# Refactoring, A First Example

> When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.

## The First Step in Refactoring

The first test is always to build a set of tests. The goal of having these tests upfront is so that if you make a mistake, you will hopefully catch it in one of the tests. Ideally these tests are self-checking.

# Principles of Refactoring

The first purpose of refactoring is to make the software easier to understand and modify. Only changes made to the software to make it easier to understand is considered a refactor. Optimization, on the other hand, guises itself as a type of refactoring, but its underlying purpose is different. You are attempting to make the code more efficient rather than focusing on the maintainability of the code.

## The Two Hats

When using refactoring to develop software, you divide your time between adding function and refactoring. When adding functionality, you shouldn't be changing existing code. You can measure progress utilizing tests. When you refactor, you make a point of not adding function. Your goal is for existing tests to continue to work with minimal changes, such as updating the application interfaces.

## Why Refactor?

- improves the design of software
- makes software easier to understand
- helps you find bugs

## When to Refactor?

Setting time aside to simply refactor is something that isn't typically done. Usually it is done in small bursts when trying to accomplish another task. If you are having to duplicate code, modify existing code, when you need to fix a bug, or it is difficult to implement your current functionality maybe it is a good time to refactor. A not-so-obvious time to refactor is during a code reviews.

A lot of people skim over code reviews, raise their hands and say, "It looks good to me." Rather than doing that, why not take some time to make it a learning experience? > Code reviews help spread knowledge through a development team. Reviews help more experienced developers pass knowledge to less experienced people. Code reviews are also meant to give feedback, and if there is a better way of accomplishing the same task why not suggest it? You may have to deal with that code later on anyways.

Before you refactor the code to review, read the code and understand it to some degree and make suggestions (if not only to yourself). Now begin thinking of alternative routes of implementing the code, and if it is easy enough do it yourself and see how it behaves. Not only now have you potentially found a better way of performing the task, you now *understand* how the code works and can make better suggestions.

## Bad Smells In Code

- Duplicated Code
- Long Method
- Large Class
- Large Parameter List
- Divergent Changes
  - Divergent change occurs when one class is commonly changed in different ways for different reasons. If you look at a class and say "I will have to change these methods every time I get a new database; I have to change these four methods every time there is a new financial instrument," you likely have a situation where two objects are better than one.
- Shotgun Surgery
  - This is the same as divergent changes, but opposite. This is when every time you make a change, you have to make a bunch of small changes to a lot of different classes.
- Feature Envy
  - This is where a method seems like it uses a lot of data from another class, and should probably be a part of the other class.
- Data Clumps
  - Groups of data that are being reused can be bundled into a class for reuse and consistent accessibility and usage.
- Primitive Obsession
  - Grouping primitives using a base class to mask its type so different primitives that are used for the same thing can be grouped.
- Switch Statements
  - If you are reusing switch statements, considering placing it in a class.
- Parallel Inheritance
  - In this case, every time you make a subclass one one class, you also

have to make a subclass of another.
- Lazy Class
  - If you have subclasses that are not doing enough to pay for itself, it should be eliminated.
- Speculative Generality
  - Don't add API for something that will want to be added someday, add API for what is going to be applied now.
- Temporary Field
  - If you are using temporary variables that may be used that are associated with a class, consider putting them in the class.
- Message Chains
- Middle Man
  - Encapsulation often comes with delegation. This can go to far, if you have a class's interface that is used to defer you to another class, consider removing the middleman.
- Inappropriate Intimacy
  - If classes are too interconnected, consider removing their commonalities to another class or try to make the classes unidirectional.
- Alternative Classes with Different Interfaces
- Incomplete Libraries
- Data Class
  - Try to group methods that utilize simple data classes
- Refused Bequest
  - If you have inherited classes that don't need attributes that they are inheriting, you are doing it wrong.
- Comments
  - If you need a comment to explain what the block of code does, simplify the block. If the method is already extraced but you still need a comment to explain, try renaming the methods.

# Building Tests

## The Value of Self-Testing Code

If you have test that run automatically every time you compile, you can see if the changes you made affect the entirety of the program by looking at the tests. And if you did have an adverse affect on the program, a test should fail and you will know immediately that there is a problem and have a place to start in order to correct it.

## Adding More Tests

A good style to follow is to look at what the class (or section of code) is supposed to do, and test those things. Not necessarily test every public method. This will reduce the amount of tests you need to create while also keeping your project

fully tested. The key thing here it to be testing areas where you are worried things will go wrong. In this way, you get the most out of your testing.

A tricky thing about objects is that inheritance and polymorphism can make testing harder, because there are many combinations to test. If you have three abstract classes that collaborate and each has three subclasses, you have nine alternatives but twenty-seven combinations. You don't necessarily have to test each combination, but it a good practice to try and at least test the alternatives.

## Composing Methods

A large part of refactoring is composing methods to package code properly. Long methods are troublesome because they often contain lots of information that gets buried by the complex logic that usually gets dragged in.