哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 9: Software Refactoring
9.1 Code Smells ⇒ Refactoring

Wang Zhongjie
rainy@hit.edu.cn

May 15, 2018

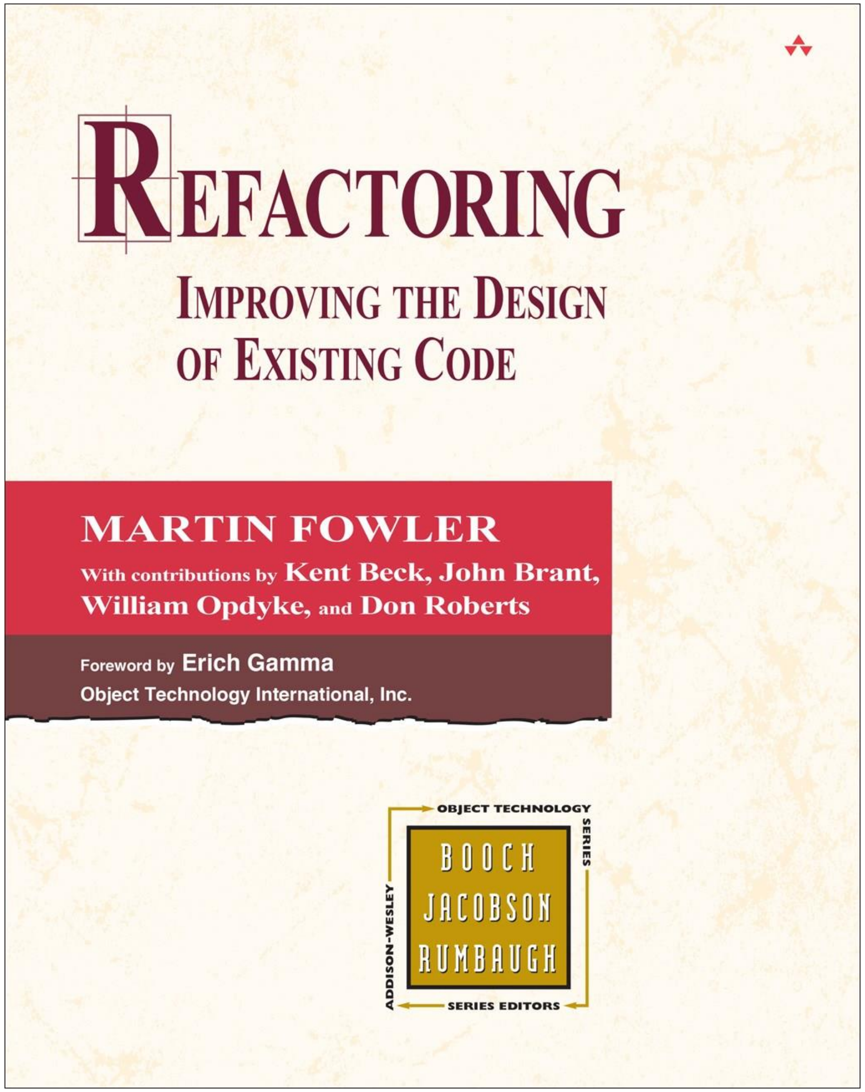
Outline

- **What is refactoring?**
- **Code smells**
- **Refactoring process**
- **Summary**

到本章为止，已学习了面向五个质量指标的软件构造技术。如果你严格遵循相应的构造原则，那么代码质量会很高。

但总有时候，代码引入了各式各样的问题。或者是为了评审他人代码，发现问题，让代码变得更好。

本章关注：代码重构。
本节：什么样的代码是“不好的”？重构的基本过程是什么样？


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.
OBJECT TECHNOLOGY SERIES
BOOCH JACOBSON RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

Reading

- **重构--改善既有代码的设计：第1-3章**
- **代码大全：第24章**

1 What is refactoring?

Problem: "Bit rot"

- **After several months and new versions, many codebases reach one of the following states:**
  - ***Rewritten*** : Nothing remains from the original code.
  - ***Abandoned*** : Original code is thrown out, rewritten from scratch.

- **Why?**
  - Systems evolve to meet new needs and add new features
  - If the code's structure does not also evolve, it will "rot"

  - This can happen even if the code was initially reviewed and well-designed at the time of check-in

Code maintenance

- **Maintenance: Modification or repair of a software product after it has been delivered.**
- **Purposes:**
  - Fix bugs
  - Improve performance
  - Improve design
  - Add features
- **Studies have shown that ~80% of maintenance is for non-bug-fix-related activities such as adding functionality (Pigosky 1997).**

Maintenance is hard

- **It's harder to maintain code than write your own new code.**
  - “House of cards" phenomenon (don't touch it!)
  - Must understand code written by another developer, or code you wrote at a different time with a different mindset
  - Most developers dislike code maintenance
- **Maintenance is how developers spend much of their time.**
- **It pays to design software well and plan ahead so that later maintenance will be less painful.**
  - Capacity for future change must be anticipated

Refactoring

- **Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.**
  - Incurs a short-term time/work cost to reap long-term benefits, and a long-term investment in the overall quality of your system.
- **Refactoring is:**
  - restructuring (rearranging) code...
  - ...in a series of small, semantics-preserving transformations (i.e. the code keeps working)...
  - ...in order to make the code easier to maintain and modify
- **Refactoring is not just any old restructuring**
  - You need to keep the code working
  - You need small steps that preserve semantics
  - You need to have unit tests to prove the code works

What is "refactoring"?

- **Chikosfky and Cross:** "the transformation from one representation to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)"
- **Griswold:** "source-level structural transformations that are guaranteed to preserve the meaning of programs"
- **Opdyke:** "program restructuring operations that preserves the behavior of a program"

```
g[] sett
settings[0].compare
  if (name.compareTo("")
    name += " _";
  }
  name += etr.getString(settings[1]);
else if (settings[0].compareTo("d") == 0) {
  if (name.compareTo("") != 0) {
    name += " _";
  }
  name += DateUtils.format(etr.getDate(settings[1])
} else if (settings[0].compareTo("n") == 0) {
  if (name.compareTo("") != 0) {
    name += " _";
  comSysNumber = etr.getDouble
  f = NumberFormat.ge
  d(false);
```

Code refactoring

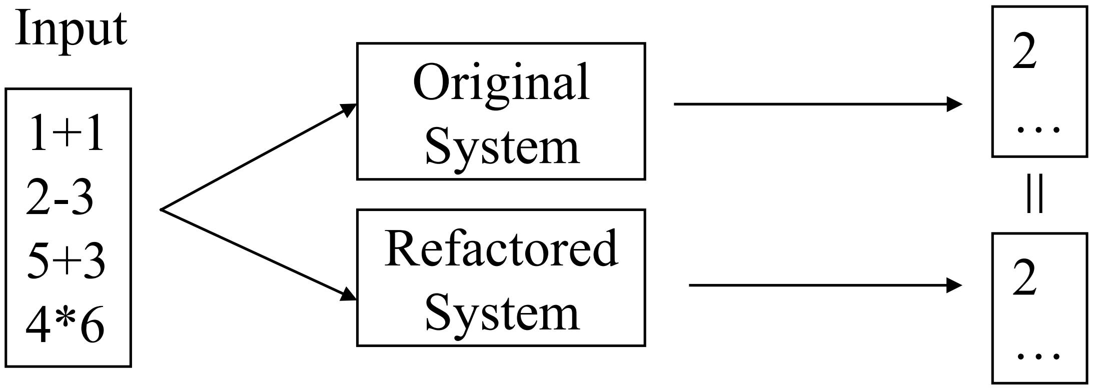
- **Code refactoring is the process of restructuring existing source code without changing its external behavior.**
- **Refactoring improves nonfunctional attributes of the software.**

- **Advantages include improved code readability and reduced complexity;**
- **These can improve source-code maintainability and create a more expressive internal architecture or object model to improve extensibility.**

Martin Fowler (1963-)

Refactoring

- **Refactoring is not the same thing as:**
  - Adding features
  - Debugging code
  - Rewriting code
- **Refactoring is "Behavior Preservation"**
  - A refactoring is a parameterized behavior-preserving program transformation
  - Approaches to behavior preservation may perform checks either statically or dynamically



Why refactor?

- **Why fix a part of your system that isn't broken?**
- **Each part of your system's code has 3 purposes:**

  1. To execute its functionality,
  2. To allow change,
  3. To communicate well to developers who read it.

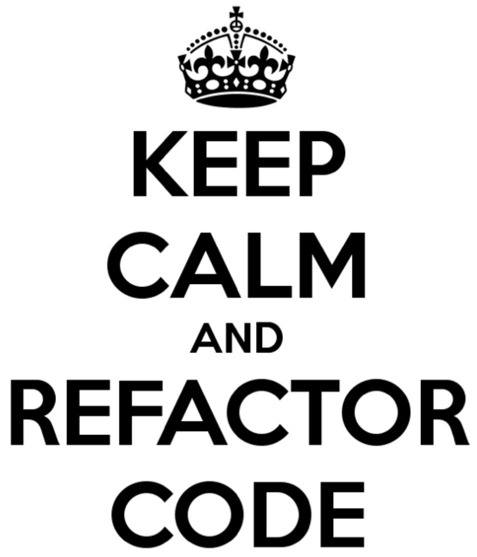
  – If the code does not do one or more of these, it is "broken."
- **Refactoring:**
  – Improves software's design
  – Makes it easier to understand

Any fools can write code that a computer can understand.

Good programmers write
code that humans can understand.

Objectives of Code refactoring

- **Make it easier to maintain**
- **Make it easier to comprehend**
- **Breakdown of modularity --- crosscutting modification (non-localized)**
- **Painful & tedious to write (to add code for a bug fix / feature addition)**
- **Generalize / keeping the program OO**
- **Design conformance**
- **Performance**



When to refactor?

- **When is it best for a team to refactor their code?**
  - Best done **continuously** (like testing) as part of the process
  - Hard to do well late in a project (like testing)
  - Or say, any time that you see a better way to do things (“Better” means making the code easier to understand and to modify in the future)
- **Refactor when you identify an area of your system that:**
  - Isn't well designed
  - Isn't thoroughly tested, but seems to work so far
  - Now needs new features to be added
  - Detect a “bad smell”
- **You should *not* refactor:**
  - Stable code (code that won’t ever need to change)
  - Someone else’s code (Unless you’ve inherited it and now it’s yours)

Example 1: switch statements

- **`switch` statements are very rare in properly designed object-oriented code**
  - Therefore, a `switch` statement is a simple and easily detected “bad smell”
  - Of course, not all uses of `switch` are bad
  - A `switch` statement should *not* be used to distinguish between various kinds of object
- **There are several well-defined refactorings for this case:**
  - Replace conditional with polymorphism
  - Motivation: You have a conditional that chooses different behavior depending on the type of an object.
  - Technique: Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

Example 1, continued

```
class Animal {
   final int MAMMAL = 0, BIRD = 1, REPTILE = 2;
   int myKind;  // set in constructor
   ...
   String getSkin() {
      switch (myKind) {
         case MAMMAL: return "hair";
         case BIRD: return "feathers";
         case REPTILE: return "scales";
         default: return "integument";
      }
   }
}
```

Example 1, improved

```
class Animal {
     String getSkin() { return "integument"; }
}
class Mammal extends Animal {
     String getSkin() { return "hair"; }
}
class Bird extends Animal {
     String getSkin() { return "feathers"; }
}
class Reptile extends Animal {
     String getSkin() { return "scales"; }
}
```

How is this an improvement?

- **Adding a new animal type, such as Amphibian, does not require revising and recompiling existing code**
- **Mammals, birds, and reptiles are likely to differ in other ways, and we've already separated them out (so we won't need more switch statements)**
- **We've gotten rid of the flags we needed to tell one kind of animal from another**
- **Basically, we're now using Objects the way they were meant to be used**

JUnit tests

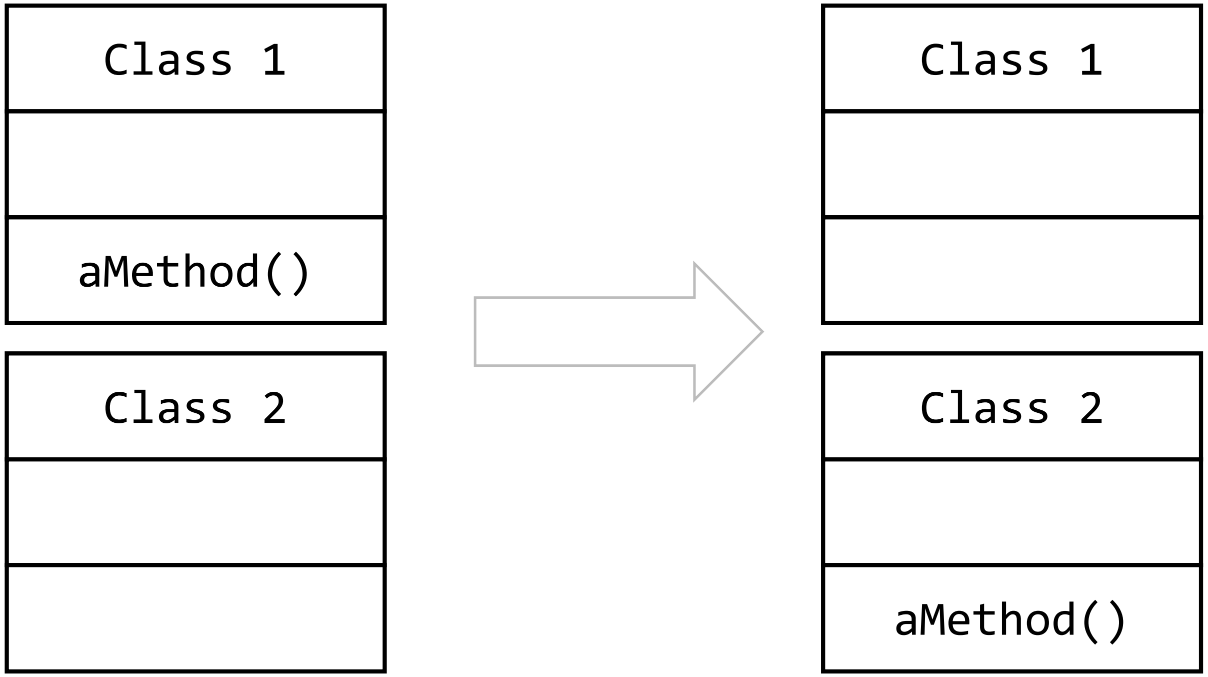
- **As we refactor, we need to run JUnit tests to ensure that we haven't introduced errors**

```
public void testGetSkin() {
    assertEquals("hair", myMammal.getSkin());
    assertEquals("feathers", myBird.getSkin());
    assertEquals("scales", myReptile.getSkin());
    assertEquals("integument", myAnimal.getSkin());
}
```

- **This should work equally well with either implementation**
- **The `setUp()` method of the test fixture may need to be modified**

Refactoring example, 2

- **Move a method:**
  - Motivation: A method is, or will be, using or used by more features of another class than the class on which it is defined.
  - Technique: Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.



Refactoring example, 3

- **Introduce `Null` Object**
  - Motivation: You have many checks for null
  - Technique: Replace the null value with a null object.

```
Customer c = findCustomer(...);
...
   if (customer == null) {
        name = “occupant”;
   } else {
        name = customer.getName();
   }

if (customer == null) {
…
```

Completely eliminated the if statement by replacing checks for null with a null object that does the right thing for “null” values.

```
public class NullCustomer extends Customer {

   public String getName() {
        return “occupant”;
}

--------------------------------------------

Customer c = findCustomer();
name = c.getName();
```

Refactoring example 4

- **Replace Conditional with Polymorphism**
  - Motivation: You have a conditional that chooses different behavior depending on the type of an object.
  - Technique: Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

```
double getSpeed() {
     switch (_type) {
       case EUROPEAN:
            return getBaseSpeed();
       case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * _coconuts;
       case NORWEGIAN_BLUE:
             return (_isNailed) ? 0 : getBaseSpeed(_voltage);
}    throw new RuntimeException ("Should be unreachable");
}
```

Refactoring example 5

```java
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

Don’t Repeat Yourself (DRY)

- Duplicated code is a risk to safety. If you have identical or very similar code in two places, then the fundamental risk is that there’s a bug in both copies, and some maintainer fixes the bug in one place but not the other.
- Copy-and-paste is an enormously tempting programming tool, and you should feel a frisson of danger run down your spine every time you use it. The longer the block you’re copying, the riskier it is.
- **Don’t Repeat Yourself**, or DRY for short, has become a programmer’s mantra.
  - Some of the repetition in `dayOfYear()` is repeated values. Suppose our calendar changed so that February really has 30 days instead of 28. What will happen?
  - Another kind of repetition in the code is “`dayOfMonth+=`…“

Fail Fast

- **Failing fast means that code should reveal its bugs as early as possible.**
  - The earlier a problem is observed (the closer to its cause), the easier it is to find and fix.
  - Static checking fails faster than dynamic checking, and dynamic checking fails faster than producing a wrong answer that may corrupt subsequent computation.
- **The `dayOfYear` function doesn't fail fast — if you pass it the arguments in the wrong order, it will quietly return the wrong answer.**
  - In fact, the way `dayOfYear` is designed, it's highly likely that a non-American will pass the arguments in the wrong order.
  - It needs more checking — either static checking or dynamic checking.

Fail Fast

- **What are the results of the following invocations? (Expected input is February 9, 2019)**
  - `dayOfYear(2, 9, 2019)`
  - `dayOfYear(1, 9, 2019)`
  - `dayOfYear(9, 2, 2019)`
  - `dayOfYear(“February”, 9, 2019)`
  - `dayOfYear(2019, 2, 9)`
  - `dayOfYear(2, 2019, 9)`

- **Static error? Dynamic error? Wrong answer? Right Answer?**

Fail Fast

- Which of the following changes would make the code fail faster if it were called with arguments in the wrong order?

```
public static int dayOfYear(String month, int dayOfMonth, int year) {
    ...
}
```

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month < 1 || month > 12) {
        return -1;
    }
    ...
}
```

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month < 1 || month > 12) {
        throw new IllegalArgumentException();
    }
    ...
```

```
public enum Month { JANUARY, FEBRUARY, MARCH, ..., DECEMBER };
public static int dayOfYear(Month month, int dayOfMonth, int year) {
    ...
}
```

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 1) {
        ...
    } else if (month == 2) {
        ...
    }
    ...
    } else if (month == 12) {
        ...
    } else {
        throw new IllegalArgumentException("month out of range");
    }
}
```

Avoid Magic Numbers

- There are really only two constants that computer scientists recognize as valid in and of themselves: 0, 1, and maybe 2.
- All other constants are called `magic` because they appear as if out of thin air with no explanation.
- One way to explain a number is with a comment, or to declare the number as a named constant with a good, clear name.
  - 2, …, 12 would be far more readable as FEBRUARY, …, DECEMBER.
  - 30, 31, 28 would be more readable if they were in a data structure like an array, list, or map, e.g. MONTH_LENGTH[month].
  - The mysterious numbers 59 and 90 are particularly pernicious examples of magic numbers. Not only are they uncommented and undocumented, they are actually the result of a computation done by hand by the programmer. Explicit computations like 31 + 28 make the provenance of these mysterious numbers much clearer.

One Purpose For Each Variable

- **The parameter `dayOfMonth` is reused to compute a very different value — the return value of the function, which is not the day of the month.**
- **Don't reuse parameters, and don't reuse variables.**
  - Variables are not a scarce resource in programming.
  - Introduce them freely, give them good names, and just stop using them when you stop needing them.
  - You will confuse your reader if a variable that used to mean one thing suddenly starts meaning something different a few lines down.
- **Method parameters should generally be left unmodified.**
  - It's a good idea to use `final` for method parameters, and as many other variables as you can.
  - The `final` keyword says that the variable should never be reassigned, and the Java compiler will check it statically.

But, refactoring is dangerous!

- **Refactoring CAN introduce problems, because anytime you modify software you may introduce bugs!**
- **Management thus says:**
  - Refactoring adds risk!
  - It's expensive – we're spending time in development, but not "seeing" any external differences? And we still have to retest?
  - Why are we doing this?

Who initiated "refactoring"?

- **Ward Cunningham and Kent Beck influential people in Smalltalk**
- **Kent Beck – responsible for Extreme Programming**
- **Ralph Johnson a professor at U of Illinois and part of "Gang of Four"**
- **Bill Opdyke – Ralph's Doctoral Student**
- **Martin Fowler - http://www.refactoring.com**
  - Refactoring : Improving the Design of Existing Code

2 Code smells (code hygiene)

Signs you should refactor

- **Code is duplicated**
- **A routine is too long**
- **A loop is too long or deeply nested**
- **A class has poor cohesion**
- **A class uses too much coupling**
- **Inconsistent level of abstraction**
- **Too many parameters**
- **To compartmentalize changes (change one place → must change others)**

Signs you should refactor

- **To modify an inheritance hierarchy in parallel**
- **To group related data into a class**
- **A "middle man" object doesn't do much**
- **Poor encapsulation of data that should be private**
- **A weak subclass doesn't use its inherited functionality**
- **A class contains unused code**

Code "smells“ (hygiene)

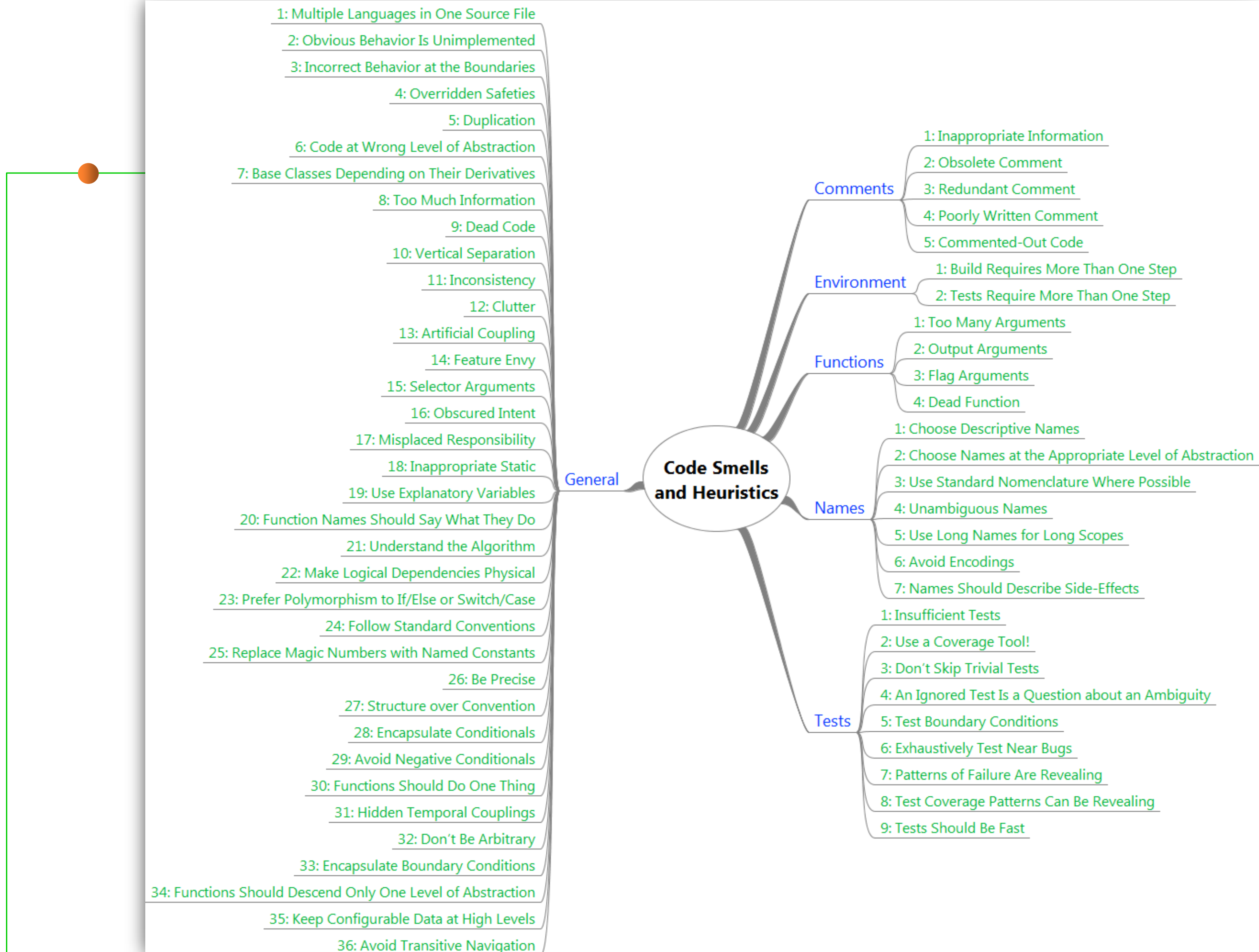
“A code smell is a surface indication that usually corresponds to a deeper problem in the system”.

— —Martin Fowler

- **Duplicated Code**
- **Long Method**
- **Large Class**
- **Long Parameter List**
- **Divergent Change**
- **Shotgun Surgery (change one place → must change others)**
- **Feature Envy**
- **Data Clumps**
- **Primitive Obsession**
- **Switch Statements**
- **Parallel Inheritance Hierarchies**

Code "smells"

- **Lazy Class**
- **Speculative Generality**
- **Temporary Field**
- **Message Chains**
- **Middle Man**
- **Inappropriate Intimacy**
- **Alternative Classes with Different Interfaces**
- **Incomplete Library Class**
- **Data Class**
- **Refused Bequest (subclass doesn't use inherited members much)**
- **Comments**


Code Smells and Heuristics
General
1: Multiple Languages in One Source File
2: Obvious Behavior Is Unimplemented
3: Incorrect Behavior at the Boundaries
4: Overridden Safeties
5: Duplication
6: Code at Wrong Level of Abstraction
7: Base Classes Depending on Their Derivatives
8: Too Much Information
9: Dead Code
10: Vertical Separation
11: Inconsistency
12: Clutter
13: Artificial Coupling
14: Feature Envy
15: Selector Arguments
16: Obscured Intent
17: Misplaced Responsibility
18: Inappropriate Static
19: Use Explanatory Variables
20: Function Names Should Say What They Do
21: Understand the Algorithm
22: Make Logical Dependencies Physical
23: Prefer Polymorphism to If/Else or Switch/Case
24: Follow Standard Conventions
25: Replace Magic Numbers with Named Constants
26: Be Precise
27: Structure over Convention
28: Encapsulate Conditionals
29: Avoid Negative Conditionals
30: Functions Should Do One Thing
31: Hidden Temporal Couplings
32: Don't Be Arbitrary
33: Encapsulate Boundary Conditions
34: Functions Should Descend Only One Level of Abstraction
35: Keep Configurable Data at High Levels
36: Avoid Transitive Navigation
Comments
1: Inappropriate Information
2: Obsolete Comment
3: Redundant Comment
4: Poorly Written Comment
5: Commented-Out Code
Environment
1: Build Requires More Than One Step
2: Tests Require More Than One Step
Functions
1: Too Many Arguments
2: Output Arguments
3: Flag Arguments
4: Dead Function
Names
1: Choose Descriptive Names
2: Choose Names at the Appropriate Level of Abstraction
3: Use Standard Nomenclature Where Possible
4: Unambiguous Names
5: Use Long Names for Long Scopes
6: Avoid Encodings
7: Names Should Describe Side-Effects
Tests
1: Insufficient Tests
2: Use a Coverage Tool!
3: Don't Skip Trivial Tests
4: An Ignored Test Is a Question about an Ambiguity
5: Test Boundary Conditions
6: Exhaustively Test Near Bugs
7: Patterns of Failure Are Revealing
8: Test Coverage Patterns Can Be Revealing
9: Tests Should Be Fast

Code Smells

- **Duplicated Code**
  - The same code structure in more than one place (caused by “code clone”)
  - Bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!
- **Long Method**
  - Long methods are more difficult to understand
  - Performance concerns with respect to lots of short methods are largely obsolete
- **Large Class**
  - Classes try to do too much, which reduces cohesion
  - Often shows up as too many instance variables

Whenever we feel the need to comment something, we write a method instead

Code Smells

- **Long Parameter List**
  - Hard to understand, can become inconsistent and difficult to use.
- **Divergent Change**
  - Related to cohesion
  - Symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset (One class is commonly changed in different ways for different reasons).
- **Shotgun Surgery**
  - Every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.
  - It's hard to identify and make a complete change.

Code Smells

- **Feature Envy**
  - A method seems more interested in a class other than the one it actually is in (e.g., to invoke mane `getter()` methods on another object to calculate some value.
- **Data Clumps**
  - Bunches of data that hang around together, e.g., fields in a couple of classes, parameters in many method signatures
- **Primitive Obsession**
  - Code uses built-in types instead of application classes
  - Consequences: reduces understandability, long methods, code duplication, add complexity
- **Switch Statements**
  - Often you find the same switch statement scattered about a program in different places. If you add a new clause to the switch, you have to find all these switch, statements and change them.

Code Smells

- **Parallel Inheritance Hierarchies**
  - A special case of shotgun surgery: every time you make a subclass of one class, you also have to make a subclass of another.
- **Lazy Class**
  - A class that no longer “pays its way”, e.g. may be a class that was downsized by a previous refactoring, or represented planned functionality that did not pan out
- **Speculative Generality**
  - “Oh I think we need the ability to do this kind of thing someday”
- **Temporary Field**
  - An attribute of an object is only set in certain circumstances; but an object should need all of its attributes

Code Smells

- **Message Chains**
  - When a client asks one object for another object, it then asks for yet another object, and so on.
  - The client is coupled to the structure of the navigation. Any change to the intermediate relationships causes the client to have to change.
- **Middle Man**
  - Most of the methods of a class are delegating to this other class.
- **Inappropriate Intimacy**
  - Classes become far too intimate and spend too much time delving in each others‘ private parts.
- **Alternative Classes with Different Interfaces**
- **Incomplete Library Class**

Code Smells

- **Data Class**
  - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data AND behavior
- **Refused Bequest**
  - A subclass ignores most of the functionality provided by its superclass
  - Subclass may not pass the "IS-A" test
- **Comments (!)**
  - **Comments are sometimes used to hide bad code**
    - **"…comments often are used as a deodorant" (!)**

Many more code smells

- **Many more smells at:**
  - http://c2.com/cgi/wiki?CodeSmell

- **Given a smell, what refactorings are likly?**
  - http://wiki.java.net/bin/view/People/SmellsToRefactorings

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

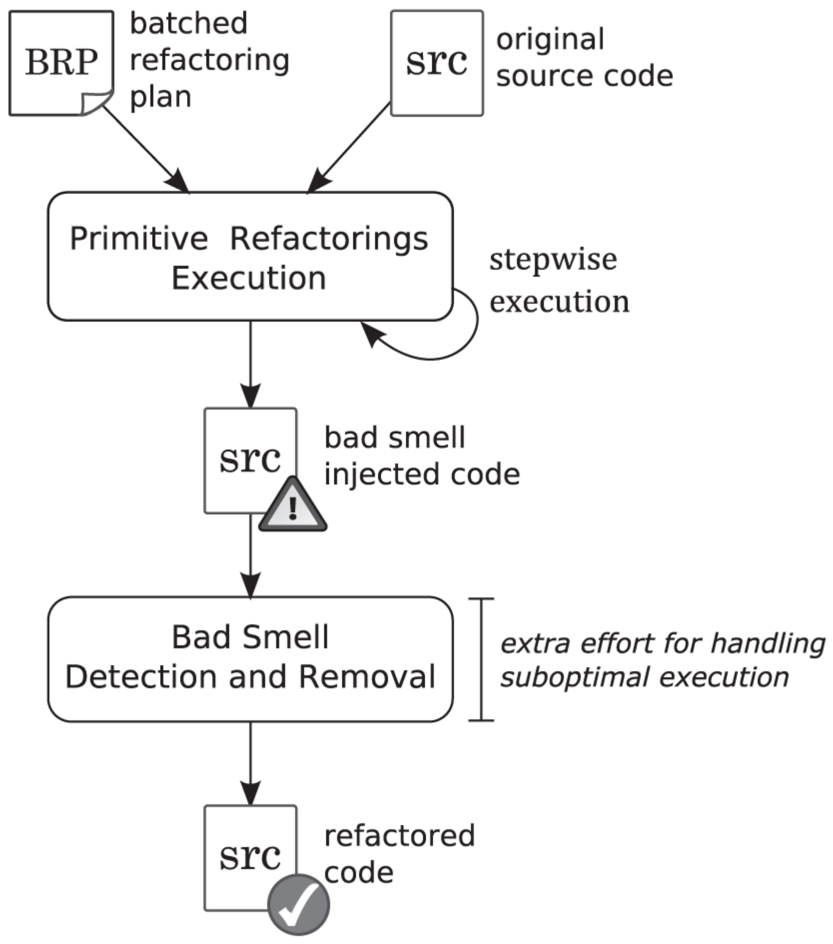
3 Refactoring process

Assertions-based refactoring

- **Identify a set of invariants in the program that preserve its behavior for refactoring**
- **Create a set of conditions which guarantee that these invariants hold**
- **Pre- and postconditions are therefore formulated and checked before and after refactorings are applied**
- **Are a lightweight and automatic means of verification**

Refactoring process

- Identify where the software should be refactored
- Determine which re-factorings should be applied to the identified places
- Guarantee that the applied refactoring preserves behavior
- Apply the refactoring
- Assess the effect of the refactoring on quality characteristics of the software
- Maintain the consistency between the refactored program code and other software artifacts



Refactoring plan, pt 1

- **Save / backup / checkin the code before you mess with it.**
  - If you use a well-managed version control repo, this is done.
- **Write unit tests that verify the code's external correctness.**
  - They should pass on the current poorly designed code.
  - Having unit tests helps make sure any refactor doesn't break existing behavior (regressions).
- **Analyze the code to decide the risk and benefit of refactoring.**
  - If it is too risky, not enough time remains, or the refactor will not produce enough benefit to the project, don't do it.

Refactoring plan, pt 2

(after completing steps on the previous slide)

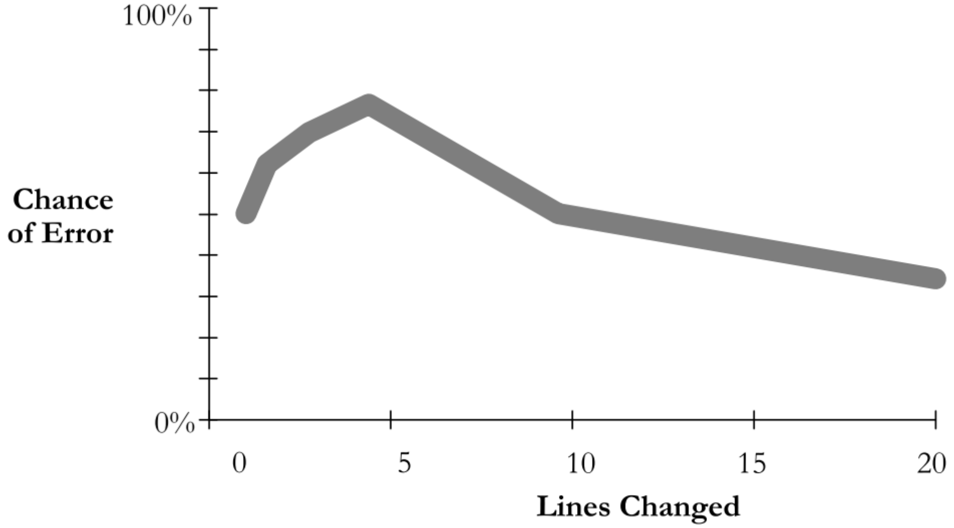
- **Refactor the code.**
  - Some unit tests may break. Fix the bugs.
  - Perform functional and/or integration testing. Fix any issues.
- **Code review the changes.**
- **Check in your refactored code.**
  - Keep each refactoring **small**; refactor one issue / unit at a time.
    - helps isolate new bugs and regressions
  - Your checkin should contain *only* your refactor.
    - NOT other changes such as adding features, fixing unrelated bugs.
    - Do those in a separate checkin.
      - (Resist temptation to fix small bugs or other tweaks; this is dangerous.)

"I don't have time!"

- **Refactoring incurs an up-front cost.**
  - many developers don't want to do it
  - management don't like it; they lose time and gain "nothing" (no new features)
- **But...**
  - well-written code is more conducive to **rapid development** (some estimates put ROI at 500% or more for well-done code)
  - refactoring is good for **programmer morale**
    - developers prefer working in a "clean house"

Dangers of refactoring

- **Code that used to be ...**
  - well commented, now (maybe) isn't
  - fully tested, now (maybe) isn't
  - fully code reviewed, now (maybe) isn't
- **Easy to insert a bug into previously working code (regression!)**
  - a small initial change can have a large chance of error



Regressions

What if refactoring introduces new bugs in previously working functionality ("regressions")? How can this be avoided?

- **Code being refactored should have good unit test coverage, and other tests (system, integration) over it, *before* the refactor.**
  - If such code is not tested, **add tests** first *before* refactoring.
  - If the refactor makes a unit test not compile, **port it**.
  - If the method being tested goes away, the underlying functionality of that method should still be somewhere. So move the unit test to cover that new place in the code.

Company/team culture

- **Organizational barriers to refactoring:**
  - Many small companies and startups don't do it.
    - "We're too small to need it!" ... or, "We can't afford it!"
  - Many larger companies don't adequately reward it.
    - Not as flashy as adding features/apps; ignored at promotion time

- **Reality:**
  - Refactoring is an investment in quality of the company's product and code base, often their prime assets.
  - Many web startups are using the most cutting-edge technologies, which evolve rapidly. So should the code.
  - If a team member leaves or joins (common in startups), ...
  - Some companies (e.g. **Google**) actively reward refactoring.

Refactoring and teams

- **Amount of overhead/communication needed depends on size of refactor.**
  - *small refactor:* Just do it, check it in, get it code reviewed.
  - *medium refactor:* Possibly loop in tech lead or another dev.
  - *large refactor:* Meet with team, flush out ideas, do a design doc or design review, get approval before beginning.

- **Avoids possible bad scenarios:**
  - Two devs refactor same code simultaneously.
  - Refactor breaks another dev's new feature they are adding.
  - Refactor actually is not a very good design; doesn't help.
  - Refactor ignores future use cases, needs of code/app.
  - Tons of merge conflicts and pain for other devs.

Phased refactoring

- **Sometimes a refactor is too big to do all at once.**
  - Example: An entire large subsystem needs redesigning. We don't think we have time to redo all of it at once.

- **Phased refactoring: Adding a layer of abstraction on top of legacy code.**
  - New well-made System 2 on top of poorly made old System 1.
  - System 1 remains; Direct access to it is *deprecated*.
  - For now, System 2 still forwards some calls down to System 1 to achieve feature parity.
  - Over time, calls to System 1 code are replaced by new System 2 code providing the same functionality with a better design.

Refactoring at Google

- **"At Google, refactoring is very important and necessary/inevitable for any code base. If you're writing a new app quickly and adding lots of features, your initial design will not be perfect. Ideally, do *small* refactoring tasks early and often, as soon as there is a sign of a problem."**
- **"Refactoring is unglamorous because it does not add features. At many companies, people don't refactor because you don't get promoted for it, and their code turns into hacky beasts."**
- **"Google feels refactoring is so important that there are company-wide initiatives to make sure it is encouraged and rewarded."**

Refactoring at Google

- **"Common reasons not to do it are incorrect:**
  - a) '*Don't have time; features more important*' -- You will pay more cost, time in adding features (because it's painful in current design), fixing bugs (because bad code is easy to add bugs into), ramping up others on code base (because bad code is hard to read), and adding tests (because bad code is hard to test), etc.
  - b) '*We might break something*' -- Sign of a poor design from the beginning, where you didn't have good tests. For same reasons as above, you should fix your testing situation and code.
  - c) '*I want to get promoted and companies don't recognize refactoring work*' -- This is a common problem. Solution varies depending on company. Seek buy-in from your team, gather data about regressions and flaws in the design, and encourage them to buy-in to code quality."
- **"An important line of defense against introducing new bugs in a refactor is having solid unit tests (*before* the refactor)."**

-- Victoria Kirst, Google

In-Class Exercises

- **Example from http://www.codeproject.com/KB/architecture/practicalexp.aspx**
- **Not written up as an inclass exercise, but could start as an example to find code-smells from.**
- **Article shows refactoring and code smells in action**

Question

- **How to identify "code smells"?**

- **Manually**
- **Automatically**

Summary

Summary

- **Refactoring improves the design of software**
  - without refactoring, a design will "decay" as people make changes to a software system
- **Refactoring makes software easier to understand**
  - because structure is improved, duplicated code is eliminated, etc.
- **Refactoring helps you find bugs**
  - Refactoring promotes a deep understanding of the code at hand, and this understanding aids the programmer in finding bugs and anticipating potential bugs
- **Refactoring helps you program faster**
  - because a good design enables progress

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

May 15, 2018