Code Evolution

Objectives

One of the primary ways in which the "softness" of software manifests is in the ease with which it can be changed: in just a few minutes or hours we can take a piece of software and change, evolve, morph it into something different. This ease of change cannot be found in any other form of engineering outside software engineering. In this chapter we learn how to evolve software through techniques such as ***refactoring***. In the real world, it is extremely rare that you can afford to start a software project from scratch – most often you will face an existing, unfamiliar code base, and you need to evolve it into the software you want. We shall learn more about ***being productive with legacy code***, i.e., approaching, learning, and living with existing code bases. Inevitably, when changing existing code, the software engineer faces the risk of introducing new bugs; this is at its most annoying when, in the process of fixing a bug, new bugs are introduced inadvertently. In this chapter you will learn how to ***safely evolve code*** without breaking it. Since evolving code has its fair share of pitfalls, sometimes it turns out to be better to completely throw away some part of the code and rewrite it – we shall understand this ***trade-off between evolving vs. rewriting code*** and be in a position to decide intelligently when to do one or the other. We will conclude the chapter with a ***primer on software licenses***, which is the primary way to govern what code can be reused by others and in what way, without infringing upon the rights of the creator – this is something that every modern software engineer needs to be well aware of.

In this module, you will learn:

- How to “refactor” existing code to make it easier to understand without breaking it
- How to navigate an unfamiliar codebase
- How to decide whether to rewrite existing code or to slowly evolve it

- What kinds of software licenses exist, and what they allow you to do

Refactoring

Refactoring is an essential part of software engineering, and it presents a disciplined technique for evolving an existing body of code by altering its internal structure without changing its external functional behavior. There are other ways in which code evolves (e.g., by adding features), but what distinguishes refactoring from all other ways is that it is not supposed to change the externally visible functional behavior.

> *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."*
>
> Refactoring: Improving the Design of Existing Code, by Martin Fowler et al.

A software engineer refactors their code in order to improve it, and this can be triggered for a variety of reasons. The most common ones are to improve readability, make it more extensible and manageable, to reduce complexity, to improve performance, to reduce resource utilization, etc. Often refactoring is done in reaction to so-called "code smells."

Code Smells

A code smell is a sign that something is probably wrong with your code. This term was introduced by Kent Beck while working on the Refactoring book with Martin Fowler. There are two things that are important to keep in mind:

- A code smell is quick to spot (M. Fowler refers to it as being "sniffable"): by just looking at the code, it jumps out at you. For example a data clump, a duplicate class, a method that is too long, a method with too many parameters, etc.
- A code smell doesn't always mean there is a problem, it is just a warning flag. The programmer has the responsibility to look deeper and see if indeed there is a problem. In other words, the code smell itself is not a problem, but rather it is most commonly a symptom of an underlying problem.

To illustrate the smell-to-refactoring transition, consider the code smell known as *data classes*, i.e., classes consisting of all data and no behavior. When you read the code, you ask yourself what behavior is embodied in this class? And you cannot answer. Then you will likely find that other classes access the fields of this data class in order to perform computations or make decisions, instead of asking this class to do that. Those other classes may present a code smell known as *feature envy*.

For instance, say you have a `Rectangle` class with a field for `width` and a field for `height`. Users of `Rectangle` may need to compute the area, and do so by multiplying its `width x height`. In most cases, this makes the code stink, because it breaks encapsulation. Instead, the `Rectangle` ought to keep the `width` and `height` fields private, and provide accessors for them along with a `getArea()` method. We just performed what is called the *move method* refactoring. In general, whenever you make extensive use of fields from another class to perform any sort of logic or computation, that logic probably belongs as a method in that class.

There are many code smells, and we will not enumerate them all here. Instead, we want you to take another step toward being an auto-didact, and use web search to learn more about them. To get you started, we ask that you use refactoring.guru, Wikipedia, and web search to understand the basic smells listed below and search for code examples that illustrate them. You will find that most code smells fall into one of 5 classes (Bloaters, OO Abusers, Change Preventers, Dispensables, Couplers).

- Bloaters

    - Data clumps: A group of variables that are passed around together in different parts of the code, suggesting that they are a clump that belongs together.
    - Long method: If there are more than 10 lines of code (LOC) in a method, you should start asking yourself whether that's not too long.
    - Long parameter list: There are more than 4 parameters being passed to a method.
    - Large class: A class that has evolved to doing too much, having too much code, and too many fields and methods.
    - Primitive obsession: Using primitive types instead of small, custom objects (e.g., passing around primitive string instead of street addresses).
- OO Abusers
    - Alternative classes with different interfaces: Even though two classes do pretty much the same thing, their interfaces are different, their methods have different names, etc.
    - Switch statement: The same switch statement (or, alternatively, long "if - else if - else if … " statement) appears in different parts of the code.
    - Temporary field: Objects with fields that are not used all the time and, when not used, contain empty or contains irrelevant data.
    - Refused bequest: An object inherits from another object some code that it doesn't want, and thus overrides a method in such a way that the contract of the base class is not honored by the derived class (thus violating the LSP).
- Change Preventers
    - Shotgun surgery: If you want to change one piece of code, you then need to follow dependencies and change a bunch of other parts of the code (akin to having to use a shotgun instead of a rifle).
    - Divergent change: The reverse of shotgun surgery: a class that is commonly modified in multiple ways for different reasons (akin to shooting it with different rifles).
- Dispensables: Generally, these are things that are unnecessary and whose removal would improve code readability and/or performance.
    - Unnecessary comments
    - Duplicated code (class, method, etc.): Either identical code or very similar code appears in multiple parts of the software.
    - Freeloader (a.k.a. lazy class): A class that doesn't justify its existence.
    - Speculative generality: Code that was written in anticipation of some future use, which however is not being used.
- Couplers
    - Feature envy: An object accesses the data fields of another object more than its own.
    - Inappropriate intimacy: A class that inappropriately uses the internal methods (and sometimes even fields) of another class.
    - Message chain: A sequence of method invocations in the same expression, such as `Widget.foo().bar().baz()`.

As you have discovered by now, for each code smell there are a number of refactorings that you can do in order to fix them (and here is a cheat sheet you can use to get the mappings). So what is a refactoring?

Refactorings

The "dictionary" definition of "refactoring" is as follows:

- *noun*: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior
- *verb*: to restructure software by applying a series of refactorings without changing its observable behavior.

Refactoring consists of improving the code without changing the software's external functional behavior (or "behavior" for short). To do this effectively, refactoring consists of a series of small behavior-preserving transformations. Each transformation touches little code, but eventually the little changes can add up to a significant restructuring. Keeping each refactoring small helps reduce the likelihood that it goes wrong (and

perhaps introduces undesired behavior changes). Having a solid and thorough The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.

Refactoring is not another word for cleaning up code. It is a technique for improving the health of a codebase. Refactoring the code in small increments, and doing it fairly often, avoids the accumulation of *technical debt*. This concept reflects the implied cost of changing a given design and/or implementation to make it become better. As with financial debt, technical debt increases over time if you don't repay it, due to "interest". Refactoring often and in small increments is equivalent to paying your credit card debt on-time every month. If you want to learn more about technical debt, you can start here.

We've seen how we can use code smells to determine where to look deeper for refactoring opportunities. We looked at a simple `Rectangle` class example that was merely a data class that invited feature envy, and this triggered the thought of changing it by moving the `getArea()` method into the `Rectangle` class. Move method is an example of refactoring.

There exists an entire catalogue of refactorings and, similarly to code smells, they are grouped in categories. As we did for code smells, we invite you to use refactoring.guru, Wikipedia, and web search to understand the following basic refactorings and to search for code examples that illustrate them. This will provide you with the stepping stones for discovering many other refactorings on your own, and with time become proficient in using them.

- Composing methods
  - Extract method: You have a bunch of code that belongs together, as a group, so you turn it into a separate method and replace the original code with a call to the method.
  - Inline method is the inverse of extract method: You replace a call to a method with the body of code itself, because that code cannot justify its existence (e.g., it's a one-liner).
  - Substitute algorithm: You found a better way to do something that a method implements, so you change the algorithm inside that method and nothing else.
- Moving features
  - Move method: You move a method X from class A to class B, and then replace the original method with a call to B.X.
  - Extract class: You have a class that does more than it's supposed to, so you extract the added functionality and create a separate class.
  - Inline class: You transfer the feature of a class A to the class B that actually calls into this class, because A's individual existence is no longer justified.
  - Hide delegate: Instead of a method A.X that returns an object of class B, on which most callers then invoke method B.Y, you create a method A.Y that delegates the call to B.Y directly.
- Organizing data:
  - Replace magic number: You replace a constant with a symbolic name that is human-readable.
  - Replace array with object: You replace an array that contains different types of data with an object that has separate fields for those array elements.
- Simplifying conditionals
  - Decompose conditional: You replace a complex `if` or `switch` statement into method calls that are suitably named to better convey the logic of the conditional.
  - Consolidate conditional fragments: You replace a duplicated piece of code that appears both on the `then` and the `else` branch with that code pulled outside the conditional statement.
  - Replace conditional with polymorphism: Instead of a conditional that is based on the type of an object you implement subclasses for each case and rely on runtime dispatch to obtain the conditional behavior.
- Simplifying method calls
  - Rename method: Replace the name of a method with something more representative of what it actually does.

    - Parameterize method: Replace multiple methods that differ in just some internal value with a single method that takes that value as a parameter.
    - Introduce parameter object: Replace a set of parameters that often appear together with a single object containing those parameters.
    - Replace construction with factory method: Instead of an overly smart constructor that does more than initialize fields, you use a factory method.
- Dealing with generalization
    - Pull up method: Replace identical methods in subclasses with one corresponding method in the superclass.
    - Push down method is the inverse of the above: a superclass method used by only one subclass is pushed down into the corresponding subclass.
    - Extract superclass: two classes that have common fields and methods become subclasses of a common superclass that provides those fields and methods.
    - Replace inheritance with delegation: This is the inheritance to composition transformation we argued for in the "Inheritance vs. composition and containment" lecture.
    - Replace delegation with inheritance: This is the inverse of the above, when the LSP is satisfied.

Now you see how code smells signal places that might be ready for a refactoring. You don't have to refactor everything in one shot, you can just do a little bit at time. Since you don't change the program's external behavior, refactorings can often be done in a highly localized fashion. Remember to check out the cheat sheet that suggests, for each code smell, what possible refactorings you might consider.

Note that these transformations do not necessarily have to wait until refactoring time, you can even do them while writing the code. For example, you might see that you're writing comments that start smelling like unnecessary comments, so you can immediately refactor the code so that you don't need those comments. Remember, comments should not explain what the code does but rather what you are thinking.

Refactoring is an important part of writing code. When you want to add a feature to a program, and the code isn't conveniently structured, then it must have accumulated some technical debt (in other words, it is harder to change than you'd like), so you might choose to first refactor the program to make it easy to add the feature, and only afterward add the feature.

Using an IDE to automate/speed up refactoring

This section is here just to help you be more productive. We will not test you on this material.

We provide a cheat sheet for common Android Studio operations, to which you are welcome to add your tips.

The official documentation for IntelliJ, on which Android Studio is based, lists all of the available refactorings.

You can watch this short video from the IntelliJ authors demonstrating common refactorings, if you wish.

For a longer example, this video from the IntelliJ authors is a good example of refactoring legacy code into a more maintainable shape. You may think that the existing code in that video is unreasonably convoluted; it is not. Consider that the widely-used GCC compiler contains an if statement whose *condition* is 40 lines of code. Even its authors call that file “the equivalent of Satan”.

How to deal with legacy code?

The meaning of "legacy code" has evolved a lot. It used to refer to code or software that is no-longer supported or is incompatible with modern hardware. More recently, the interpretations of the term have evolved to be less specific. The most prevalent ones are "*source code inherited from someone else*", "*code that developers are afraid to change*" or more famously "*code without tests*" (Working Effectively with Legacy Code, by Michael

Feathers). They all point to a common characterization that is that legacy code is code that is poorly written and that developers have a hard time working with because they don't know if they will break the existing behavior.

Why keep legacy code?

Legacy code can be a common occurrence when dealing with **backwards compatibility**, which can mean support for old hardware, file formats, protocols, etc. Developers might be tempted to rewrite a lot of it because they may see a better way to achieve the same behavior. But there is usually a reason why the code is written in a certain way, and changing it might cause more problems than it solves.

Changing legacy code is particularly **expensive**. It requires a lot of time and effort to read it, understand it and make the appropriate changes while always checking that the behavior is still correct. Furthermore, the programmer might have to write the tests himself, adding on the time required to essentially rewrite a feature that already exists.

Another issue with legacy code that supports old technologies is **availability**. For example, rewriting an interface to an old piece of hardware might be hard to verify, as the availability of the specific hardware might be low. In that case, it is perhaps preferable to keep the code as-is to support those rare use cases.

Why modify legacy code ?

A maybe trivial reason to modify legacy code is **behavioral change**. If the current code does not behave as the user or client expects, has bugs, or does not have a particular feature, then changes need to be made. At the end of the day, “*behavior is the most important thing about software*” and the presence of legacy code should not get in the way of requirements.

By definition, legacy code is hard to understand and to work with, which qualifies it as "*bad code*". It is in the best interest of the programmer to fix the code in order to **increase the overall quality** of the codebase. Refactoring legacy code should not modify the intended behavior of the program, and should only be used to improve the structure and maintainability of legacy code.

Another type of refactoring is **optimization**. Very old code is often optimized to work with specific architectures. As hardware has evolved, these optimizations can become obsolete and even hurt the performance of the code on modern architectures. For this reason, legacy code can be evolved so that it is better optimized for the systems it is likely to run on, which often means performing the same task but in less time and/or using less memory.

Dealing with an unfamiliar codebase

Unless a project is written entirely from scratch, a codebase will contain some amount of third party code. Nowadays, this is very likely, due to the wide availability of third-party code and the chronic time restrictions faced by software engineers. It is almost certain that any programmer will have to deal with unfamiliar code, either to add new features, fix bugs or do some refactoring. The problem is to know how to approach it efficiently. This can be an overwhelming task depending on the amount of code, but there are a few methods that help to not get lost.

> "*In examining the tasks of software development versus software maintenance, most of the tasks are the same—except for the additional maintenance task of "understanding the existing product." This task consumes roughly 30 percent of the total maintenance time and is the dominant maintenance activity.*" (Fact 44, Facts and Fallacies of Software Engineering by Robert L. Glass)

The first step is to **read through** the code and understand its structure. Program comprehension is a topic that is often overlooked but it is very important in the day-to-day job of a software engineer. As a developer, you will read orders of magnitude more code than you will write.

One thing to do (described by Michael Feathers as *Exploratory Refactoring*) is to freely refactor the code during a fixed time period of 30 minutes without thinking about whether the code still compiles or not. Once the timer is finished, the changes are discarded. This method prevents the programmer from getting stuck too long on a particular problem and instead trying to understand as much of the code as possible.

Once mostly familiar with the codebase, the next step is to **write tests** for that code. The goal is to validate the understanding of the code while having the added benefit of checking that future changes will not impact the current behavior of the software. However, it can be hard to select what to test and how. A strategy is to start testing from the outside, which means writing tests from the highest level of the functionality, and then work your way down. This ensures a safety net that captures the overall behavior of the code. One way to do this is to use "*Approval Testing*" which is divided into four steps:

- *Arrange*: sets up the test environment.
- *Act*: executes the tested code.
- *Printer*: generates an output string from the effect of the code.
- *Assert*: compares the new output with the existing one.

This method does not require the programmer to understand everything about the code, but still provides the guarantee that behavior is not altered between modifications.

A final step can be to **clean up the code**. This can be achieved by writing documentation or using static analysis tools to identify common coding standard violations, and then fixing them.

Once these few steps are applied, modifying legacy code becomes a safer and less time-consuming task. It is clear that the process of cleaning up an unfamiliar codebase can take a considerable amount of time, but this is time that the team will not have to reinvest each time a change is needed. The key is to evaluate whether modifications are absolutely necessary, and perform those changes in small but controlled steps.

Dealing with legacy code is such a common occurrence that there exist a lot of different methods to go about it. You can find more advice here if you are interested.

Importance of documentation for future modifications

Decisions that seem obvious at the time they are made often get lost to history if nobody explicitly writes them down. This is particularly true in programming, because it can be hard to distinguish between personal preference and intentional decisions when looking at code. Consider the following questions:

- Why does the code use library A to parse JSON, and not libraries B or C?
- Why is the class D abstract, instead of an interface with default methods?
- Why does the layer E validate arguments, when it could be done in the upper layer F instead?

All of these questions could have answers that relate to personal preference. Maybe the developer who wrote the JSON handling was familiar with library A, so they chose it out of habit. Maybe the developer who wrote class D prefers abstract classes. Maybe the same developer wrote layers E and F, and put argument validation in one without much thought.

But they could also come from intentional decisions. Maybe library A is particularly fast at the specific operation the code needs, allowing the software to meet its performance requirements. Maybe the team agreed that class D will need to keep its own state in upcoming changes, which an interface cannot do. Maybe the next sprint will add an alternative layer F2, which should not have to duplicate argument validation from layer F.

Documenting changes allows your future coworkers, and your future self, to avoid having to guess. If you believe that library B would be a better choice to parse JSON because it has some convenient operations that your code needs, knowing that library A was chosen for performance reasons allows you to test the performance of B to

know how to proceed. It would be a waste of time to refactor the entire codebase to move to B, only to realize a month later that your software no longer meets its performance requirements in production. On the other hand, if it turns out library B meets your performance needs, then you can proceed with confidence.

One way to avoid future issues is to keep notes, sometimes called *Architecture Decision Records* or ADRs for short. They contain:

- The decision
- The date at which the decision was taken
- The context: why you took the decisions, what constraints you had in mind, why you believe the decision is the right one at that moment in time
- The consequences: what changes happened because of the decision at the time it was taken, such as refactoring the code in some way

When you revisit a decision, you can add a new record for the new decision, and edit the old one to point to the new one, so that future readers know that the old version is no longer in use. You can then link to these descriptions in code comments. Future maintainers can then know why the code looks the way it does.

For small decisions, you can also write them entirely in a code comment. For instance, “As of 2020-11-02, this class is expected to require its own state for the anonymous login feature that will be added soon, thus it is an abstract class and not an interface”.

Here is an example of decisions taken during the development of the UK Ministry of Justice’s “Form builder”: https://github.com/ministryofjustice/form-builder/tree/master/decisions

Primer on software licenses

All software belongs to their author, the author's employer, or the customer for which the code was written. The owner gets to decide what people can or cannot do with it. Usually, the things you can or can’t do are defined in a software license, that you can find with the code (in the case of open source software) or with the binary. These licences often contain a lot of legal terms and can be tricky to understand. Thankfully, in the open-source world, there are some license archetypes that are used by lots of projects, so you can be confident of your rights when you see one. We will present here some of these archetypes.

> If you want to use code you found somewhere (e.g., on the Web), make sure that you have the permission from the license owner, either through the terms of a license, or because the owner explicitly granted you the permission. Make sure that the usage you make of the code stays within the boundaries defined by the license.

This section can also help you find which license you want to use on your personal projects. GitHub has also created a tool (https://choosealicense.com/) to help developers find which license suits their project best.

GNU Licenses

The GNU project (GNU’s Not Unix) is one of the pioneering projects of the free software movement, backed by the Free Software Foundation (FSF). They promote software that grants four major freedoms to the user: the freedom to execute the software, to see the code (to understand how it works), to modify the code, and to redistribute modified versions of the software.

The GNU licenses defend this vision, and their entire text can be found on the FSF’s website.

Here is a quick look at the 3 main licenses they offer:

- The GNU GPL (General Public License), currently in version 3. This license grants the four freedoms described above to all users. In particular, the source code of the software must be easy to access to any person who has a binary copy of the software. This license is considered as a “copyleft” license, as it is "**contagious**". This means that any software that uses GPL code must use a license that grants the same liberties as the GPL. This may include programs that link to a GPL library (but there is legal debate on that point). This license gives the guarantee to contributors on a GPL-licensed project that their work will not be used to develop closed-source (proprietary) software.
  There is one main exception: if the modified version is only used internally (for example, you modify the code for your own use), you don’t have to share the source code with others. This means that you can, for instance, operate a Web service using GPL software without having to provide the source code to the users of that service.
  You are allowed to sell software licensed under GPL, as long as you provide the source code with the binary. The license grants your user the right to share the software with others, and you are not allowed to prevent them from giving your software to others.
- The GNU LGPL (Lesser General Public License) is a less strict version of the GPL. The main difference is that a program under LGPL can be used by or linked to by a program that is not under GPL or LGPL. This is a license that has been adopted by lots of popular libraries.
- The GNU AGPL (Affero General Public License) extends the copyleft of the GPL to server-side applications. As mentioned above, if you write a web server and license it under GPL, a company can modify it and run it on their servers without sharing the modified version, even if the servers are accessible to the public. If the server is under AGPL, however, the fact that the service is accessible to the public means that the source code must be accessible as well.

Permissive licenses: MIT, BSD, Apache...

These licenses are quite permissive, in that they don’t impose many obligations for the users of the software: they are free to use code in any way they want, including writing proprietary software.

Their differences are mostly in the protection they give to the authors of the software, and in the wording. Most of these licenses, for instance, require that the original author be acknowledged in the credits.

European licenses

All the previously mentioned licenses are written in English. This is usually good, as software is distributed widely. However, this may be an issue at the time of enforcement. If you want to enforce a provision of the Apache license in a non-English speaking country, for example, you may need to hire a lawyer to provide an official translation for the judges.

For this reason, alternative licenses exist. For example, the European Union has created the EUPL (EU Public License) that has been translated in all the official languages of the EU. It also defines the competent jurisdiction to rule in case of a problem. In France, the CeCILL license does the same, but is only available in French. Both these licenses are *compatible* with the GPL, meaning you can use EUPL code in GPL software, and GPL code in EUPL software.

Sublicensing

The license under which the code is licensed is decided by the owners of the software. An open-source project, for instance, that has hundreds of contributors, cannot change license unless all contributors agree, as each contributor is the owner of their contribution.

In 2015, the GameCube/Wii emulator Dolphin chose to change its license. This required contacting all contributors to get the permission. As some contributors could not be contacted, they had to rewrite some parts of the code to not use those contributions. They explained this process in a blog post.

When you take code from the Internet, you cannot change the license, unless the existing license gives you this permission. The code you write can indeed be under a different license, as long as that license is compatible with the license of the original code. Compatibility here means that one license doesn't forbid something that is mandated by the other one. For example, if you release a modified version of the popular GCC compiler, all the original GCC compiler will remain under its original license (GPLv3). Your additions can be under a different license, as long as it doesn't violate the provisions of the GPL (it cannot restrict the freedoms granted by the GCC, but it can grant new ones: for example, you can allow your modifications to be used in closed source software).

Some licenses explicitly allow *sublicensing*. For example, code under GPLv2+ can be licensed under GPLv2 and GPLv3 (this is what the "+" means here). Code under EUPL can be licensed under GPL, and so on. This means that, if you use EUPL code in your GPL software, you can simply switch that EUPL code to GPL, instead of having to maintain two different licenses for different chunks of your code.

Finally, the permissive licenses usually grant you almost all rights on the code, so you can sublicense them as you wish.

Multi-licensing

As the author of the software, you are often free to publish it under multiple licenses (unless, for instance, your employment agreement prohibits this). For example, MySQL is published for free under the GNU GPL license, but can also be sold under a proprietary commercial license that enables commercial distributors to change the software and redistribute it without the need to share the modifications.

Beware, however, that as long as you get contributions on a project you cannot multi-license it without the agreement of the contributors. And if your software is distributed with multiple licenses, contributors must agree to share their modifications under all those licenses. That is why big, open-source projects often require their contributors to sign an agreement on the code they write.