

Refactoring

CITS4401 Software Requirements and Design

Week 11

Dr Tingting Bi

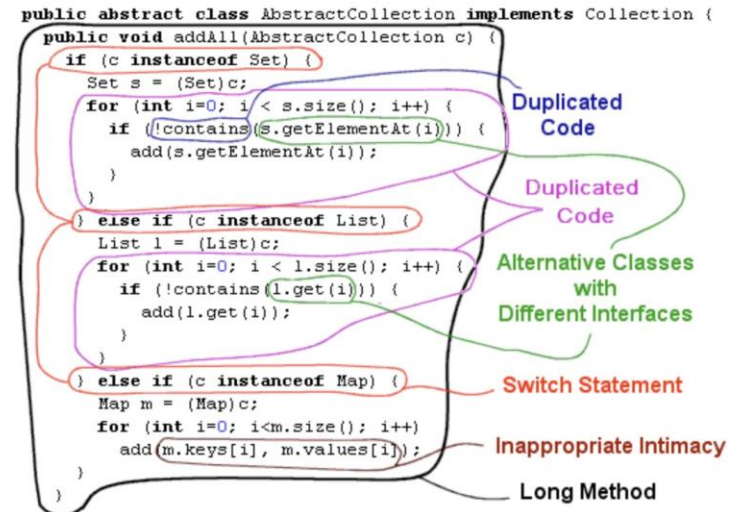
Lecture Outline

What is refactoring?

Refactoring during development

Why refactor?

- It improves design
- It makes software easier to understand
- It helps you find bugs
- It helps you program faster



```
public abstract class AbstractCollection implements Collection {  
    public void addAll(AbstractCollection c) {  
        if (c instanceof Set) {  
            Set s = (Set)c;  
            for (int i=0; i < s.size(); i++) {  
                if (!contains(s.getElementAt(i))) {  
                    add(s.getElementAt(i));  
                }  
            }  
        } else if (c instanceof List) {  
            List l = (List)c;  
            for (int i=0; i < l.size(); i++) {  
                if (!contains(l.get(i))) {  
                    add(l.get(i));  
                }  
            }  
        } else if (c instanceof Map) {  
            Map m = (Map)c;  
            for (int i=0; i<m.size(); i++)  
                add(m.keys[i], m.values[i]);  
        }  
    }  
}
```

Annotations in the diagram:

- Duplicated Code**: Points to the `if (!contains(s.getElementAt(i)))` condition in the Set branch.
- Duplicated Code**: Points to the `if (!contains(l.get(i)))` condition in the List branch.
- Alternative Classes with Different Interfaces**: Points to the `Set` and `List` interfaces.
- Switch Statement**: Points to the `if-else if-else if` structure.
- Inappropriate Intimacy**: Points to the `m.keys[i]` and `m.values[i]` access in the Map branch.
- Long Method**: Points to the entire `addAll` method.

What Is Refactoring?

Refactoring is the process of modifying software so that:

- The **external behaviour of the code is not modified**
- The internal structure is improved to make it **easier to understand** and **cheaper to modify** without changing its observable behavior
[Fow1999]

Refactoring does not fix bugs, but it may help find bugs by scrutinizing code. It may also reduce the further introduction of bugs by cleaning-up code

- Refactoring does not add new functionality to the system, but it will ease the further adding of new functionality.

Refactoring is

“Improving the design after it has been written ”

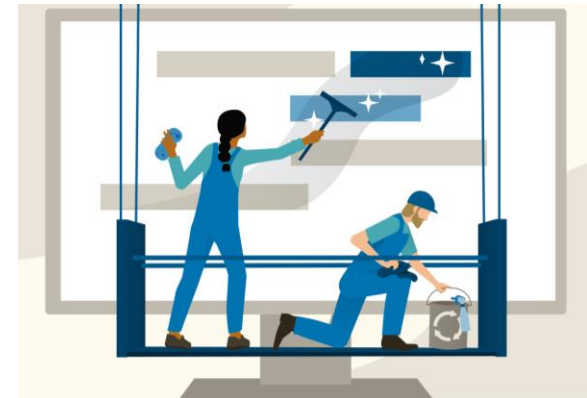
[Fow1999]

Why would we want to “improve the design after it has been written”?

Don't we design, *then* code?

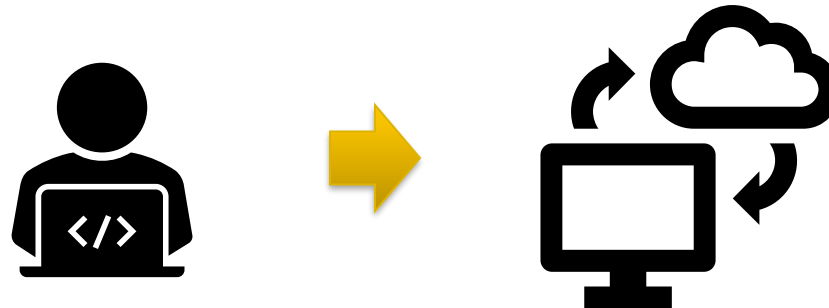
Yes... but real software systems are subject to requirements that change over their lifetimes

- The code is modified. The integrity of the structure according to the original design fades away (remember “design entropy”)
- We risk sinking from software engineering to hacking



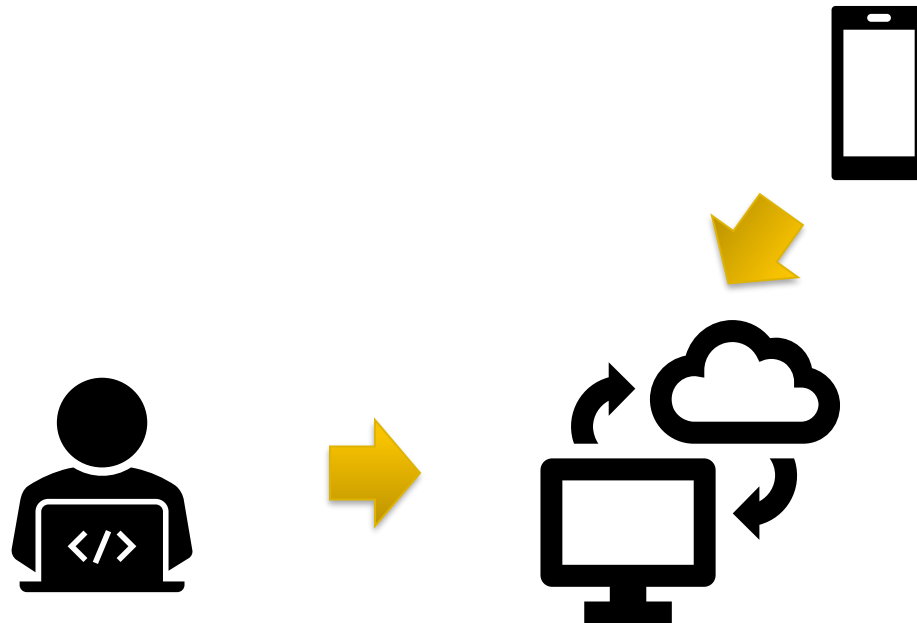
Example :Why “improve the design after it has been written”?

In January, your team deliver a new client-server-based unit enrollment system, written in NodeJS to the university. The database is stored on the cloud in JSON format



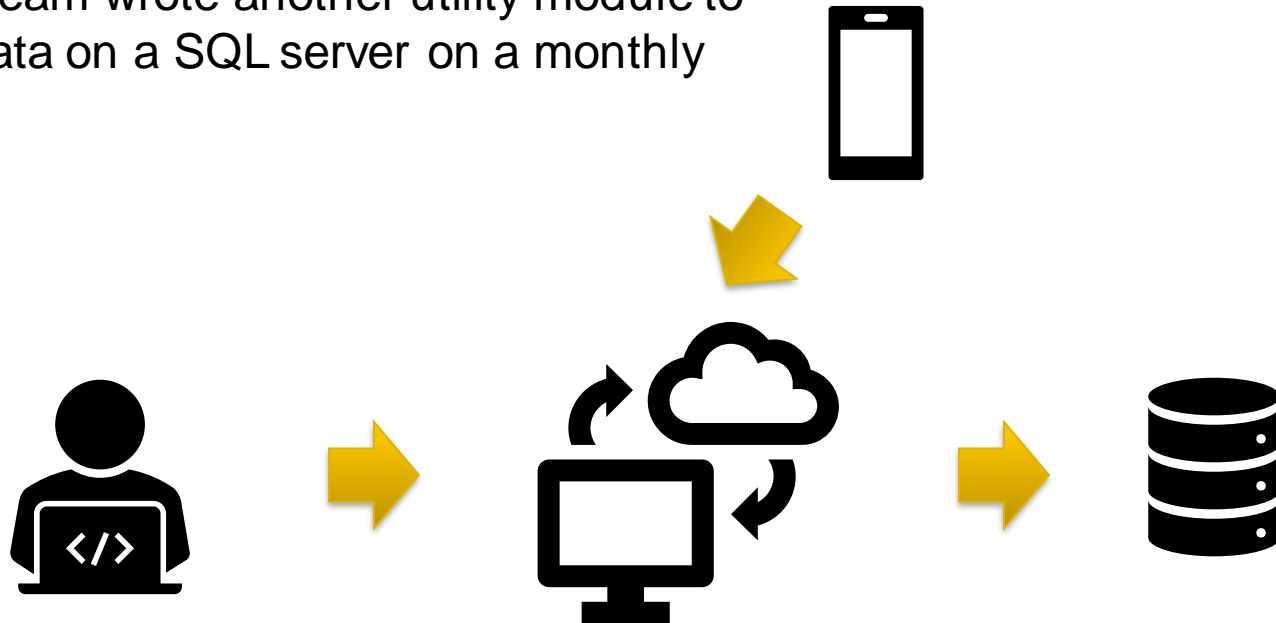
Example (cont.)

In April, the university requested for a native mobile app for the enrollment system. Thus, your team rewrite some of the front-end code and imported Android plugin and libraries for NodeJS



Example (cont.)

In September, the finance department requested that a persistent copy of the students' enrollment data should be stored internally for safekeeping. Hence, your team wrote another utility module to backup the data on a SQL server on a monthly basis



Example (cont.)

What does this entail?

As new requirements are introduced to the system, the system tends to “drift away” from its original design

With more new codes added to the system, problems (such as “bad smells” or “anti-patterns”) might be introduced unintentionally that can negatively impact the quality of the software

- Sluggish performance
- Hard to remove code (because they are entangled)
- Hard to add new code (because too many hard-coding)

Refactoring

In refactoring we take a bad design, perhaps even code with no known design, and rework it into well-designed code

Each individual step in refactoring is simple, e.g. common refactorings are

- *Extract Method, Move Method, Replace Temp with Query, Replace Conditional with Polymorphism*, etc. [will discuss some of them later]

The cumulative effect of all these small changes, however, can be a greatly improved design

- This is the complete opposite of the normal design entropy effect, where design quality degrades over time

Refactoring during Development



We have discussed refactoring so far as if it was always done to a finished piece of software

- This does not have to be the case

Refactoring can be done during initial development

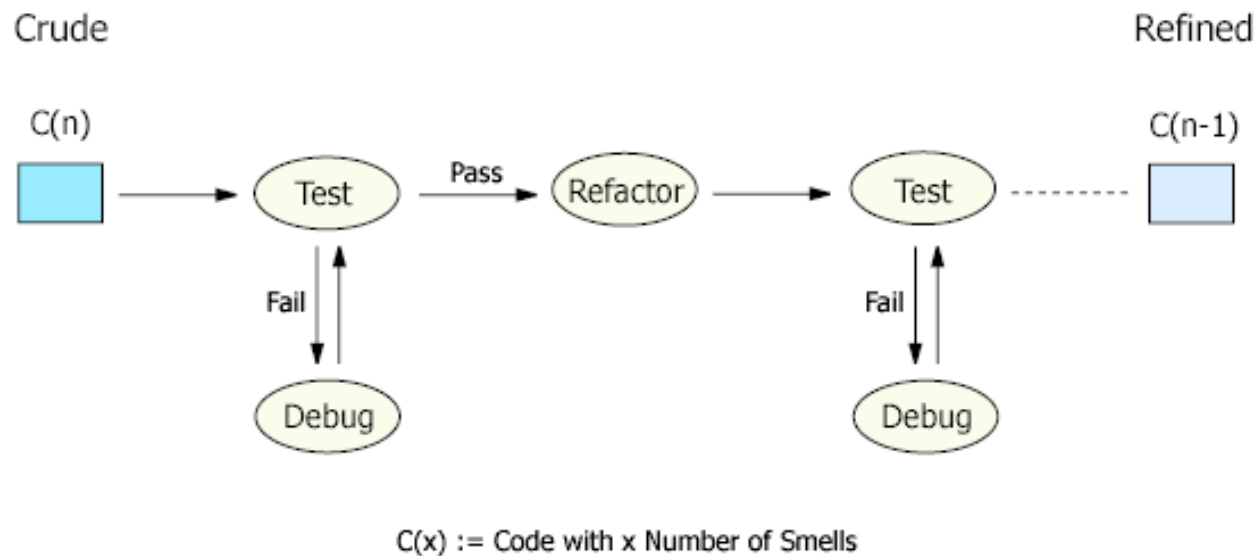
- Agile methods, such as eXtreme Programming (XP), encourage this

How to do Refactoring?

Each refactoring is implemented as a small behavior-preserving transformation.

Behavior-preservation is achieved through pre- and post-transformation testing.

Refactoring process: test-refactor-test



The Two Hats

When using refactoring during development, you divide your time between:

- Adding functionality
- Refactoring

When adding functionality, you should not be changing existing code

- You write tests for the new functionality
- You add the new functionality
- You measure progress by getting the new tests to work

The “adding functionality” hat and the “refactoring” hat frequently

You start off trying to add new functionality

- You realize that this would be a lot easier if the code had a different structure

You swap hats

- You refactor until the code has the structure you want

You swap hats again, and add the new functionality

Now you think the new functionality is hard to understand, so you swap hats and...

It is important always to be aware of which hat you have on

Why refactor?

It Improves Design

Without refactoring, the **design of the software will decay**

Changes are often made to meet short-term goals

- Often, people who maintain the software are not the original software developers who write the code!
- The maintainers might not be aware of the intended overall design

It becomes harder to understand the design from the code itself



It Improves Design

Every time this happens, the effects add up:

- If it is hard to see the design in the code, it will be hard to preserve it
- This will cause the design to decay more rapidly
- > Eventually **the original design may fade away** completely

Refactoring helps to keep the code, and the design, in shape



It Improves Design

Badly designed code usually **uses more code to do the same thing** as well-designed code

- Often because it literally does the same thing in multiple places!
- Shotgun surgery on the code just to meet deadlines!

An important goal of refactoring is thus to remove duplicate code

- This won't make the system any faster, but it will make it easier to maintain
- The more code there is, the more there is to understand... or misunderstand

It Helps You Find Bugs

The use of refactoring as an aid to understanding can also help in finding bugs

- Not everyone is good at simply reading large chunks of other people's code and spotting bugs

The process of refactoring forces you to focus on deeply understanding the design

- As a result of this process the bugs can become almost impossible to miss

It Helps You Program Faster

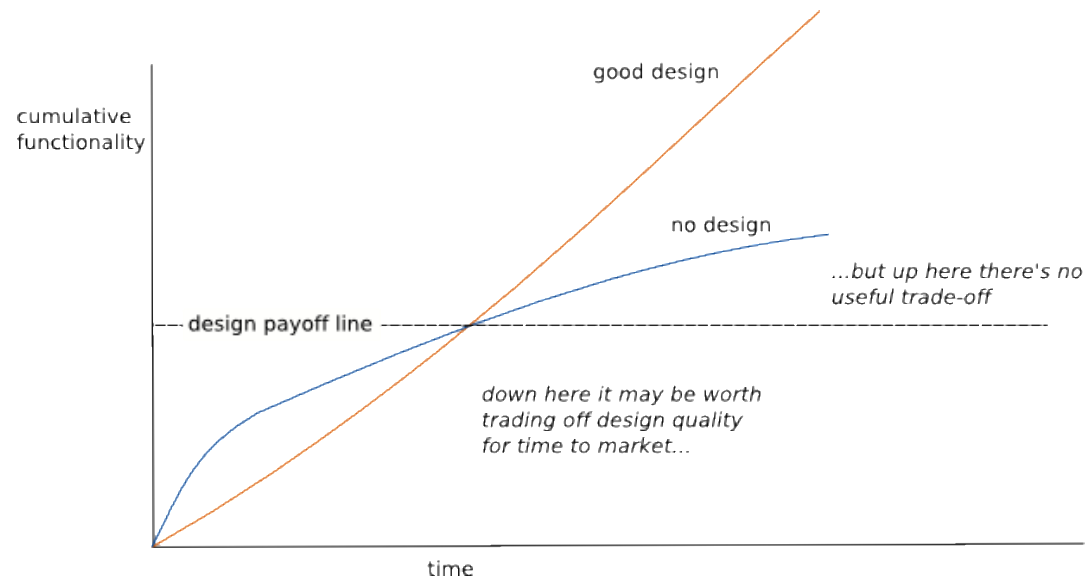
How do I justify spending time making changes to code that, by definition, have no externally observable effect?

It may be easy to argue that refactoring will help to improve quality...

- Better design, improved readability, fewer bugs

... but doesn't this slow down development?

- No!



Known Refactorings

Fowler Catalogues More Than 70 Refactorings in “Refactoring” [Fow2019]



Composing Methods

- *Extract Method, Replace Temp with Query, Introduce Explaining Variable, Replace Method with Method Object, etc.*

Moving Features Between Objects

- *Move Method, Move Field, Extract Class, Hide Delegate, Remove Middle Man, etc.*

Organizing Data

- *Replace Data Value with Object, Change Value to Reference, Duplicate Observed Data, Encapsulate Field, Replace Type Code with Subclasses, etc.*

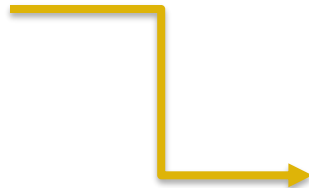
We are going to discuss a few of them but if you are interested, you can have a look at the book [Fow1999]

Refactoring: Examples

Rename Method: The name of a method does not reveal its purpose. Refactor it by changing the name of the method. Most straightforward and easiest refactoring technique (but very useful!!)

```
int getCrLit() {  
...  
}
```

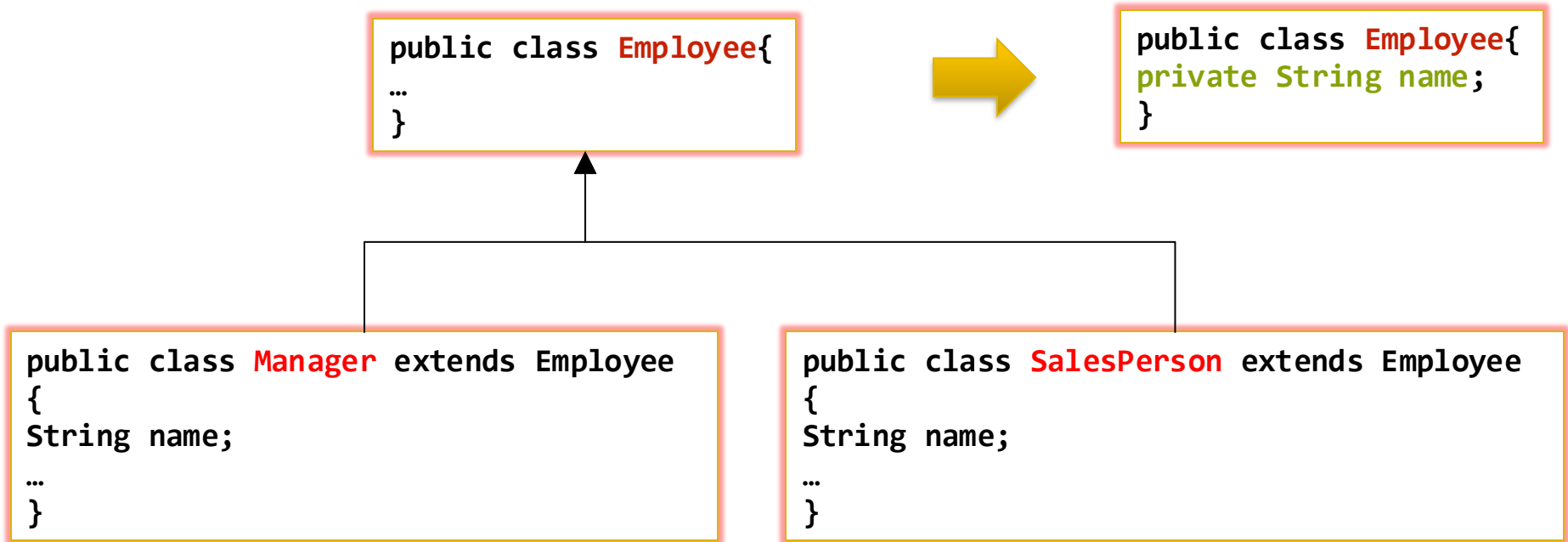
What is CrLit?! Only the original developer understand its purpose



```
int getCreditLimit() {  
...  
}
```

Refactoring: Examples

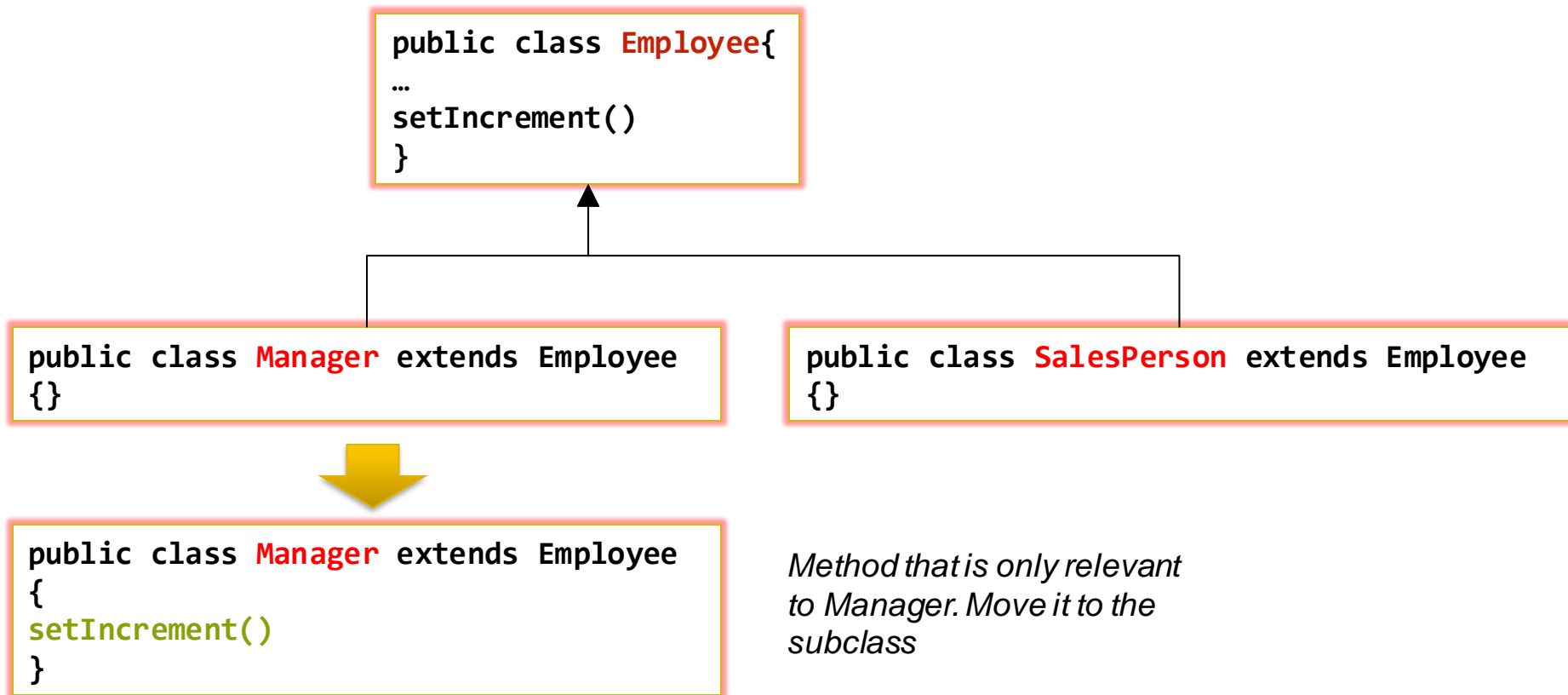
Pull Up Field: Two subclasses have the same field. Refactor it by moving the field to the superclass.



*Attribute that is relevant to
both subclasses – Move it up
to the superclass*

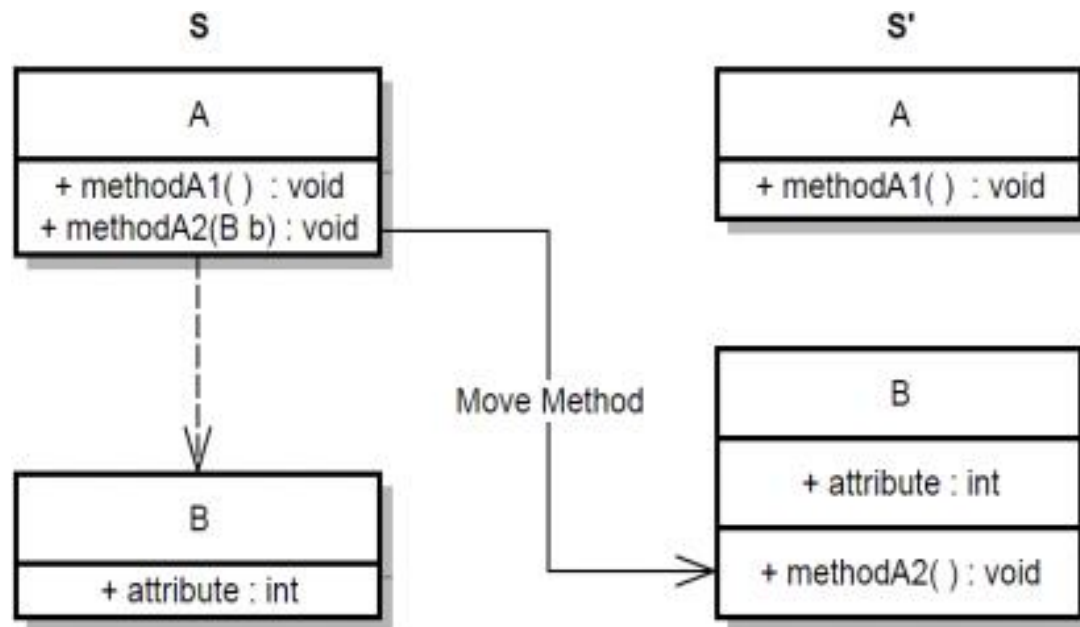
Refactoring: Examples

Push Down Method: Behavior on a superclass is relevant only for some of its subclasses. Refactor it by moving it to those subclasses.



Refactoring: Examples

Move Method: A method is, or will be, using or used by more features of another class than the class on which it is defined. Solution? 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



Example: *Move Method* Motivation

Move methods when

- A class has too much behaviour
- Classes are collaborating too much and are too highly coupled

By moving methods, we can

- Make classes simpler
- End up with classes that are more crisp implementations of a set of responsibilities

Refactoring : Examples

Inlining: reduce the number of unnecessary methods/variables and simplify the code

```
boolean hasDiscount(Order order) {  
    double basePrice = order.basePrice();  
    return basePrice > 1000;  
}
```

basePrice is a temporary variable that is assigned as the result of a simple expression. Not being used at other parts of the code

hasDiscount is a boolean function that returns true if the basePrice of the order is more than 1000

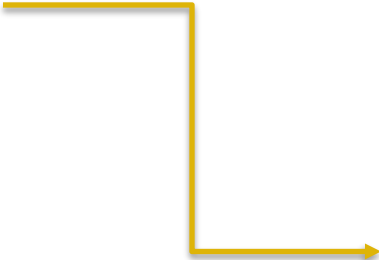
```
boolean hasDiscount(Order order) {  
    return order.basePrice() > 1000;  
}
```

Refactoring : Examples

Self-encapsulate Field : use of direct access to private fields inside a class, which is not flexible and prone to security issues. Solution? Create a getter and setter for the field and use only them for accessing the field.

```
class Range {  
    private int low, high;  
    boolean includes(int arg) {  
        return arg >= low && arg <= high;  
    }  
}
```

NOTE: Refactoring DO NOT always reduce the number of lines of code. It might increase as a result of refactoring!



```
class Range {  
    private int low, high;  
    boolean includes(int arg) {  
        return arg >= getLow() && arg <= getHigh();  
    }  
    int getLow() {  
        return low;  
    }  
    int getHigh() {  
        return high;  
    }  
}
```

Refactoring Software Architecture?

Refactoring is not constrained to just code. It is applicable to software architecture as well.

Architectural refactoring improve the design of an existing software, which changes the structure but not the functionality.

The goal is to improve a set of quality attributes such as **performance**, **scalability**, **extensibility**, **testability**, and **robustness**.

For example, if you have a monolithic application. In order to improves its scalability and extensibility, you might want to **split the functionalities into multiple submodules or services** (microservices).

- You don't add new functionality to the system, but improves its overall design and quality

Controversy around Refactoring..

Researchers have found evidence that refactoring does not always result in higher quality code (measured using software metrics such as cohesion, coupling, etc.)

- Because more lines of code are added

The work by Alsha2009 and Strog2007 both found that

“refactoring improved a quality attribute in some classes, only to reversely weaken that same quality attribute in other classes of the same system”

The researchers suggested that selecting the appropriate refactoring techniques to address certain software quality attribute is important

- Which highly depends on the experience and skills of the developers/maintainers

It is well worth becoming familiar with these known refactorings

- The checks are particularly valuable
 - > They are the distilled wisdom of expert developers who have seen what can and does go wrong when attempting certain refactorings

Refactoring tools are now appearing in most popular IDEs, including Eclipse, VisualStudio, NetBeans, etc.

Explore the refactoring tools available in your development environment

- Learning to use them can make a big difference to your productivity

Finally...

This lecture is primarily about the motivations for refactoring

I encourage you to

- Read about specific refactorings, such as the *Move Method* refactoring we have just seen
- Investigate the refactoring tools available in your chosen development environment

Good refactoring tools are what make agile development approaches such as XP truly effective

References

- [Fow2019] Martin Fowler, “Refactoring : improving the design of existing code”, Martin Fowler, Second edition. 2019 (Chs. 1, 2, 7)
- [Alsha2009] Alshayeb, M. (2009). Empirical investigation of refactoring effect on software quality. Information and software technology, 51(9), 1319-1326.
- [Strog2007] Stroggylos, K., & Spinellis, D. (2007, May). Refactoring--does it improve software quality?. In Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007) (pp. 10-10). IEEE.