Dr. Saba Alimadadi

CMPT 276: Intro to Software Engineering

Lecture 17: Refactoring

Thanks go to Dr. Frank Tip for allowing the use of his lecture materials.

Some materials are taken from: Refactoring: Improving the Design of Existing Code by Martin Fowler, Kent Beck

Anti-Patterns

- Common coding practices that are not necessarily, but often correlated with bugs.

- If they occur in your software, you're doing it wrong.

The Blob

- One object ("blob") has the majority of the responsibilities, while most of the others just store data or provide only primitive services.
- aka “God Class”
- Solution: refactoring

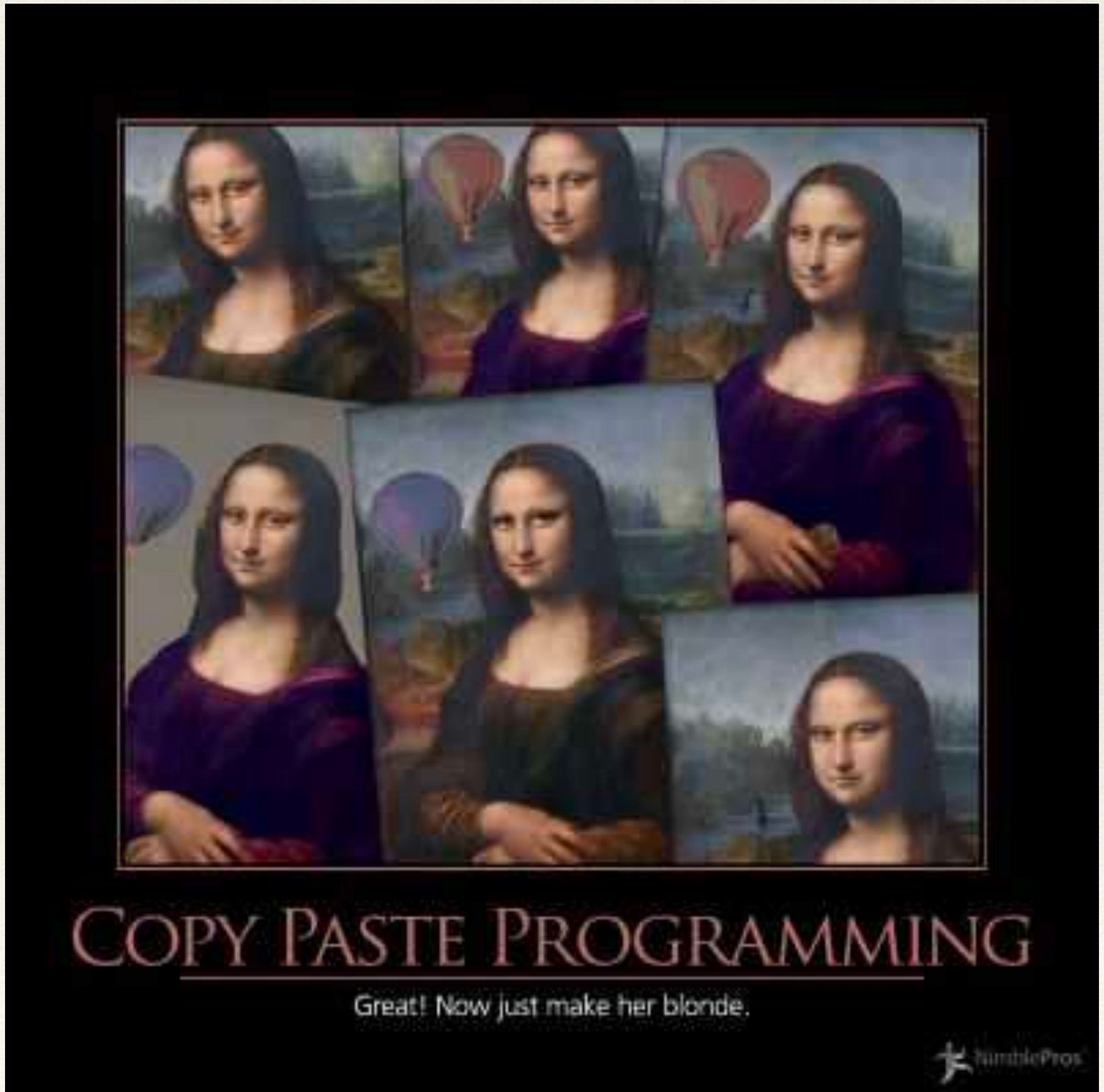
The Golden Hammer

- A favourite solution ("Golden Hammer") is applied to every single problem: With a hammer, every problem looks like a nail.

- Solution: improve education

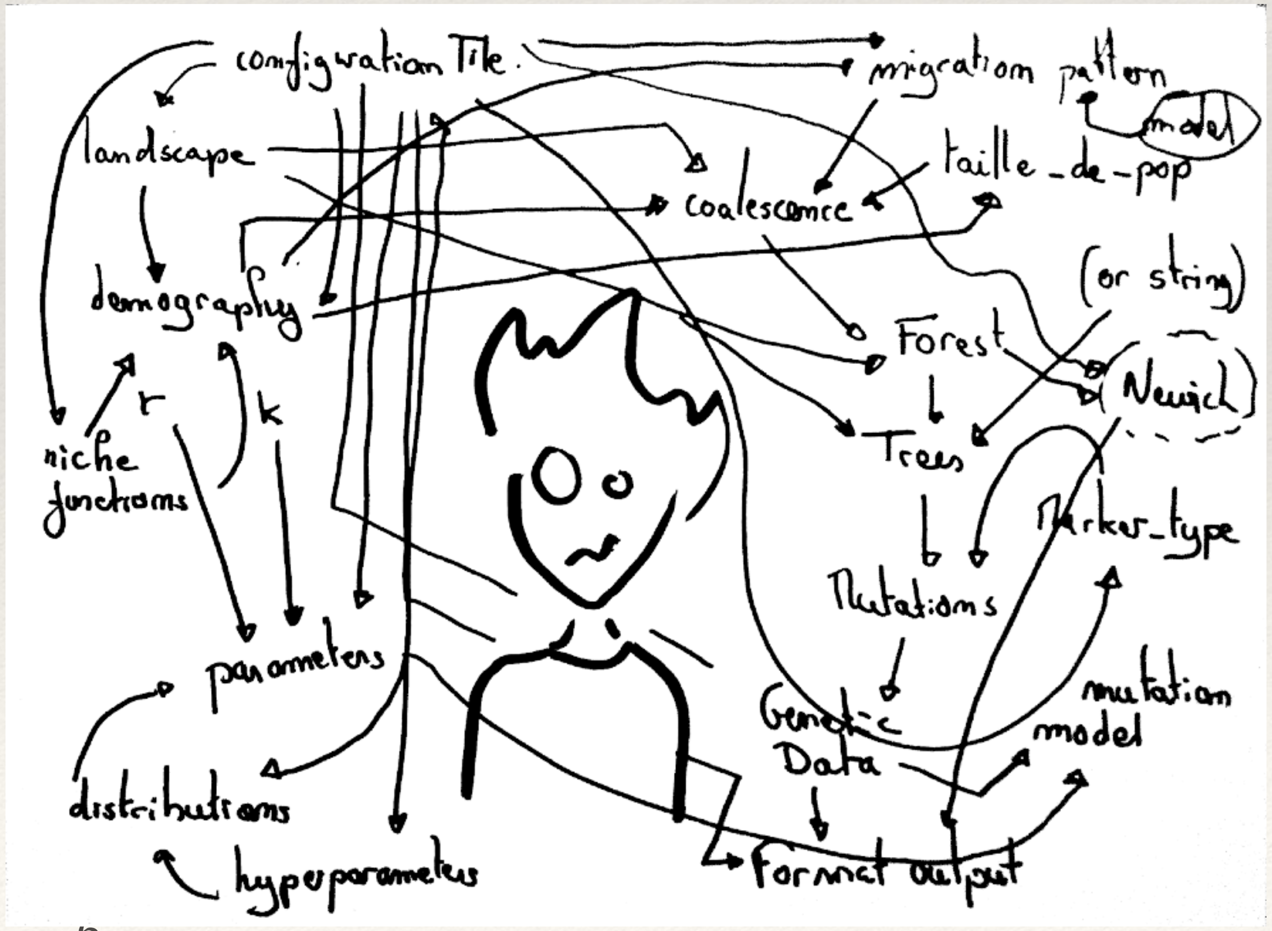
Copy and Paste Programming

- Code is reused in multiple places by being copied and adjusted, causing maintenance problems.
- Solution: refactoring



Spaghetti Code

- The code is mostly unstructured; it's neither particularly modular nor object-oriented; control flow is obscure.
- Solution: Prevent by designing first, and only then implementing. Existing spaghetti code should be refactored.



Mushroom Management

- Developers are kept away from users.

- Solution:
  Improve communication.

Vendor Lock-In

- A system is dependent on a proprietary architecture or data format.
- Solution: Improve portability, introduce abstractions.

Design by Committee

- The typical anti-pattern of standardizing committees, that tend to satisfy every single participant, and create overly complex and ambivalent designs ("A camel is a horse designed by a committee").

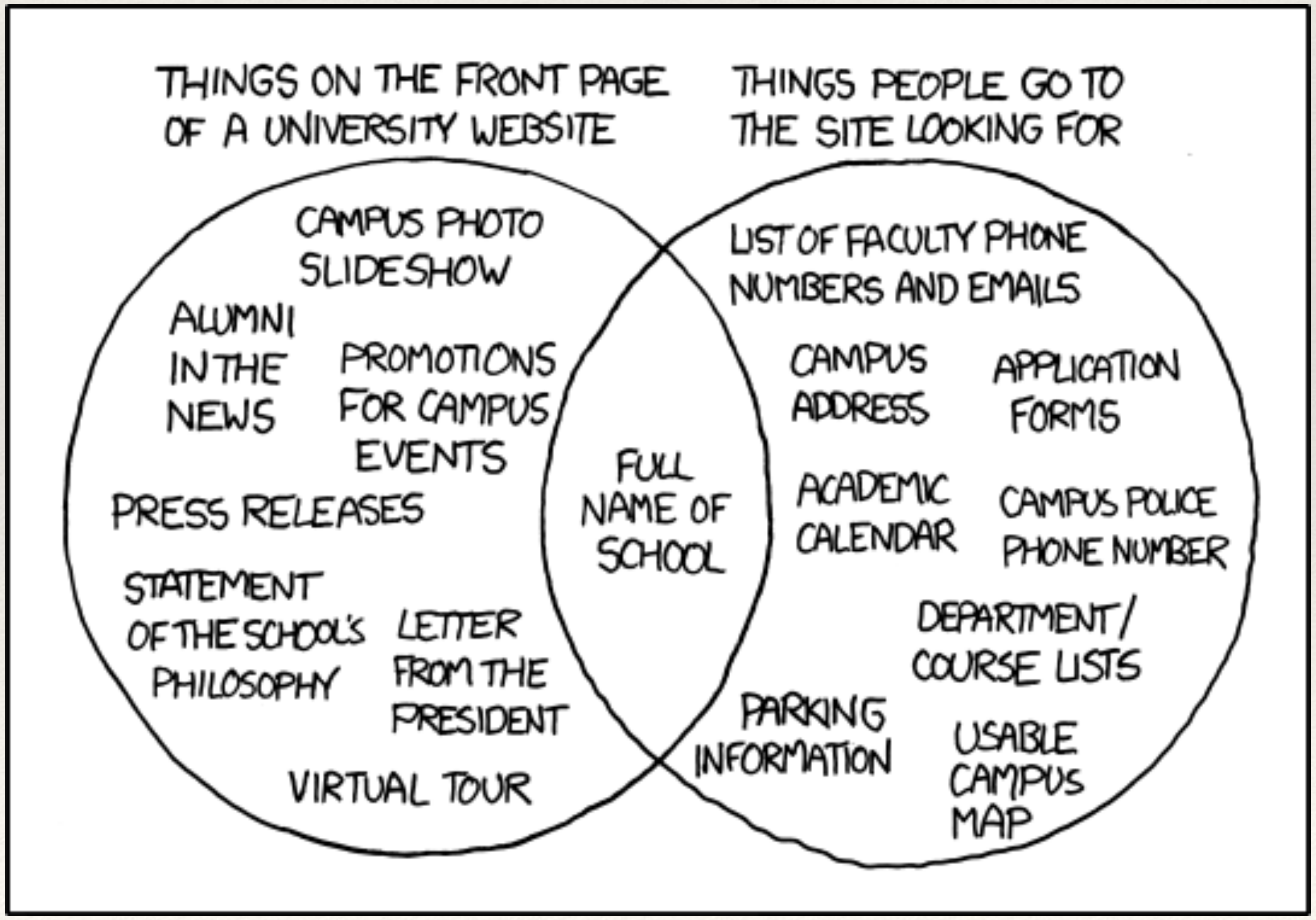
- Known examples:

  SQL and CORBA.

- Solution:

  Improve group dynamics

  and meetings (teamwork)


THINGS ON THE FRONT PAGE OF A UNIVERSITY WEBSITE
THINGS PEOPLE GO TO THE SITE LOOKING FOR
CAMPUS PHOTO SLIDESHOW
ALUMNI IN THE NEWS
PROMOTIONS FOR CAMPUS EVENTS
PRESS RELEASES
STATEMENT OF THE SCHOOL'S PHILOSOPHY
LETTER FROM THE PRESIDENT
VIRTUAL TOUR
FULL NAME OF SCHOOL
LIST OF FACULTY PHONE NUMBERS AND EMAILS
CAMPUS ADDRESS
APPLICATION FORMS
ACADEMIC CALENDAR
CAMPUS POLICE PHONE NUMBER
DEPARTMENT/ COURSE LISTS
PARKING INFORMATION
USABLE CAMPUS MAP

Reinvent the Wheel

- Due to lack of knowledge about existing products and solutions, the wheel gets reinvented over and over, which leads to increased development costs and problems with deadlines.

- Solution: Improve knowledge management.

Intellectual Violence

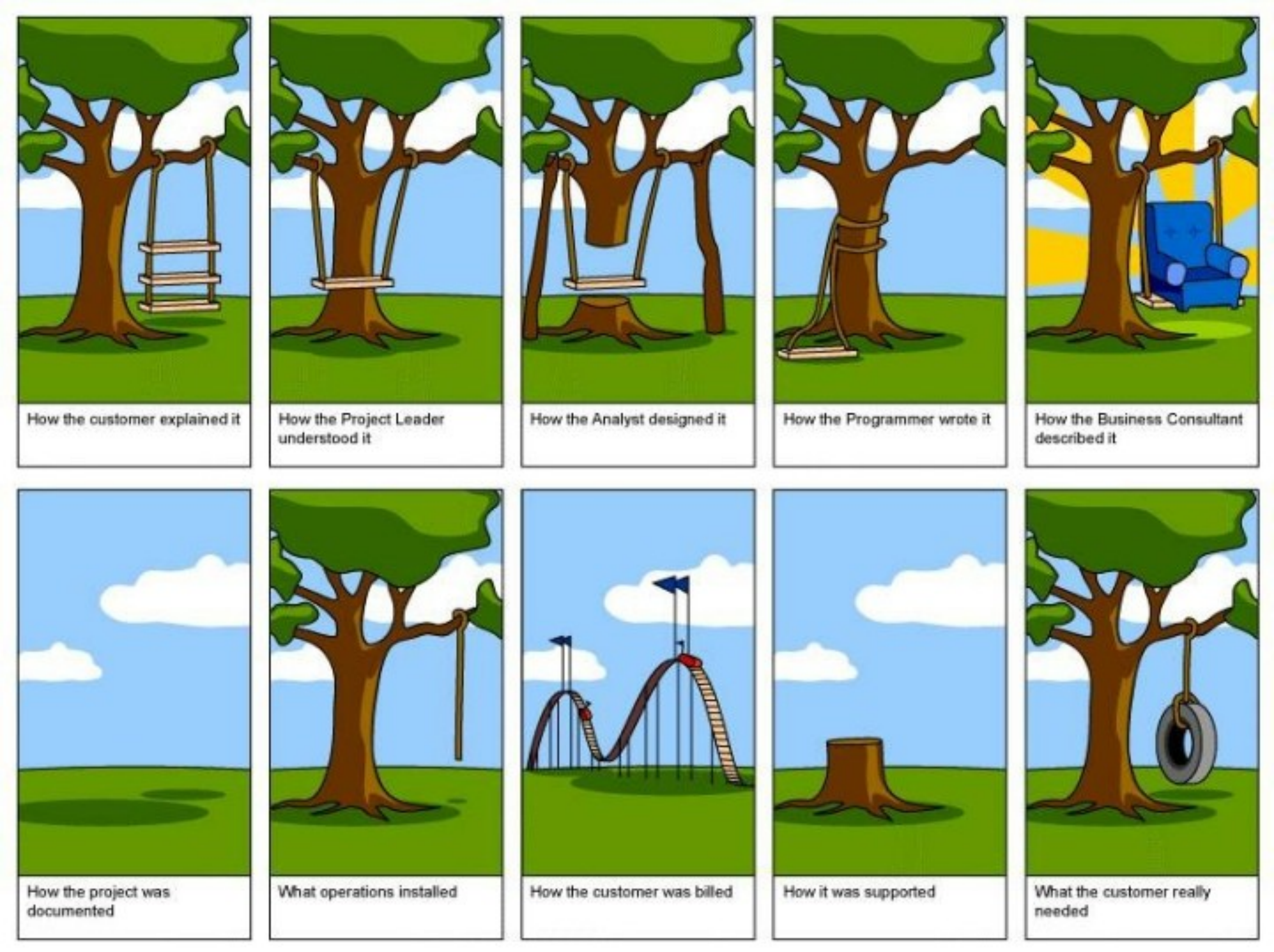
- Someone who has mastered a new theory, technique or buzzwords, uses his knowledge to intimidate others.

- Solution: Ask for clarification!

Project Mismanagement

- The manager of the project is unable to make decisions.

- Solution: Admit having the problem; set clear short-term goals


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

> “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”
>
> *–Martin Fowler*

What is refactoring?

- **Refactoring (noun):**

  a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.

- **Refactor (verb):**

  to restructure software by applying a series of refactorings without changing its observable behaviour.

What is refactoring?

"Is refactoring just cleaning up code?”

Yes, but:

goes further:

- provides a technique for cleaning up code in a more efficient and controlled manner.
- purpose: to make the software easier to understand and modify.

can make many changes in software that make little or no change in the observable behaviour.

Why refactor?

Why refactor?

- Improves the Design of Software
    - Without refactoring, the design of the program will decay.
- Makes Software Easier to Understand
- Helps You Find Bugs
- Helps You Program Faster
    - sounds counterintuitive
    - but a good design is essential for rapid software development
        - Without a good design, you can progress quickly for a while, but soon the poor design starts to slow you down.
        - Changes take longer as you try to understand the system and find the duplicate code.
        - New features need more coding as you patch over a patch that patches a patch on the original code base.
        - Refactoring: stops the design of the system from decaying & can even improve a design.

When to refactor?

- Refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts.
- You refactor because you want to do something else, and refactoring helps you do that other thing.

- **The rule of three**
  - The **first time** you do something, you **just do it**.
  - The **second time** you do something similar, you wince at the duplication, but you do the duplicate thing anyway.
  - The **third time** you do something similar, you **refactor**.

When to refactor?

When to refactor?

- Refactor When You Add Function
  - Helps you understand some code you need to modify
  - Fix the design that does not help you add a feature easily: make future enhancements easier
- Refactor When You Need to Fix a Bug
  - Make code more understandable: code was not clear enough for you to see there was a bug
  - Active process of working with the code helps in finding the bug
- Refactor As You Do a Code Review
  - Code reviews help spread knowledge through a development team
  - Very important in writing clear code: your code may look clear to you but not to your team
  - Gives the opportunity for more people to suggest useful ideas
  - Refactoring also helps the code review have more concrete results

What shouldn’t you refactor?

What shouldn’t you refactor?

- When the **existing code is such a mess** that although you could refactor it, it would be easier to start from the beginning
  - When the current code just does not work
  - Code has to work mostly correctly before you refactor
- A **compromise** route is to refactor a large piece of software into components with strong encapsulation.
  - You can make a refactor-versus-rebuild decision for one component at a time.
- The other time you should **avoid refactoring** is when you are **close to a deadline**
- Other than that, you should not put off refactoring because you haven't got time.
  - Technical debt: pay parts of it off by means of refactoring
  - Refactoring results in increased productivity
  - Not having enough time usually is a sign that you need to do some refactoring.

Bad Smells in Code

- Look for certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring.
- ***No** precise criteria* for when a refactoring is overdue.
  - No set of metrics rivals informed human intuition
  - But we have indications that there is trouble that can be solved by a refactoring
    - You will have to develop your own sense of how many instance variables are too many instance variables and how many lines of code in a method are too many lines.

Duplicated Code

If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

- **same expression**
  - in **two methods of the same class**: *Extract Method* and invoke the code from both places.
  - in **two sibling subclasses**: eliminate this duplication by using *Extract Method* in both classes then *Pull Up Field*.
- **similar code** but not the same:
  - use *Extract Method* to separate the similar bits from the different bits. You may then find you can use *Form Template Method*.
- duplicated code in **two unrelated classes**:
  - consider using *Extract Class* in one class and then use the new component in the other.
  - Another possibility is that the method really belongs only in one of the classes and should be invoked by the other class or that the method belongs in a third class that should be referred to by both of the original classes.
  - You have to decide where the method makes sense and ensure it is there and nowhere else.

Duplicated Code

If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

- **same expression**
  - in **two methods of the same class**: *Extract Method* and invoke the code from both places.
  - in **two sibling subclasses**: eliminate this duplication by using *Extract Method* in both classes then *Pull Up Field*.
- **similar code** but not the same:
  - use *Extract Method* to separate the similar bits from the different bits. You may then find you can use *Form Template Method*.
- duplicated code in **two unrelated classes**:
  - consider using *Extract Class* in one class and then use the new component in the other.
  - Another possibility is that the method really belongs only in one of the classes and should be invoked by the other class or that the method belongs in a third class that should be referred to by both of the original classes.
  - You have to decide where the method makes sense and ensure it is there and nowhere else.

Duplicated Code

If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

- **same expression**
  - in **two methods of the same class**: *Extract Method* and invoke the code from both places.
  - in **two sibling subclasses**: eliminate this duplication by using *Extract Method* in both classes then *Pull Up Field*.
- **similar code** but not the same:
  - use *Extract Method* to separate the similar bits from the different bits. You may then find you can use *Form Template Method*.
- duplicated code in **two unrelated classes**:
  - consider using *Extract Class* in one class and then use the new component in the other.
  - Another possibility is that the method really belongs only in one of the classes and should be invoked by the other class or that the method belongs in a third class that should be referred to by both of the original classes.
  - You have to decide where the method makes sense and ensure it is there and nowhere else.

Long Method

- The object programs that live best and longest are those with short methods.
- The longer a procedure is, the more difficult it is to understand.
- Real key to making it easy to understand small methods is good naming.
- You should be much more aggressive about decomposing methods.
  - A heuristic: whenever we feel the need to comment something, we write a method instead that is named after the intention

Large Class

- **Too many instance variables**:
    - Class is trying to do too much: duplicated code cannot be far behind.
    - You can *Extract Class* to bundle a number of the variables.
        - Choose variables to go together in the component that makes sense for each.
    - If the component makes sense as a subclass, you'll find *Extract Subclass* often is easier.
- **Too much code**:
    - Prime breeding ground for duplicated code, chaos, and death.
    - Eliminate redundancy in the class itself.
    - Usual solution: *Extract Class* or *Extract Subclass*
    - A useful trick: determine how clients use the class -> use *Extract Interface* for each of these uses.
    - If your large class is a GUI class, you may need to move data and behaviour to a separate domain object. This may require keeping some duplicate data in both places and keeping the data in sync: *Duplicate Observed Data*

Large Class

- **Too many instance variables**:
  - Class is trying to do too much: duplicated code cannot be far behind.
  - You can *Extract Class* to bundle a number of the variables.
    - Choose variables to go together in the component that makes sense for each.
  - If the component makes sense as a subclass, you'll find *Extract Subclass* often is easier.
- **Too much code**:
  - Prime breeding ground for duplicated code, chaos, and death.
  - Eliminate redundancy in the class itself.
  - Usual solution: *Extract Class* or *Extract Subclass*
  - A useful trick: determine how clients use the class -> use *Extract Interfac*e for each of these uses.
  - If your large class is a GUI class, you may need to move data and behaviour to a separate domain object. This may require keeping some duplicate data in both places and keeping the data in sync: *Duplicate Observed Data*

Long Parameter List

- In OO programs parameter lists tend to be much smaller than in traditional programs.
  - **long parameter lists**: hard to understand | become inconsistent and difficult to use | you are forever changing them as you need more data.
- How to refactor?
  - *Replace Parameter with Method:* when you can get the data in one parameter by making a request of an object you already know about. This object might be a field or it might be another parameter.
  - *Preserve Whole Object*: to take a bunch of data gleaned from an object and replace it with the object itself.
  - *Introduce Parameter Object*: if you have several data items with no logical object.
- Exception:
  - When you explicitly do not want to create a dependency from the called object to the larger object.
  - Still if the parameter list is too long or changes too often, you need to rethink your dependency structure.

Long Parameter List

- In OO programs parameter lists tend to be much smaller than in traditional programs.
  - **long parameter lists**: hard to understand | become inconsistent and difficult to use | you are forever changing them as you need more data.
- How to refactor?
  - *Replace Parameter with Method:* when you can get the data in one parameter by making a request of an object you already know about. This object might be a field or it might be another parameter.
  - *Preserve Whole Object*: to take a bunch of data gleaned from an object and replace it with the object itself.
  - *Introduce Parameter Object*: if you have several data items with no logical object.
- Exception:
  - When you explicitly do not want to create a dependency from the called object to the larger object.
  - Still if the parameter list is too long or changes too often, you need to rethink your dependency structure.

Divergent Change

- When one class is commonly changed in different ways for different reasons.
- E.g., if you look at a class and say, "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument”
- you likely have a situation in which two objects are better than one => *Extract Class*
- That way each object is changed only as a result of one kind of change.

Divergent Change

- When one class is commonly changed in different ways for different reasons.
- E.g., if you look at a class and say, "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument”
- you likely have a situation in which two objects are better than one => *Extract Class*
- That way each object is changed only as a result of one kind of change.

Shotgun Surgery

- Similar to Divergent Change, but opposite
- = every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.
- *Move Method* and *Move Field* to put all the changes into a single class.
- If no current class looks like a good candidate, create one. Often you can use *Inline Class*

Divergent Change

- When one class is commonly changed in different ways for different reasons.
- E.g., if you look at a class and say, "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument”
- you likely have a situation in which two objects are better than one => *Extract Class*
- That way each object is changed only as a result of one kind of change.

Shotgun Surgery

- Similar to Divergent Change, but opposite
- = every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.
- *Move Method* and *Move Field* to put all the changes into a single class.
- If no current class looks like a good candidate, create one. Often you can use *Inline Class*

Divergent change: one class that suffers many kinds of changes.
Shotgun surgery: one change that alters many classes.
Ideally: you want a one-to-one link between common changes and classes.

Feature Envy

- A classic smell: a method that seems more interested in a class other than the one it actually is in.
  - The whole point of objects: to package data with the processes used on that data.
  - Most common focus of the envy: the data.
    - E.g, a method that invokes half-a-dozen getter methods on another object to calculate some value.
- **Cure** = *Move Method*: the method clearly wants to be elsewhere
  - Sometimes only part of the method suffers from envy: use *Extract Method* on the jealous bit and *Move Method* to give it a dream home.
- Sophisticated patterns that break this rule. E.g., Strategy and Visitor from the Gang of Four.
  - Fundamental rule of thumb: put things together that change together.
  - Strategy and Visitor allow you to change behaviour easily, but come at a cost.

Feature Envy

- A classic smell: a method that seems more interested in a class other than the one it actually is in.
    - The whole point of objects: to package data with the processes used on that data.
    - Most common focus of the envy: the data.
        - E.g, a method that invokes half-a-dozen getter methods on another object to calculate some value.
- **Cure** = *Move Method*: the method clearly wants to be elsewhere
    - Sometimes only part of the method suffers from envy: use *Extract Method* on the jealous bit and *Move Method* to give it a dream home.
- Sophisticated patterns that break this rule. E.g., Strategy and Visitor from the Gang of Four.
    - Fundamental rule of thumb: put things together that change together.
    - Strategy and Visitor allow you to change behaviour easily, but come at a cost.

Data Clumps

- Data items tend to be like children; they enjoy hanging around in groups together.
    - E.g., you'll see the same 3-4 data items together in lots of places: fields in a couple of classes, parameters in many method signatures.
- Data items that hang around together -> should be made into their own object.
    - 1. look for where the clumps appear as fields. Use *Extract Class* on the fields to turn the clumps into an object.
    - 2. Focus on method signatures using *Introduce Parameter Object* or *Preserve Whole Object* to slim them down.
- A good test: consider deleting one of the data values: if you did this, would the others make any sense? If they don't -> refactor

Data Clumps

- Data items tend to be like children; they enjoy hanging around in groups together.
  - E.g., you'll see the same 3-4 data items together in lots of places: fields in a couple of classes, parameters in many method signatures.
- Data items that hang around together -> should be made into their own object.
  - 1. look for where the clumps appear as fields. Use *Extract Class* on the fields to turn the clumps into an object.
  - 2. Focus on method signatures using *Introduce Parameter Object* or *Preserve Whole Object* to slim them down.
- A good test: consider deleting one of the data values: if you did this, would the others make any sense? If they don't -> refactor

Primitive Obsession

- Primitive types are your building blocks.
- Objects: blur or even break the line between primitive and larger classes.
  - Java does have primitives for numbers, but strings and dates, which are primitives in many other environments, are classes.
- Don't be reluctant to use small objects for small tasks
  - Replace Data Value with Object on individual data values.
    - If the data value is a type code, use Replace Type Code with Class if the value does not affect behaviour.
    - If you have conditionals that depend on the type code, use Replace Type Code with Subclasses or Replace Type Code with State/Strategy.

Switch Statements

- A symptoms of object-oriented code: comparative lack of switch (or case) statements.
  - The problem with switch statements is essentially that of duplication.
- Most times you see a switch statement you should consider **polymorphism**.
- Often the switch statement switches on a type code.
- If you only have a few cases that affect a single method, and you don't expect them to change, then polymorphism is overkill. In this case *Replace Parameter with Explicit Methods* is a good option.

Other Code Smells

- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
- Comments

The Value of Self-Testing Code

If you look at how most programmers spend their time, you'll find that writing code actually is quite a small fraction. Some time is spent figuring out what ought to be going on, some time is spent designing, but most time is spent debugging.

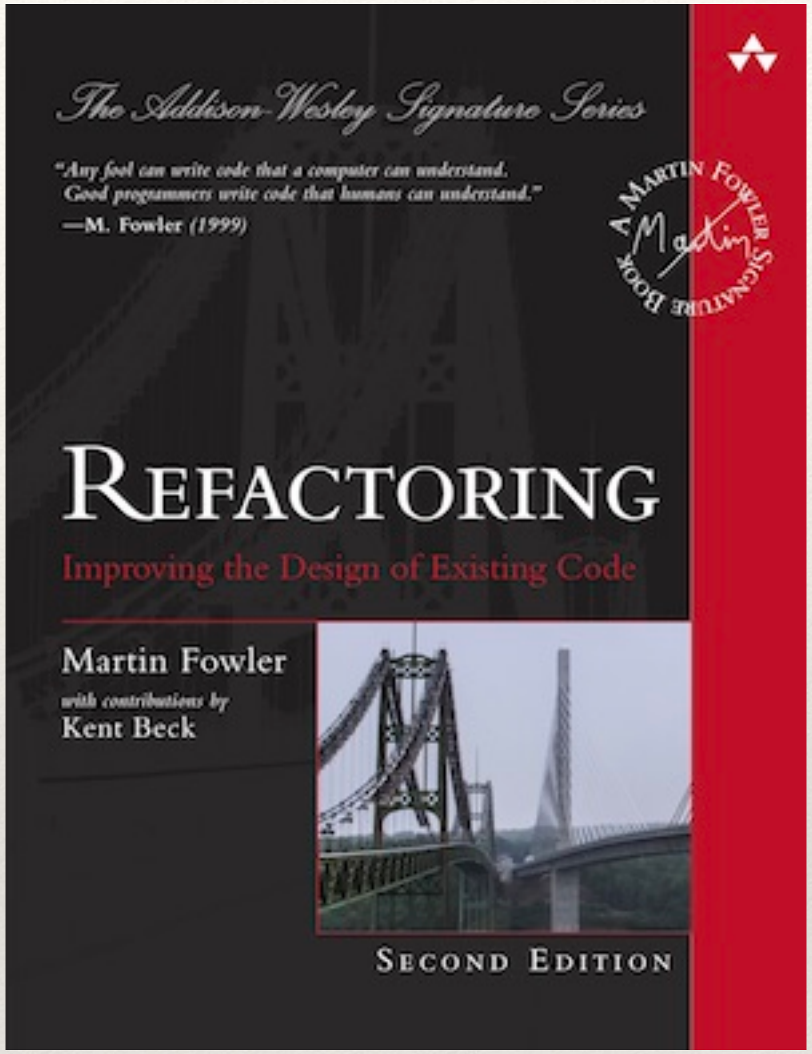
Characteristics of refactoring:

Small Steps + Behaviour-preserving

A main source of the slides:

Refactoring
Improving the Design of Existing Code

by Martin Fowler, with Kent Beck


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Martin
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition