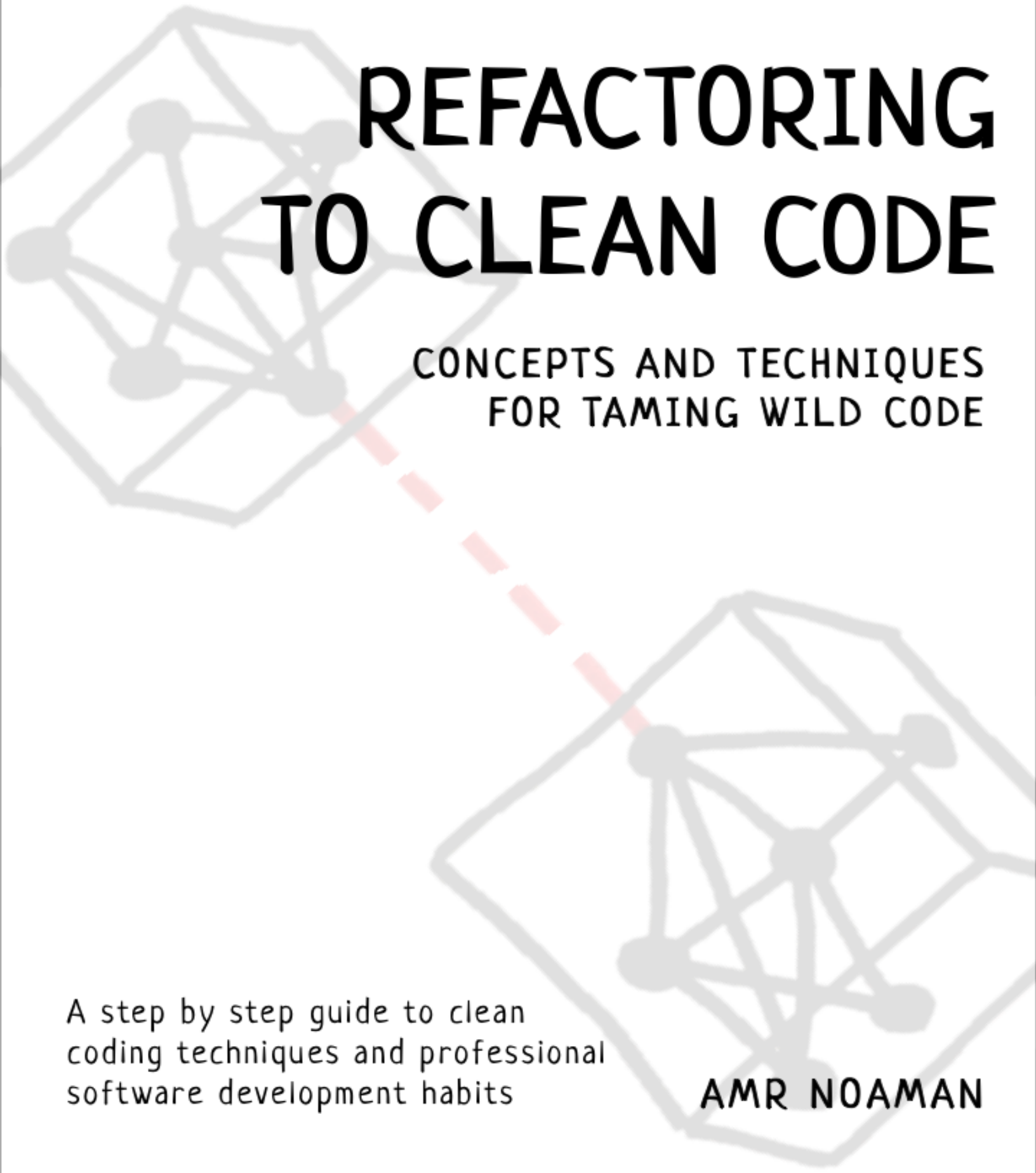


REFACTORING TO CLEAN CODE

CONCEPTS AND TECHNIQUES FOR TAMING WILD CODE

A step by step guide to clean coding techniques and professional software development habits

AMR NOAMAN

Refactoring to Clean Code

Concepts and Techniques for Taming Wild Code

Amr Noaman

This book is for sale at http://leanpub.com/RefactoringToCleanCode

This version was published on 2018-07-15

Tweet This Book!

Please help Amr Noaman by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought 'Refactoring to Clean Code' !

Contents

Introduction and Background

The story of this book started long ago at 2009, while helping organizations and coaching teams to adopt more agile ways of work. I faced lots of problems working with teams whose code is extremely poor. Such teams suffered from frequent code failures and intermittent outages. Code was literally stopping them from transitioning to shorter iterations or even smaller releases.

This book is an assimilation of the large amount of good advice available in books and online; summarized and organized into a roadmap. It is an easy read for juniors and seniors responsible for maintaining software regardless of their business domain or technology.

This book is **not** a reference for all clean coding techniques and refactoring best practices. I have intentionally left over many of the good advice and tools, because in some cases they are not universal and may only apply in special cases, or because I had to chose only a small set of techniques. This small set of techniques will help teams kick off refactoring rather than overwhelm them with so many ideas.

Why refactoring matters?

In the 90's, 70 billion of the 100 billion expenditure on software products were spent on maintenance; and 60% of which was consumed to *locate defective code* [1]. Using simple algebra, reducing the amount of time to locate defective code by 30% would reduce the overall expenditure on software by 15%, which is a huge improvement.

How to refactor - the "old" way

Over the course of several years, I have struggled with teams to refactor their application code to be easier to understand and cheaper to modify. These attempts followed one or more of the old-way patterns described below. I have to say that most of these attempts failed or achieved very little value.

1. Re-write the whole

This solution looms like the easiest solution from both technical and management point of views. If code is cluttered and causing lots of trouble, trash it and start from scratch. The question is: If you start from scratch, what makes you confident that you'll not hit the same wall again?

The question is: If you start from scratch, what makes you confident that you'll not hit the same wall again?

It's like the next picture of two accidents. Car capabilities are different, one is ordinary sedan car while the other is an expensive sports car. However, the accidents are similar, because the *driving habits* are also similar.

Bad driving habits may cause very bad accidents, similar to bad development habits, which may also lead to very poor code with lots of technical debt

So, if you would like to start over again and not hit the same wall, you have to change your development habits, rather than change the code. But, why not change development habits while still maintaining the same code? This would definitely save us huge costs of development, and this book gives you a roadmap how to do that.

2. Technical hero

This is very common: Hire a highly qualified technical person and give him full authority and power. People usually expect that such a person is magically capable (someway or another) to fix things up and turn the code from cluttered and spaghetti to structured, readable, understandable and changeable code.

There are two fundamental problems in this approach:

1. Such persons are very expensive and may not be affordable by many organizations. If this is the only way to go, then refactoring will become an expensive activity monopolized by some wealthy teams. What we aspire, instead, is to enable all software people to refactor their own code themselves; regardless of their seniority or technical expertise. This is more sustainable, though requires more time and effort.
2. Beware that a technical hero may well take wrong refactoring decisions, especially if nobody is reviewing his/her work. Such decisions may lead the whole team to hit the wall and waste so much time and effort. Having full power and authority may also lead very easily to what I call a Technical Glut Trab - described below.

3. As per the book

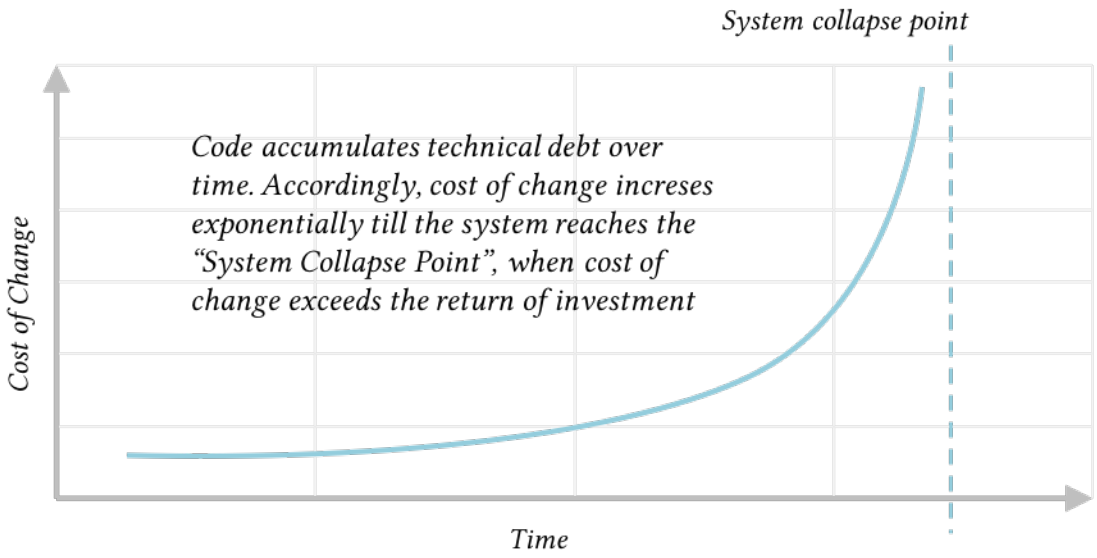
This is another pattern of people who tend to follow advice *as per the book*. There are so many excellent books about clean coding, refactoring, design patterns, etc. While all these books are extremely helpful, they might not be used as step-by-step guide to refactoring poor code. Rather, they are like a buffet of so many good advice and proven practices and techniques; however, they don't tell you which of these practices and techniques has of higher priority and would realize higher value in *your* specific case.

Even this book. Although I tried to give a generic roadmap which should apply to any type of project (I hope). Although I did that, I believe that you should think and question and experiment a bit before you blindly follow the advice in any book.

4. Try-then-retry

This is another pattern which I see very frequently. If you have large amounts of technical debt, try to fix a part of the product once there is a change request related to this part. If it works, then fine. If not, retry with the next change request.

Although this is the most pragmatic approach to handling technical debt, but it incorporates a lot of risks postponing and accumulating technical debt over time. At one point of time, the system may reach what we call a *system collapse point* when refactoring code due to one change may become so expensive and not affordable:



Cost of change increases exponentially due to accumulating technical debt

Why refactoring fails?

I have documented some of the root causes of these failures in an experience report published 2013 [2]:

1. Vague and hazy objectives

The first reason was due to hazy and unclear objectives. We didn't specify what "good code" looks like or how to measure "goodness". The decision of what to refactor or whether or not the code is "good" was based on the gut feeling of the engineering staff, rather than clear and indicative measures.

2. Start with covering poor code with fragile tests

We always thought that an automated test suite with high test coverage is a safety net for a team refactoring poor code, because it picks any side effects or regression issues caused by refactoring old code. Without such safety net, most of the time, we didn't have the courage to change the code and integrate it to the mainline. However, in all three projects, **covering poor code with automated tests turned out to be not possible due to many factors**. In two projects I worked with I faced these challenges:

- Product code lacked clear system interfaces and suffered from scattered business logic in all layers, including the database.
- Writing automated tests incurred very high costs, not only in development, but in toolset and training.

In time, some team members viewed automated tests as an impediment to refactoring!

3. It's non of the managers' business!

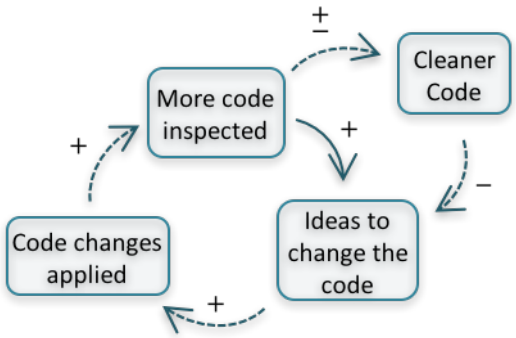
Technical teams had the attitude that refactoring was "none of the managers' business". They did not spend any effort to involve busy managers and get their support. This attitude created a counter effect from managers towards refactoring and refactoring effort which was viewed by senior managers as a non-value adding activity and was only allowed due to pressure from the development teams. Once the planned time for refactoring elapsed, management became more and more resistant to spending any more effort on refactoring.

Managers will not sponsor an activity which they cannot track or control.

4. The Technical Glut Trap

Sometime, technical teams indulge in deep technical review and merciless code refactoring with no limits to their technical imagination and creativity.

As shown in the next figure, this may form an endless positive feedback loop and refactoring might never end. This dynamic intensifies when the outer dashed loop pushes in the same direction and the changes deteriorates the code rather than enhance it.



The more you change the code, the more ideas you'll get to further change the code. Unless you keep your changes small and integrated frequently, the cycle will keep escalating and the code will be finally thrown away. The dashed lines are another positive feedback loop. If changes make the code less clean (or less readable and more complex), this may escalate the effect of the inner positive feedback loop.

One very frequent example of the technical glut trap is when the team focuses more and more on refactoring to patterns to make the code more "robust" with respect to patterns, but less readable and more complex to maintain!

5. Unsustainable Development Pace

In early refactoring failed experiments, the development pace was not sustainable by developers, managers, or customers. Teams were developing new features, fixing bugs, and supporting customers on one branch while on another branch, they were applying large refactorings, experimenting with design patterns, doing architectural spikes, and other refactoring fixes.

Managers and customers started to receive delayed fixes and prolonged plans for new features. To accommodate release fixes, the development team had to apply the same fix twice, once on every branch. After a while, team members couldn't sustain the huge effort of maintaining multiple versions of the code. Managers, on the other hand, stopped supporting them because they didn't see tangible results. Eventually, the refactoring effort was discontinued and the refactored code branch was abandoned altogether.

Is there a better way?

After these failures, I realized that there has to be another way. This triggered my research and experiments with some volunteering teams till we reached what we believe to be a better way.

This new approach of refactoring is put in a roadmap format, and is explained in detail in the rest of this book. But, before we do that, these are some guiding principles which governed my thinking when I designed this roadmap.

Simplicity

Throughout my development live, I always detested complex (and probably genius) solutions. One reason is that I always considered my self an average developer who is spending hard time trying to understand complex solutions developed by other "geeks". Another reason is that I believe simple and effective solutions need not be explained in more than 10 minutes. If you can't do that, then probably your solution is complex and it might be better to throw it away and look for another solution with a fresh mind. Guess what, in all cases which I can remember in my 16+ years of development, this worked and I managed to find simpler and more innovative solutions.

Now, I find this fact, that I'm an average developer who doesn't feel good towards complex solutions; I find this a gift really. Because, when I started thinking of better ways of refactoring poor code, I thought of things which "average" developers (like me) can understand, do, and appreciate.

This is why you may find most of the information in this book is simple, and probably this is why it is very effective!

Disclaimer: Most of the information in this book is simple, and probably this is why it is very effective!

Sustainability

Refactoring should always be sustainable for developers, managers, and users.

It's unsustainable for developers to always stay up at night to carry out refactoring tasks. It's unsustainable for managers not to see or "feel" the value of refactoring. It's unsustainable for users and customers to wait for months with no updates or fixes till developers finish a refactored version of the product.

The refactoring roadmap in this book is designed to be sustainable for all; for developers, managers, customers, and end-users.

1. Before You Start - Prepare a Healthy Environment

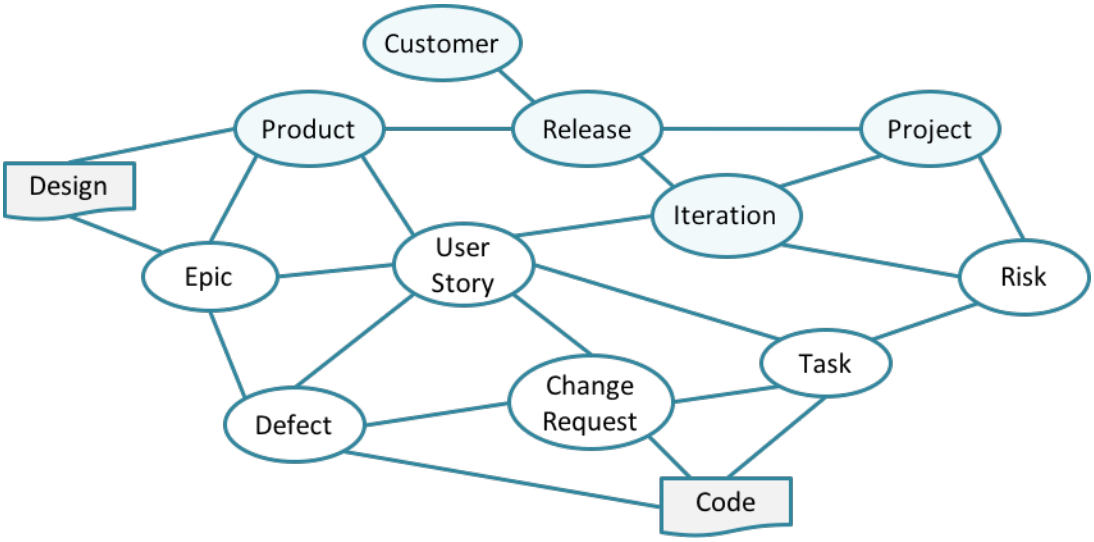
1.1 Work item tracking

Work items (aka issues) are *tasks or things which we do at work to reach a greater goal or carry out a value-adding job.* Examples of work items are: user stories, epics, bugs, tasks, change requests, etc. Work item tracking (aka issue tracking) is an important topic in configuration management and a basic constituent in a healthy software development environment. However, many software teams may not have a clear idea about what work item types and workflows they should track, what state information they should collect, or what kind of relations to be maintained between work items.

One sign of a mature development team is that they start creating new work item types and adjust their workflows to suite their specific needs. They also use it not only to track their everyday development activities, but also to generate rich progress reports, build traceability networks, do impact analysis, and improve their internal process.

1.2 End-to-end traceability

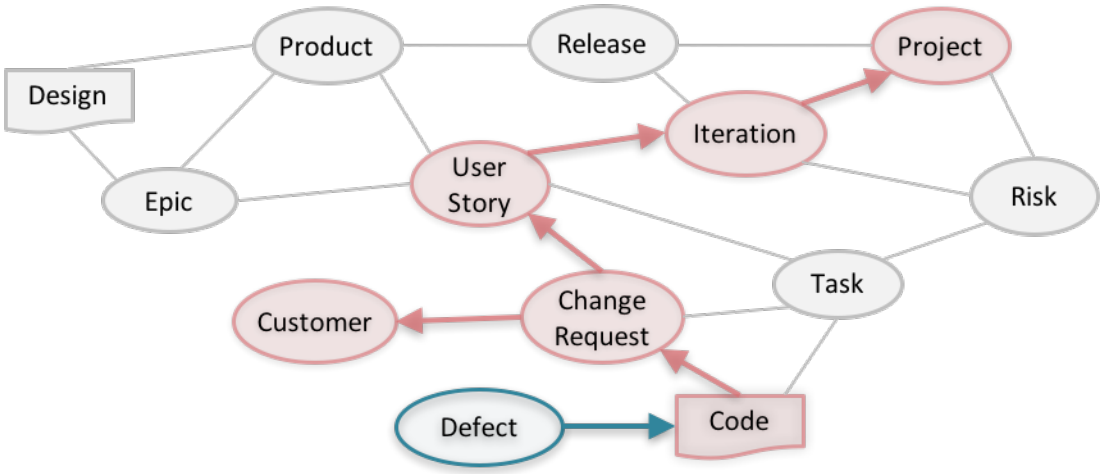
Traceability is a dynamic network of relationships which you can use to trace or relate work items and software artifacts:



Example traceability network which should possible links between a group of work items

As you can see, any artifact can be related to so many other concepts, artifacts, or work items. A user story can trace to many other work items (like tasks of defects), trace to other information (like iteration or release), or trace to physical artifacts (like code or design document).

Traceability enables the team to study the impact of a change and assess its costs, risks, and root-causes. In the scenario below, a defect reported by a customer is traced back to the defected code. This code is traced back to the reason of its existence, which is found to be a change requested by the customer sometime ago. In turn, more information about the change request can be deduced:



Defected code is traced back to the reason of change and hence the root cause behind the defect

This is particularly important for a team trying to refactor and enhance existing code. The team should be able to distinguish regression bugs resulting from refactoring old code from other bugs resulting from developing new features or fixing bugs.

1.3 Continuous integration

A Continuous Integration (CI) server is the companion of the development team. There are so many repetitive tasks which can be automated one way or another and a CI server will relieve the team pains of running and monitoring such tasks. As a result, the team will keep their focus on development and refactoring efforts.

1.4 Starting with characterization (aka pin-down) tests

Micheal Feathers first introduced the idea of **characterization tests**, which are **tests "that characterize the actual behavior of a piece of code"** [18]. The aim of adding these tests are not to discover bugs, but rather to understand how the system behaves. This is why characterization tests are particularly useful if you are maintaining code which you have little very little experience with and know very little about its behavior. **Adding some tests around the area you're refactoring helps you understand and preserve the actual behavior of the code**.

Here are some examples for tests that you may consider:

- Does calling a specific interface requires logged in user?
- Does it work with null parameters?
- Should objects be initialized?
- What possible values which will not throw errors?
- What return values are expected?
- Does this combination of parameters raise an error?

Adding some of these tests will help you *characterize* the code under investigation and will help you *pin-down* and preserve some of its key behaviors.

While you're introducing these tests, you may find weird behaviors or even bugs. In this case, be cautious when fixing bugs or changing things because you still can't anticipate side effects. Moreover, some bugs might have become featured in the system, and fixing/changing them may not be accepted by end user! My advice is to wait until you have good understanding of the code.

1.5 Ground rules for sustainable refactoring

Before you start, this is a final step in preparing a healthy refactoring environment: to agree on this set of ground rules. These ground rules are necessary to alleviate some of the issues discussed earlier in the 'Why refactoring fails?' section:

> **Rule 1: Refactorings are committed daily on the development mainline, not a dedicated branch.**
>
> Maintaining two branches of code is a nightmare. Development teams cannot sustain manually integrating and merging features, fixes, and patches from one branch to the other, especially after applying profound refactorings in one of them.
>
> This is why I deliberately advice teams to integrate refactorings on their mainline of development not on a dedicated branch. This is not easy, but it is possible.

> **Rule 2: Timebox an agreed upon percentage of the development effort to refactoring.**
>
> Agree on this in advance. Negotiate this with senior management if necessary. Let refactoring be a development habit rather than an unpleasant mandatory task to do.

Rule 3: Refactoring effort and outcome should be visible to everybody, including management.

This is crucial to keep the momentum and maintain the sponsorship of refactoring as an expensive activity, especially when refactoring old systems with tons of technical debt.

Rule 4: Any change *must* be reviewed.

Review can take place either by pair programming the change, mob programming the change, or peer reviewing it later on.

Rule 5: Large refactorings are not allowed throughout the roadmap.

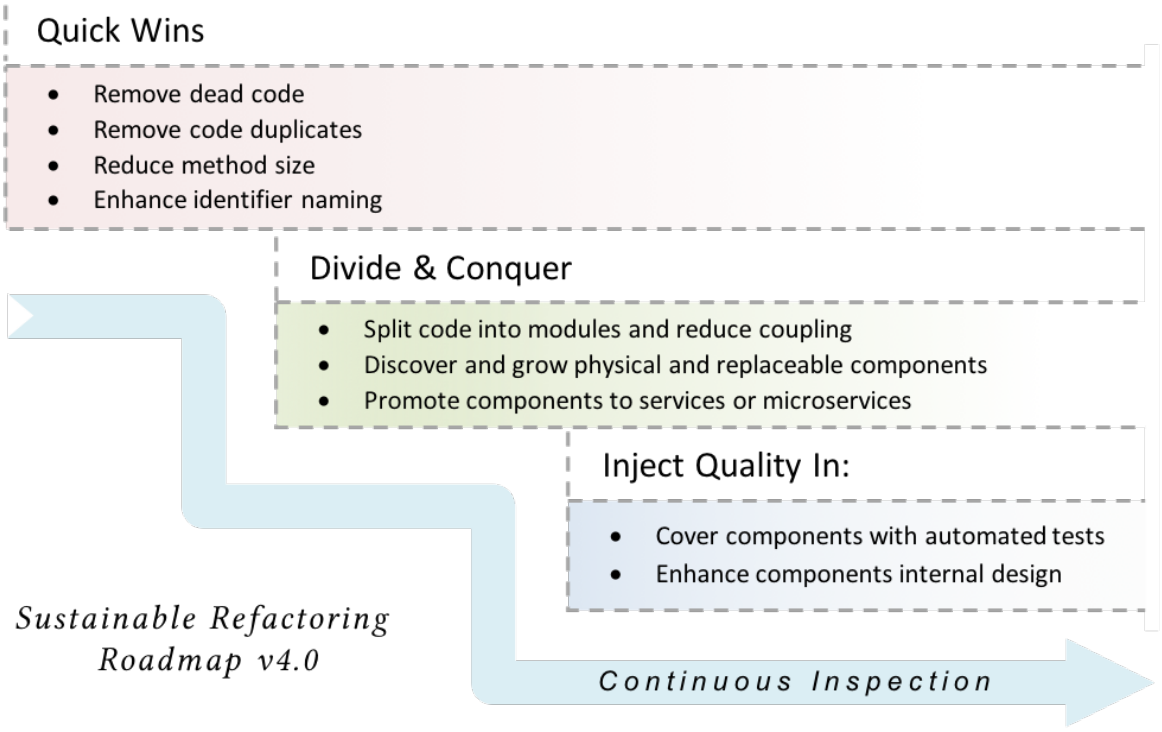
Example large refactorings is to introduce an architectural enhancement which may require changes in many places in code. These refactorings need special care and must be handled differently.

These are two rules of thumb to detect whether or not a refactoring is large:

- A refactoring is large if you spend much time thinking and designing before you start working on it
- A refactoring is large if it takes more than 20-30 minutes for you to get things up and running

2. Refactoring Roadmap Overview

In a nutshell, start with the highest value-add and least risky fixes, then work on re-organizing code chunks into components, and finally wrap everything with automated tests. In all stages, automate checks to make sure what is fixed will remain fixed.



Stage 1: Quick-wins - Simple and least risky enhancements

In this early stage of refactoring, we rely heavily on tools to detect and fix issues with code. As appears in the roadmap, the kind of issues we are tackling involve the whole code base. **So, when we *remove dead code* or *remove code duplicates*, we do this for the whole code base, not part of the code.**

Working on the whole code base magnifies the impact and signifies the improvement. You may put it this way: It may be better to move the whole code one foot forward, rather than to move part of the code a thousand feet forward.

It may be better to move *the whole code one foot forward*, rather than to move part of the code a thousand feet forward.

From my experience, teams working on the quick wins stage for a while usually start feeling “more confident” in enhancing the code and applying refactoring ideas. They also have better grasp and ownership for the code. In my opinion, this is one of the most important by-products of this approach to refactoring.

Stage 2: Divide & Conquer - Split code into components

After getting rid of most of the fat during the last stage, we gradually start introducing structure into the code. The key idea is to move *similar* code together and let components with clear interfaces emerge gradually.

In most cases, you may find that code is already organized into high level modules. However, such modules may have grown in size and collected so much responsibilities for a middle-size code module. Part of these responsibilities may be perfect candidates to move to another module or form a new one.

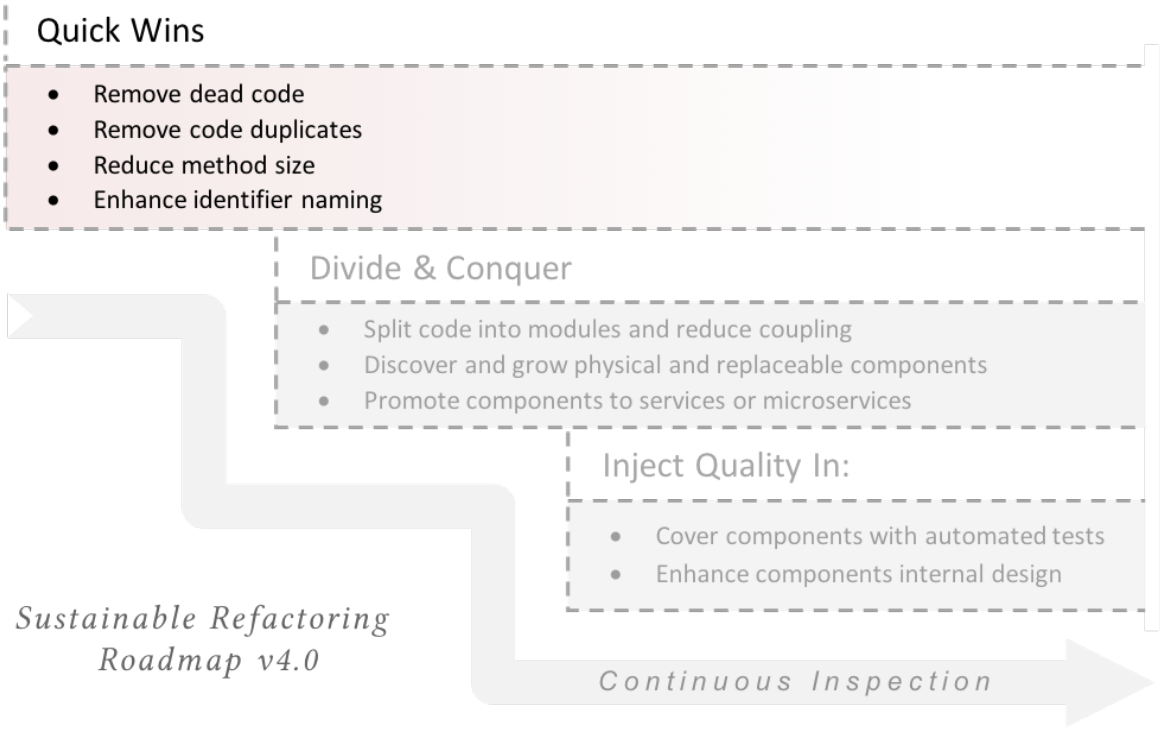
Stage 3: Inject quality in - Cover components with automated tests

Automated tests is one of the key enablers of quality code. From a development perspective, automated tests empowers the team to refactor entangled code safely. From a customer perspective, automated tests enables fast changes by detecting regression bugs early in the process. At this stage in the roadmap, After working on splitting code into components with clear interface methods, it is a perfect timing to start covering components with automated tests.

Continuous Inspection - Ensure what is fixed will remain fixed

Throughout the roadmap, enabling continuous inspection is key to sustainable refactoring. Continuous inspection ensures that the newly-introduced development habits are kept and enforced and ensures that what is fixed will remain fixed. In other words, it ensures that we do not hit the same wall again.

3. Stage 1: Quick-Wins



Stage 1: Quick-wins - Simple and least risky enhancements

3.1 Remove dead code

> **Deleting dead code is not a technical problem; it is a problem of mindset and culture**
>
> *- Kevlin Henney* [1]

Dead code is the "unnecessary, inoperative code that can be removed without affecting program's functionality". These include "functions and sub-programs that are never called, properties that are never read or written, and variables, constants and enumerators that are never referenced, user-defined types that are never used, API declarations that are redundant, and even entire modules and classes that are redundant." [10]

[1] Alistair Cockburn, a famous agile author and one of the 17. The quote is from his book: Crystal Clear [20].

It is fairly intuitive (and was shown empirically) that as code grows in size, it needs more maintenance [4][9]. This can be attributed to three factors:

1. More time needed to analyze code and locate bugs
2. Larger code implies bigger amount of functionality, which, in turn, requires more maintenance
3. Software size has significant influence on quality. This was shown in an empirical study which researched the relationship between several project parameters (including size) and project quality. "Information systems project size was found to be a **significant influence on quality**. That is, as project size increased, project quality decreased." [9, p.6]. This, in turn, has significant effect on maintenance cost

What's evil about dead code?

There are many reasons why dead code is bad. First of all, it increases the code size, and thus, as described above, increases the maintenance effort [4][9]. For example, Do you recall a case when you kept staring at a piece of code trying to understand why it is commented out? Did you or anyone of your teammates wasted hours of work trying to locate a bug in a piece of code which turned out to be unreachable?

While these are very good arguments, there is another reason which makes removing dead code more compelling. Fortune magazine tells a story about Knight Capital Group (KCG), which "nearly blew up the market and lost the firm $440 million in 45 minutes". After investigation, it turned out that the code mistakenly set a flag which enabled the execution of a piece of dead code.

This piece of dead code "had been dead for years, but was awakened by a change to the flag's value. The zombie apocalypse arrived and the rest is bankruptcy" [5].

How to detect dead code?

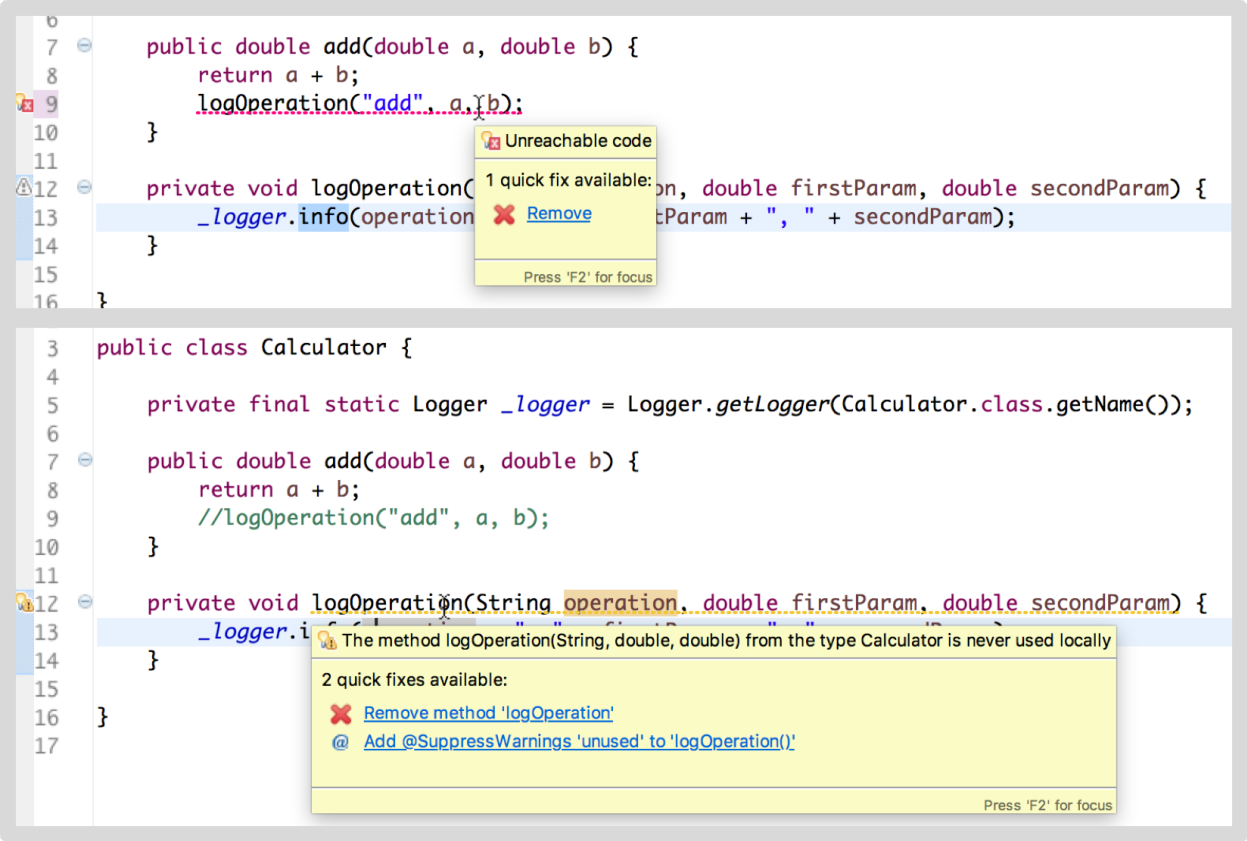
There are plenty of ways to detect dead code. It is as put by Kevlin Henney[2]: "Deleting dead code is not a technical problem; it is a problem of mindset and culture." [5]

To help you start, here are some ideas how to detect dead code:

1. Static analyzers

Static analyzers detects unused code by semantic analysis of static code at compile or assembly time. For example:

[2]Alistair Cockburn, a famous agile author and one of the 17. The quote is from his book: Crystal Clear [20].



Examples of unreachable code detected by Eclipse. In the first method, the method returns and the rest of the code is ignored. The second one is a private method which nobody calls in this class

Examples of unreachable code

These are also called *Unreachable Code* and it is only one type of dead code. There are many other programming errors which may result into unreachable code, like:

- Exception handling code for exceptions which can never be thrown
- Unused parameters or local variables
- Unused default code in switch statements, or switch conditions which can never be true
- Objects allocated and probably does some internal construction logic, but the object itself is never used
- Unreachable cases in if/else statements

All these cases are simple and straight forward to catch using compilers and static analyzers. However, if your program allows for dynamic code changes, reflection, or dynamic loading of libraries and late binding; in such cases, static analyzers may not help.

2. Files not touched for so long

One easy and very effective technique is to search for files that has never changed since a while. These are three main reasons why code files did not change for so long [5]:

- it's just right
- it's just dead
- it's just too scary

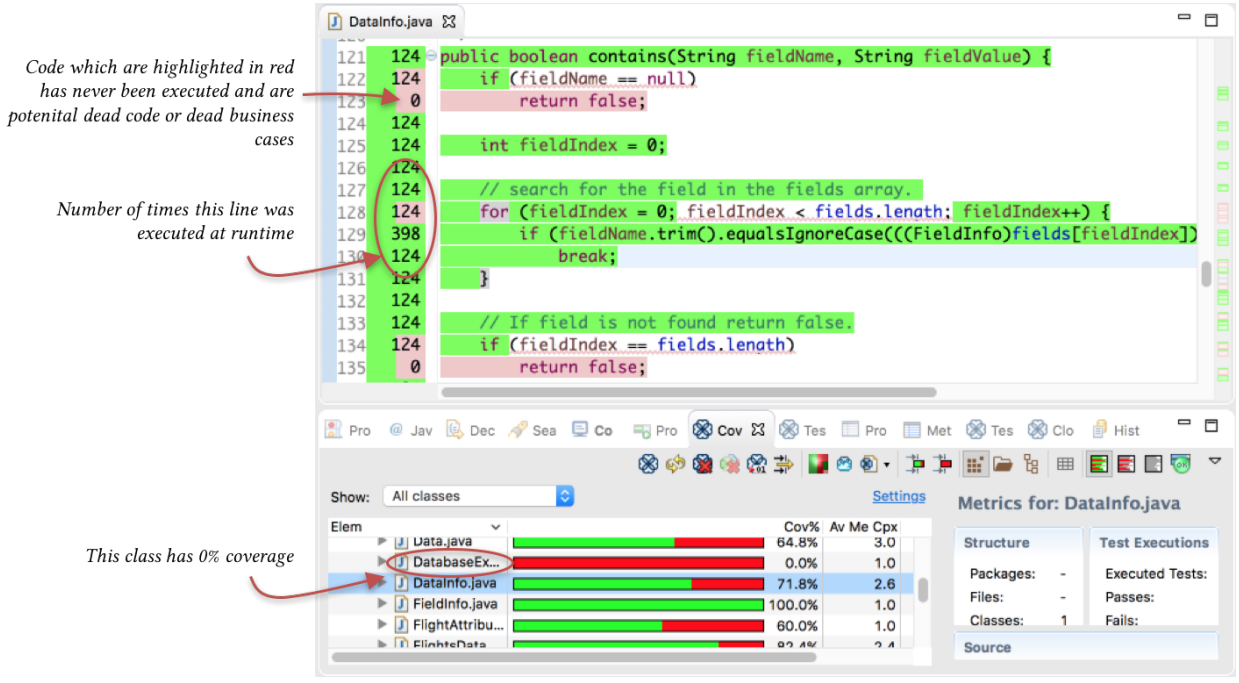
Of course, this requires that you investigate and check whether the code is dead or not. This may take some time, but unless you investigate you'll never know.

3. Dynamic program analysis

Runtime monitoring or dynamic program analysis may be used to rule out parts of the code which are **not** dead code. This effectively reduces the amount of code to be inspected.

The idea is the same as measuring test coverage. In test coverage, tools help you pinpoint lines of code which are *not covered by any test*. In dynamic program coverage, tools help you pinpoint lines of code which are *never run by users*, either because the code is dead or because the features themselves are never used.

This is an example of dynamic code coverage report generated by Clover-for-Eclipse:



The data gathered shows which lines of code and how many times they were run by users. It also shows in red lines of code which has never been run. The horizontal bars below show several classes that were never run altogether.

A final note on dead code

Removing dead code is a quick win by all means. It doesn't take time and gives a big relief for the team. In my experience, it took us no more than 2-3 days removing crap and end up with this feeling of achievement! On average, in this small period of time, teams managed to remove 4% to 7% (and in one case 10%) of the total lines of code [2].

3.2 Remove code duplicates

> **Duplication may be *the* root of all evil in software**
>
> *- Robert C. Martin*

It is interesting to read what gurus write about code duplication. You feel like reading about a plague or a catastrophe which you should avoid by all means.

Andrew Hunt, one of the 17 signatories of the Agile Manifesto, and David Thomas, in their book *"The Pragramtic Programmer"*, have put down several principles for Pragmatic Programming, the first of which is: **Don't Repeat Yourself!**

SonarQube, the famous tool for continuous inspection of code quality, lists duplication as one of the **seven deadly sins of developers!**[3]

Robert C Martin (aka uncle Bob), the famous author, speaker and developer, says that "Duplication may be the root of all evil in software"[4]. In another article[5], he is no longer hesitant and asserts that "**Duplicate code *IS* the root of all evil in software design**."

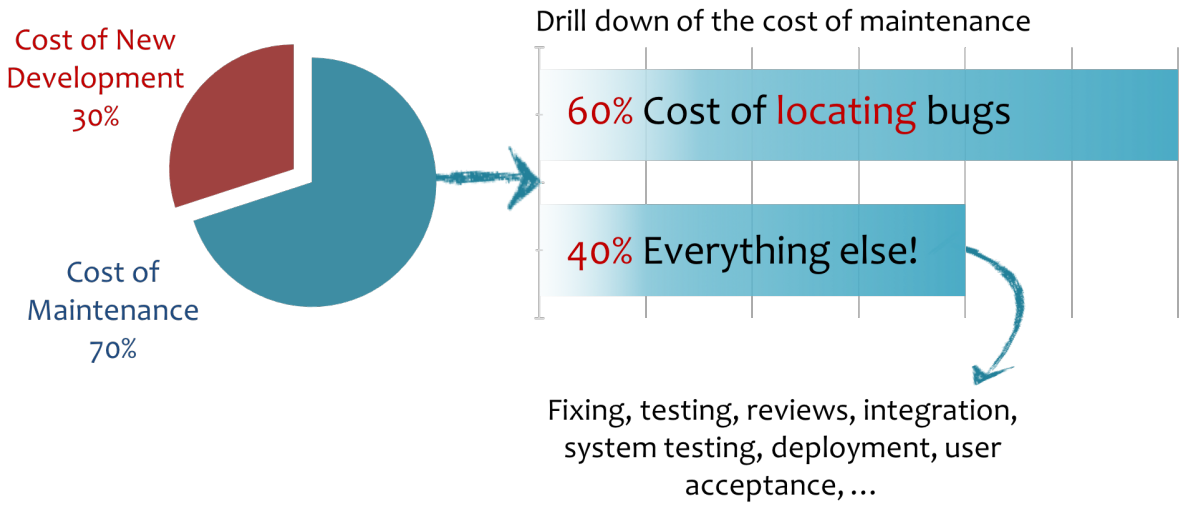
What's evil about code duplication

In a study conducted on software expenditure during the 90's, they found that 70 billion of the 100 billion expenditure on development were spent on maintenance; and 60% of which is consumed to *locate defective code* [1]:

[3] Developers' Seven Deadly Sins

[4] This is mentioned in his famous book: *Clean Code: A Handbook of Agile Software Craftsmanship*

[5] Uncle Bob mentioned this explicitly in his article at infoq: Robert C. Martin's Clean Code Tip of the Week #1: An Accidental Doppelgänger in Ruby



60% of the maintenance effort is spent on locating bugs. That is, debugging and chasing code lines till you finally point to a lines of code and say 'I found the bug'. The remaining 40% are for everything else: Fixing, testing, reviews, integration, system testing, deployment, user Acceptance,…

Duplication further magnifies time for locating bugs. If you have a defective piece of code duplicated two or three times, then it's not enough to spend the cost of finding the defect once (which already takes 60% of overall defect handling and resolution time). Rather, you'll need to find each and every copy of this defect elsewhere in the code, which is sometimes very expensive or even not possible. what usually happens is that we get an illusion that the bug is fixed upon fixing the first clone, ship the *fixed* software to the customer, who probably become very annoyed and backfire on us that the bug is still there.

Why developers copy and paste code?

Well, if code duplication is that evil. Why do we duplicate code all the time? Throughout my career, I noticed developers follow this pattern one way or another: Copy some code, change it to suite your new behavior, and finally test all changes.

Usually, developers start by copying some code, change it to suite the new behavior, then test.

This is pretty natural. Actually, I myself always followed this pattern and I'm still following it. And, I've been doing excellent work with teams I worked with. So, where is the problem? The problem is that I always do a forth step which is necessary and cannot be neglected: refactoring. It's ok to copy and paste code only if you're going to refactor this code later on.

What's missing is to refactor before committing changed code. Overlooking this step results in a huge amount of duplicate code.

Neglecting this step is a fundamental mistake which rightly is one of the "deadly sins of developers", as put by SonarQube.

Type of code clones

There are four types of code clones: *Exact, Similar, Gapped, and Semantic*. They are also known as type 1, 2, 3, and 4 of clones. In the following sections, we will shed light on each of them to help you detect and remove them mercilessly!

Note: All examples of code clones are detected by ConQAT, a **Con**tinuous **Qua**lity monitoring tool developed by the Technical University of Munich.

Type 1: Exact Clones

These are the most straight forward and the easiest to detect type of clones. Here is an example of an exact clone:

```
    * to mark the database in use. */
private static Properties serverProperties = null;

/** Properties file that saves the properties */
private static File propsFile = null;

// initialize and load the properties of the database server.
static {
    serverProperties = new Properties();
    propsFile = new File("./server.properties");

    try {
        if (propsFile.exists()) {
            serverProperties.load(new FileInputStream(propsFile));
        } else {
            propsFile.createNewFile();
        }
    } catch (IOException e) {
        e.printStackTrace(System.err);
    }
}
```

```
IvjEventHandler ivjEventHandler = new IvjEventHandler();
/** Properties object that holds the server properties. */
private static Properties serverProperties = null;
/** Properties file that saves the properties */
private static File propsFile = null;

// initialize and load the properties of the database server.
static {
    serverProperties = new Properties();
    propsFile = new File("./server.properties");

    try {
        if (propsFile.exists()) {
            serverProperties.load(new FileInputStream(propsFile));
        } else {
            propsFile.createNewFile();
        }
    } catch (IOException e) {
        e.printStackTrace(System.err);
    }
}
```

Exact clones: Copies of the code is exactly the same

Type 2: Similar Clones

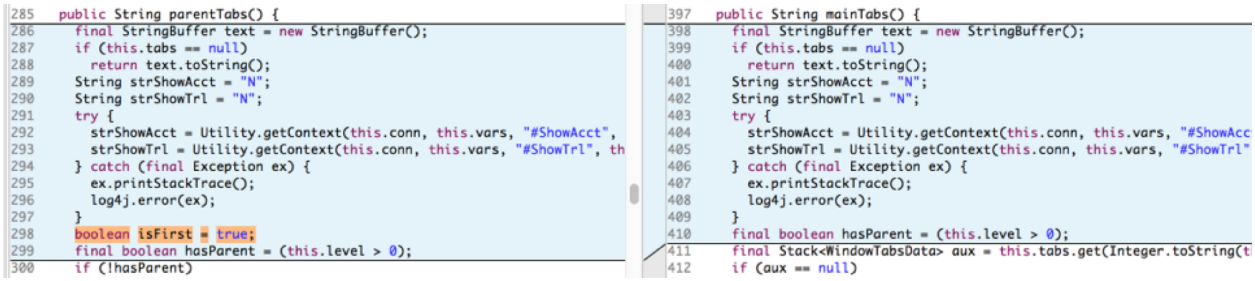
Similar clones are more common than exact clones because most probably, when a programmer copies some code, he/she changes or renames some of the variables or parameters:

Notice that `Locator` is renamed to `AttributeSetInstance` and `PROPERTY_SEARCHKEY` is renamed to `PROPERTY_DESCRIPTION`

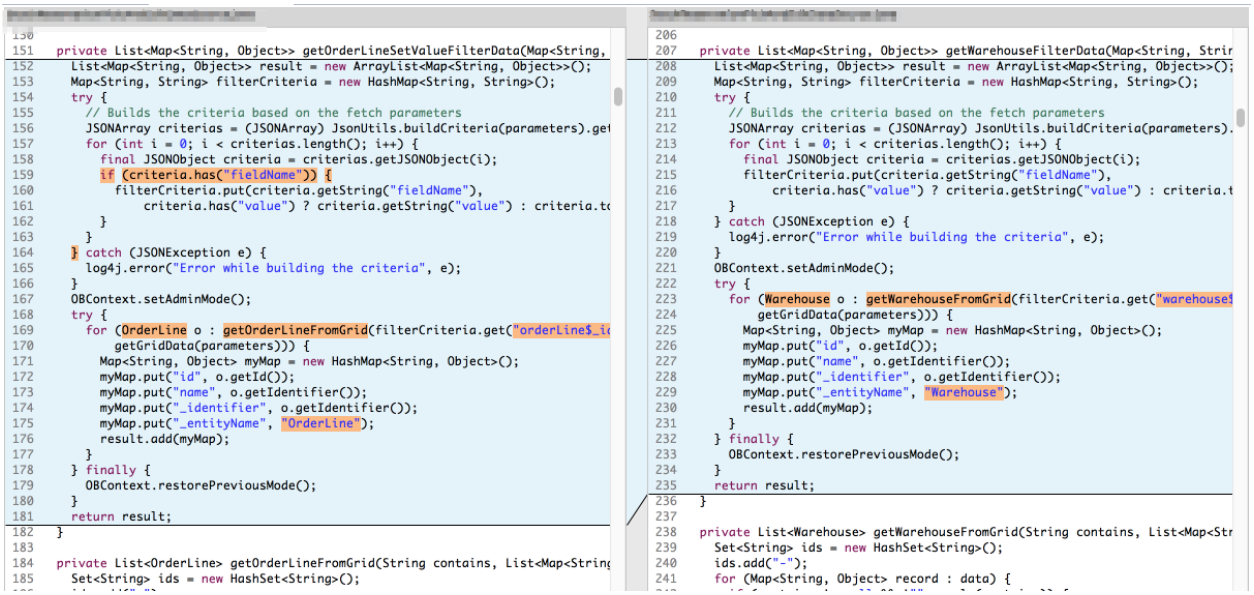
As you can see in the above example, clones are exactly the same except for some renamed identifiers. Note that the structure of the code is the same, and the positions of the renamed identifiers are all the same.

Type 3: Gapped Clones (aka inconsistent clones)

This type of clones are very interesting. These are exact or similar code clones with one or two lines of code changed (either added, deleted, or modified). These changes are called *Gaps*. Why are they interesting? Because probably they are defects fixed in one location and wasn't fixed in the others!



Two exact clones with only one line change (or gap). With minimal review, one may discover that this was a bug fixed in the left hand clone, and not in the other.



Two similar clones (with some renames), but also with one gap: `if (criteria.has("fieldName"))` check.

In both above examples, you need to review the code before fixing anything. It may be a valid business case or a *dormant bug*. Unless you review, you will never know.

Type 4: Semantic clones

The forth type of clones deals with fragments of code doing the same thing but not sharing similar structure. For example, implementing a routine which calculates the factorial of a number, one using loops and another using recursion:

```
int factorialUsingLoops(int n){
  int factorial = 1;
  for(int i = 1; i <= n; i++)
    factorial = factorial * i;

  return factorial;
}
```

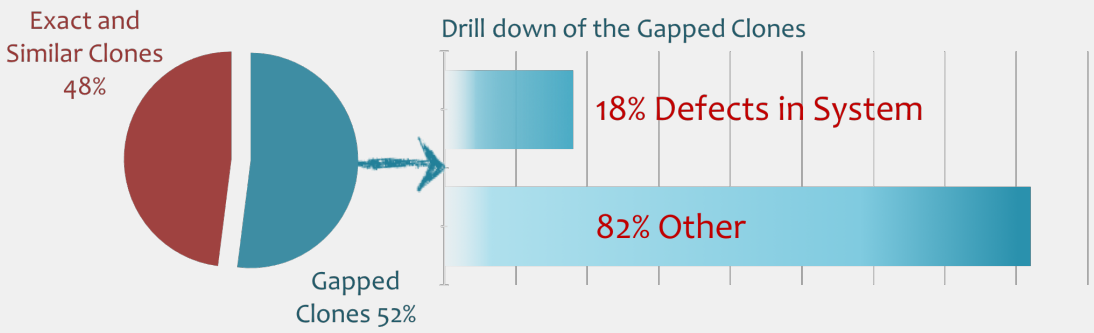
```
int factorialUsingRecursion(int n){
  if (n == 0)
    return 1;
  else
    return(n * factorialUsingRecursion(n-1));
}
```

There are lots of efforts in the academia to research whether it is possible to detect type 4 of code clones or not. Till they reach something tangible, let's focus our attention to detect and remove the first three types of code clones.

Dormant Bugs and Gapped Clones

Dormant bugs are bugs which have lived some time on production before they are discovered. Recent studies found that 30% of bugs are dormant. This is scary, because this indicates that there are dormant bugs with each and every deployment. You have no idea when they will fire back; you have no idea what would be the side effects [6].

Now, think about gapped clones. These are typically probable dormant bugs on production. Another study shows that the percentage of gapped clones in software systems running in large enterprises are 52%. Amongst these clones, 18% are defects [7]:



If there are 100 code clones, 52 of them are gapped clones. If you drill into these gapped clones, you'll find 18% of them are defects

This means that if you managed to remove 100 gapped clones, then congratulations! You've removed **18 dormant bugs!**

Removing code duplicates

There are several refactoring techniques for removing duplicate code. The safest and most straight forward technique is to 'Extract Method', and point all duplicates to it. This is relatively a safe refactoring specially if you rely on tool support to automatically extract methods.

In all projects that I've worked on, we were very cautious while removing duplicates. These are several pre-cautions to keep in mind:

- Rely on automatic refactoring capabilities in IDE's to extract methods. Sometimes, it is the most obvious mistakes which you may spend hours trying to discover. Relying on automatic refactoring support will reduce or even eliminate such mistakes.
- Any change, what so ever, must be reviewed.

Keeping these two pre-cautions in mind will save you, especially that we are refactoring on the mainline, not on a separate long living branch. More on this in this previous chapter on how to prepare a healthy environment section.

3.3 Reduce method size

> **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.**
>
> *- Martin Fowler [8]*

One thing I like about this definition is the clearly-stated objectives of refactoring, which are to make software:

1. Easier to understand
2. Cheaper to modify

Having these two objectives in mind, it's possible to develop your "gut feeling" about the correct length of a method. So, let's agree for now that **a method is *good* and needs no further refactoring when it fulfills these two criteria of being understandable and modifiable**.

An experiment on method size

To measure the effect of the method length on the code readability, I have done an experiment with university students. I gave them three variants of a method: without comments, with comments, and refactored into a small 5-line method. I have measured the time it takes them to understand the intent of the method. Results were as follows:

- Method without comments: ~ **2 minutes**
- Method with comments: ~ **1 minute**
- Refactored short method: ~ **10 seconds**

I advice you to do this experiment with your team. Get a stopwatch and use the sample code below or any piece of code from your project. **It is stunning how much time you save by just reducing methods sizes into smaller ones with readable private method calls.** It realizes the core objective of refactoring: to make the code "easier to understand and cheaper to modify".

Ready? Go!

Method with no comments:

```
1  public List criteriaFind(String criteria) {
2    if (criteria == null)
3      criteria = "";
4
5    List criteriaList = scanCriteria(criteria);
6    List result = new ArrayList();
```

```
7     Iterator dataIterator = getDataCash().iterator();
8     Iterator criteriaIterator = null;
9     DataInfo currentRecord = null;
10    List currentCriterion = null;
11    boolean matching = true;
12
13    while (dataIterator.hasNext() && !interrupted) {
14      currentRecord = (DataInfo) dataIterator.next();
15
16      criteriaIterator = criteriaList.iterator();
17      while (criteriaIterator.hasNext() && !interrupted) {
18        currentCriterion = (List) criteriaIterator.next();
19        if (!currentRecord.contains((String) currentCriterion.get(0),
20          (String) currentCriterion.get(1))) {
21          matching = false;
22          break;
23        }
24      }
25      if (matching)
26        result.add(currentRecord);
27      else
28        matching = true;
29    }
30    if (interrupted) {
31      interrupted = false;
32      result.clear();
33    }
34    Collections.sort(result);
35    return result;
36  }
```

This is a 36-line method. It seems to be small. However, you've spent some time (probably around 1-2 minutes) to grasp how the code works. So, **according to our definition, is this method *maintainable*? The answer is no.**

Now, consider this enhanced version of the method:

Method with explanatory comments:

```
1  public List criteriaFind(String criteria) {
2   if (criteria == null)
3     criteria = "";
```

```
4
5   // convert the criteria to ordered pairs of field/value arrays.
6   List criteriaList = scanCriteria(criteria);
7   List result = new ArrayList();
8
9   // search for records which satisfies all the criteria.
10  Iterator dataIterator = getDataCash().iterator();
11  Iterator criteriaIterator = null;
12  DataInfo currentRecord = null;
13  List currentCriterion = null;
14  boolean matching = true;
15
16  while (dataIterator.hasNext() && !interrupted) {
17    currentRecord = (DataInfo) dataIterator.next();
18
19    // loop on the criteria; if any criterion is not fulfilled
20    // set matching to false and break the loop immediately.
21    criteriaIterator = criteriaList.iterator();
22    while (criteriaIterator.hasNext() && !interrupted) {
23      currentCriterion = (List) criteriaIterator.next();
24      if (!currentRecord.contains((String) currentCriterion.get(0),
25        (String) currentCriterion.get(1))) {
26        matching = false;
27        break;
28      }
29    }
30    if (matching)
31      result.add(currentRecord);
32    else
33      matching = true;
34  }
35
36  // clear results if user interrupted search
37  if (interrupted) {
38    interrupted = false;
39    result.clear();
40  }
41
42  // Sort Results
43  Collections.sort(result);
44  return result;
45 }
```

Adding some comments are generally perceived to enhance code understandability. It may clutter the code a bit, but at least in this example, the code is a little more readable. But, wait a minute, if we are adding comments to make the code more readable, isn't this an indication that the code is not maintainable? According to our definition of maintainability, the answer is yes. This is why *explanatory comments* are generally considered a code smell, or a sign of bad code.

If we are adding comments to make the code more readable, isn't this an indication that the code is not maintainable? According to our definition of maintainability, the answer is yes. This is why *explanatory comments* are generally considered a code smell, or a sign of bad code.

Now, let's work on this method. If you notice, comments are placed at perfect places. They give you a hint of the *Boundaries of Logical Units* inside the method. Such logical units are functionally cohesive and are candidate to become standalone methods. Not only that, the comment itself is a perfect starting point for naming of the newly born method.

So, by extracting each chunk into a standalone method, we will reach this version of the method:

After extracting method steps into private methods:

```
1  public List criteriaFind(String criteria) {
2    List criteriaList = convertCriteriaToOrderedPairsOfFieldValueArrays(criteria);
3    List result = searchForRecordsWhichSatisfiesAllCriteria(criteriaList);
4    clearResultsIfUserInterruptsSearch(result);
5    sortResults(result);
6    return result;
7  }
```

This is a 5-line method which narrates a story. No need to write comments or explain anything. It is self-explanatory and much easier now to instantly capture the intent of the code.

Logical units of code

Notice that the original form of the `criteriaFind` method in the above example is functionally cohesive and follows the Single Responsibility Principle (SRP) in a perfect way. However, if you look inside the method, you may notice what I call *Logical Units of Code*, which are *steps of execution*; each one is several lines of code. A single step does not implement the full job, but it implements a conceivable part towards the goal.

Examples of logical units may be an if statement validating a business condition, a for loop doing a batch job on a group of data records, a query statement which retrieves some data from the database, several statements populating data fields on a new form, etc. In my experience, sometimes the logical

unit are as small as two or three lines of code. More frequently, they are bigger (like 5 to 12 lines). On very rare occasions I see logical units which are bigger than that.

This is an example of logical units of code, extracted from the famous OpenBravo open source ERP solution. Notice how comments help you identify these units:

Steps of execution

```
}

// Initialize current stock qty and value amt.
BigDecimal currentStock = CostAdjustmentUtils.getStockOnTransactio
BigDecimal currentValueAmt = CostAdjustmentUtils.getValuedStockOnT
log.debug("Adjustment balance: " + adjustmentBalance.toPlainString

// Initialize current unit cost including the cost adjustments.
Costing costing = AverageAlgorithm.getProductCost(trxDate, basetrx
if (costing == null) {
  throw new OBException("@NoAvgCostDefined@ @Organization@: " + ge
}

// If current stock is zero the cost is not modified until a relat
// the stock is found.
BigDecimal cost = null;
if (currentStock.signum() != 0) {
  cost = currentValueAmt.add(adjustmentBalance).divide(currentStoc
}

log.debug("Starting average cost {}", cost == null ? "not cost" :
if (AverageAlgorithm.modifiesAverage(trxType) || !baseCAL.isBack
```

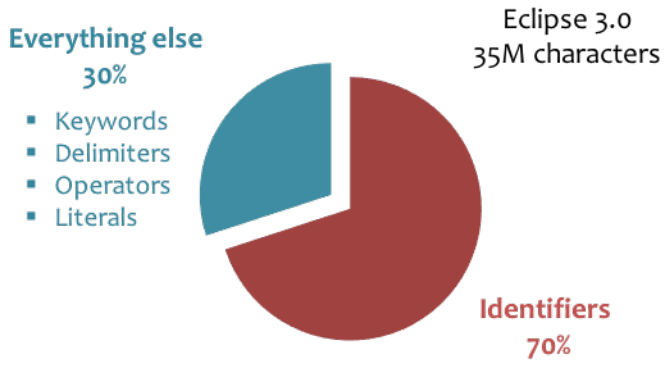
logical units of code or steps of execution.

Such logical units are perfect candidates to be extracted into *private* methods. If you adopt this practice for a while, you'll start noticing some private methods which are similar in nature or shares the same "interest". In such case, you may extract and group them into a new logical component. More about this in the Divide and Conquer stage.

3.4 Enhance identifier naming

> **You know you are working on clean code when each routine you read turns out to be pretty much what you expected.** [6]
>
> *- Ron Jeffries*

In an interesting study titled: *Concise and Consistent Naming*, the authors has conducted token analysis on Eclipse 3.0 code, and found that *"Approximately 70% of the source code of a software system consists of identifiers"* [17]:



Token analysis of Eclipse 3.0 source code shows that 70 of the code is identifiers which developers coin their names*

This is why "the names chosen as identifiers are of paramount importance for the readability of computer programs and therewith their comprehensibility". Imagine that every class, method, parameter, local variable, every name in your software is indicative and properly named, imagine how readable your software will become.

The good news is that renaming has become a safe refactoring which we can apply with minimal side effects; thanks to the automatic rename capability available in most modern IDE's.

Explanatory methods and fields

One of the interesting tools to enhance code readability is to use *explanatory methods and fields*. The idea is very simple: if you have a one line code which is vague and not self-explanatory, consider extracting it into a standalone method and give it an explanatory name.

Similarly, if you have a piece of calculation whose intent is not clear, consider extracting it into a field and give an explanatory name.

```
public Boolean bookSeats(Request request) {
```

[6] Quoted in *Leading Lean Software Development: Results Are not the Point*, by Mary and Tom Poppendieck.

```
  Boolean bookingResult = new Boolean(dataHandler.book(dataHandler
      .getRecord(((Integer)request.getParametersList().get(0)).intValue()),
        ((Integer)request.getParametersList().get(1)).intValue()));
  return bookingResult;
}
```

`dataHandler.book` parameters are not clear. There is a difficulty understanding what kind of parameters we are passing. In stead, we can use **explanatory methods** as such:

```
public Boolean bookSeats(Request request) {
  Boolean bookingResult = new Boolean(dataHandler.book(getFlightRecord(request),
    getNumberOfSeats(request)));
  return bookingResult;
}
private DataInfo getFlightRecord(Request request){
        return dataHandler
    .getRecord(((Integer)request.getParametersList().get(0)).intValue());
}
private int getNumberOfSeats(Request request) {
        return ((Integer)request.getParametersList().get(1)).intValue();
}
```

Or, we can use **explanatory fields** as such:

```
public Boolean bookSeats(Request request) {
  DataInfo flightRecord = dataHandler.getRecord(
    ((Integer)request.getParametersList().get(0)).intValue());
  int numberOfSeats = ((Integer)request.getParametersList().get(1)).intValue();
  Boolean bookingResult =
    new Boolean(dataHandler.book(flightRecord, numberOfSeats));
  return bookingResult;
}
```

My advice is to always use explanatory methods and fields. They are extremely simple and astonishingly helpful tool to enhance program readability.

3.5 Considerations related to the quick-wins stage

Avoid introducing inheritance trees

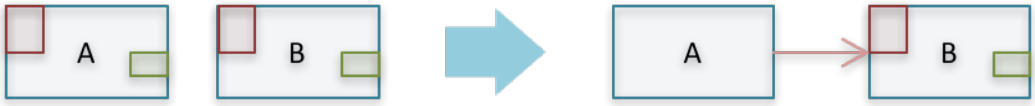
One tempting technique to remove duplication is to introduce a parent type which gathers common behavior among two or more child types:

Highlighted parts represent duplicated behavior among A and B

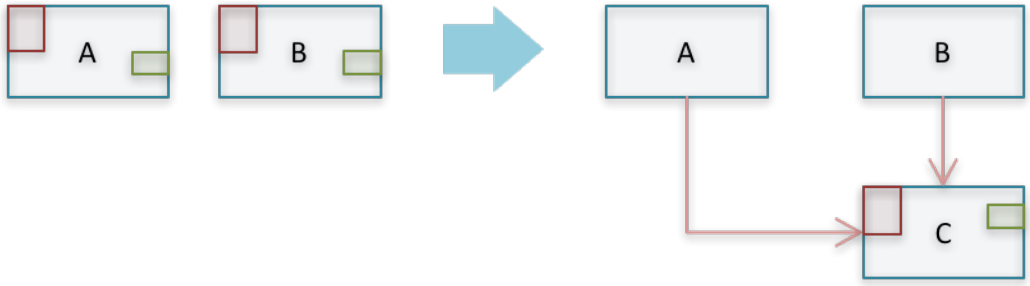
In general, I prefer composition over inheritance. Inheritance hierarchies are notorious for their complexity and difficulty of understanding polymorphic behavior of sub-types. Moreover, they are especially not recommended at this very early stage of refactoring.

Instead, you may chose one of the following three simple alternatives:

1. *Inline Classes* into one class, especially if the coupling and/or level of duplication is high between the two of them.
2. *Extract Methods* in class B and reuse them in class A. This creates a dependency on B, which may or may not be a bad thing.



3. If coupling between A and B is bad, then *Extract Methods* in A and then *Move Methods* to an existing common class. If no candidate common class is available, use the *Extract Class* refactoring to extract the common methods to a new class C. In both cases, A and B will depend on the common class C:



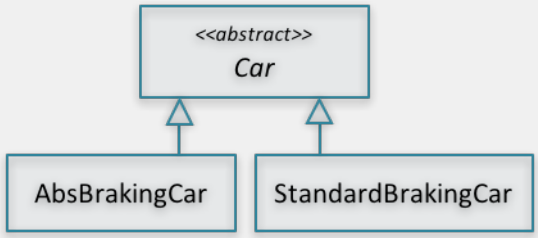
Why we should favor composition over inheritance

This is a controversial topic since the inception of object-oriented design. A lot has been said about when to use inheritance and whether you should favor composition and when. However, it seems there is a general "impression" that overuse of inheritance causes problems and deteriorates program

clarity; something which we are already trying to avoid. Here are some references:

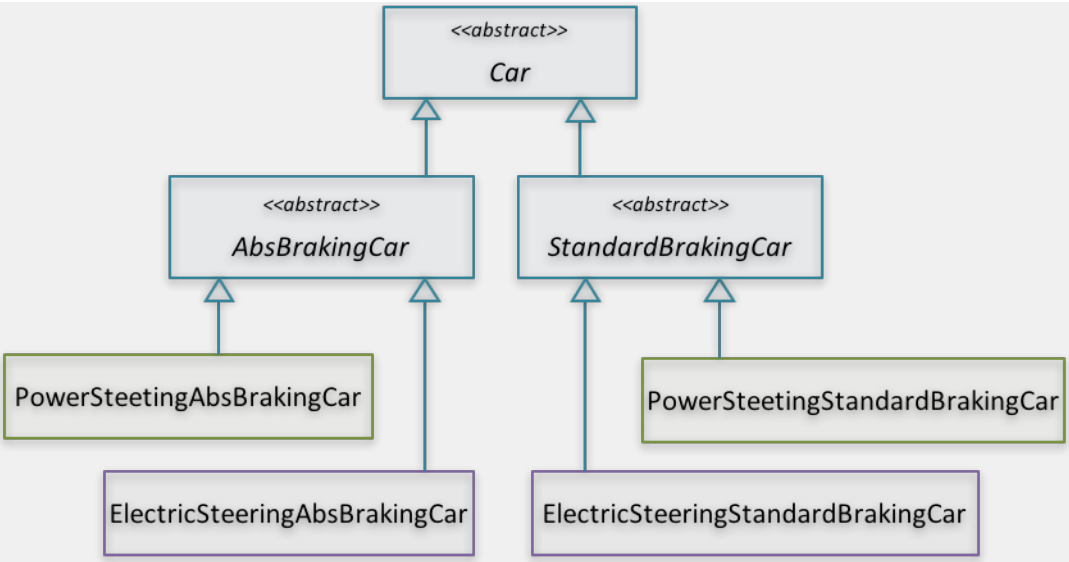
- The GOF book, way back in 1995, advices us to "Favor 'object composition' over 'class inheritance'." They rightly argue that "because inheritance exposes a subclass to details of its parent's implementation, it's often said that 'inheritance breaks encapsulation'" [12]
- Eric S. Raymond, in his book *The Art of Unix Programming*, argues that the overuse of inheritance introduces layers in code and "destroys transparencies" [13]. I absolutely agree on this. From my experience, looking for a bug in a pile of inheritance hierarchy with five or six layers of polymorphic behavior is like searching for a needle in a haystack!

Im many cases, using composition with the Strategy pattern hits a sweet spot between composition and inheritance. Consider this example: We are building a car system simulator in which a car may have two breaking systems: standard and ABS. In this case, it may be straight forward to use inheritance:



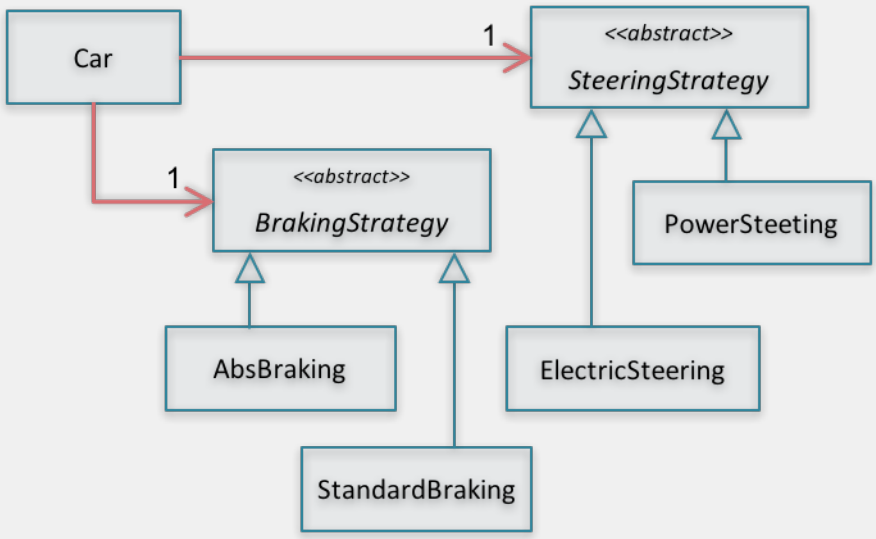
Car abstract class in a car system simulator. The car may be specialized by the type of breaking system it has: standard and ABS

Now, consider adding a capability to simulate two steering systems: Power and Electric. If we continue using inheritance, we will have to introduce duplication, the enemy of clean code. In the example, the logic of power steering is now duplicated in both `PowerSteeringAbsBrakingCar` and `PowerSteeringStandardBrakingCar`, and the logic of electric steering is duplicated in both `ElectricSteeringAbsBrakingCar` and `ElectricSteeringStandardBrakingCar`



Further specialization results in duplication, as in the case of `PowerSteeringAbsBrakingCar` and `PowerSteeringStandardBrakingCar`

Instead, let's collapse this inheritance tree, and use composition with the Strategy pattern. Here, we will design a Car with many components, each component is an *abstract strategy* which may have several *concrete implementations*:



Using composition with the Strategy pattern hits a sweet spot between composition and inheritance

Generally, maintaining code with lots of components is much easier than maintaining code with hierarchies of inheritance trees.

Always rely on tools support

One important consideration in this stage is that **no manual refactoring is allowed!**. Detecting dead code, detecting and removing code clones, extracting methods to reduce method size, renaming identifier names; you can carry out all such tasks with the assistance of strong IDE features or add-on tools.

Using automated refactoring tools contributes to safety and makes developers more confident when dealing with poor and cluttered code.

Should we do them in order?

Yes, with little bit of overlap. This is logical and practical. For example, removing dead code, removes about 10% of your code duplicates[7].

Another example is working on reducing method size before removing duplicates. This actually is a bad practice. Because you may split a method apart while it is actually a duplicate of another. In this case, you have lost this similarity and may not be able to detect this duplication anymore.

Are these refactorings safe?

Sometime, applying any change to production code is scary. Changes may result in unexpected flows and incorrect side effects, especially if the code is entangled. If this is the case, is it safe to carrying out those changes the quick-wins stage?

In one of my experiments, the team applied the quick-wins refactorings side by side while developing new features. I have compared the results of this release with the previous release which witnessed new features development only. Table 4 compares some quality metrics of both releases. Note that effort spent on both releases are exactly 4 months, team members are the same, and they did not introduce any improvements in their process except their work on refactoring:

TABLE 1. Quality metrics for two releases: 5.5 (released before working on refactoring), and 5.6 (released while working on refactoring)

Metric	Release 5.5	Release 5.6
Total bugs detected	128	176
% of Regression bugs	29.7%	25.1%
Average bug fixing cost (hours)	1.97	1.8

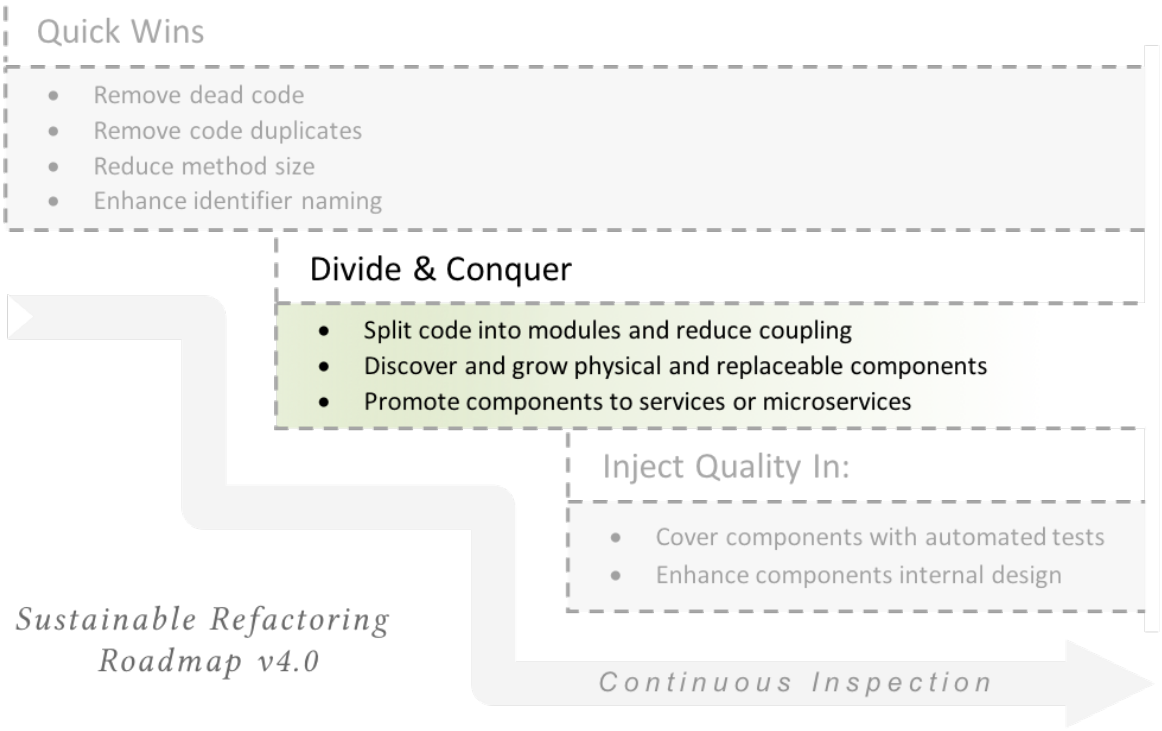
Two important observations from this table:

- Percentage of regression bugs and average cost of bug fixing decreased. Although this may give an indication of better code quality, the difference in numbers is not significant

[7]This was validated in one of our experiments. We found that removing dead code removes also 10% of duplicate code [2]. This is totally reasonable, because a good portion of duplicated code are eventually abandoned.

- Refactorings applied during release 5.6 did not produce higher rates of regression bugs. This is a proof that refactorings did not impact existing functionality or introduce further defects.

4. Stage 2: Divide and Conquer



Stage 2: Divide & Conquer - Split code into components

Software design is all about components and their relationships. The better you divide your software into loosely-coupled and highly-cohesive parts, the more comprehensible, more responsive to change, and more agile your software design becomes. The act of partitioning your software in this manner is described by George Fairbanks as authoring "a story at many levels", which results in a software design that will "tell a story to whoever looks at it, and it will be easy to understand":

> **To be comprehensible, your software should be structured so that it reveals a story at many levels. Each level of nesting tells a story about how those parts interact. A developer who was unfamiliar with the system could be dropped in at any level and still make sense of it, rather than being swamped.**
>
> *- George Fairbanks [14]*

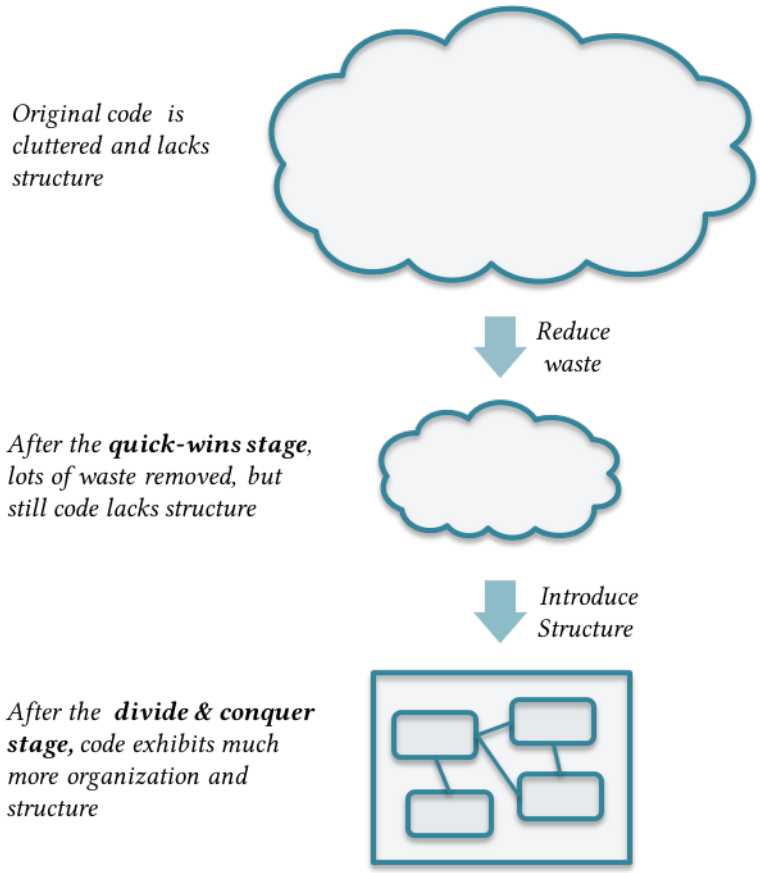
Now, here is a question: If software partitioning is that important, why didn't we start with it right away in the refactoring roadmap?

We can describe what we achieved so far as **removing "fat" from the application's body of code**; namely, removing dead code, and reducing code duplication, plus applying some very basic and intuitive enhancements which makes the code slightly more readable, like reducing method size and using proper naming conventions. This is like *preparing the scene* or *organizing our backyard* before we work on partitioning the code and re-organizing the parts. This has two very important side effects:

1. We have saved the time that we would have spent working on dead or duplicate code.
2. The team reached a better grasp of the code while scanning and reviewing duplicates and suspect dead code. Also, after working on breaking large methods and trying to give better names to identifiers and code constructs, they formed better understanding about the intent of the code.

As I explained before in the Introduction and Background section, I have noticed the effect in the second point above many times while working with teams on refactoring, and teams become more courageous and bold in enhancing the code especially after the quick-wins stage.

Although all the enhancements in the quick-wins stage made the code *better*, but the code still *lacks structure*. The mission now is introducing structure by discovering/uncovering components and enhance their interfaces:

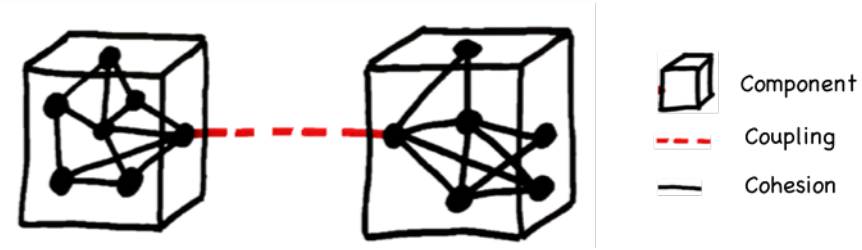


Code evolution throughout the quick-wins and the divide & conquer stages

4.1 Coupling and Cohesion

Coupling and Cohesion are two principal concepts in software design. Coupling between two code modules indicates the level of connectedness and interaction between them. We can measure coupling between two classes, packages, modules, components, or any two parts of the system. Low coupling between any two modules is a good system property because it reduces unnecessary dependencies and minimizes change ripple effect in the system.

Cohesion, on the other hand, indicates the level of connectedness and interaction among constituents of one part of the system. High cohesion is a good quality of a module or component. A high cohesive module means that parts of the module needs each other to carry out the responsibilities of this module, and thus may not be broken apart into two distinct modules.



Low coupling and high cohesion are among the most important principles of good design. Image adapted from: https://www.planetgeek.ch/2011/07/08/presentation-agile-code-design-how-to-keep-your-code-flexible/

In the next sections, we will explore more ideas and techniques to reduce coupling and increase cohesion in existing code.

4.2 Modules, components, services, or micro-services?

Now, are we splitting our code into modules, components, or services? let's first agree on what a module, component, and service are.

Module

The *UML Reference Manual* provided a very brief and broad definition of what a module is. It is a *"software unit of storage and manipulation"* [19, pp 334].

To elaborate on this definition, **a module is any logical grouping of cohesive code which provides access to its functions in a uniform manner**. This can be as big as a sub-system, like an accounting or HR module, or as small as a class, like a calculator or an xml parser.

Component

> **A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. It is intended to be easily substitutable for other components that meet the same specifications.**
>
> *- The UML Reference Manual [19, pp. 10]*

From this definition, we understand that a component is a physical standalone file; a jar, war, dll, gem, etc. Also, it is replaceable, meaning that it can be deployed/redeployed on its own. Finally, we understand that a component may contain one or more smaller modules; and vice versa, a big module may contain one or more components.

Service

In essence, web services (or just services), are components. They are physical, replaceable, provides clear interfaces, and easily substitutable. However, there is value in differentiating services from components.

> **A service is similar to a component in that it's used by foreign applications. The main difference is that I expect a component to be used locally (think jar file, assembly, dll, or a source import). A service will be used remotely through some remote interface, either synchronous or asynchronous (eg web service, messaging system, RPC, or socket.)**[1]
>
> *- Martin Fowler - Inversion of Control Containers and the Dependency Injection pattern*

One very important difference between a component and a service is that a component cannot run on its own. It has to be integrated in a bigger whole to achieve any value out of it. Unlike a service, which is available and standalone. It can be located and used whether on its own or as part of a bigger application.

Then what?

So, what's the value of outlining these differences between modules, components, and services? This distinction is important because there are higher level of decoupling and increased formality in defining interfaces when moving from modules to components to services.

Modules may co-exist in the same (physical) deployable package; unlike components or services, which are physically standalone. Modules and components run typically in the same process; unlike services, where every service resides in its own process. Modules and components may communicate through in-memory method calls, while services requires inter-process communication through web-service requests, remote procedure calls, etc.

So, services enjoy the maximum level of decoupling. You can view them as standalone applications which could be glued together in order to provide greater value for some end user.

So, **should we divide code into modules or components or services?** The answer is to start by splitting your code into modules. Then, assess whether or not it is useful and safe to upgrade them to components or services.

> **Divide the code into modules and components. Then, assess whether or not it is *useful and safe* to upgrade them to services or microservices.**

[1] This quote is from Martin's article: Inversion of Control Containers and the Dependency Injection pattern. Also, you will find other interesting distinctions between components and services in this other famous article by Martin Fowler: Microservices: a definition of this new architectural term

About Microservices

In the above discussion, I have talked about web-services (or just services). The main difference between services and microservices is that services share a common datastore, whereas each microservice has a separate standalone datastore.

When looking from the angle of refactoring legacy or monolithic application code bases, it is very risky and may not be feasible splitting a large backend database collecting data over the years into smaller ones and move towards a microservices architecture. Other challenges loom in the way like handling distributed transactions and understanding and supporting the call chain for every business transaction[a].

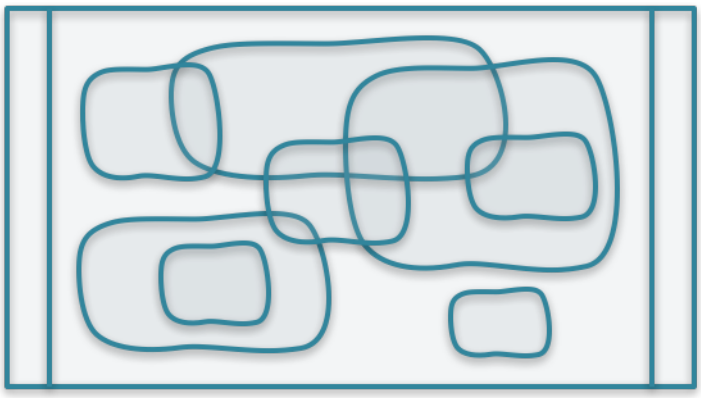
I'll discuss some considerations related to microservices later in this chapter. For now, let's accept that a middle-way is a course-grained service, following a service-based architecture of some kind.

[a]More discussion about why microservices architecture is not suitable when refactoring monolithic applications is at this talk by Neal Ford: Comparing service-based architectures.

4.3 Moving from spaghetti to structured code

The journey from spaghetti and tangled to structured code and from Big Ball of Mud[2] to modular design is progressive and multi-stage. Code with large amount of technical debt usually looks like this figure. No clear boundaries between modules, high level glimpses of module interfaces, unstructured or tangled module communication, etc.

[2]A https://en.wikipedia.org/wiki/Big_ball_of_mud is a conventional name for the architecture of a system which lacks "conceivable structure". Usually, the code as well lacks structure and all parts are tangled and highly coupled.



Module: *any grouping of cohesive code functions; can be as big as a sub-system, like an accounting or HR module, or as small as a class, like a calculator or an xml parser*

Process or container *running standalone applications. Applications may be a full-fledged enterprise application or just a simple web-service*

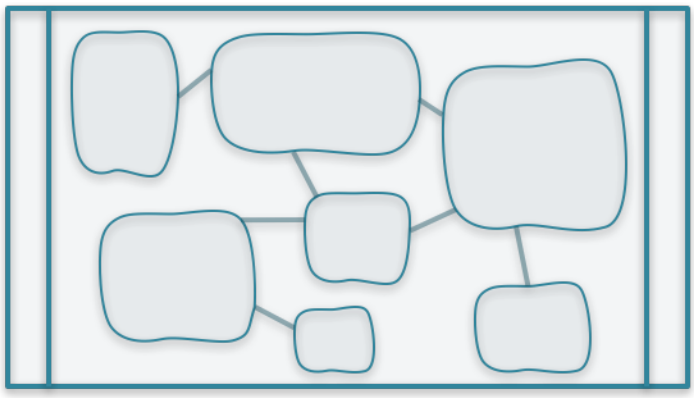
Unstructured module communication*. This is a result of undefined boundaries and high level of coupling among modules*

Gradually, we start moving methods and classes around to let modules emerge and become more apparent. This process is usually referred to as *sprout classes* described by Micheal Feathers [18]. This process can be generalized to *sprout modules* or *sprout components*. The idea is that modules and components emerge while refactoring code.

To do that, we use safe refactorings with support of automated refactoring tools. In most cases, you can depend on the following refactorings:

- Rename
- Extract Method/Class/Interface
- Move Method/Class

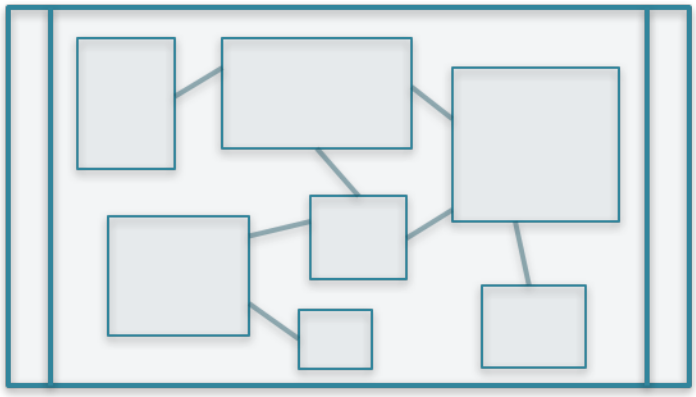
This results in clearer module boundaries and better manifestation of module interfaces:



Structured module communication. *Modules interact through defined interfaces, interface classes, or interface functions.*

Next, we should concentrate on more decoupling modules and create a solo-deployable components. At this stage, we should work more on polishing interfaces, move away un-needed interface methods and interface parameters. Some useful refactorings at this stage are:

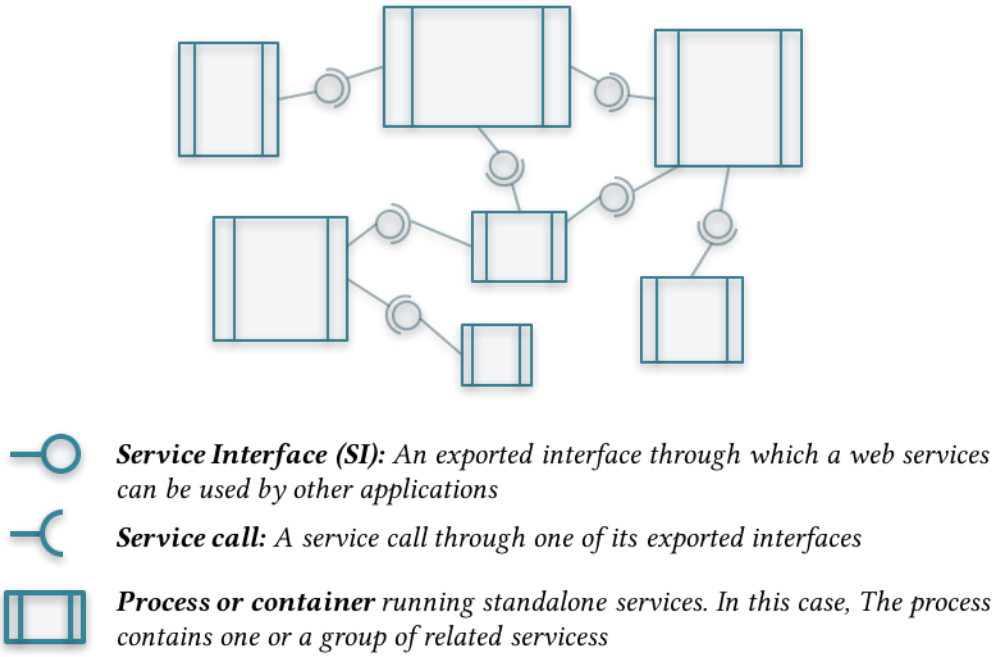
- Change Method Signature (to remove or reorder method parameters)
- Introduce Parameter
- Introduce Parameter Object
- Turn Public Methods Private



Component*: a physical replaceable part of the system which can be accessed through defined interfaces. This can be a jar, war, ear, gem, dll, etc.*

Structured component communication*. Components interact through defined interfaces, interface classes, or interface functions.*

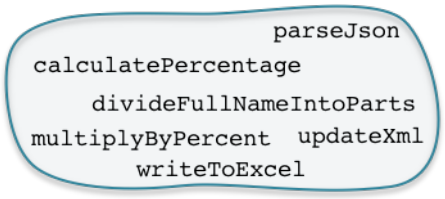
You may stop at this stage. Or, you may move to the next step and turn components into services. Remind you that you may chose to do so only if you find it *valuable and safe*.



In the next section, let's take an example of how code is gradually structured, and how sprout modules and components emerge.

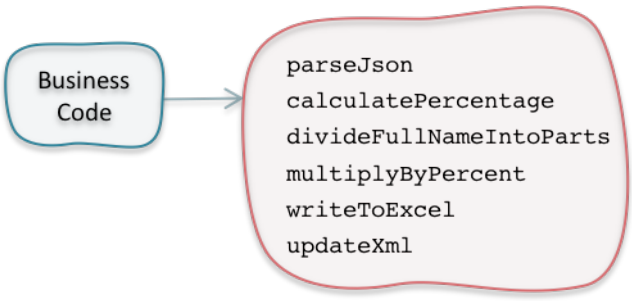
Sprout Classes, Modules, and Components

Consider a business application with all its business coded in one big module. The code base is becoming very large and maintenance is definitely taking so much time. This image depicts some of the distinctive methods in this module:



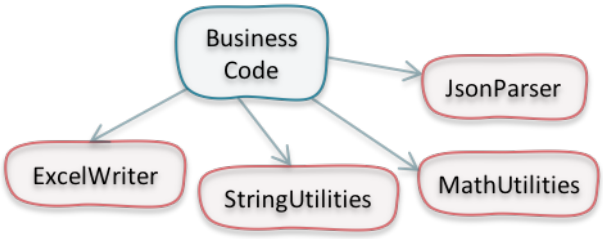
A business module with many utility methods.

The first step might be to group such utility methods and move them to a standalone "sprout" utility class. Take into account that till now we are not concerned with the best design of this module or the overall system. We are just grouping similar code together:



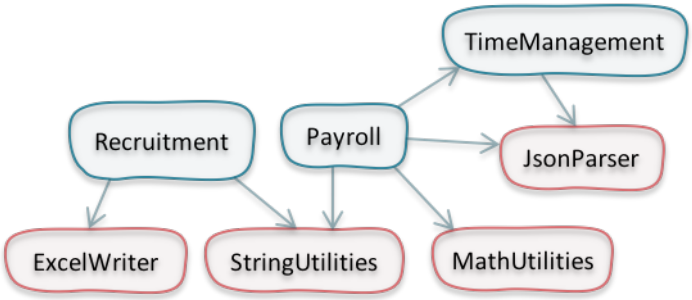
Utility methods are separated into a new sprout utility class. Initially, this is one class contains all types of utilities.

After a while, you may notice that the utility class has grown in size and is candidate for being split into separate more specialized utility classes:



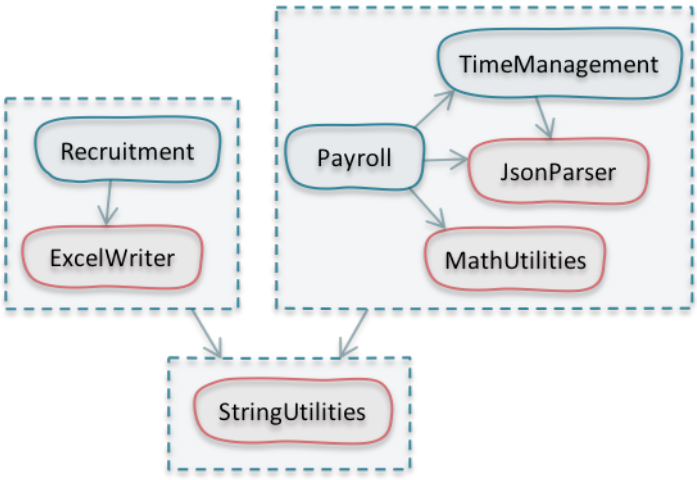
Specialized utility classes emerge due to the big size of the original utility class.

In the meanwhile, you might be tempted to split the business module itself into more cohesive and standalone parts. This will definitely enhance the readability of the code and alleviate some of the maintenance pain:



Business module is broken down into smaller more cohesive business modules.

Now, you have much better grasp of the system, you may draw the boundaries of your components more smoothly. Splitting the system into physical and replaceable components will divide the overall complexity of the system into smaller and manageable parts. It will make any future change more contained in one or two components. It will also make your live much easier while deploying the system to production. You don't need to replace the whole system because now you have the luxury of deploying the changed one or two component only:



Draw the boundaries of physical and replaceable components. Now, you can control the interfaces for accessing components more deliberately and enhance the overall structure of the application.

Notice how we let the design of the system emerge. With very small steps of extracting methods and classes and moving things around we were able to *sprout* new modules and components and *see* the new structure of the system.

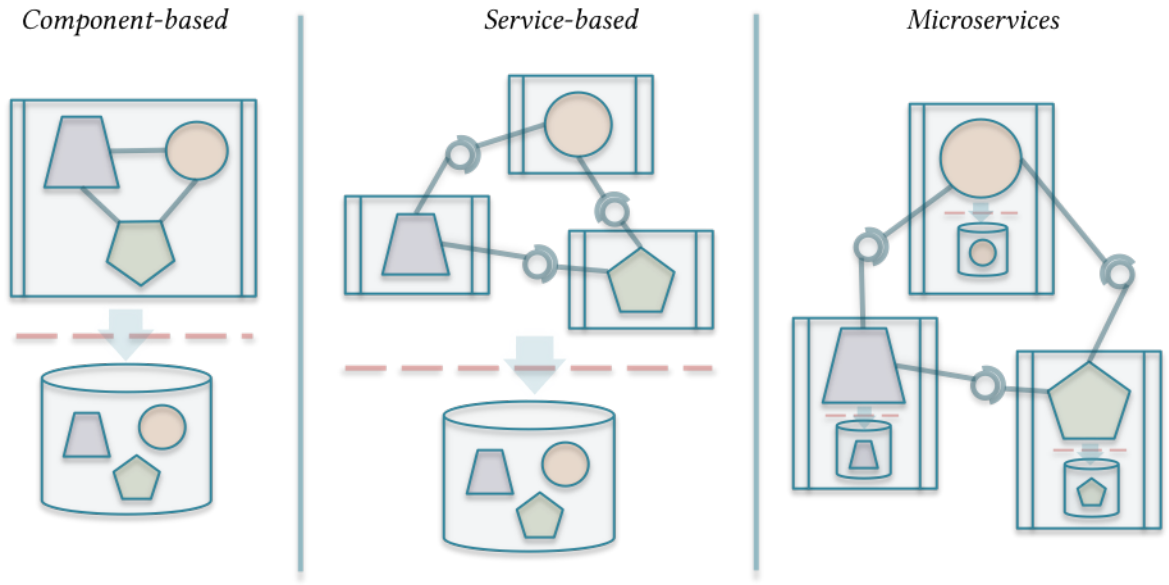
From component-based architecture to microservices

Now, let's consider this valid question: **Shall we move forward to services or microservices or stop at this stage?** The bottom line is to reach a component-based architecture, where all components are disjoint and can be deployed each one on its own. Sometimes, it's wise to stop and reap the benefits. In other cases (for instance, when scalability is a concern), it may be a good choice upgrading the system to several (3-4) big web-services. In this case, you'll get some of the benefits of service-based architectures without paying the cost of moving to microservices.

For example, if your objectives out of refactoring are around these ideas:

- Better quality and more structured code.
- Better maintainability. Changes need not take ages to be developed.
- Disjoined parts, so that updating one won't blow-up others!
- Faster time to develop and deploy new features.

If these are your objectives, then you don't need to move to microservices, because most probably you have already achieved these objectives by refactoring to component-based or service-based architectures.



Moving from component-based or service-based architectures to microservies requires splitting the data-store into multiple smaller ones, and incurs more costs and risks to be paid and mitigated.

In the meanwhile, there are some benefits that may not be achieved unless you move to microservices; namely, scalability and resilience in your development organization. Microservices promote separate development cycles, independent deployments, loosely coupled code bases, and technical and functional separation of concerns. All these are pre-requisites for "large scale software projects with a high degree of parallelism"[^microservicesimpact]. However, as put by Martin Fowler, its the **Microservice Trade-Offs**[3]. Meaning that nothing is for free and you need to pay some costs and mitigate some risks when moving to microservices:

1. Risks of splitting the monolithic data store into multiple smaller ones, most probably organized around the idea of *Bounded Contexts*. With databases collecting huge amount of data over the years, splitting this database definitely incurs huge risks.
2. Synchronization and aggregation of data among several data-stores
3. Handling distributed transactions and understanding and supporting the call chain for every business transaction.
4. Costs for hosting the new architecture.
5. Required skills and calibers.
6. Tools needed to operate and maintain the new development and production environments.
7. Security risks and issues. The increased number of services also increases the number of vulnerable points hackers may attack and compromise.

[^microservicesimpact] Microservices and the organizational Impact

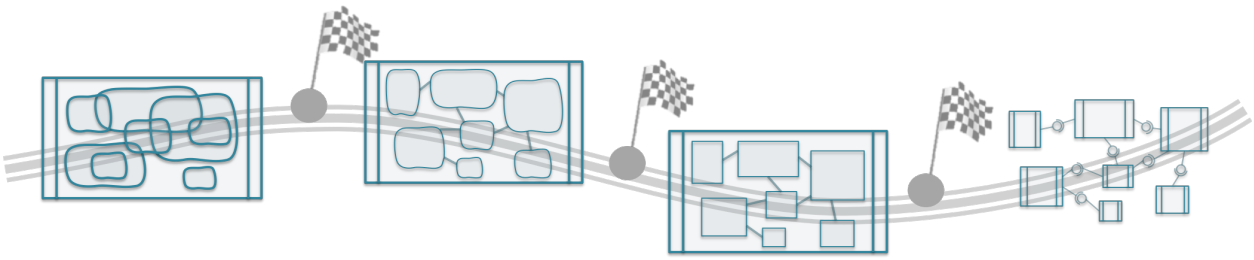
[3] Microservice Trad-Offs is an article by Martin Fowler which highlights what trade-offs you may consider when moving to microservices.

Think twice before moving to microservices. You might be over-engineering your solution and adding unnecessary complexity. In many cases, it may not be the right decision, considering the costs and overheads.

Stations not stages

A quick answer for the dilemma of refactoring to microservices is to deal with it as a **journey with many stations not stages**.

Consider moving your code to modules, components, and probably services as stations in the journey from monolithic application architecture to microservices. Each step is an achievement and results in a better overall code structure. Taking one step forward is rewarding and manageable as opposed to moving all the way till the end. After each step, you may pause, inspect and adapt, then decide whether to move forward or break off if you're satisfied with what you've achieved:



Moving to microservices has many intermediate stations. You don't need to continue the journey till the end. Rather, you may embark at any station if you're satisfied with what you've achieved.

4.4 Types of software components - Strategies for code decomposition

This section is a primer about types of software components, which follows some universal types that many experienced developers have noted. Being aware of these types will help you detect/uncover modules and enhance your code structure more effectively and efficiently.

The following two guidelines are *the general strategy for code decomposition*:

- **Guideline 1: Let modules emerge by grouping similar code together.**
- **Guideline 2: If a module becomes large, zoom into it and reapply the first guideline.**

Determining whether or not a module is large is a subjective decision. One of the very important factors which guides this decision is the level of cohesion of the module. A highly cohesive module is a good one which needs not to be split regardless its size. In the meanwhile, the *3-30* rule of thumb may give an indication whether a module is becoming very large. It states that a module may provide at least 3 and at most 30 interface methods or functions.

Factors which drive code decomposition

The two main factors which drive your thinking about code decomposition are:

First, *code artifacts which change together should be kept together*. This is known as the Common Closure Principle [15], and states that "Classes that change together are packaged together". Packaging such files together will reduce the overall coupling in the system and will reduce the change "ripple effect" on other packages or components in the system.

The second factor is that *code artifacts which are released together belong together*. Again, this is derived from the Release-Reused Equivalency Principle [15] which states that "the granule of reuse is the granule of release".

In a sense, both factors co-exist in most cases. If two code artifacts change together, then most probably they will be released together. Vice versa, if two code artifacts are reused together, then most probably they will both change together, or at least they will be affected by each other's change.

Types of software components

Next, in the remaining part of this section, we will cover the following types of software components. These are the most universal and commonly used ones[4]:

1. Functional (or business)
2. Utility
3. Port
4. View
5. Archtypes
6. Architectural style

Type 1: Functional (or Business)

The easiest type of similarity to detect and results in the most cohesive module type is to group code related to the same business area together. This results in a system abstraction which is more comprehensible and easier to read and understand.

Type 2: Utility

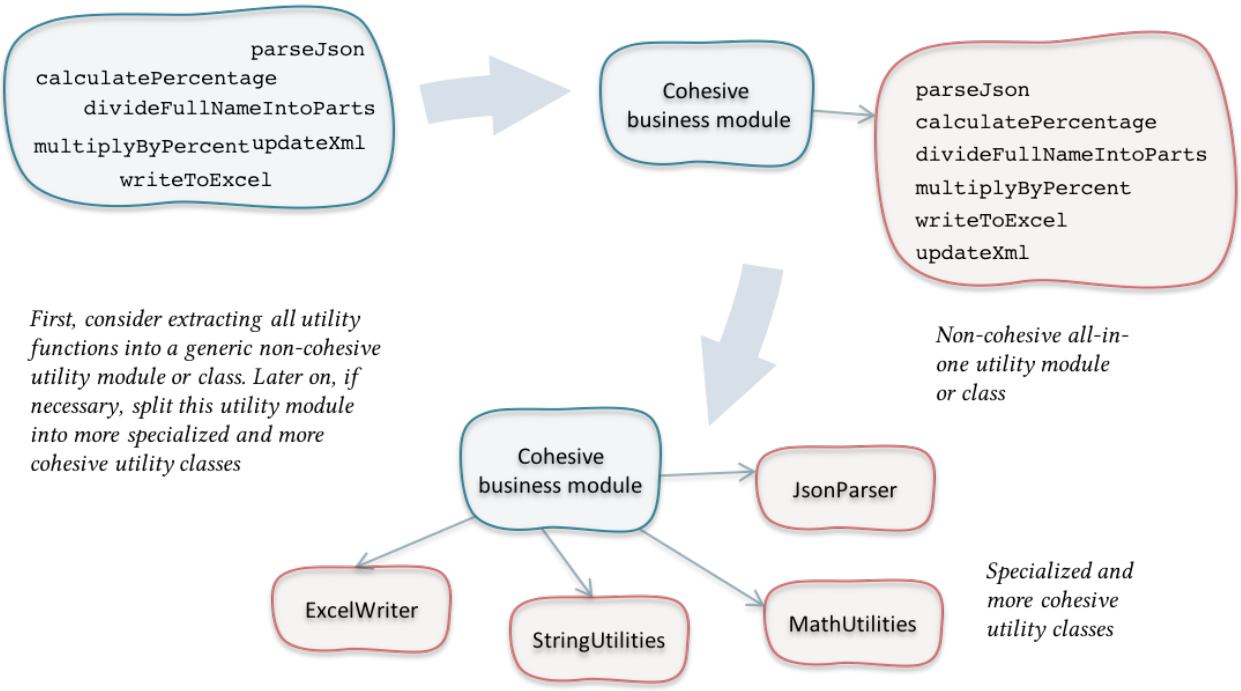
When grouping business functions together, you'll notice parts of the code doing some redundant type of work. Sometimes, this is part of the business function itself, like `validateEmployeeId` for validating an multi-part employee id is correctly formed; or `formatEmployeeName` for preparing a special print name of employees based on their name, department, and hiring date. For this type of redundancy, No need to move them to a separate module.

In other cases, you may detect utility functions which are distinctive and may not relate to the core business functionality. These are some examples of utility functions:

[4]George Fairbanks in his excellent book: **Just enough Software Architecture [14]** discussed most of these component types in detail.

- Standard mathematical or string calculations, like `calculatePercentage(base, percent)` for calculating a percentage out of a base number, or `divideFullNameIntoParts` which returns person first to last names organized into an array
- Batch operations on collections of raw data, like `multiplyByPercent`, which receives a collection of values and returns the same set multiplied by a parameter value
- Reading or writing records from an excel file
- Parsing XML or JSON structures
- etc.

In these cases, the first step is to group all utility functions in a separate generic "utility" module or class. Then, revisit this module and see what groups of utilities emerged and need to be grouped in a separate more cohesive utility class:



Type 3: Port

Port modules are those which encapsulate communication logic to and from a special resource. For example, communicating with web-services, RMI/IIOP, databases, file system, network resources, etc. So, any type of communication which may be needed by more than one *functional modules* should be encapsulated in a standalone module.

Port modules may be considered a subtype of the *Utility modules* because at the end of the day, a port module is a group of utility functions specialized to do a logical job. Even though, I find it very important to think about port modules as a separate type for two reasons:

- Sometimes, it encapsulates some business logic related to how objects or data are prepared or serialized before sending or after receiving. So, it may not be pure utility functions.
- This type is almost in all applications and is very commonly used every where. This is why it deserves a special type.

Type 4: View

Any software with a graphical user interface needs one or more view modules. Usually, views are tightly coupled with its corresponding functional modules; therefore, it is tempting to package them together in one deployable component. On the other hand, the *Release Reuse Equivalency Principle* states that *"The granule of reuse is the granule of release"* [15]. Meaning that you should keep an eye on how your components are reused. If part of the component is reused more than another, then it should be placed in a separate deployable release, or component.

The *Common Closure Principle* gives another dimension. It states that: *"Classes that change together are packaged together"* [15]. Sometimes, change in business requires a change in view and vice versa. In this case, following the principle, you should keep both view and business classes in the same component. In contrast, if the changes are usually confined to view or business, you should place each one of them in a separate component.

In summary, package the view or UI code with the business code in one component if they are reused together and change together. If this is not the case, separate them into two components.

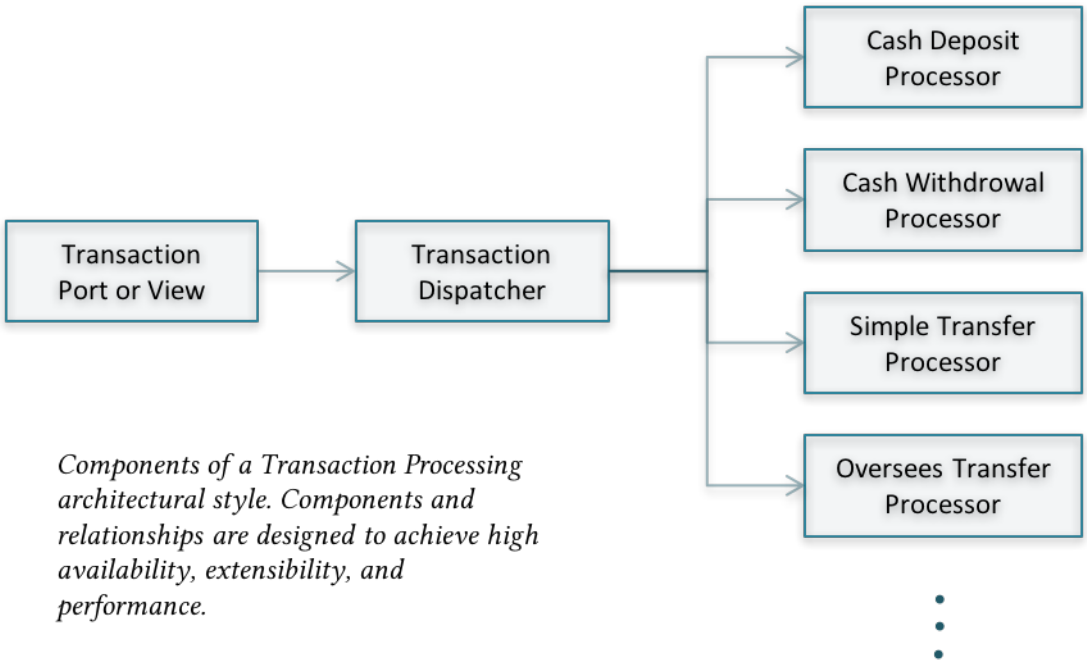
Type 5: Archtypes (aka Core types or Model)

Archtypes are the most noticeable or important data types [14]. Usually, these types are gathered in one core module used by almost all others. Although this raises coupling between this module and the rest of the system, gathering core types in one module reduces the overall coupling among all other modules in the system.

Type 6: Architectural style

You can also partition your code following an architectural style. For example, if you're maintaining a heavy transactional system, a banking system for instance, then probably it will follow a *Transaction Processing* style[5]. In this architectural style, transactions are recorded and processed later on. This is a high level diagram of typical components in transaction processing application:

[5]For a description of the Transaction Processing architectural pattern, refer to Philip Bernstein and Eric Newcomer book: *Principles of Transaction Processing* [14]



Components of a Transaction Processing architectural style. Components and relationships are designed to achieve high availability, extensibility, and performance.

Banking application following the Transaction Processing architectural style

This architectural style has four component types: * **Transaction Port**: Typically a port or view component which receives or input transactions information * **Transaction Dispatcher**: A mediator component responsible for dispatching logged transaction to processors * **Transaction Processors**: Components for processing or handling different types of transactions

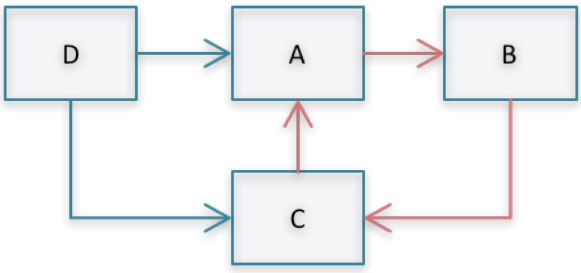
As you may notice, these components can sometimes be considered functional (Transaction Processors), Port or View (Transaction Ports), or Utility (Dispatcher). Even though, the reason of their existence is the architectural style itself; and components and relationships are defined to fulfill a set of constraints and promote some pre-defined system quality attributes. This is why these components are categorized as "architectural components".

In most cases, you may find glimpses of these architectural styles while you are refactoring old code. Try to honor this structure and enhance its encapsulation.

4.5 Considerations while breaking code apart

Break circular (aka cyclic) dependencies

Circular dependencies occurs when one component depends on another component which in turn depends directly or indirectly on the first one:

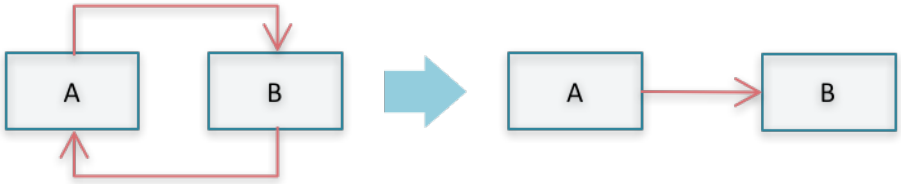


A dependency cycle causing circular dependency between components A, B, and C

You can live with circular dependencies for some time. However, in time, your code may become very complex with higher levels of coupling between components. This will result in more regression type of defects, upfront load time, and possible memory leaks due to cyclic references which never releases used objects. A perfect recipe for how to create spaghetti code!

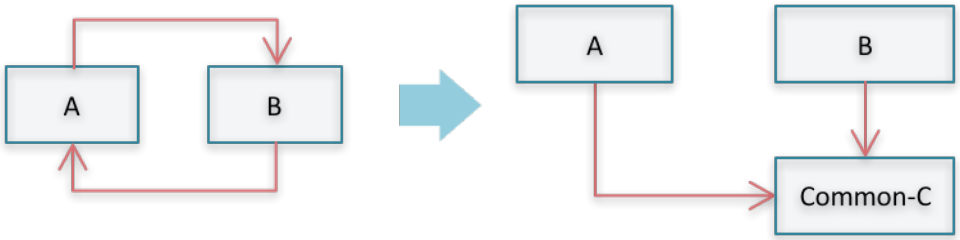
Here are some strategies to break circular dependencies:

1. **Move variable/method/class to the dependent component**. This should always be the first solution to think about, because very often this variable or method was created by a lazy programmer who didn't bother to place things in the proper place. Using an IDE's automated refactoring for moving things around would be the safest, fastest, and cleanest solution.



Move code from A to B, so that B is no longer dependent on A

2. **Extract common logic into a standalone component**, on which both original components depend

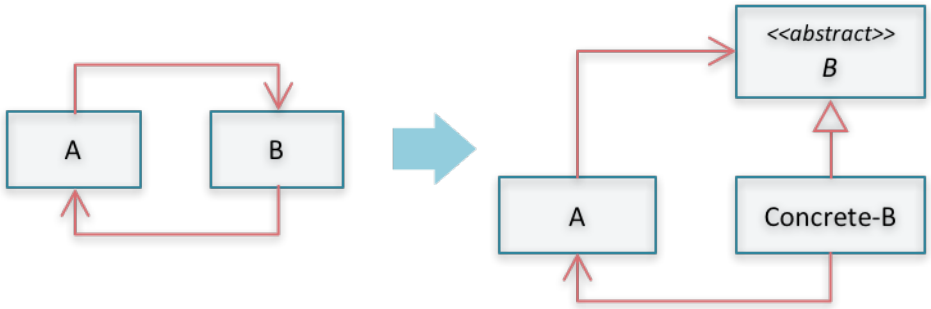


Extract common logic into a standalone component

3. **Apply the Dependency Inversion Principle**[6], which states that *High-level modules should not depend on low-level modules. Both should depend on abstractions.* To do that, split one component (component B in the example below) into two components: One holds the abstractions (the generic definitions of types and interfaces) and the other provides the

[6]This is the sixth principle of the famous SOLID principles of object oriented design by Robert C. Martin [11]

concretions (one default implementation). Then, component A and B depends on the newly-cerated abstract component.



Apply the Dependency Inversion Principle

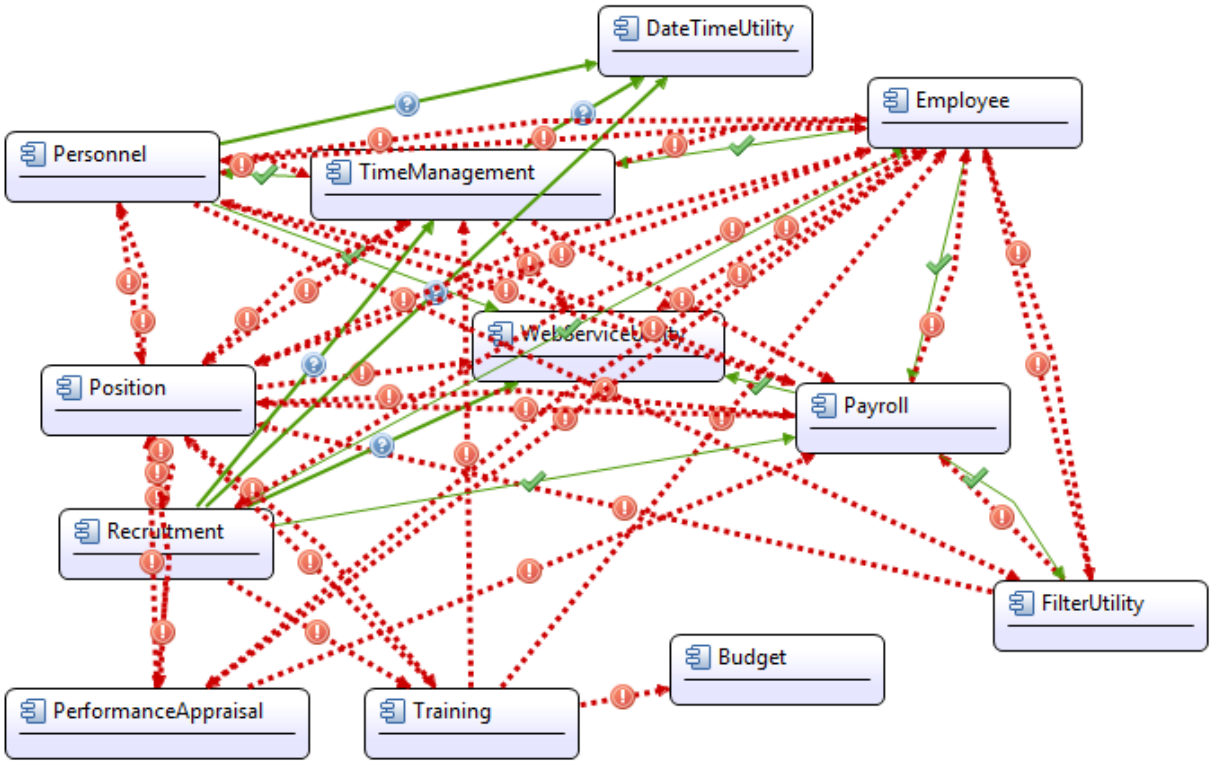
4. **Make use of some architectural patterns**, like the Observer pattern. In my experience, this may be considered a large refactoring at this stage and breaks the ground rules upon which we have agreed at the beginning of this book. In stead, I would resort to one of the previous three solutions.

Start from (and honor) existing architecture

Developers tend to deviate from existing initial architecture for many reasons: lack of design clarity, insufficient documentation, or emergent design consideration which was not handled properly. The volume of these "violations" to initial architecture was found to be from 9% to 19% of all dependencies in the system for healthy project (projects with updated reference architecture) [17].

For poor and tangled code projects, the percentage is much higher. The diagrams below present the amount of violations found in a project I worked with. We first drew the architectural modules and the expected dependencies between them. Then, we used ConQat [7] to check the architecture validity and detect any violations:

[7] Architectural analysis is done by ConQAT, a **Con**tinuous **Qua**lity monitoring tool developed by the Technical University of Munich.



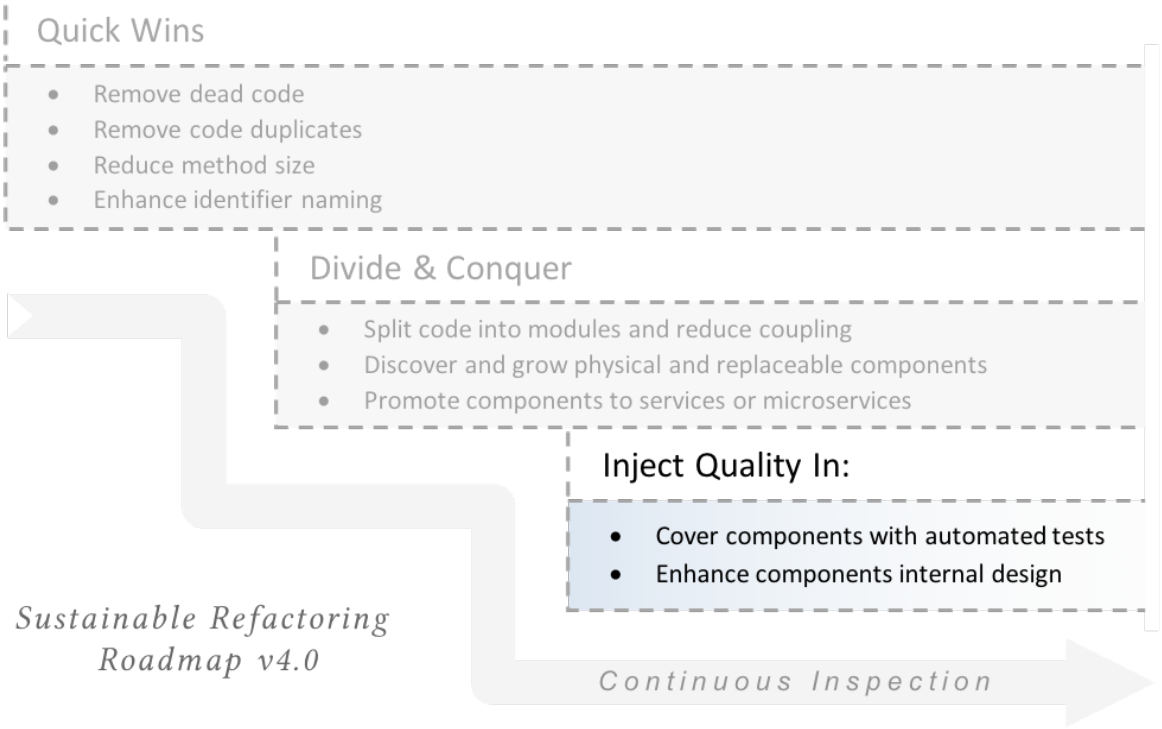
Green lines are the expected dependencies between components. Red lines are relationships and dependencies which do not confirm to expected dependencies, and therefore, they are considered violations.

The above system suffered from so many violations, circular dependencies, many un-necessary calls, and high level coupling among components.

To fix this situation, we started from existing architecture and gradually worked on moving classes and methods around to reduce dependencies and remove violations. Using two simple refactorings: *Move Class* and *Move Method*, we managed to remove most of the violations.

The key takeaway of these experiences is that **existing architectural components should be honored and refined during first attempts to reduce dependencies and lower coupling between components**.

5. Stage 3: Inject Quality In

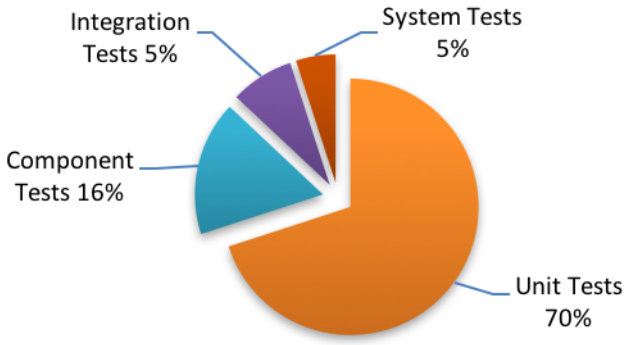


Stage 3: Inject quality in - Cover components with automated tests

The final stage in the roadmap is to cover components with automated tests and make it self-testing. This, in return, enables the team do more profound and risky refactorings.

5.1 Which type of tests?

There are several types of automated developer tests. The following diagram is a typical distribution of automated tests for a “healthy” product:



Distribution of automated tests

In case of high technical debt and poor code structure, coding unit tests on method level while mocking/faking everything else would have very little Return On Investment (ROI) and would take so much time and effort before the team feels any value.

Instead, at this stage, we will concentrate on component, integration, and system tests. These are the 20% of tests which will realize 80% of the value. Also, there are some other reasons which makes such higher-level tests more appealing:

1. In the previous stage (divide & conquer), we have already prepared component interfaces, and they became ready for getting covered by tests
2. Component tests makes the component code *self-testing* and raises our confidence in the component interfaces
3. Still, the internal complexity of the component code is high. Remember that we refrained from doing any risky refactorings so far. This is why unit tests may not be feasible at this stage

5.2 Tracking coverage

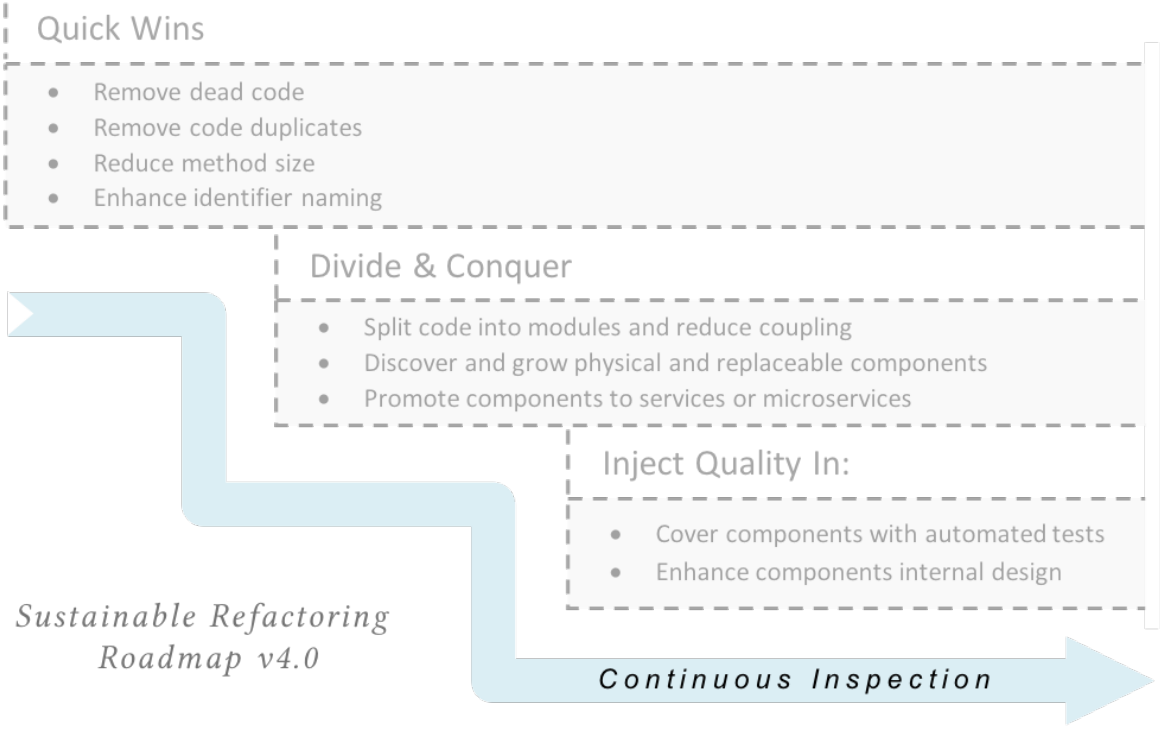
One of the important considerations to take while refactoring is how will you track this work. As mentioned in the Ground rules for sustainable refactoring section, *refactoring effort and outcome should be visible to everybody, including management.* This is particularly important at this stage.

The most famous measure in this stage is test coverage. However, there are several types of test coverage; Line, branch and path coverage.

- **Line coverage** measures the percent lines of code covered by automated tests
- **Branch coverage** measures the percent branches covered by automated tests
- **Path coverage** measures the number of paths/scenarios covered out of the all possible business scenarios

Tracking a single measure is not enough. For example, I can easily achieve 60-70% line coverage, with just 20% path coverage. Also, it's important to review the branches which are not covered. This may give excellent insight about unnecessary conditions in code.

6. Continuous Inspection Throughout the Roadmap



Continuous Inspection - Ensure what is fixed will remain fixed

6.1 Why continuous inspection is important?

< under development >

6.2 Which conventions should be put under continuous inspection?

< under development >

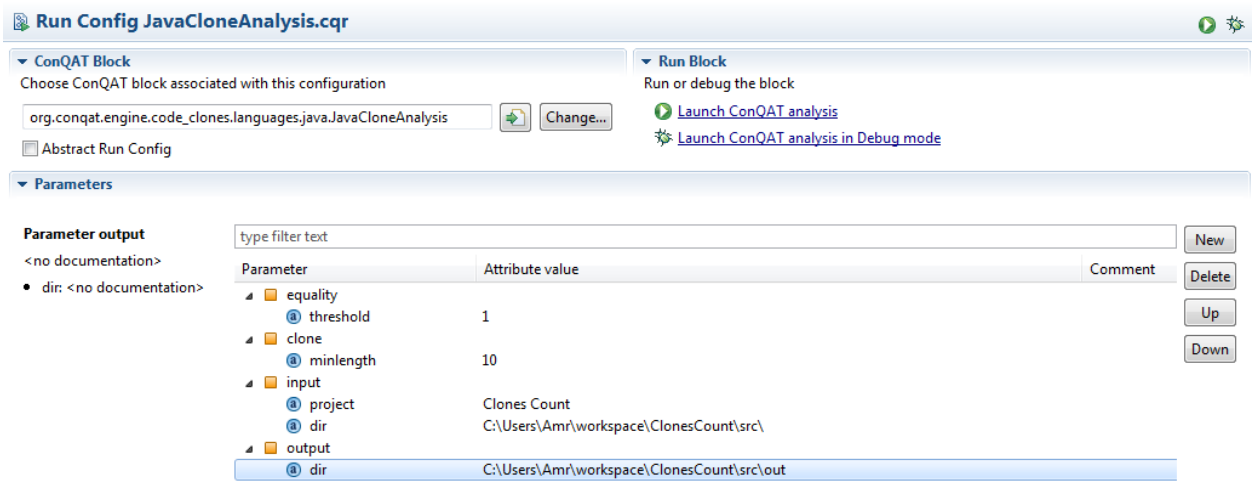
6.3 Example: Using Jenkins and ConQAT to enable continuous clone detection

ConQAT has an excellent engine for detecting code clones (duplicates). Jenkins, on the other hand, can automatically run external tools and continuously check for code quality. If we can merge both, this will become an invaluable tool for technical teams and will save them a lot of review effort.

Step 1: Define Your Policy of Preventing Code Clones

This is done by specifying the ConQat Run configuration files (.cqr) using eclipse. That is, create a cqr file which detects kinds of duplicates that you don't want to see in the code altogether.

For example, you may define a cqr file which detects exact code clones of length greater than 10 lines of code. This is a screen shot of a cqr file which does exactly this job:



ConQat "Run conf" file with parameters

Step 2: Invoke ConQAT cqr from Jenkins

Ant can do this using shell scripts or command lines, as follows:

```
cd <ConQat Location>\conqat\binconqat -f <cqr file location>\JavaCloneAnalysis.cqr
```

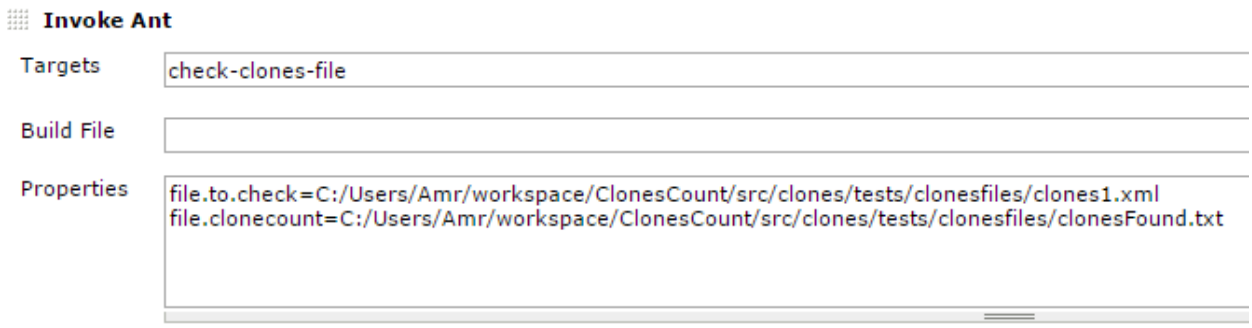
Note: Make sure that you specify the output directory in the configuration file appropriately, because Jenkins will be accessing it later in the next step

Step3: Introspect the clones.xml file, and fail the build if it has clones

Let Ant check the clones.xml file generated. This is an ant target that does this job, add it to your build.xml file:

```
<!--
Check the file named in the property file.to.check (the clons.xml file) to see
if there are any clones.

The way this works is to find all lines containing the text "cloneClass" and put the\
m into a separate file.  Then it checks to see if this file has non-zero length. If \
so, then there are errors, and it sets the property clones.found. Then it calls the \
fail-if-clones-found target, which doesn't execute if the clones.found property isn'\
t set.
-->
<target name="check-clones-file"
        description="Checks the file (specified in ${file.to.check}) for clones">
       <property name="file.to.check" description="The file to hold the clones" />
       <copy file="${file.to.check}" tofile="${file.clonecount}">
              <filterchain>
                     <linecontains>
                           <contains value="cloneClass" />
                     </linecontains>
              </filterchain>
       </copy>

       <condition property="clones.found" value="true">
              <length file="${file.clonecount}" when="gt" length="0" />
       </condition>

       <antcall target="fail-if-clones-found" />
</target>

<!-- Fail the build if clones detected-->
<target name="fail-if-clones-found" if="clones.found">
       <echo message="Code clones found - setting build fail flag..." />
       <fail message="Code clones detected during ${codeline} build.  Check logs." />
</target>
```

Note: In order to pass parameters to the ant script, click on the advanced button and add parameters as in the following screen:



Passing parameters to Ant

7. Measuring Code Quality and Reporting Progress

Measuring code gives you visibility and helps you set clear objectives. As put by Deming:

> **Without data you're just another person with an opinion**.
>
> *- W. Edwards Deming*

Measuring code and making sense of code metrics drives us away from almost all common failure patterns of code refactoring. Mainly, the are very useful in reporting progress and gain support from busy managers. They also help us visualize areas with highest code issues and maximize the gain of our limited time of work.

In this chapter, I will list some of the code metrics which are invaluable in refactoring. I will also explain how to make sense of these metrics and will give several examples of progress and quality indicators which I used in my previous projects. Finally, I will talk about harmful use of metrics in team evaluation.

7.1 Useful code metrics

Code metrics are useful when they indicate the amount of **code smells** you have in code. It's not enough to measure code, what's important is the kind of smell these metrics detect or surface.

The idea of tracing code metrics to code smells is fundamental and crucial in code refactoring. You may become very easily overwhelmed with the amount of metrics about your code, while you only need a couple of them in this stage of refactoring.

For example, if your code has lots of dead code, you may concentrate on code size and nothing else. If you have so many conditionals, you may focus on the cyclomatic complexity and nothing else. If you are covering code with tests, you may focus on code coverage or even one type of coverage metrics and nothing else.

The next table summarizes some of the important code metrics and how they can be useful and what code smells they expose:

TABLE 2. A listing of useful code metrics

Metric	Description	Usage	Related code smells
Code size	Can be measured either in lines of code or number of statements. Lines of code excludes whitespace and preferably excludes comments. Number of statements is a better metric because it is not affected by grouping multiple statements on the same line.	Used throughout the product lifecycle. In case of refactoring poor legacy code, we target to reduce this metric till it reaches a stable lower limit.	Large methods. Large Classes. Unused code. Unnecessary code. Extra features.
Methods with size > 10 LOC	Lengthy methods is a sign of poor code. When a method exceed the threshold of 10 lines of code, most probably it violates the Single Responsibility Principle (SRP). Also, lengthy methods are no longer self explanatory and much less maintainable accordingly. It results in multitudes of problems just because of the lengthy methods.	Should be controlled throughout the project. However, it is so much needed in the Quick-wins stage and specifically in the step: Reducing method size.	Big Methods. Too many conditionals.
Duplication level	Percentage of code duplicated. There are several ways to calculate this number. The idea is to use the same tool and the same set of parameters every time. Basically, this measure takes into account exact and similar clones only.	Used heavily in the quick-wins stage. We rely on it to assess whether we need to continue working on *removing code duplicates* or not.	Duplication is the enemy of clean code

Metric	Description	Usage	Related code smells
Cyclomatic complexity (CC)	Cyclomatic complexity is an indicator of how execution paths one method has. The more execution paths, the more logic and complexity the method contains.	Mainly, it's used during the quick-wins to pinpoint big and complex methods which needs to be refactored. Usually, you may find that CC and method length are both high. So, sometimes I prefer to look at the method length first before the CC	Long Method. Too many conditionals. Switch statement
Code Coverage	Percentage of code covered by automated tests.	Used mainly in the *Inject Quality In* stage.	It helps identify part of code which are not covered by any tests.
Build time	Time elapsed to build, package and deploy the product.	This number is used throughout the project lifecycle. In poor legacy code projects, it may be several days of manual effort to package, test, and deploy. The target is to reach a less-than-an-hour process end-to-end	Long build time. Manual repetitive work.
Method parameters	number of parameters in methods.	Used in the *Divide and Conquer* stage while reviewing and improving the components interfaces.	Long Parameter List.
Class coupling	A measure of how many calls back and forth between two classes or packages. It pinpoints high coupling between classes and packages.	This measure is used mainly in the *Divide and Conquer* stage. It guides you while resolving dependencies and reducing overall coupling in your code.	Feature Envy. Inappropriate Intimacy. Middle Man.

7.2 Making sense of code metrics

A metric is not an indicator. It doesn't indicate anything. What if I told you that your code size is 1.5 million lines of code. Is this good or bad? Is it a big or small number? No body knows, and it is incorrect to start reasoning using single readings of any metric.

To make sense of code metrics, you need to organize or present them in a visual way which elicit more information about your code. Usually, you need to do one or more of these things:

- **Compare metrics to known benchmarks**. For example, method size is bad when it exceeds 10 LOC
- **Compare metrics to desirable targets**. For example, code coverage should be 100%
- **Relate metrics to each other**. For example, build time relation to module code size.
- **Show the metric trend over time**. For example, burn down for the level of duplication over time.

A metric is just a number, while the indicator is a data visualization technique which tells a story around these numbers.

7.3 Examples of progress/quality indicators

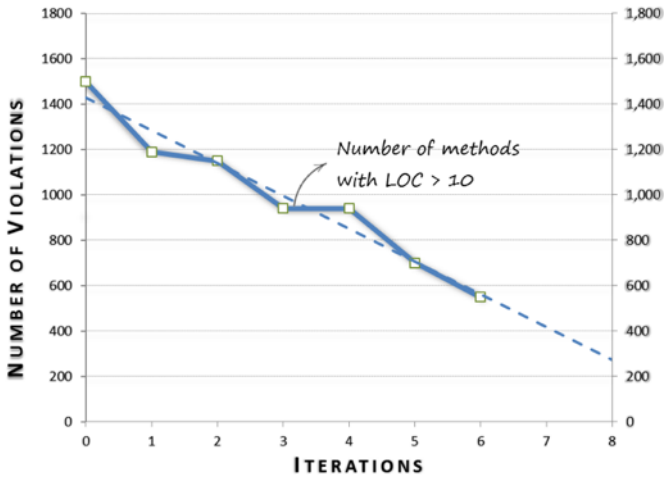
In several refactoring projects, I have used different types of indicators to show progress and assist the team in taking refactoring decisions.

Burn up/down charts

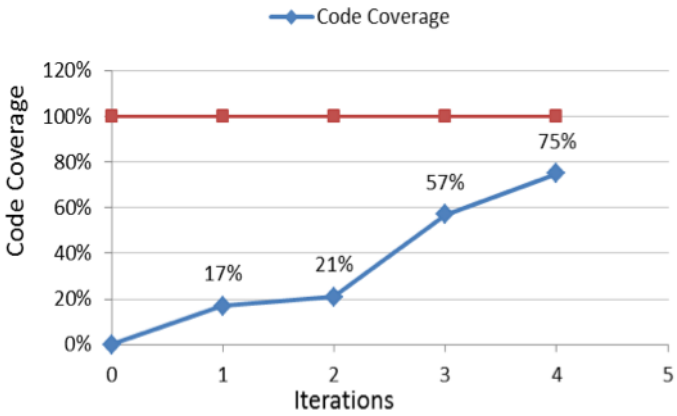
One of the most effective techniques was the burn up/down charts, about which Alistair Cockburn says:

Burn charts have become a favorite way to give visibility into a project's progress. They are extremely simple and astonishingly powerful.

- Alistair Cockburn [^alistair]



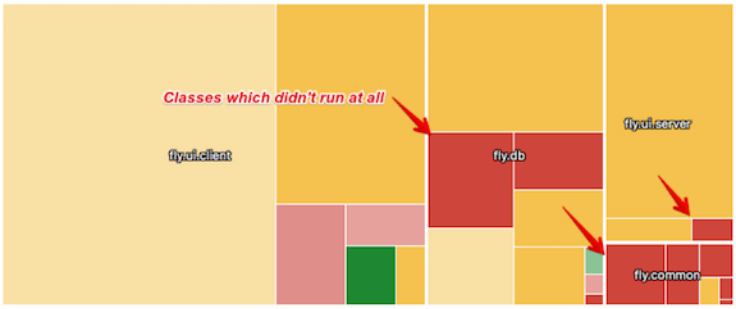
This is an example of using burn down charts when your target is 0. In this case, the target was to clear out all violations of clean code. One example of these violations was methods larger than 10 LOC.



Code coverage burn up chart. The target is to cover all the code with automated tests. Using this chart started with the third stage, and in several iterations, we reached 75%

Tree maps

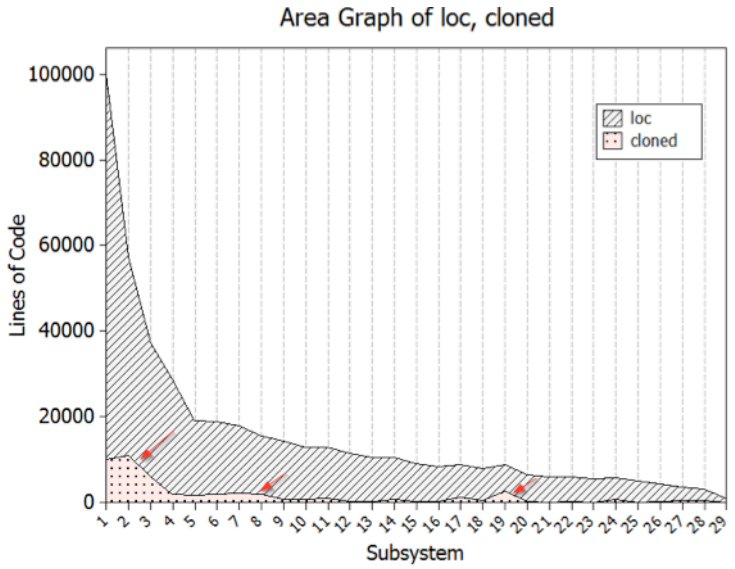
Tree maps is a way of visualizing the topology of code issues throughout the code packages/namespaces. It is very helpful in visualizing the contribution of code parts to the overall issues in the code.



Tree map of the results of dynamic code coverage generated by Clojure. Notice the red areas. These are parts of the code which has never been used by users.

Area graphs

Area graphs are very similar to tree map because they also show the distribution of issues throughout the code parts, but in a different way:



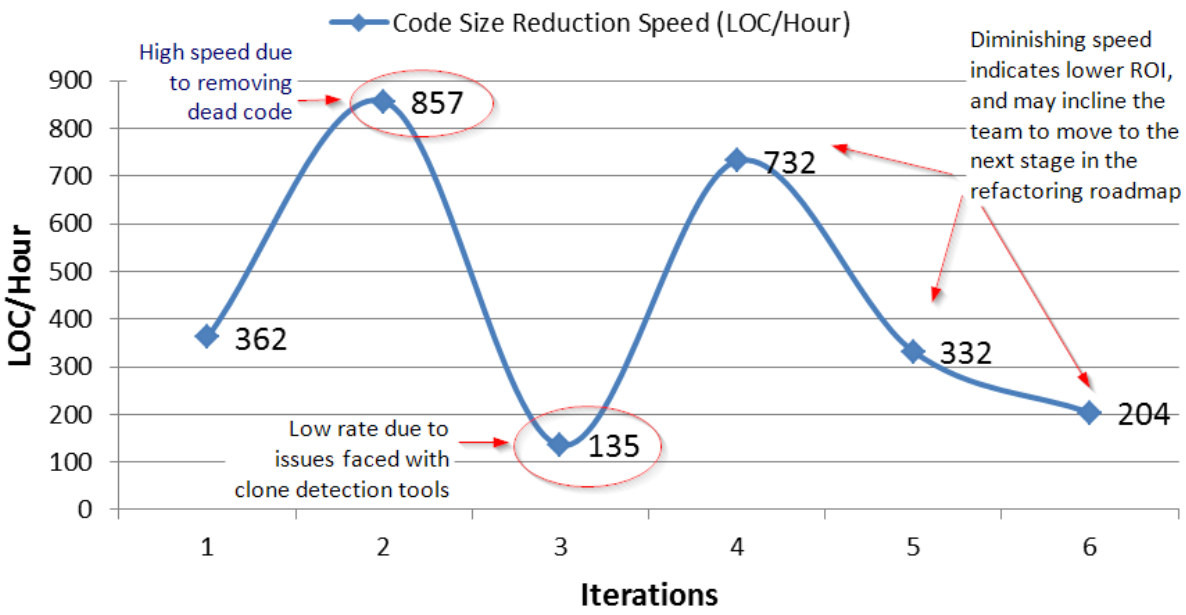
Area graph of modules with highest code duplication level.

Run charts

Run charts are very important because they show the status of code quality over time. They may reveal so much information by analyzing the trend and studying the outliers, if any.

The following is a run chart in one of the refactoring projects I worked on. The code contained more than 52% dead and duplicated code [3]. We started working on removing them but the challenge was to determine when to stop.

We used the following run chart which indicates how much code we remove per one hour of work. This gave us a hint about when to stop working on dead and duplicate code and start working on the next step in the refactoring roadmap:



Run Chart for the Code Size Reduction Speed (CSRS). This chart indicates the amount of return on investment we get out of working on removing dead and duplicate code.

7.4 Using code metrics for team evaluation

Metrics are very powerful tools. If they are used in the wrong direction, they drive negative behaviors, demotivate individuals, and undermine team values.

Code metrics should only be used for enhancing code and showing progress, not for anything else. You should never use them to judge personal capabilities, individual performance, or team productivity, especially if the team is enhancing poor code with already lots of problems. Using code metrics in evaluation and ranking people and teams creates a very unsafe environment and drives negative behaviors. Eventually, you may get better code metrics, but less maintainable code:

> **People with targets and jobs dependent upon meeting them will probably meet the targets – even if they have to destroy the enterprise to do it.**
>
> *- W. Edwards Deming*

One example for that is the code size. As a general rule, you should keep your code base simple and remove any unnecessary code. If you reward the behavior of reducing code size (or punish the otherwise behavior), probably you'll get code like this first code sample below, which is three line of code, instead of the second more readable and self-explanatory code sample, which is fourteen lines of code:

```
public double getSalary(Employee employee) {
        return employee.getBasicSalary() + employee.getChildren().count * (employee.getBasi\
cSalary() * utils.getEduAllownacePer(employee)) + utils.getTransFees(employee.getAdd\
ress());
}
```

Instead of this one:

```
public double getSalary(Employee employee) {
  double basicSalary = employee.getBasicSalary();
  double educationAllowance = getEducationAllowance(employee);
  double transportationAllowance = utils.getTransFees(employee.getAddress());

  double totalSalary = basicSalary + educationAllowance + transportationAllowance;
  return totalSalary;
}

private double getEducationAllowance(Employee employee){
  int numberOfChildren = employee.getChildren().count;
  double allowancePercentage = utils.getEduAllownacePer(employee);
  return numberOfChildren * (employee.getBasicSalary() * allowancePercentage);
}
```

This behavior is widespread, and appears in every single organization. This was noted by Eli Goldratt, the father of Theory of Constraints, who said:

> **Tell me how you measure me, and I will tell you how I will behave!**
>
> *- Eli Goldratt*

My story with peer reviews

A few years ago, I led product development of a business process management suite. My main responsibilities were technical design, supervision, and mentorship. I used to do code reviews for team members and record what we called *review issues* on our issue tracking tool. Recording issues was very healthy because every now and then we used to collect similar issues, think about their root causes, and take prevention actions to stop them from re-occurring.

Everything was ok till the organization designed an employee appraisal system held twice a year. Employee's performance is determined by a complicated formula of many contributing measures, and one of these measures was the number of review issues opened on the person's work! After this point of time, I almost stopped reporting review issues on the issue tracking system. Instead, I was leaving my notes on a piece of paper with the person, or sending them by email.

Although this broke the continuous improvement cycle I described above, my Inner self convinced me that these issues were very minor and not worth the time of reporting them on the issue tracking, especially if these issues would negatively impact my colleagues' evaluation!

This is an example of the negative subtle effect of metrics in organizations when used for individual evaluation. You may not detect their downsides until the damage is already done in the organization.

8. Starting a New Project? Important Considerations

Start with your production environment ready!

One of the most important things you should care about when you start a new project is to get your production environment (or at least a staging one) ready from day one!

Previously, we used to code then deploy. Just before deployment, we start thinking about how we are going to do it. We start answering questions like: Which server? shall we start with a testing environment or create a staging one? Do we need to create a production-like environment or just care about the core services? What would a typical production environment look like in the first place? etc. And usually, we have to wait for some good amount of time till these logistics are sorted out.

In contrary, to start right, flip things around and start by setting up and preparing a production environment, link it to you development environment using proper automatic integration and deployment toolset and scripts.

Dedicate 10% of your time to refactoring

Refactoring is not an activity to do when code become cluttered and full of technical debt. Rather, it is a continuous maintenance activity which keeps the code clean and protects it from deterioration.

Usually, 10% of your time is justified and can be sponsored by higher management, especially if they are educated about the concept of technical debt and the deadly cycle of adding features and accumulating technical debt.

Setup your continuous integration server to check coding conventions and development guidelines

You may chose to do peer reviews to check coding conventions and development guidelines. However, when code grows in size, it may not be efficient nor effective to do it by peer reviews. What you should do is to automate as much as you can, then do peer reviews to pick defects a machine cannot check or detect.

9. Tools of Great Help

As a general rule for all refactoring effort, the more you rely on toolset to carry out the refactoring the safer you are. Not only it will save you a lot of effort chasing and fixing manual errors and unseen side effects, but it will help you see opportunities of refactoring which you may never notice or think about.

In this chapter, I will list some tools which I used and found invaluable while working on refactoring projects. Although these tools may have already been mentioned before in the book, I still see value in listing all tools in one place as a reference, and keep updating this chapter every now and then.

I'm pretty sure that there are tens of other very useful tools which I've never used or even heard about. So, don't limit yourself to this set of tools. These are just examples!

Calculating code metrics

In any improvement activity, measurement is everything, and refactoring is no exception. If you don't measure, it's like traveling in the deserts without a compass. May be you'll travel very far, but no guarantee you're in the right direction.

Eclipse Metrics plugin

This is a very useful eclipse plugin for java-based projects. I always use it for several purposes:

- Count the total lines of code
- Pinpoint lengthy methods, which usually accumulate large technical debt and are first candidates for refactoring
- Get a feeling about coupling between classes and packages in the system. This is just a high level view. Later on, you may use more tools to better guide you while breaking the system apart and reducing coupling between components

Nitriq code analysis

This is another excellent tool. Nitriq uses the familiar LINQ query language to extract metrics from your code. For example:

```
from m in Methods
where m.Calls.Contains(m)
select new { m.MethodId, m.Name, m.FullName };
```

Which lists all recursive methods in your code! I have used Nitriq to measure some very interesting metrics. For example, measure the amount of business logic lines of code and rule out all auto-generated and UI code. Another example is to count all the public methods which are neither constructors nor setters or getters.

Dead code - Static code analysis

There are tons of tools to do static code analysis. I've always used Eclipse and Visual Studio for this purpose. Other tools like SonarQube, FindBugs, and PMD are also very popular. For every other technology, you'll find many tools which embody some interesting code recommendations.

Dead code - Dynamic code analysis

Tools in this category can monitor a live application running in a test or production environment and build a production code usage report exactly similar to the test coverage report.

Clover

Clover is typically used for test code coverage. However, it can be used for dynamic code analysis.

Coverband

For Ruby on Rails, Coverband is a nice and easy tool. It produces output which is SimpleCov compatible and thus can be formatted the same way as test coverage reports produced by SimpleCov.

OpenCover

For .Net applications you can use OpenCover for code usage analysis[1].

Duplicate code

Identifying and removing duplicate code is one of the very important and early steps towards clean code. Without a good tool, your ability to pinpoint and remove duplicates is very minimal.

ConQAT

ConQAT is probably the best tool I've evaluated to detect code clones. It has a very powerful algorithm to detect type 1, 2 and 3 of code clones. It also comes with nice plugin to analyze and visualize duplicates on eclipse, even if you're analyzing languages other than java.

The second best thing about ConQAT is that it is extremely powerful when working with large code bases. I have used it to analyze duplicates for a code base of 5 million LOC. Guess what? it took less than 3 minutes to finish!

The only downside of ConQAT is the very limited community using and supporting this tool. Also, ConQAT 2015 is the last released open-source version of the tool, and TeamScale is the replacement for ConQAT, although you can still use ConQAT and get paid support for it.

Visual Studio

Visual Studio has a very powerful code detection tool which detects duplicates live while you're typing! However, there are two downsides I found. First, it only detects exact and similar code clones (type 1 and 2). The second drawback is that it is not fast enough when analyzing large code

[1] This blog post describes step by step how to do dynamic dead code detection using OpenCover. Accessed May 13, 2018.

bases. I've tried it with 400k lines of code, it took around 35 minutes to complete the code analysis and generate the duplicates list.

SonarQube

SonarQube has a basic component for detecting code clones. I found ConQAT much more powerful in detecting clones. However, with the vast documentation and huge user community, sonarqube may well be a good choice for enabling continuous detection of clones.

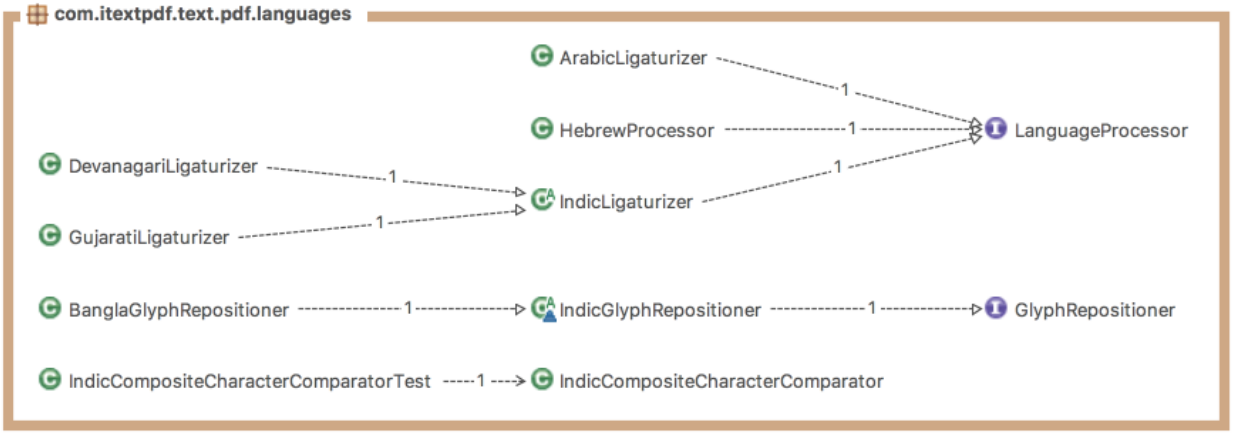
Structural analysis and componentization

ConQAT

ConQAT help define and test existing dependencies in your code. First, you start with visually creating the blueprint of your components, assign code parts (classes or packages) to them, and finally define access rules. ConQAT helps you test whether these rules are really followed and pinpoint any access violations[2].

Stan4J

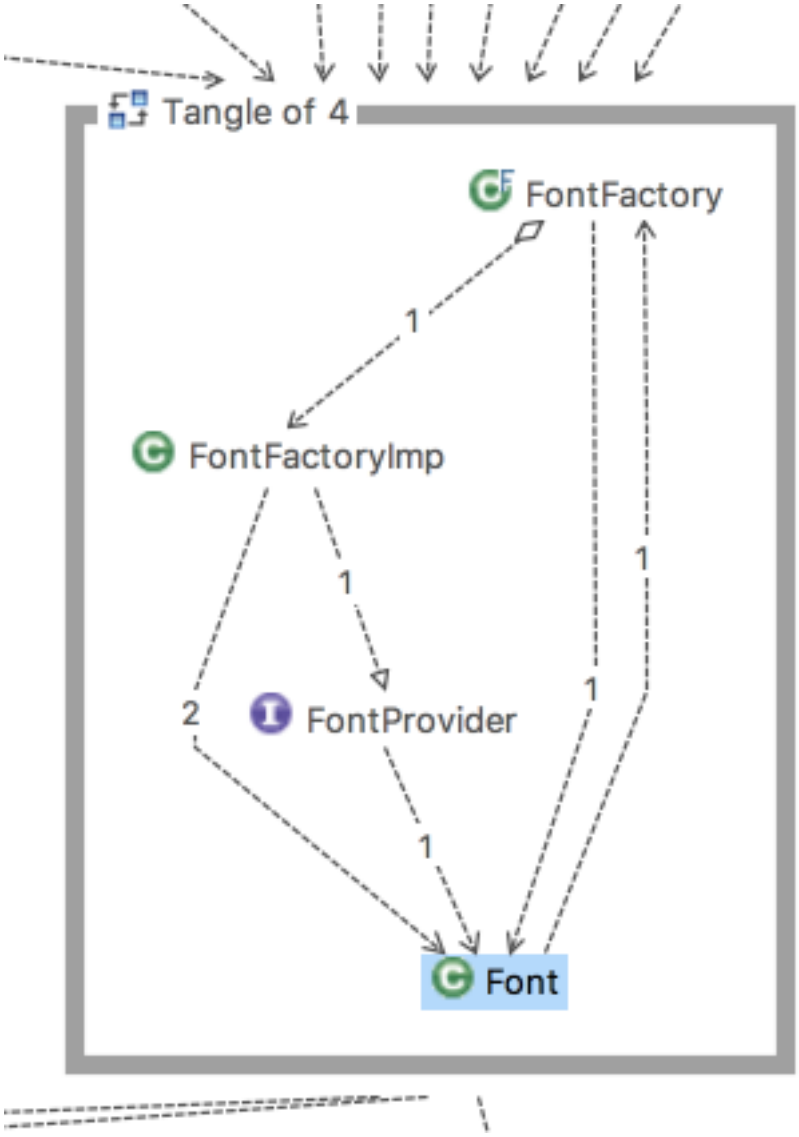
Stan4J does a great job in visualizing dependencies in your code. You can very quickly see dependencies in a nice directed graph with the number of calls put on each arrow:



Directed graphs of dependencies drawn by Stan4J

What it does really good, is organizing code in a way which help you resolve violations. So, if there is a **circular dependency**, Stan4J isolates the interacting classes/packages in what is called **Tangles** in a way which minimizes the number of arrows need to be inverted:

[2]This is a step by step guide from my blog about How to detect architectural violations using ConQAT. Accessed July 15, 2018.



Notice the circular dependency between `Font` *and* `FontFactory` *in this example.*

Catalogue of Useful Refactorings

This chapter is still under development. Basically, these are the most useful refactorings which are commonly used throughout the refactoring roadmap:

Quick-wins stage

- Extract Method
- Move Method
- Rename

Divide and conquer stage

- Extract Class
- Move Class
- Inline Class
- Turn Public Methods Private
- Extract Interface
- Change Method Signature
- Introduce Parameter
- Introduce Parameter Object

References

[1] Dennis D. Smith, *Designing Maintainable Software*, Springer, 1999.

[2] Amr Noaman Abdel-Hamid (2013). *Refactoring as a Lifeline: Lessons Learned from Refactoring*. Agile Conference (AGILE).

[4] Rajiv D. Banker, Srikant M. Datar, and Dani Zweig. *Software Complexity and Maintainability*, pp 251.

[5] Kevlin Henney, *Dead Code Must Be Removed*. An interview with infoq.

[6] *An Empirical Study of Dormant Bugs*, Queen's University, Canada, Rochester Institute of Technology, USA

[7] Elmar Juergens et al. (2009). *Do Code Clones Matter?*. Institut fur Informatik, Technische Universitat Munche.

[8] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, 1999.

[9] Nancy L, John M, Kimberly A (2005) *IS Project Management: Size, Complexity, Practices and the Project Management Office*, Proceedings of the 38th Hawaii International Conference on System Sciences.

[10] Edward E, *On the Relationship between Software Complexity and Maintenance Costs*, Journal of Computer and Communications, 2014, 2, 1-16.

[11] Robert C. Martin (2003). *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall. pp. 127–131

[12] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. pp. 19.

[13] Eric S. Raymond (2003). *The Art of Unix Programming: Unix and Object-Oriented Languages*. Retrieved 2017-9-6.

[14] George Fairbanks (2010). *Just Enough Software Architecture: A Risk-Driven Approach*. Marshall & Brainerd. pp.199

[15] Robert C. Martin. *Principles of Object Oriented Design*. Retrieved 9-9-2017.

[16] Philip A. Bernstein and Eric Newcomer (2009). *Principles of Transaction Processing*. The Morgan Kaufmann Series in Data Management Systems, 2nd Edition.

[17] Florian Deißenbock and Markus Pizka *Concise and Consistent Naming*. Institut fur Informatik, Technische Universitat Munchen. Retrieved 12-September-2017.

[18] Michael C. Feathers (2005). *Working Effectively with Legacy Code*. Prentice Hall PTR.

[19] James Rumbaugh, Ivar Jacopson and Grady Booch (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley.

[20] Alistair Cockburn (2004). *Crystal Clear: A Human-Powered Methodology for Small Teams*. pp. 113