a very small change in code.

Another technique is the wrap method and class. On the method level, say a new logging method log() should be added to a(), developers could create a new method alog() that calls log() and a() independently. Wrap method avoids the modification of the original methods completely, where sprouts still have to at least add a call to the new method. The drawback of the approach would be the poor names used for the new method, like alog() in the example above. On the class level, a new child class that extends the existing class can be created. Wrap class like this is similar to those of decorator pattern, where a parent controller triggers all methods in the related class. This can be used in cases where the new feature is completely independent and the original code should not be polluted or when the existing code is too large.

*Chapter 6 Experience*

When creating a new feature for any application, time always exists as a constraint. The deadline of other ongoing projects or any matter could greatly influence the developer's familiarity with the codespace they will work in. However, as proven by history, tests are still worth the time and effort to complete. Thus, to further simplify the process, new code is often introduced independent of the original work.

The sprout and wrap approach streamlines what developers have been using in adopting new features. In an internship project, the team has little to no familiarity with the incoming member. Thus, their first

project is often completely independent of the team. When adopting the new feature, the team will then use something similar to the sprout method to connect the repository. This saves a lot of time for the team as they would need to invest a ton of resources in creating a port for the intern's project. The intern on the other hand as well does not need to understand the large codebase completely and risk the potential of breaking the original work.

## Chapter 7: It Takes Forever to Make a Change

*Chapter 7 Summary*

Deciding to make a change, developing the actual changes, and moving it into production can take a very long time and various potential obstacles. The first part is understanding. A legacy system has a disorganized code base and is very difficult to dissect and locate where the changes need to be made and how. In comparison, a well-maintained system that is broken up into small, well-named, and understandable microservices would allow developers to ramp up knowledge way faster and propose the modifications.

Another aspect that takes time is the lag between the completion of the change and the time when it is launched into production. This is often caused by the building time that is required related to the size of the project. When this time gets larger, developers need to wait for the build to complete before receiving feedback for the change and further improvements. However, in most mainstream programming languages, developers could invest the time to have files built concurrently. While many files have dependencies, they do not all depend on each other. This could potentially decrease building time to as short as five seconds, which greatly reduces the lag the developer needs to wait.

The building time often is extended due to the dependencies being recompiled when rebuilding the system. This can be shortened through having the files call the interfaces instead of actually classes themselves. In most integrated development environments, there exists an option to choose a class and create its interface and have all calls to be referenced to the interface. In C++, the extract implementer can also replace the extract interface tool. This also adds a border between the caller and the actual file, such that when the file changes the calling methods still references the interface. This is the dependency inversion principle and allows files to be modified freely without affecting others.

In summary, as more interfaces and packages are introduced to the repository and dependencies are broken, the total cost of rebuild will inevitably increase; however, the average time will fall to promote a faster development cycle.

*Chapter 7 Experience*

When developing CS427's instrumented test the lag between receiving feedback and time when the test is implemented is fairly large. Due to the nature of an instrumental test, where it mimics the actions of an actual user, both setting up the environment and executing the animation will be time consuming. This increases the time needed to develop the tests significantly, while also providing visual insights to what has gone wrong during the testing from an user perspective.

Making interfaces out for classes can bring many other benefits aside from reducing the rebuild time. While it does complex the system, it adds another layer of structure to the system that allows other classes to be built on top. The interface also does not need to be updated and maintained as frequently as an actual class while providing an easy area of code to work with.

## Chapter 8: How Do I Add a Feature

*Chapter 8 Summary*

In general when adding a new feature, developers face the challenge and fear of modifying the original and complex code. Thus, often falling back to produce few features using the sprout or wrap approach as mentioned in chapter six. However, as introduced in prior chapters, creating a proper testing environment would benefit the codebase in the long term. Whereas if continue to be ignored, the legacy code will eventually have functionalities that are duplicates of new features that cause further issues.

The most powerful technique in creating a feature is test-driven development (TDD). This includes the steps of:

(1) writing a failing test case. This creates a task for the new feature to complete. Writing the test as the first step also avoids the chances that tests are created for the program to pass.
(2) get it to compile. Having the test alone would not compile as it likely depends on the feature to be actually implemented.
(3) make the test pass. This is the stage where the feature implementation takes place. There would be multiple small steps involved according to the specific function to be added. This could also include the development of an algorithm or use of some external library to support the task.
(4) remove the duplication. In most situations this is not required as features are implemented to solve the feature; however, if following the steps specifically, one would likely face the implementation to solve exactly the test case in step 1. In such cases, the code produced will likely need to be improved for generalization or removed on duplication.

This progress could also involve the section of getting the class under test. This will bring original legacy code under coverage and improve understanding before development.

Another technique that has been used in programming by difference. This mainly focuses on creating a new structure that relates to the original code with some kind of inheritance. This could be creating a subclass that overrides the original with the differences or creating an additional configuration class and use it as a parameter. It allows developers to make quick changes and a cleaner design in comparison to test-driven development.

However, there exists issues like the Liskov Substitution Principle (LSP) when overusing inheritance. One example of which is Square being a subclass of Rectangle. Rectangle would have both width and height; though, its subclass Square can be accidentally set with width of 3 and height of 4. When executing ```
Rectangle r = new Square();
r.setWidth(3);
r.setHeight(4);
r.area()
```, we can receive 9 or 12 instead of the anticipated 12 since r is a rectangle. This should be avoided whenever possible when implementing the Square class or attempt to call the overridden method in Square.area(). For example, an abstract class can be created for Square and Rectangle and two implementations of SquareArea.area and RectangleArea.area can be added.

*Chapter 8 Experience*

This chapter introduces formatted ways to add a new feature. While the test-driven development methodology is highly recommended from all sources, it is rarely practiced. This step requires the implementation of the feature to be completed before the test can properly compile. This adds a lot of complexity to the development phase. There are often interactions of implementation that occur that
```

have different classes each with varying functionality in each cycle. This will require the test to also be updated every time the code changes for variable or method modifications.

In other cases like CS427's weather app, instrumented tests are almost impossible to compile without the completion of the knowledge of element IDs, function title, intent passed into the activity, and many more. While the final coverage report still reaches near 80%, the order of development has been reversed for simplicity and ease of mind during the development phase.

DUE TO PAGE LIMIT, THE NEXT SECTION WILL ONLY SUMMARIZE REMAINING SECTIONS IN THE BOOK

### Chapters 9 to 24

Chapter 9 to 24 of the book continues in similar fashion as in chapter 6, 7, and 8 where the author provides real-world complexities and examples that emphasize on testing mindsets and taking incremental improvements in legacy code. In the earlier chapters of 9 to 12, approaches like splitting responsibilities, extracting smaller classes to prevent bloated code, and targeting tests are covered. Chapter 13 to 18 shifts the focus to making safe changes when implementing features and creating tests. This includes implementation of fast and frequent commits, creating tests for learning purposes, refactoring tangled methods and classes, and isolating external dependencies. The final chapters in part II then dives into the design mindset, motivating developers to have a structured approach and rely on the compiler. Most importantly, as stated in the last chapter, one should find joy and purpose in the work of refactoring ugly code for a morale boost.

## Part III: Changing Software

### Chapters 25: Dependency-Breaking Techniques

Part III devotes its entirety to chapter 25 to talk about numbers of ways to break dependencies. One of which is to extract and override factory method, getter, and call method. These overrides will move the control of the built in methods to the developer and allow tests to be implemented. Another is extracting interfaces like mentioned in chapter 8 to allow dependency injection, while being fully automatable. In most object oriented programming codebases, subclassing and override still serves as a powerful tool that avoids structural change to the original files. When having singleton or global service, one could also introduce a static setter when no other seams are available. This chapter includes dozens more techniques that enables the developer to create safe and incremental steps on progressing through the refactorization of legacy code.

## Appendix: Refactoring

In Martin Fowler's book Refactoring: Improving the Design of Existing Code, the idea of refactoring has been introduced and remained as a core method in improving code quality. The chapter focuses on the exact method which is deemed the most useful. This method allows developers to systematically split large systems into pieces for easy understanding and reuse. In comparison, poorly designed code will have long methods and low useability. Thus, engineers will often find themselves recreating the same functionalities over and over. To avoid such a situation, the extract method can be executed with: (1) commenting the lines of code one wishes to extract. (2) creating a new method with a good name. (3) call the new method from the old code. (4) migrate the old to the new method. (5) set up parameters and return values. (6) adjust method signature to fit the larger style. (7) execute tests to confirm accuracy. (8) clean up the workspace. This technique remains the core of refactoring legacy code.