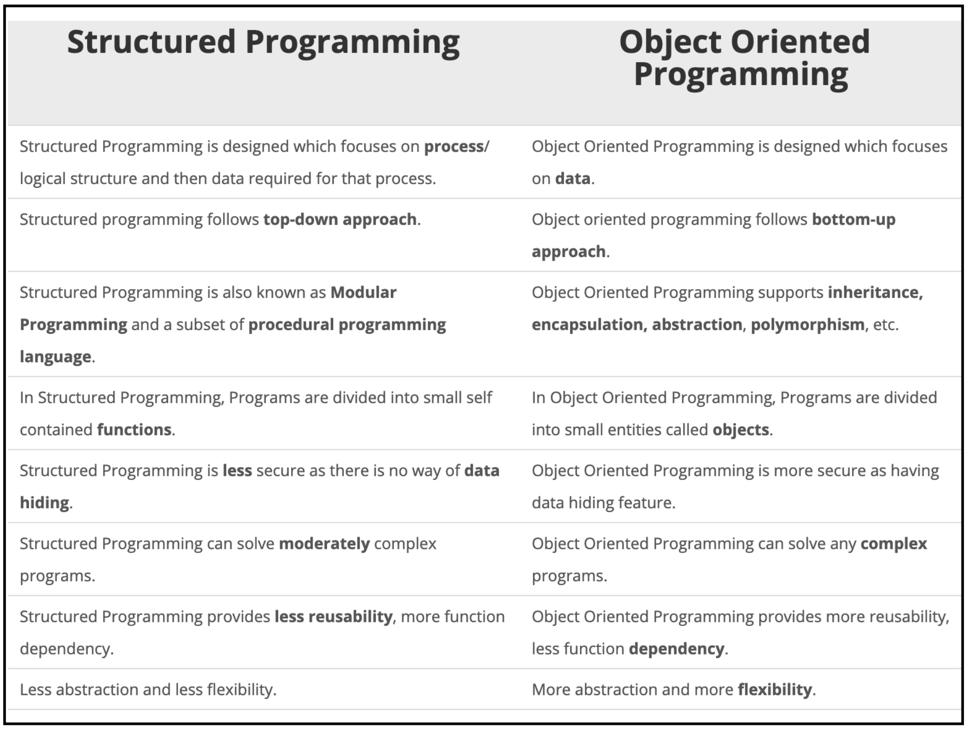
Refactoring

Dr. Nguyen Thanh Hung

1



Structured Programming	Object Oriented Programming
Structured Programming is designed which focuses on **process**/ logical structure and then data required for that process.	Object Oriented Programming is designed which focuses on **data**.
Structured programming follows **top-down approach**.	Object oriented programming follows **bottom-up approach**.
Structured Programming is also known as **Modular Programming** and a subset of **procedural programming language**.	Object Oriented Programming supports **inheritance, encapsulation, abstraction**, **polymorphism**, etc.
In Structured Programming, Programs are divided into small self contained **functions**.	In Object Oriented Programming, Programs are divided into small entities called **objects**.
Structured Programming is **less** secure as there is no way of **data hiding**.	Object Oriented Programming is more secure as having data hiding feature.
Structured Programming can solve **moderately** complex programs.	Object Oriented Programming can solve any **complex** programs.
Structured Programming provides **less reusability**, more function dependency.	Object Oriented Programming provides more reusability, less function **dependency**.
Less abstraction and less flexibility.	More abstraction and more **flexibility**.

2

Why do good developers write bad software?

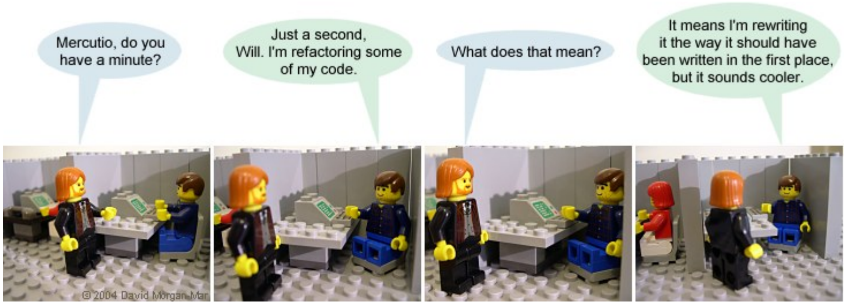
- Requirements change over time, making it hard to update your code (leading to less optimal designs)
- Time and money cause you to take shortcuts
- You learn a better way to do something (the second time you paint a room, it's always better than the first because you learned during the first time!)

Two questions:
1.How do we fix our software?
2.How do we know our software is "bad"… when it works fine!

3

Refactoring

- Definition: Refactoring modifies software to improve its readability, maintainability, and extensibility without changing what it actually does.
- External behavior does NOT change
- Internal structure is improved


Mercutio, do you have a minute?
Just a second, Will. I'm refactoring some of my code.
What does that mean?
It means I'm rewriting it the way it should have been written in the first place, but it sounds cooler.
© 2004 David Morgan-Mar

4

Refactoring

- The goal of refactoring is NOT to add new functionality
- The goal is refactoring is to make code easier to maintain in the future

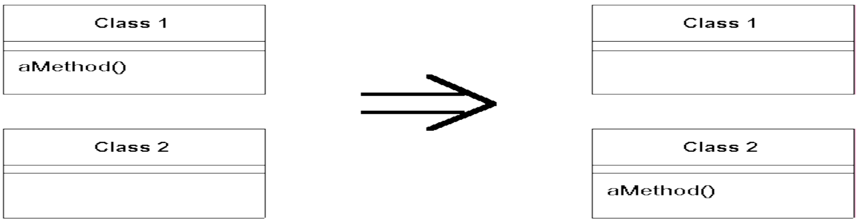
5

Refactoring Simple Example

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.



6

Danger!

- 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?

7

Motivation

- We refactor because we understand getting the design right the first time is hard and you get many benefits from refactoring:
  - Code size is often reduced
  - Confusing code is restructured into simpler code
  - Both of these greatly improve mainatainability! Which is required because requirements always change!

8

Refactoring: Complex Example

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) {
…
```

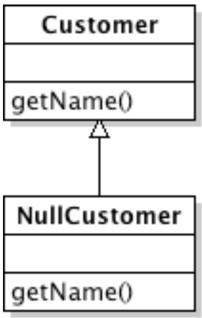
Refactoring: Complex Example

```
public class NullCustomer extends Customer {

  public String getName() {
        return “occupant”
}

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

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



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

Refactoring Example: Replace Conditional with Polymorphism

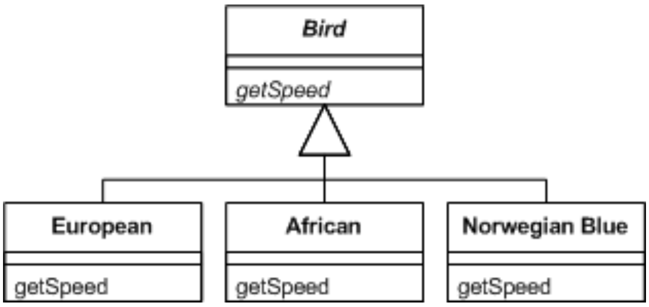
Motivation: *You have a conditional that chooses different behavior depending on the type of an object.*

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() * _numberOfCoconuts;
         case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
  }    throw new RuntimeException ("Should be unreachable");
}
```

11

Refactoring Example: Replace Conditional with Polymorphism



12

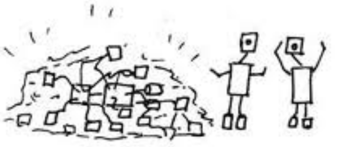
When do I refactor?

- When you add functionality
  - Before you add new features, make sure your design and current code is “good” this will help the new code be easier to write
- When you need to fix a bug
- When you do a peer review

13

How do I identify code to refactor?

- Martin Fowler uses “code smells” to identify when to refactor.
- Code smells are bad things done in code, somewhat like bad patterns in code
- Many people have tied code smells to the specific refactorings to fix the smell



14

Code Smells

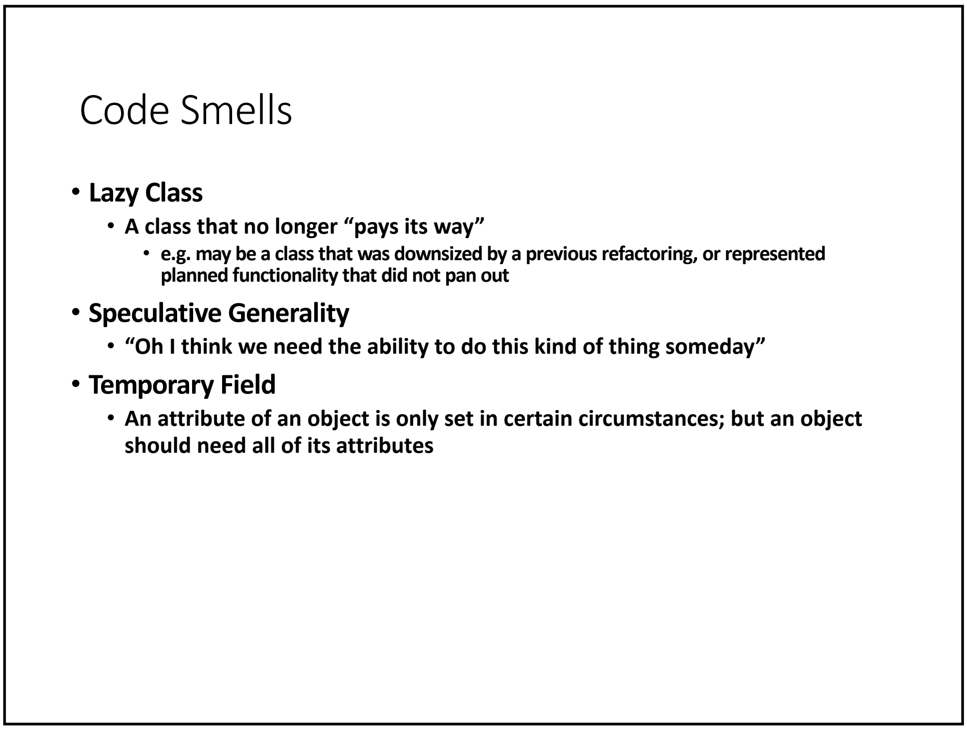
- **Duplicated Code**
  - **bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!**
- **Long Method**
  - **long methods are more difficult to understand**
  - **performance concerns with respect to lots of short methods are largely obsolete**

15

Code Smells

- **Large Class**
  - **classes try to do too much, which reduces cohesion**
- **Long Parameter List**
  - **hard to understand, can become inconsistent**
- **Divergent Change**
  - **Related to cohesion: symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset**

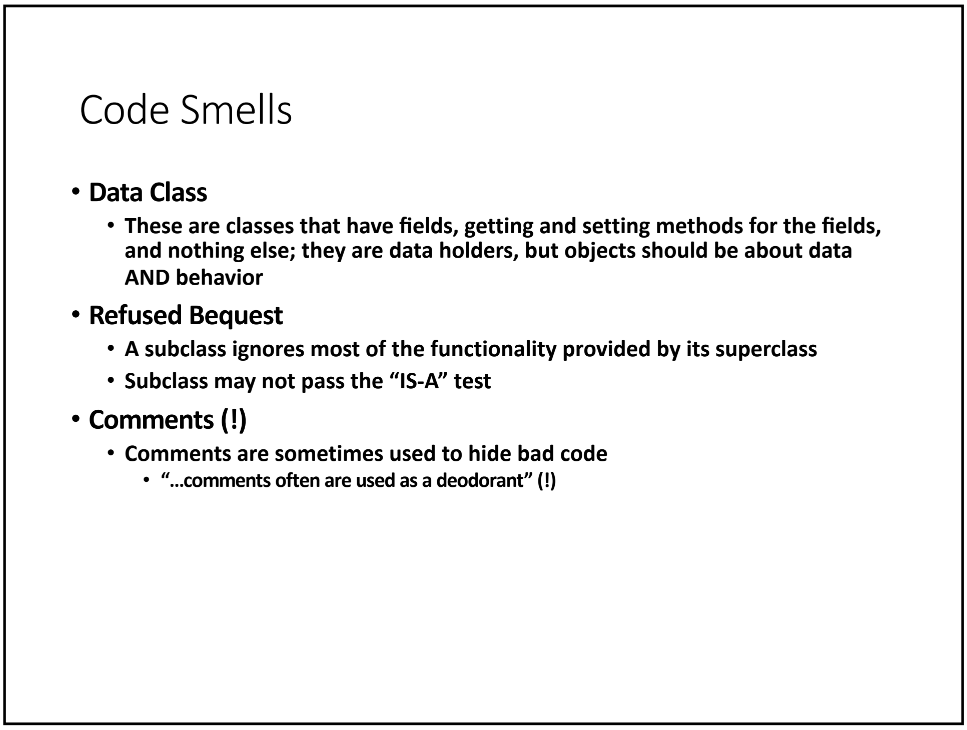
16



Code Smells

- **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**

17



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” (!)**

18

SMELLS EXAMPLE – See which smells you find in the sample code

Duplicated Code
Long Method
Large Class
Long Parameter List

Divergent Change- Related to cohesion: symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset

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

Comments (!) - Comments are sometimes used to hide bad code

Lazy Class - A class that no longer "pays its way"

Speculative Generality - "Oh I think we need the ability to do this kind of thing someday"

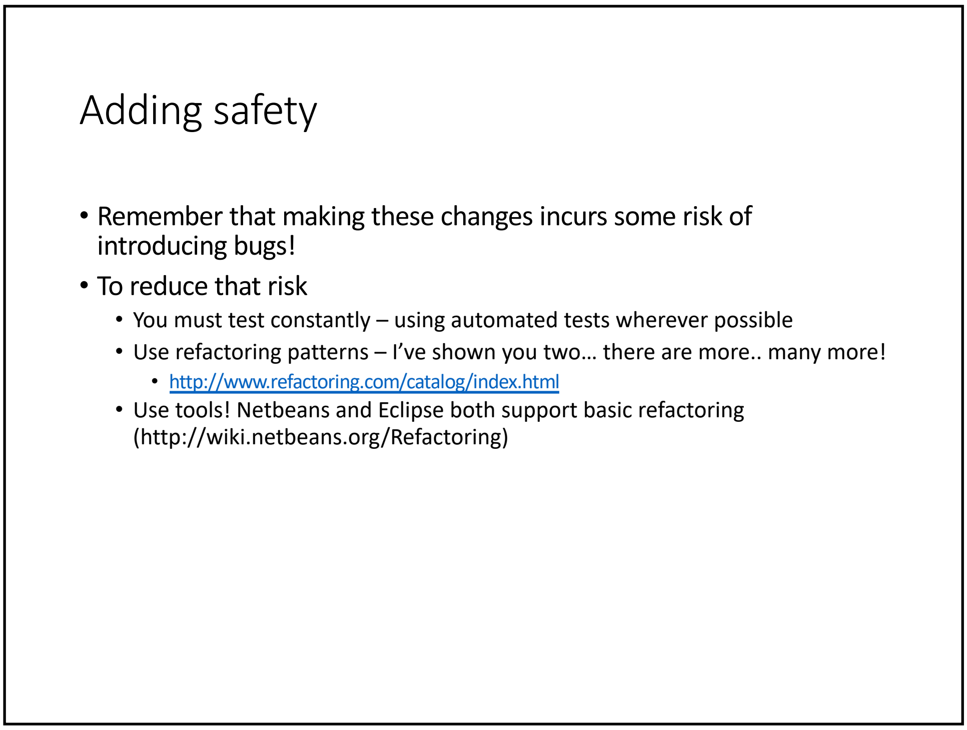
Temporary Field - An attribute of an object is only set in certain circumstances; but an object should need all of its attributes

19

Why use them?

- Code smells and refactoring are techniques to help you discover problems in design and implementation and apply known solutions to these problems
- Should they be used all the time? You should always think about them, but only apply them when they make sense… sometimes you need a long method… but think about it to make sure!

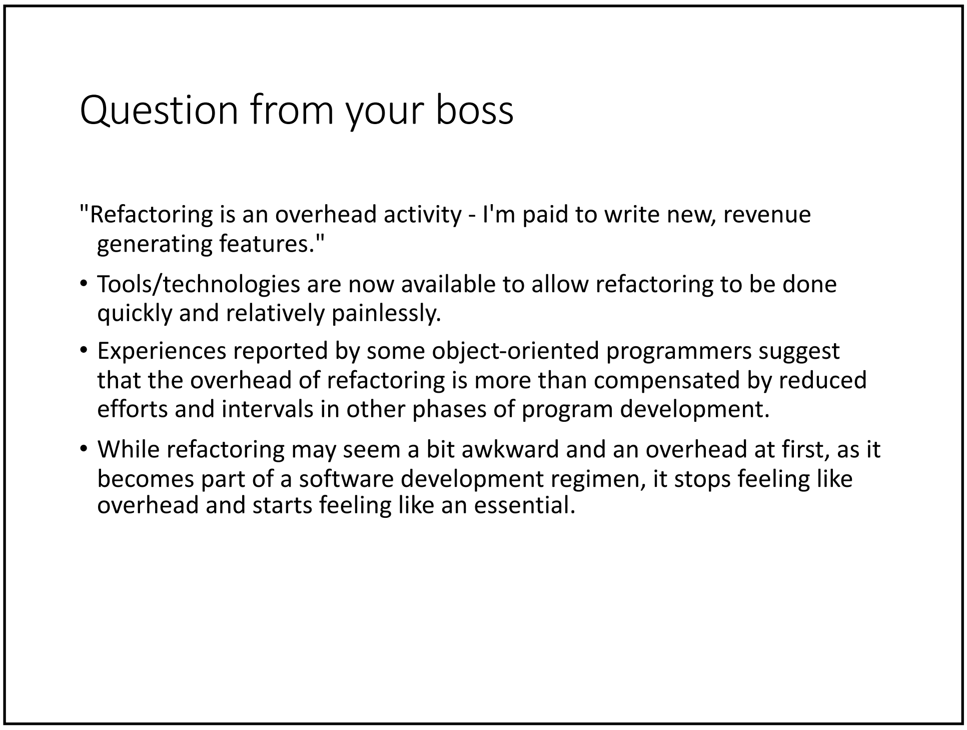
20



Adding safety

- Remember that making these changes incurs some risk of introducing bugs!
- To reduce that risk
  - You must test constantly – using automated tests wherever possible
  - Use refactoring patterns – I've shown you two… there are more.. many more!
    - http://www.refactoring.com/catalog/index.html
  - Use tools! Netbeans and Eclipse both support basic refactoring (http://wiki.netbeans.org/Refactoring)

21

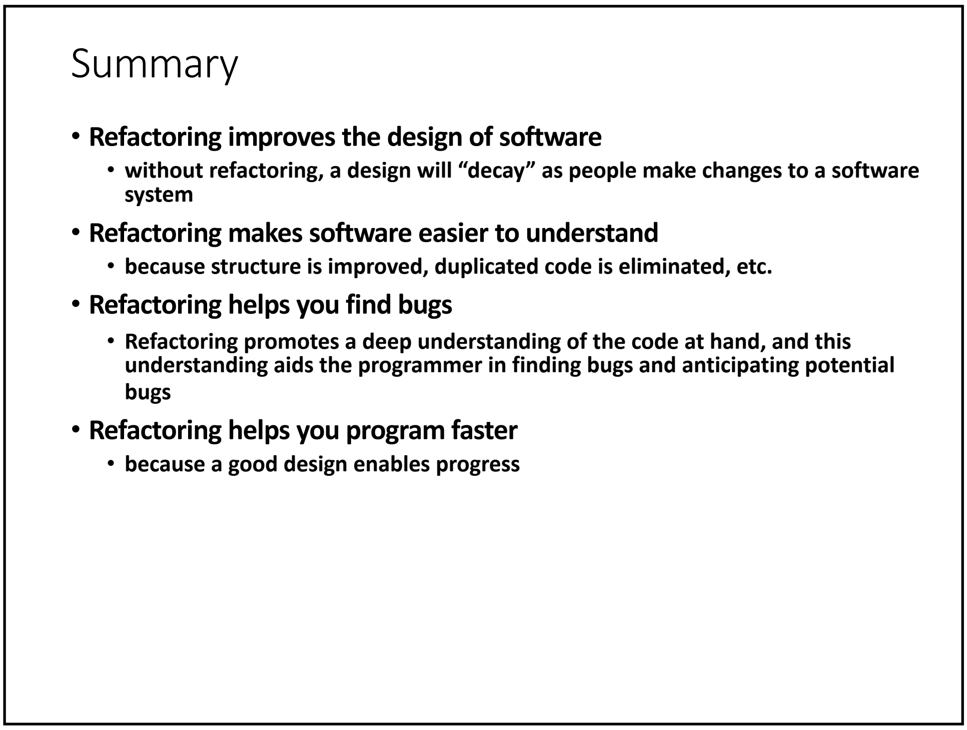


Question from your boss

"Refactoring is an overhead activity - I'm paid to write new, revenue generating features."

- Tools/technologies are now available to allow refactoring to be done quickly and relatively painlessly.
- Experiences reported by some object-oriented programmers suggest that the overhead of refactoring is more than compensated by reduced efforts and intervals in other phases of program development.
- While refactoring may seem a bit awkward and an overhead at first, as it becomes part of a software development regimen, it stops feeling like overhead and starts feeling like an essential.

22



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**

23