

*CS5351 Software Engineering*  
*2024/2025 Semester B*

# Code with Quality

---

Prof. Zhi-An Huang

Email: `huang.za@cityu-dg.edu.cn`

# Coding

---

- ◆ In an Agile method, developers sketch the **software design** write **codes incrementally**, and focus on delivering *the tasks we pay attention to*.
  - We unavoidably **make mistakes** during this “no-long-term-planning” (i.e., unsystematic) process.
  - Our code is **(un)intentionally** coupled with the customized nature of these tasks.
- ◆ The design and code will likely change as developers’ understanding of the system to build is deepened.
- ◆ The **basic strategy** is: Our code should be **clean, minimal, and maintainable**.
- ◆ **Question:** How?

# Outline

---

- ◆ Basic Low-Level Design Principles at Code Level
- ◆ Code Smells
- ◆ Code Refactoring

# Basic Code Design Principles

---

There are a few basic design principles recommended to developers when they write code. We look at the core set of these design principles.

- Abstraction
- Information Hiding
- Encapsulation
- Coupling
- Cohesion
- S - Single-responsibility principle
- O - Open-closed principle
- L - Liskov substitution principle
- I - Interface segregation principle
- D - Dependency Inversion Principle

# Example of Abstraction

## A piece of code

```
// balls[] is an integer array
int n = balls.length;
int temp = 0;
for(int i=0; i < n; i++){
    for(int j=1; j < (n-i); j++){
        if(balls[j-1] > balls [j]){
            //swap elements
            temp = balls [j-1];
            balls [j-1] = balls [j];
            balls [j] = temp;
        }
    }
}
```

Abstract the  
well-defined  
functional  
purpose

Abstraction

## Function signature

```
int[] sort(int[] balls)
//The function f will take X as
input and output Y = f(x)
```

## Function body

```
{
    // balls[] is an integer array
    int n = balls.length;
    int temp = 0;
    for(int i=0; i < n; i++){
        for(int j=1; j < (n-i); j++){
            if(balls[j-1] > balls [j]){
                //swap elements
                temp = balls [j-1];
                balls [j-1] = balls [j];
                balls [j] = temp;
            }
        }
    }
}
```

# Abstraction: Functional Purpose

---

- ◆ **Abstraction** is a fundamental concept in programming theory.
- ◆ It aims to avoid duplication of information and/or human effort.
  - Let C be a piece of code with a **well-defined purpose**.
    - e.g., C is the code listing to sort an array of integers.
  - Why not put C into the body of a function having a function signature F?
    - e.g., the function signature F is `int[] sort(int[] balls)`
  - We say that **F is an abstraction of C**. [More precisely, F is a functional abstraction of C]
- ◆ A program P can include C **once** in the body of F and include one or more statements to call F instead of including multiple copies of C.
  - Developers of the client code U of F need not comprehend the code C; rather, they can rely on the abstraction F and its well-defined functional purpose to complete their own tasks (i.e., write the code for their purpose).
  - Change/revision in C can be local to F; these client codes should be transparent.
  - U becomes **more readable** as its code logic can focus on achieving its own purpose.

# Abstraction: Generalization

---

- ◆ Apart from hiding  $C$  in the function  $F$  with a functional purpose, abstraction can also include the process of (1) finding similarities and/or common aspects and (2) forgetting unimportant differences.
- ◆ Let  $\{C_1, \dots, C_i, \dots, C_n\}$  be codes with *similar* functional purposes.
- ◆  $F$  is a (common) abstraction of  $C_1, \dots, C_i, \dots, C_n$ .
  - The differences in parameters for individual ones among  $C_1, \dots, C_i, \dots, C_n$  are forgotten and unified as the same set of parameters in  $F$ .
  - Similarities among  $C_1, \dots, C_i, \dots, C_n$  are structured in  $F$ 's function body, and differences among them are put under different parts of the control flow structure in  $C$ .
    - Or in OO programming, we create a class hierarchy
  - The functional purpose of  $F$  is the **common** functional purpose of the individual functional purposes for  $C_1, \dots, C_i, \dots, C_n$ .

# Abstraction

---

- ◆ Rather than including  $\{C_1, \dots, C_i, C_j, \dots, C_n\}$  in the client code  $U$ , calling  $F$  can make developers of  $U$  incur less effort in integration
  - E.g., understanding the functional purpose of  $F$  is easier than reading all the code details; avoiding maintaining the code list explicitly
  - $\Rightarrow$  Make  $U$  **more maintainable**.
  - Note that  $F$  may not be more maintainable
- ◆ The common abstraction  $F$  used in  $U$  helps  $U$  to ignore all differences among the details abstracted away by  $F$ .
  - i.e., **clean and minimal**
    - if we do abstraction right (i.e., the process of creating an abstraction is right) and do the right abstraction (the process deliverable is right)
- ◆ **Question:** Suppose  $F$  is a function (e.g., like our sorting example on slide 5). Is it useful to create an interface for  $F$ ?



# Example 1 on Functional Abstraction

$C_1$

```
char* IsSmall(int x){
  if (x > 0 && x < 10)
    return "small";
  else
    return "unknown";
}
```

$C_4$

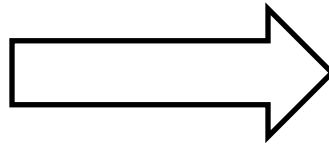
```
char* IsSmall(string x){
  if ( childlike(x))
    return "small";
  else
    return "unknown";
}
```

$C_2$

```
char* IsNotSmall(int x){
  if (x >= 0 && x <= 20)
    return "not small";
  else
    return "unknown";
}
```

$C_3$

```
char* IsOther(int x){
  if (x > 20)
    return "other";
  else
    return "unknown";
}
```



$F$

```
char * classify(int)
char * classify(string)
```

$F$

```
char* classify(int x){
  if (x > 0 && x < 10)
    return "small";
  elsif (x >=0 && x <= 20)
    return "not small";
  elsif (x > 20)
    return "other";
  else
    return "unknown";
}
```

$C$

```
....
cout << classify(8);
cout << classify("p");
....
```

$U$

# Data Abstraction

---

- ◆ Functional abstraction is to abstract the procedural steps in code.
- ◆ If we abstract and group functions **critically related to** a data structure into a set, the set represents a **single functional purpose** of the data structure.
  - Function with functional abstraction = **operators** on the **data structure**
  - functional signatures = **interfaces**
- ◆ **Data Abstraction** is to model the operators on a data structure as a set of *related* functions (called **interfaces**). In contrast, the data structure implementations are concealed by the interfaces.

# Examples of Data Abstraction

---

## ◆ Good example

<code>Stack()</code>	create a new stack
<code>s.isEmpty()</code>	is s empty?
<code>s.length()</code>	number of items in s
<code>s.push(int x)</code>	push x onto s.
<code>s.pop()</code>	remove and return the item most recently pushed onto s.

## ◆ Bad example

<code>Stack()</code>	create a new stack
<code>s.isEqual(int[] X)</code>	is s equal to the integer array X?
<code>s.length()</code>	number of items in s
<code>s.set(int x, int i)</code>	put x at the position <code>s[i]</code> .
<code>s.get(int i)</code>	remove and return the item <code>s[i]</code> , shorten the stack s.
<code>s.push(int x, int i)</code>	implemented as <code>s.set(int x, int i)</code>

The data structure in s should be compatible to an integer array

Does not obey the first-in-last-out property of stack

# Bad Examples of Data Abstraction

---

## ◆ Worse example

```
Stack()  
s.isEqual(int[] X)  
s.length()  
s.set(int[] X)  
s.get()
```

create a new stack

is s equal to the integer array X?

number of items in s

keep X as s.

return the data structure instance used by s,  
and internally reset s to empty.

Completely expose  
the implementation  
of the data structure  
of the stack

Bad example	Stack ()	create a new stack
	s.isEqual(int[] X)	is s equal to the integer array X?
	s.length ()	number of items in s
	s.set (int x, int i)	put x at the position s[i].
	s.get (int i)	remove and return the item s[i], shorten the stack s.
	s.push(int x, int i)	implemented as s.set (int x, int i)

# 1-Minute Self-Reflection and Discussion

---

- ◆ The Bad Example on slide 11 can serve wider scenarios than the Good Example on the same slide that does not expose the implementation of the data structure to the client code.
- ◆ *E.g.*, to know the value kept at the 3<sup>rd</sup> element in a stack, one call to the stack in the Bad Example can know the answer, whereas we need a sequence of calls to the stack in the Good Example to get the same answer.
  - Writing less code is productive, which is good!
  - As long as we use the operators in the Bad Example correctly, it should be OK!
- ◆ What is your opinion?

# Data Abstraction

---

Consider two pieces of client code using **Bad Example**:

```
for (int i = 0; i < n; i++) { ...;s1.push(i,i);...}  
for (int i = n; i > 0; i--) { ...;s2.push(i,i);...}
```

← The two loops **produce equivalent sequences** of integers in s1 and s2.

A developer, after reversing the loop order for the first loop, expects the processing in the 2<sup>nd</sup> loop to be reversed, but this is not the case. [A small fix to the second loop does not resolve the problem.](#)

Consider two pieces of client code using **Good Example**:

```
for (int i = 0; i < n; i++) { ...;s1.push(i);...}  
for (int i = n; i > 0; i--) { ...;s2.push(i);...}
```

← The sequence of integers in s1 is a **reverse sequence** kept in s2 (with each pair of elements differs by one), which follows the expectations of developers to reverse the looping order in the client code.

We can change s2 to keep s1's elements by using s2.push(i-1) instead of s2.push(i).

# What Make “Good Example” Work?

Consider two pieces of client code using **Bad** Example:

```
for (int i = 0; i < n; i++) { ...;s1.push(i,i);...}  
for (int i = n; i > 0; i--) { ...;s2.push(i,i);...}
```

Consider two pieces of client code using **Good** Example:

```
for (int i = 0; i < n; i++) { ...;s1.push(i);...}  
for (int i = n; i > 0; i--) { ...;s2.push(i);...}
```

- ◆ After each invocation of an operator (i.e., push()) of an abstract data type, the integrity of the data structure should be kept intact. To ensure intact, the changes kept on the data structure should only be performed by these operators.
  - This is the case for Good Example
  - This is **not** the case for Bad Example
  - This is also **not** the case for Worse Example
    - A client code can change the internal data structure for the stack.
  - For using the interface of stack in Good Example, a checking on data structure integrity can be more lightweight => more efficient and less complex code.

# Data Abstraction

---

- ◆ The Bad Example is a special kind of array manipulation.
- ◆ Rather than calling it “Stack” to mislead developers, we rename it to `myArray`, and implement `Stack` on top of `myArray`.

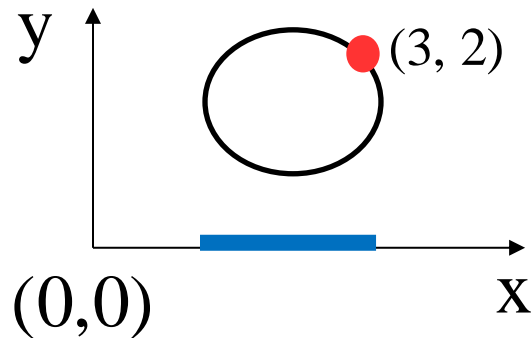
<code>Stack()</code>	create a new stack. <b>Implemented by <code>a = myArray()</code></b>
<code>s.isEmpty()</code>	is s empty? <b>Implemented by <code>a.isEqual(myArray())</code></b>
<code>s.length()</code>	number of items in s. <b>Implemented by <code>a.length()</code></b>
<code>s.push(int x)</code>	push x onto s. <b>Implemented by <code>a.set(x, a.length()+1)</code></b>
<code>s.pop()</code>	remove and return the item most recently pushed onto s. <b>Implemented by <code>a.get(x, a.length())</code></b>

<code>myArray()</code>	create a new array
<code>a.isEqual(int[] X)</code>	is a equal to the integer array X?
<code>a.length()</code>	number of items in a
<code>a.set(int x, int i)</code>	put x at the position <code>a[i]</code> .
<code>a.get(int i)</code>	remove and return the item <code>a[i]</code> , shorten the array a.



# Information Hiding and Encapsulation

- ◆ In mathematics, we have the concept of Projection.



The projection of the circle on Dimension x results in a line segment

Still able to recall what is an interface (of a data structure)?

- ◆ [Information Hiding] — is the **interface** to access ○
- ◆ [Encapsulation] Via the interface, we cover up all the information about y of ○
  - (e.g., the point ●(3, 2) of Circle is a value of 3 externally)

# Information Hiding

---

- ◆ Interfaces should be chosen to reveal **as little as possible about its internal workings**.
- ◆ We should ask whether the level of details specified in the parameters of an operator in an interface is essential.

```
// canVote1 and canVote2 provide the same functionality
// to determine whether a person holding HKID is over 18 years old
// and is a valid voter in a particular district
boolean canVote1(int HKID, int age, int district_code) {...};
boolean canVote2(Voter v) { v.votable();};
{ Voter v = new Voter(123456, 28, 12);
  canVote1(v.HKID, v.age, v.district_code);
  canVote2(v);
}
```

Which one is better?

*canVote1( )* reveals **more** details than *canVote2( )*

# Encapsulation

---

- ◆ Encapsulation on X refers to *ensuring* outsiders cannot *gain knowledge* about certain information **internal** to X.
  - X can be a class, a module, a file, a data record, etc.
  - X can also represent internal data, algorithms, etc.
- ◆ **Idea:** We achieve good encapsulation if we can describe the functional purpose using the least “working known-how” of the operators of a data structure and the functional signature does not specify more than what the functional purpose states.
  - If not, the client code U knows the details of the function. If U uses this knowledge to make customized code, the integrity of the data structure will be compromised.

# Encapsulation Violation

- ◆ Consider Stack and myArray modified from slide 16 and the client code C to use them.

```
// client code C
{
    MyArray a;
    Stack s = Stack(a);
    s.push(1);
    a.get(0);
    assert(s.isEmpty());
}
```

The code C fails its assertion. Stack s should protect its internal implementation to make it intact. But now, the variable a is manipulated outside s, which corrupts s.

Stack(a)	create a new stack. <b>Implemented by keeping a internally</b>
s.isEmpty()	is s empty? <b>Implemented by a.isEqual(myArray())</b>
s.length()	number of items in s. <b>Implemented by a.length()</b>
s.push(int x)	push x onto s. <b>Implemented by a.set(x, a.length()+1)</b>
s.pop()	remove and return the item most recently pushed onto s. <b>Implemented by a.get(x, a.length())</b>

myArray()	create a new array
a.isEqual(int[] X)	is a equal to the integer array X?
a.length()	number of items in a
a.set(int x, int i)	put x at the position a[i].
a.get(int i)	remove and return the item a[i], shorten the array a.

# Encapsulation Example

---

- ◆ We can control the visibility of each data member or data method of a class to *increase* the level of encapsulation.
  - **Question:** By controlling the visibility, does it mean that developers need not consider the information stated in the input and output parameters?
    - See the next slide for an answer.

```
public class Stack {  
    public int[] a;  
    public int    size;  
    Stack() {...}  
    public boolean isEmpty() {...}  
    public length() int() {...}  
    public void push(int x) {...}  
    public int pop() {...}  
  
}
```

```
public class Stack {  
    private int[] a;  
    private int    size;  
    Stack() {...}  
    public boolean isEmpty() {...}  
    private length() int() {...}  
    public void push(int x) {...}  
    public int pop() {...}  
  
}
```

# Encapsulation Violation Example

- ◆ We should also control the input/output parameters.

The client code C

```
{
    Stack s = new Stack();
    s.push(1);
    int n = s.getSize();
    // C knows the value of the data
    // member size of s now.
    s.setSize(n/2);
}
```

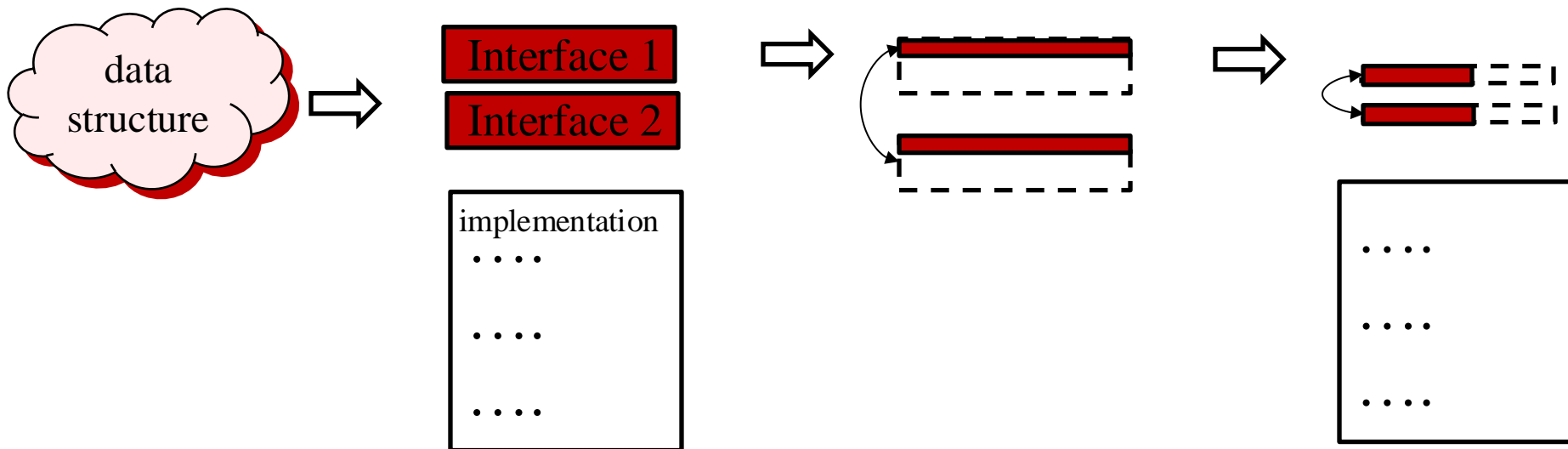
```
public class Stack {
    private int[] a;
    private int    size;
    Stack() {...}
    public boolean isEmpty() {...}
    private length() int() {...}
    public void push(int x) {...}
    public int pop() {...}
    public int getSize(){ return size; }
    public void setSize(int n) { size = n; }
}
```

# Check ...

---

## Fundamental concepts

- ◆ **Abstraction:** separate interface from implementation with a purpose
- ◆ **Information hiding:** remove details from the interfaces of a data structure (by interrelating these interfaces)
- ◆ **Encapsulation:** restrict gaining knowledge about the implementation details of the code to its client code



# Question

---

- ◆ Study the following lines of code.
  - Do they model any Functional Abstraction?
  - Do they model any Information Hiding?
  - Does the function encapsulate the details enough?

```
// computePolygonArea computes the area bounded by the consecutive  
lines connected by the points in pointList.
```

```
double computePolygonArea(int[] pointList,  
                           int    polygonType)  
{  
    if polygonType = triangle then ...//alg 1  
    elsif polygonType = square then ...// alg 2  
    ...  
}
```



# Coupling and Cohesion

---

- ◆ **Coupling** is about **syntactic** dependency.
  - **Our aim:** minimize coupling in the code listing
- ◆ **Cohesion** is about the **reasons** to place a set of entities together as a group or split them.
  - *We cannot see the dependency on cohesion from the code*
  - **Our aim:** If two entities are related, they should be grouped together and/or used together.
- ◆ **Consequence:** Higher cohesion and lower coupling *together* lead to higher quality: maintainability, reusability, and lower complexity

# Coupling

---

- ◆ An entity **X** is **structurally and directly** dependent on another entity **Y** at compile time (when the code is not yet run).
  - = there is a direct structural dependency from **X** to **Y**.
  - Programmers need such dependency info for computational efficiency
  - Compiler needs it for code optimization
- ◆ We often “hard-code” the dependency in the source code
  - Component, dataflow, control flow

```
double x1, x2, w1, w2;
...
while (x1 > 0.003) {
    ↑
    y ← x1*w1 + x2*w2;
    x1 = x1 - 0.01*y;
}
```

```
class Parent {
    //code for the parent class
}
class Child extends Parent {
    //code for the child class
}
```

# Coupling

---

- ◆ Property:

- If **X** is changed (e.g., deleted/changed its datatype), then **Y** changes too.

propagate to caller

~~double~~ int f(~~double~~ int x) { return x; }  $\Longrightarrow$  void g(~~double~~ int x)

- ◆ Consequence:

{ ~~double~~ int x = f(x); }

- If Y changes (as the result of X), elements (say Z) depend on Y changes too
- Z is indirectly dependent on X.

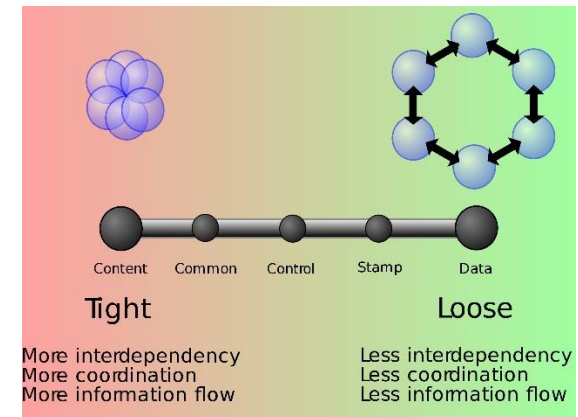
propagate and propagate

~~double~~ int f(~~double~~ int x) { return x; }  $\Rightarrow \Rightarrow$  void h(x){ g(~~2.3~~); }

- ◆ If we can either (1) reduce coupling at the compile time or (2) delay the dependency from compile time to run time (when the code is run), then we make our program have better kinds of coupling.

# Coupling

- ◆ Six basic types of coupling
  - Content (directly use a part of code)
  - Common (global variable)
  - External (interface protocol, file)
  - Control (control flow, workflow pipeline)
  - Stamp (a part of a data structure)
  - Data (the whole data structure)



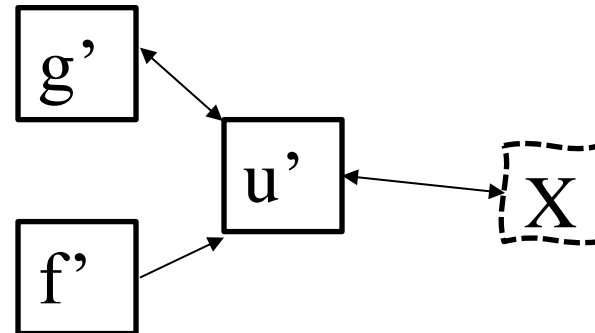
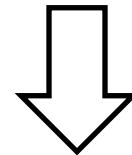
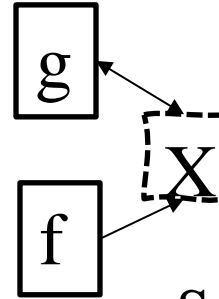
worse → better

# e.g., External Coupling

```
f(file X) {  
    read y from file X  
}  
  
g(file X, data z) {  
    read y from file X  
    y = y + z;  
    replace y by z in file X  
}
```

We can see from this example that program structure is important.

current



Suppose  $g$  and  $f$  are changed so that they pass their needs to a new entity  $u$  to access  $X$ . Then, the external coupling in the application is reduced.

# Cohesion

---

## ◆ 7 types of cohesion

- Coincidental (grouped without a reason)
- Logical (grouped because “we” treat them as one category)
- Temporal (A part of each entity in the group is processed similarly)
- Procedural (A group of functions, when composed, represent different sequences of control of abstraction (e.g., a file))
- Communicational (grouped because of operating on the same data)
- Sequential (the input of one depends on the output of another)
- Functional (to perform different aspects of a well-defined task)

worse  
↓  
better

# A bubble sorting program

```
class A {  
  greater(x, y) { return (x>y);}  
  swap(array Z, i, j) { k = z[i]; z[i] = z[j]; z[j] =k;}  
}
```

*procedural  
cohesion  
within  
class A*

*control coupling from B.sort(.) to A.greater(.)*

```
class B {  
  sort(array Z, n){  
    A a; ← data coupling from B to A
```

```
    loop i=1...n do
```

```
      loop j=1...n do
```

```
        if a.greater(Z[i], Z[j])  
          then a.swap(Z, i, j);
```

*functional  
cohesion  
within  
class B*

```
    }  
  }
```

# SOLID in Object-Orientation

---

- ◆ Abstraction, Information Hiding, and Encapsulation are fundamental.
- ◆ There are also many other more pragmatic guidelines to help developers to organize their code better. One of the most popular ones is categorized as SOLID:
  - S - Single-responsibility principle
  - O - Open-closed principle
  - L - Liskov substitution principle
  - I - Interface segregation principle
  - D - Dependency Inversion Principle



# ***SOLID***

## **S - Single-responsibility principle (SRP)**

---

- ◆ A responsibility as *a logical reason* to change the code.
  - It is an abstract concept.
- ◆ The principle states that a class should have one and only one reason to be rewritten.
  - A responsibility is not the assigned functionality of the class.
- ◆ It is a derived concept of maintaining good abstraction.

E.g., a class SortedList takes up the “logical” responsibility of “modeling a sorted sequence.” Anything related to changing how a sorted sequence is modeled, it should be done within SortedList.

If SRP is respected, then the responsibility means delivering the functional purpose of the class.

# ***SOLID***

## **Example of SRP**

---

Note that the keywords `private/public` are shown to follow the Java syntax only. They do not mean anything related to SRP.

```
public class Stack {  
    private int[] a;  
    private int    size;  
    Stack() {...}  
    public boolean isEmpty() {...}  
    private length() int() {...}  
    public void push(int x) {...}  
    public int pop() {...}  
}
```

These changes serve as the reasons to maintain the `Stack` class so that the class continues to behave like a first-in-last-out data type.

Why do we want to change the code in the class `Stack`?

- Implement a faster algorithm for the same purpose?
- Fix bugs in the existing code?
- Change to use another data structure/framework instead of `int[]`?
- Implement missing requirements required to model `Stack`?
- ...

# ***SOLID***

## **O - Open-closed principle (OCP)**

---

- ◆ Open-closed principle requires a code pattern to meet two conditions:
  - Open for extension, **and** Close for modification
- ◆ The basic rule of OCP is:
  - (1) extend without modifying the existing code or
  - (2) refine the implementation without changing the functional purpose and input/output arguments.
- Note that once a class's abstract/virtual method is set as “null” in a subclass, do not extend the method with additional code in any subclass of the former subclass.

# ***SOLID***

## **Counter-Example of OCP**

---

```
class Employee{
    enum Type {MONTHLY, DAILY }
    Type type;
    void paySalary() {
        ...
        switch (type) {
            case MONTHLY:
                doThis();
                break;
            case DAILY:
                doThat();
                break;
        }
        ...
    }
}
```

How to add more types of employees to the Contract term?

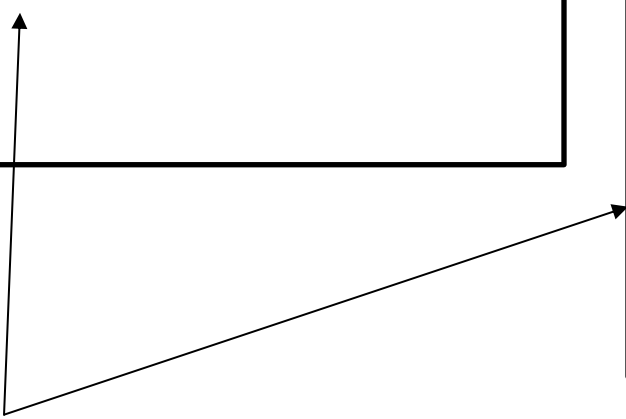
We have to modify the statement on `enum` and add an extra switch case.

# ***SOLID***

## **Example of OCP**

---

```
class Employee{
    SalaryType type;
    void receiveSalary() {
        ...
        type.doThis();
        ...
    }
}
```



```
class SalaryType
{ void doThis() {}; }

class MonthlySalaryType
extend SalaryType
{ void doThis() {...} }

class DailySalaryType
extend SalaryType
{ void doThis() {...} }
```

The code on this slide is *closed for modification* even collaborating with new (and future) subclasses of `SalaryType` to provide new behaviors.

# ***SOLID***

## **Counter-Example of OCP**

---

```
class Employee{
    SalaryType type;
    void receiveSalary() {
        ...
        if (type instanceof
            MonthlySalaryType)
            type.doThis();
        elseif (type instanceof
            DailySalaryType)
            type.doThis();
        ...
    }
}
```

```
class SalaryType
{ void doThis() {}; }

class MonthlySalaryType
extend SalaryType
{ void doThis() {...} }

class DailySalaryType
extend SalaryType
{ void doThis() {...}
}
```

The code on this slide violates OCP. To collaborate with one more subclass of SalaryType, the code in receiveSalary() should be changed.

# 5-minute Exercise on OCP

<https://stackoverflow.com/questions/37035474/am-i-violating-the-open-closed-principle/37035731>

---

**Scenario:** I stored an array of doubles in a class field `Measurements` in the class `MeasureData`. I use this data to perform some calculations (e.g., compute the arithmetic mean of the array, the maximum, and the minimum). At the moment, I don't know if in the future I'll need to do any other operation on those data (e.g., maybe I will need to get the standard deviation). I'll have many objects of the class `MeasureData`.

**Solution:** I could write a class `Calculator`, declare it `final`, use a private constructor, and use several static methods to perform the necessary calculations. This makes sense since `Calculator` acts as a utility class, without any field, much like the standard `Math` class.

**Problem:** if, in a couple of months, I'll need to do any other calculation, I'll need to write another static method in `Calculator`. Does this mean to violate the open/closed principle?

What is your suggestion to the problem?

# *SOLID*

## **L - Liskov substitution principle (LSP)**

---

- ◆ Suppose  $S$  is a subclass of  $T$ . (=  $T$  is a generalization of  $S$ )
- ◆  $S$  can be reasoned in every client code  $C$  in the same way as  $T$  is reasoned in  $C$ . (=  $T$ 's behavior is preserved by  $S$ )
  - In this case, LSP [2] is followed by the pair of  $S$  and  $T$ .
  - *E.g.*,  $S$  may refine or extend  $T$
  - $S$  refines  $T$ . *E.g.*,  $S$  and  $T$  compute the same result, but  $S$  implements a better algorithm or uses a better data structure
  - $S$  extends  $T$ . *E.g.*,  $S$  inherits everything from  $T$  and does extra things
- ◆ Duck is an LSP-kind subclass of Bird
  - We can reason a duck as a bird whenever we reason the behavior of a bird in general.
- ◆ Integer is a **not** subtype of Float. [do not have fractional math]



# ***SOLID***

## **Example of LSP**

---

```
class Employee {  
    SalaryType type;  
    void receiveSalary() {  
        ...  
        type.doThis();  
        ...  
    }  
}
```

```
class Manager extend Employee  
{ void manageEmployee()  
    {...};  
}
```

Manager is a subtype of Employee (as they are employed by a company) and a manager not only receives salary but also manage his/her subordinates.

# ***SOLID***

## **Counter-Example of LSP**

---

```
class Employee extend Manager {  
    SalaryType type;  
    void receiveSalary() {  
        ...  
        type.doThis();  
        ...  
    }  
    // overridden by a dummy  
    void manageEmployee() {null;}  
;  
}
```

```
class Manager  
{ void manageEmployee()  
    {...};  
}
```

Employee is a subtype of Manager. Since an employee in general does not supervise other employees, the function `manageEmployee` is overridden by a dummy method implementation.

# ***SOLID***

## **I - Interface segregation principle (ISP)**

---

- ◆ ISP states that a client (of an interface) should not be forced to depend on methods (i.e., implement the methods) it does not use.
  - Warning: ISP has no theoretical basis.
- ◆ It is a handy concept to avoid some problematic designs.
  - E.g., implementing too many dummy codes to conform to the interfaces for later use (which may never appear).
  - Developers are forced to implement the dummy functions in an interface due to the programming language/compiler we chose to use.

# ***SOLID***

## **Counter-Example of ISP**

---

```
interface ColorScheme {  
    void renderInRGB(byte[] image);  
    void renderInYUV(byte[] image);  
    void renderInHSV(byte[] image);  
}
```

The interface ColorScheme is not well-suitable for the code project as two methods are forced to be concretized by myDisplay even though myDisplay has no interest in them.

```
class myDisplay implements ColorScheme {  
    void renderInRGB(byte[] image) {...}; // actual need of this class  
    void renderInYUV(byte[] image) { }; // dummy, useless  
    void renderInHSV(byte[] image) { }; // dummy, useless  
}
```

# ***SOLID***

## **Example of ISP**

---

- ◆ Split `ColorScheme` into multiple interfaces

```
interface RGBColorScheme { void render(byte[] image); }  
interface YUVColorScheme { void render(byte[] image); }  
interface HSVColorScheme { void render(byte[] image); }
```

```
class myDisplay implements RGBColorScheme {  
    void render (byte[] image) {...}; // actual need of this class  
}
```

No need to add dummy implementation of the unused interface method signatures

X cannot provide a general interface for  
subclasses with incompatible calling sequence

Screen

a1  
c1  
a3  
b4

# Counter-Example of ISP

```
Client C = {  
  X x[];  
  loop:  
    code to maintain x;  
    fetch position i  
    fetch action e  
    switch (e) {  
      "f": call x[i].f(); break;  
      "g": call x[i].g(); break;  
      "h": call x[i].h(); break;  
    }  
}
```

Interface X = { f, g, h }

Subclass A

no constraint on the  
order of function calls

Subclass B

should call f before g;  
otherwise, g has no effect

Subclass C

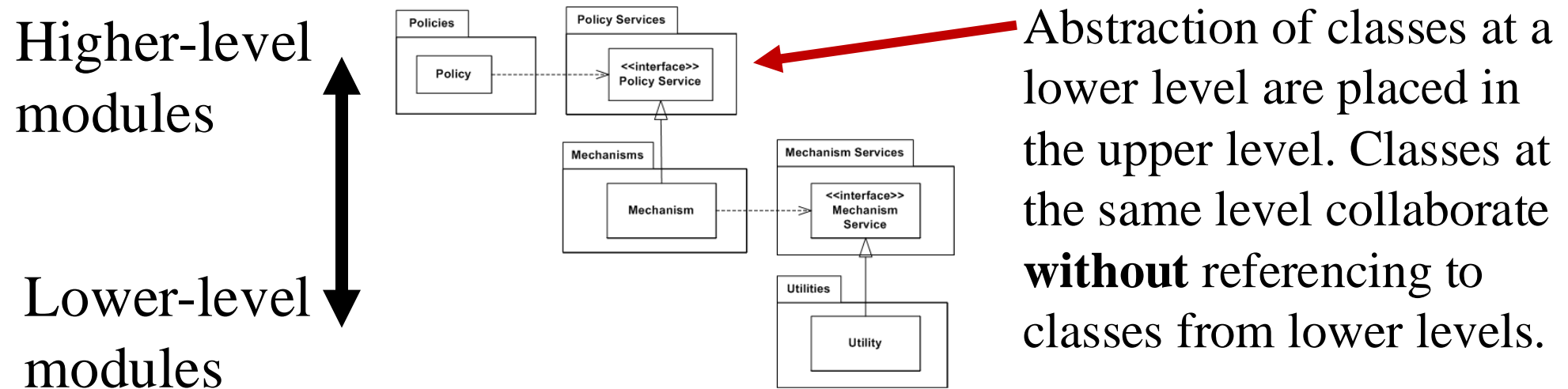
should call g before f,  
otherwise f has no effect

Error in execution

# ***SOLID***

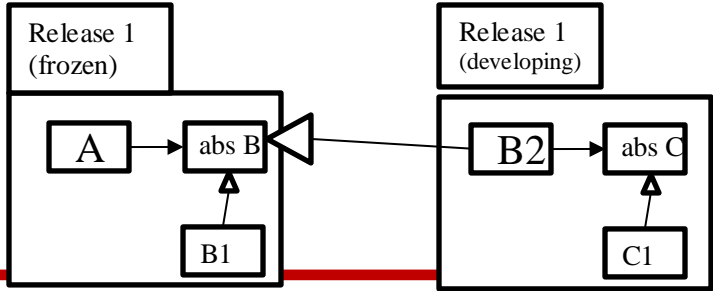
## **D - Dependency Inversion Principle (DIP)**

- ◆ DIP requires both conditions below to be held:
  1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  2. Abstractions should not depend on details. Details should depend on abstractions.

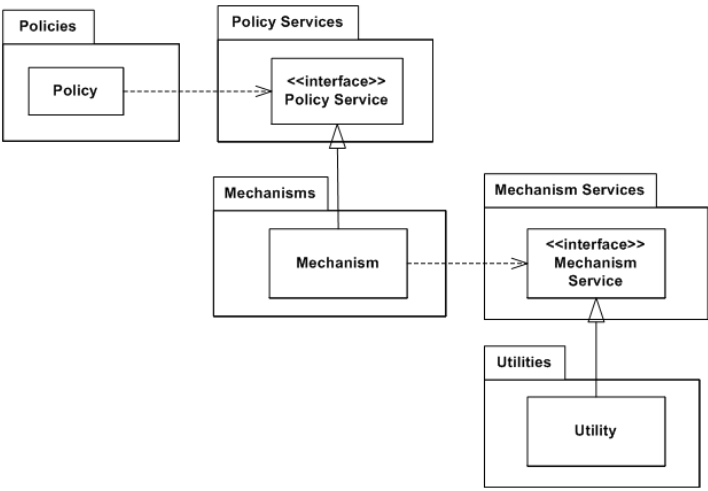


Still recall what I wrote on whiteboard in Week 1?

# Dependence Injection



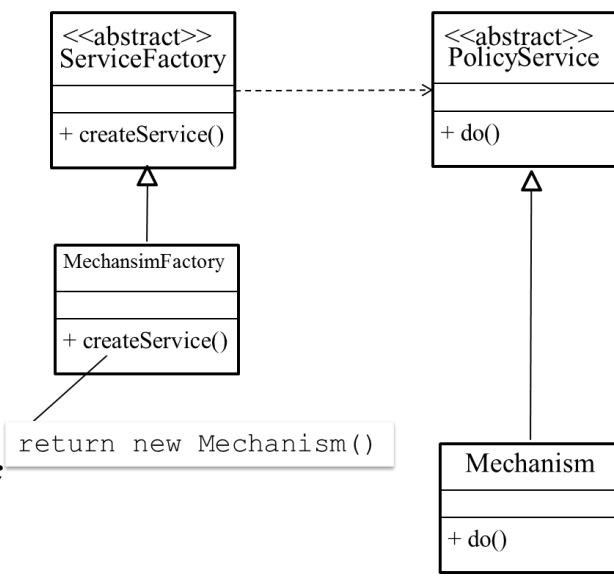
- ◆ At the top level, PolicyService is an interface, and yet a Policy object needs to call a concrete object. We may inject a dependency Mechanism object to the Policy object.



```
class Policy {
    PolicyService ps;
    Policy(PolicyService s)
    {ps = s;}
    void usePS()
    {ps.do();}
}

...
foo()
{... ServiceFactory sf = new
    MachansimFactory();
    Policy p = new
    Policy(sf.createService());
}
```

Pass an object when constructing a Policy object. We need a Mechanism factory to create the former object.





# Summary

---

## Fundamental concepts

- ◆ **Abstraction:** separate interface from implementation
- ◆ **Information hiding:** remove details from interface
- ◆ **Encapsulation:** restrict accesses to methods and info of data members; do not expose implementation details to client code

## Popular design guidelines

- ◆ **SRP:** one purpose for making change to a class
- ◆ **OCP:** extending behavior without changing the source code
- ◆ **LSP:** a subclass can do whatever a superclass does logically
- ◆ **ISP:** do not force a client code to implement unwanted methods
- ◆ **DIP:** a higher level entity does not depend on a lower level entity

# Code Smells

[also known as *anti-patterns*]

---

Code smell is a violation of a design principle and causes an adverse effect to the code.

# Code Smells

---

- ◆ *Code smell* is a violation of a design principle and (potentially) harms the code.
  - Novice developers tend to find it more difficult to judge the presence of them in their code.
    - **Code Review** is a good source for the team to manage the code
    - Some **static analysis** tools can help detect code smells.
    - Learn from the folks (large language models too?)



# Code Smell

- ◆ We have seen counter-examples of various principles on previous slides.

## Bad Examples of Data Abstraction

- ◆ Worse example

```
Stack()
s.isEqual(int[] X)
s.length()
s.set(int[] X)
s.get()
```

create a new stack  
is s equal to the integer array X?  
number of items in s  
keep X as s.  
return the data structure instance used by s,  
and internally reset s to empty.

Completely expose  
the implementation  
of the data structure  
of the stack

13

## Encapsulation Violation Example

- ◆ However, if a method violates the *principle of abstraction*, then the class does not really provide encapsulation.

The client code C

```
{
    Stack s = new Stack();
    s.push(1);
    int n = s.getSize();
    // C knows the value of the data
    // member size of s now.
}
```

```
public class Stack {
    private int[] a;
    private int size;
    Stack() {...}
    public boolean isEmpty() {...}
    private length() int() {...}
    public void push(int x) {...}
    public int pop() {...}
    public int getSize() { return size; }
    public void setSize(int n) { size = n; }
}
```

22

## SOLID Counter-Example of ISP

```
interface ColorScheme {
    void renderInRGB(byte[] image);
    void renderInYUV(byte[] image);
    void renderInHSV(byte[] image);
}
```

The interface  
ColorScheme is not well-  
suitable for the code project  
as two methods are forced to  
be concretized by  
myDisplay even though  
myDisplay has no interest  
in them.

```
class myDisplay implements ColorScheme {
    void renderInRGB(byte[] image) {...}; // actual need of this class
    void renderInYUV(byte[] image) { }; // dummy, useless
    void renderInHSV(byte[] image) { }; // dummy, useless
}
```

33

## SOLID Counter-Example of OCP

```
class Employee {
    SalaryType type;
    void receiveSalary() {
        ...
        if (type instanceof
            MonthlySalaryType)
            type.doThis();
        elseif (type instanceof
            DailySalaryType)
            type.doThis();
        ...
    }
}
```

```
class SalaryType {
    void doThis() { }; }

class MonthlySalaryType
    extend SalaryType {
    void doThis() {...} }

class DailySalaryType
    extend SalaryType {
    void doThis() {...} }
}
```

The code on this slide violates OCP. To collaborate with one more subclass of SalaryType, the code in paySalary() should be changed.

## SOLID Counter-Example of LSP

```
class Employee extend Manager {
    SalaryType type;
    void receiveSalary() {
        ...
        type.doThis();
        ...
    }
    // overridden by a dummy
    void manageEmployee() { null; }
}
```

```
class Manager {
    void manageEmployee()
    {...};
}
```

Employee is a subtype of Manager. Since an employee in general does not supervise other employees, the function manageEmployee is overridden by a dummy method implementation.

# Many Possible Code Smell Patterns

---

- ◆ Most of them are guidelines only
- ◆ 22 patterns were identified initially [7], and more were identified afterwards [8].

# Example Code Smells (don't memorize them)

	Name	Description
1	Class Data Should Be Private (CDSBP)	A class exposing its attributes
2	Complex Class (CC)	Classes having high complexity
3	Feature Envy (FE)	A method making too many calls to methods of another class to obtain data and/or functionality
2	God Class (GC)	A class having huge dimension and implementing different responsibilities
1	Inappropriate Intimacy (II)	Two classes exhibiting high coupling between them
1	Lazy Class (LC)	A very small class that does not do too much in the system
2	Long Method (LM)	A method having a huge size
1	Long Parameter List (LPL)	A method having a long list of parameters
1	Middle Man (MM)	A class delegating all its work to other classes
3	Refused Bequest (RB)	A class inheriting functionalities that it never uses
2	Spaghetti Code (SC)	A class without structure that declare long methods without parameters
3	Speculative Generality (SG)	An abstract class that is not actually needed, as it is not specialized by any other class

## References:

- [1] M. Fowler, Refactoring: improving the design of existing code. Addison-Wesley, 1999.
- [2] W. J. Brown, R. C. Malveau, W. H. Brown, et al., Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1st ed. John Wiley and Sons, March 1998.
- [3] F. Palomba et al. Do they Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In ICSME'14. pp 101-110, 2014.

1. Not perceived as design/implementation problems: CDSBP, MM, LPL, LC, II
2. Perceived as design/implementation problems: CC, GC, LM, SC
3. Gray areas: FE, RB, SG

# What is the Impact of Code Smells on Code Stability and Bug Fix?

- ◆ A study [3, 4] on ArgoUML, Eclipse, Mylyn, and Rhino, Azureus
  - Classes with code smells change more often than the otherwise.
    - They also incur more in issue-fixing or fix-bugging patches
  - Classes with more code smells change more often than the classes with less code smells
- ◆ It means that the code degraded

Table 4. Azureus: contingency table and Fisher test results for classes with at least one smell that underwent at least one change.

Releases						
	Smells-Changes	Smells-No Changes	No Smells-Changes	No Smells-No Changes	<i>p</i> -values	<i>OR</i>
3.1.0.0	220	1967	20	1433	< <b>0.01</b>	8.01
3.1.1.0	564	1686	101	1381	< <b>0.01</b>	4.57
4.0.0.0	83	2238	7	1519	< <b>0.01</b>	8.05
4.0.0.2	106	2206	12	1510	< <b>0.01</b>	6.04
4.0.0.4	435	1886	39	1484	< <b>0.01</b>	8.77
4.1.0.0	50	2297	11	1533	< <b>0.01</b>	3.03
4.1.0.2	112	2235	11	1533	< <b>0.01</b>	6.98
4.1.0.4	112	2236	12	1532	< <b>0.01</b>	6.39
4.2.0.0	37	2353	3	1580	< <b>0.01</b>	8.28

Table 5. Eclipse: contingency table and Fisher test results for classes with at least one smell that underwent at least one change.

Releases						
	Smells-Changes	Smells-No Changes	No Smells-Changes	No Smells-No Changes	<i>p</i> -values	<i>OR</i>
1.0	2042	1731	417	448	< <b>0.01</b>	1.27
2.0	3673	1373	767	236	<b>0.02</b>	0.82
2.1.1	2224	3838	193	964	< <b>0.01</b>	2.89
2.1.2	2400	3664	359	798	< <b>0.01</b>	1.46
2.1.3	2942	3125	516	642	<b>0.01</b>	1.17
3.0	3415	4880	684	1032	0.32	1.06
3.0.1	6216	2087	1294	423	0.69	0.97
3.0.2	5784	2520	1194	524	0.91	1.01
3.2	1819	9621	115	2210	< <b>0.01</b>	3.63
3.2.1	2778	8680	291	2038	< <b>0.01</b>	2.24
3.2.2	3321	8144	409	1921	< <b>0.01</b>	1.92
3.3	1778	10844	145	2364	< <b>0.01</b>	2.67
3.3.1	4337	8290	682	1830	< <b>0.01</b>	1.40

# What is the Impact of Code Smells on Program Comprehension?

- ◆ A *program comprehension* study [6] on code incurring Blob (i.e., code difficult to work with) and Spaghetti Code as code smells
- ◆ Programs incurring these two anti-patterns are more time consuming, need more effort, and lead developers to answer comprehension questions less correctly, but not reading more classes

Experiments	Systems	Characteristics	Average Times (s.)	Average Efforts	Average % of Correct Answers	# of Classes	# of SLOCs
Experiment 1: Blob	System 1: YAMM 0.9.1	With Blob	261	43.42	71%	65	11,241
		Without Blob	149	27.99	71%	62	10,923
	System 2: JVerFileSystem	With Blob	251	43.30	67%	83	8,971
		Without Blob	206	33.68	79%	85	8,750
	System 3: Aura	With Blob	189	33.31	58%	89	10,629
		Without Blob	271	42.55	66%	99	11,836
Experiment 2: Spaghetti Code	System 1: GanttProject 2.0.6	With Spaghetti	190	47.31	67%	69	6,171
		Without Spaghetti	194	53.44	51%	69	4,441
	System 2: Xerces 2.7.0	With Spaghetti	215	39.27	52%	63	4,370
		Without Spaghetti	182	34.48	52%	63	4,393
	System 3: JFreeChart 1.0.13	With Spaghetti	195	42.37	52%	76	36,949
		Without Spaghetti	218	45.84	45%	76	36,976
Experiment 3: Both APs	System 1: GanttProject 2.0.6	With both APs	187	45.36	44%	72	4,628
		Without any APs	107	27.32	83%	69	4,441
	System 2: Xerces 2.7.0	With both APs	184	42.83	73%	64	11,634
		Without any APs	140	29.33	86%	63	4,393
	System 3: JFreeChart 1.0.13	With both APs	208	48.68	43%	78	37820
		Without any APs	138	31.27	78%	76	36,976

Table IV

SUMMARY OF THE COLLECTED DATA FOR EACH EXPERIMENT (AP = ANTIPATTERN)



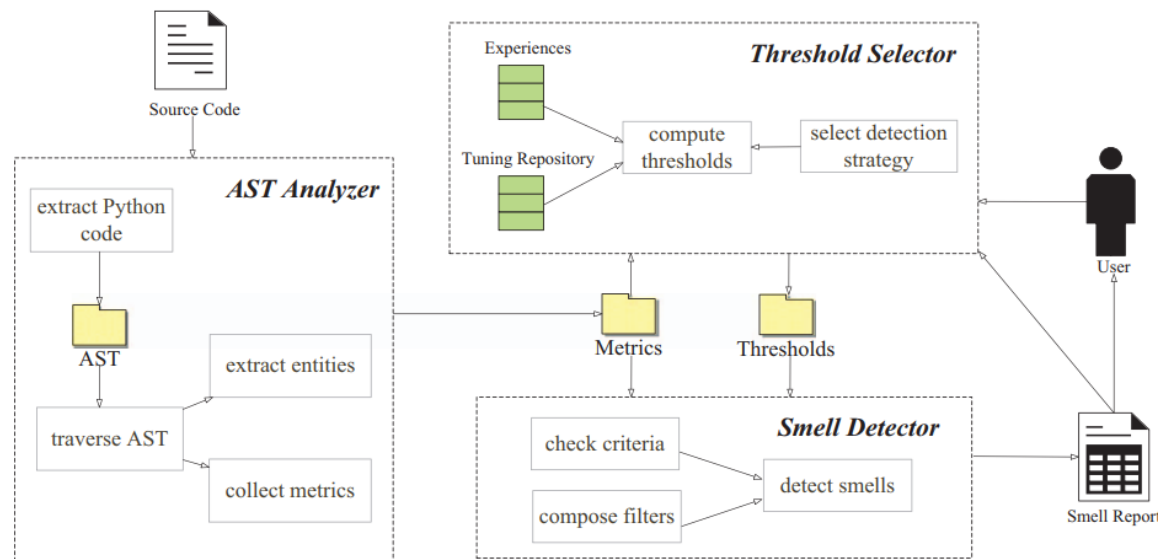
# How to Automate Code Smells Detection?

- ◆ Formulate what constitute a code small [3]

```
1 RULE_CARD : ComplexClassOnly {  
2   RULE : ComplexClassOnly { (METRIC: McCabe, VERY_HIGH, 20) };  
3 };  
4 RULE_CARD : LowCohesionOnly {  
5   RULE : LowCohesionOnly { (METRIC: LCOM5, VERY_HIGH, 20) };  
6 };
```

Listing 1. Specification of the ComplexClassOnly and LowCohesionOnly code smells.

- ◆ Then, mine the code [6] to extract entities and compute metric values



Many open-source code can generate the control flow graph or AST of a project written in any programming language.

There are also many code projects to compute various metrics

# Some Code Smell Patterns can be Formulated Successfully [don't memorize them]

Table V. Code smells and source code metrics.

Code smells	Software metrics used to recognize code smells
Large class	NAD, NDM, LOC_CLASS [97], LOC_CLASS, NOP [18], WMC [4, 46, 67, 89, 98], NOM, NOA [32], TCC [4, 89], ATFD [4, 46, 89], LOC-CLASS, NOM, NAD [99]
Data class	NACC [18], NOV, LOCProb, LOCActual [24], WOC, NOPA, NOAM(NACC) [46, 67, 97], WMC [100]
Lazy class	NOM, WMC, LOC, CBO [71, 100, 101], NOP [56], WMC, CYCLO [102], NAD [100]
Data clump	Defect density [73]
Long parameter list	PAR [24, 56, 99], NOP* [70, 81, 98]
Duplicate code	LOC, SEC, LB, SDC [103]
Divergent change	CM [104], DOCM [59], NCC [105]
Feature envy	CBO (CA, CE), LCOM [69], ATFD, LAA, FDP [46, 97], FEW, DDP [97]
Long method	LOC_METHOD [5, 56, 99, 102, 106], LOC_METHOD, CYCLO, MAXNESTING, NOAV [97]
Middle man	NFM, NOM [67]
Message chain	LMC [56], NOMCAIS [72]
Primitive obsession	NOV, VAVG [72]
Incomplete library class	—
Parallel inheritance hierarchy	DIT, NOC [65, 72], DIT [107]
Request bequest	BUR, NOP [56], NPROTM, BUR, BOVR, AMW, WMC, NOM [97]
Switch statement case	NOC* [56, 99], CYCLO [107]
Shotgun surgery	CM, CC [4, 97]
Speculative generality	—
Temporary field	IVMC [71], LOC, NOM, WM, CBO [101]
Inappropriate intimacy	—
Comments	Ratio Of comments [18, 53]
Alternative classes with different interfaces	—

**Source Code Metrics:** AMW, average method weight; ATFD, access to foreign data; BOVR, base-class overriding ratio; BUR, base-object usage ratio; CBO, coupling between objects; CC, changing classes; CM, changing methods; Ca, afferent coupling; Ce, efferent coupling; CYCLO, McCabe's cyclomatic complexity; DOCM, dependency-oriented complexity metric; DIT, depth inheritance hierarchy; FDP, foreign data providers; LAA, locality of attribute accesses; LB, line bias; LMC, length of message chain; LCOM, lack of cohesion between methods; LOCprob, number of line of code for data fields, methods, imported packages, and package declaration; LOCactual, total line of code; LOC, lines of code; IVMC, number of methods that reference each instance variable defined in a class; LOC\_CLASS, lines of code in a class; LOC\_Method, lines of code in a method; MAXNESTING, counts the maximum nesting level of control structures within an operation; NAD, number of attributes in a class; NBD, nested block deptmethods; NCC, number of concerns per component; NOM, number of methods; NACC, number of accessors (getter/setter); NOV, number of variables per class; NOMcalls, number of method calls; NOC, number of children; NOC\*, number of cases; NProtM, number of protected members; NOP\*, number of parameters; NOPA, number of public attributes; NOAM, number of accessor methods; NOP, number of properties; PAR, number of parameters; NOAV, counts the number Of accessed variables; SEC, size of exact chunk; SDC, size of duplication chain; WMC, weighted methods per class; Vavg, average count on the number of variables; WOC, weight of a class

# Refactoring

---

# Refactoring [10]

- ◆ To improve code structure and organization
  - Triggering points: code review and in code maintenance

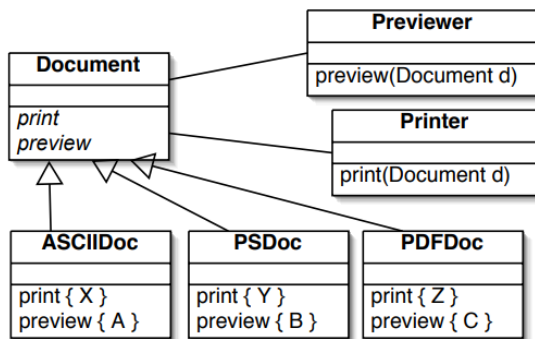


Fig. 1. *Document* class hierarchy and helper classes

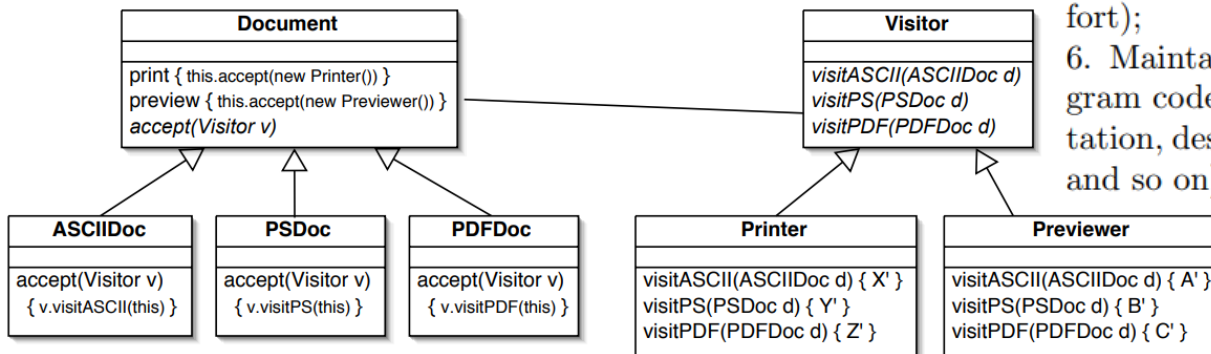


Fig. 2. Refactored design model for the *Document* class hierarchy

The refactoring process consists of a number of distinct activities:

1. Identify where the software should be refactored;
2. Determine which refactoring(s) should be applied to the identified places;
3. Guarantee that the applied refactoring preserves behaviour;
4. Apply the refactoring;
5. Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort);
6. Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests and so on)

# A Refactoring Process [10]

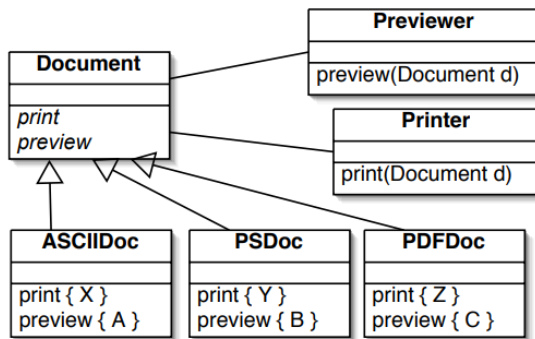


Fig. 1. *Document* class hierarchy and helper classes

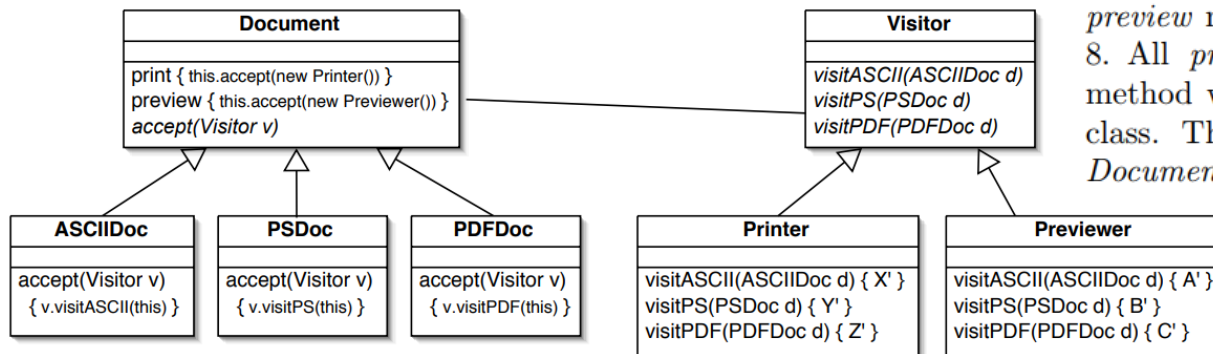
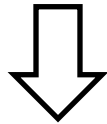


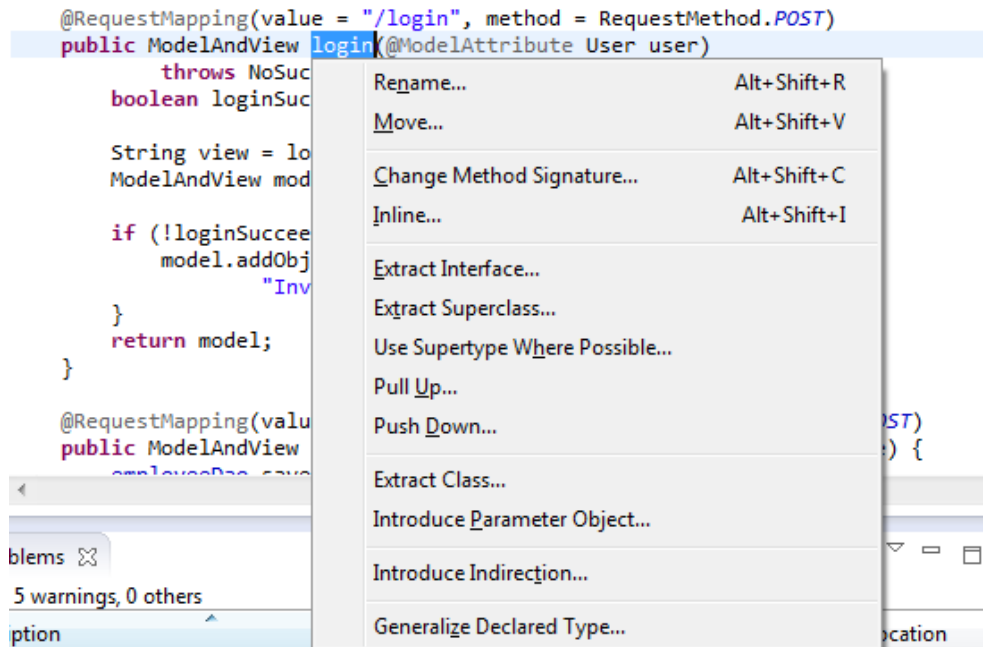
Fig. 2. Refactored design model for the *Document* class hierarchy

Although the above example is relatively simple, it already requires over twenty primitive refactorings to introduce the Visitor design pattern:

1. The *print* method in each *Document* subclass (3 occurrences) is moved to class *Printer* using a **MoveMethod** refactoring
2. To avoid name conflicts, each of the 3 moved *print* methods needs to be renamed first to a *visit\** method using a **RenameMethod** refactoring
3. The *preview* method in each *Document* subclass (3 occurrences) is moved to class *Previewer* using a **MoveMethod** refactoring
4. To avoid name conflicts, each of the 3 moved *preview* methods needs to be renamed first to a *visit\** method using a **RenameMethod** refactoring
5. An abstract *Visitor* class is introduced as superclass for *Printer* and *Previewer* using an **AddClass** refactoring
6. Three abstract *visit\** methods are introduced in the new *Visitor* class using an **AddMethod** refactoring
7. An *accept* method is introduced in all three subclasses of *Document* by extracting it from the *print* method and *preview* methods, using an **ExtractMethod** refactoring
8. All *preview* and *print* methods now call the *accept* method with an instance of the appropriate *Visitor* subclass. Therefore, their definition can be pulled up to the *Document* class, by using a **PullUpMethod** refactoring.

# Removal of Code Smells

- ◆ Every good IDE (e.g., Eclipse, Visual Studio) has a menu of *popular* refactoring to assist developers to manage their code.
  - Use them or do it manually



Press “Alt + Shift + T” in *Eclipse*

1. Highlight the code to be extracted:

- C#:

```
class Program
{
    // References
    static void Main(string[] args)
    {
        double radius = 1.23;
        double area = Math.PI * radius * radius;
    }
}
```

2. Next, do one of the following:

- Keyboard
  - Press Ctrl+R, then Ctrl+M. (Note that your keyboard shortcut may be different)
  - Press Ctrl+. to trigger the Quick Actions and Refactorings menu and select Extract Method.
- Mouse
  - Select Edit > Refactor > Extract Method.
  - Right-click the code and select Refactor > Extract > Extract Method.
  - Right-click the code, select the Quick Actions and Refactorings menu and select Extract Method.

Press “Ctrl+” in *Visual Studio*

# Why, What, and How to Refactor?

- ◆ A study [11] on Windows development team in Microsoft.
- ◆ Results: perform manually, improve code readability and maintainability, worry about regression bugs and build breaks

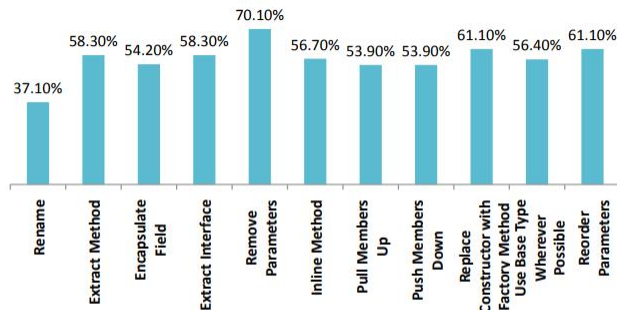


Figure 1: The percentage of survey participants who know individual refactoring types but do those refactorings manually

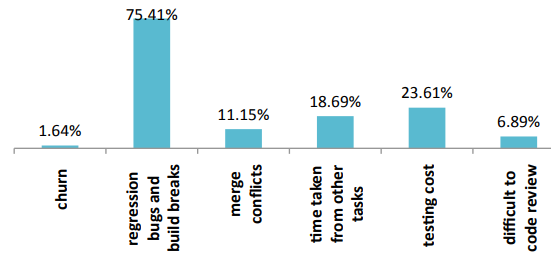


Figure 2: The risk factors associated with refactoring

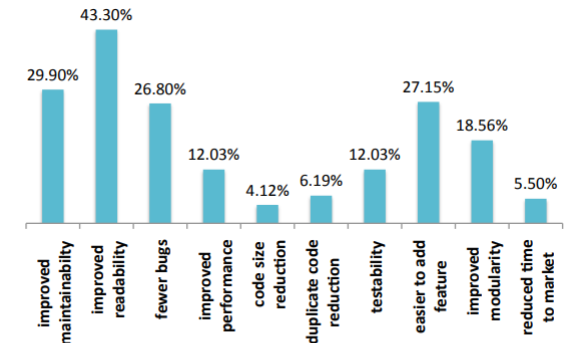
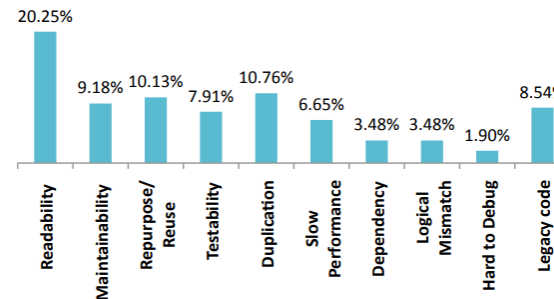


Figure 3: Various types of refactoring benefits that developers experienced

- ◆ The symptoms of code that help developers initiate refactoring :





# Refactoring correctness and implication of refactoring [13]

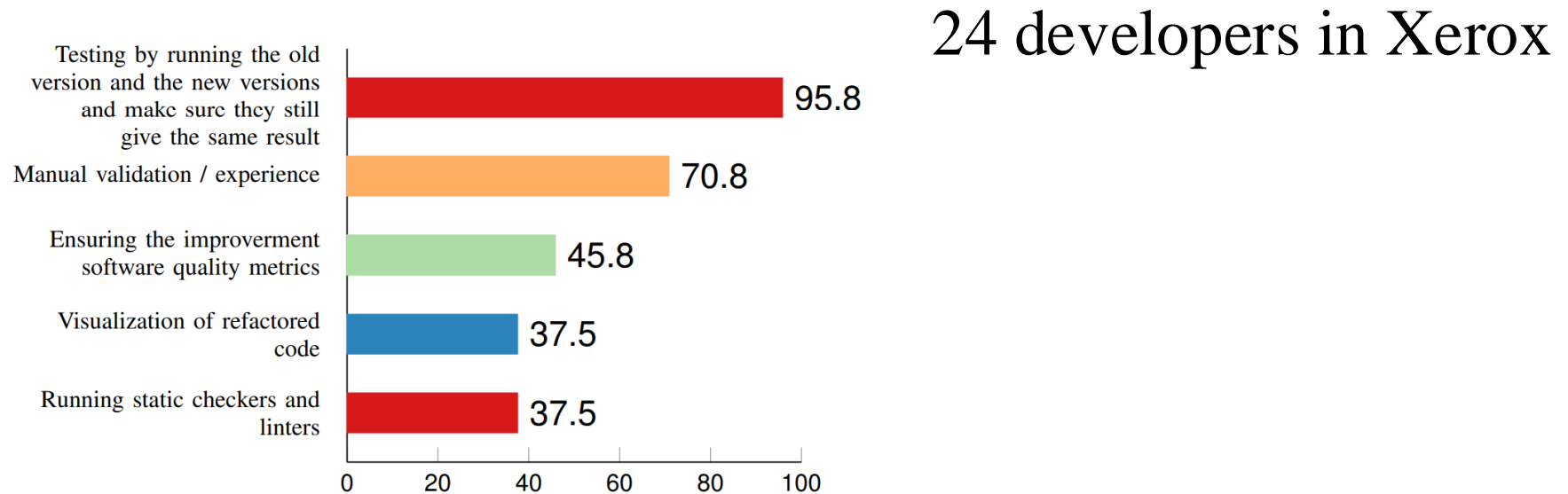


Figure (5) Mechanisms used to ensure the correctness after the application of refactoring.

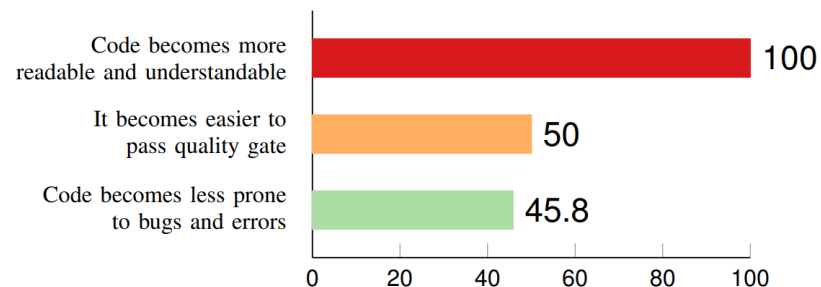


Figure (6) Implications experienced as software evolves through refactoring.



# Are Refactoring Always Useful (against Code Smells)?

by *Lacerda et al. (2020)*[12]

Refactoring is the first approach to minimize technical debt effects. However, some refactoring, when applied, negatively affect the quality of the software.

We learned technical debt (TD) in a past week. Code smell is a kind of TD

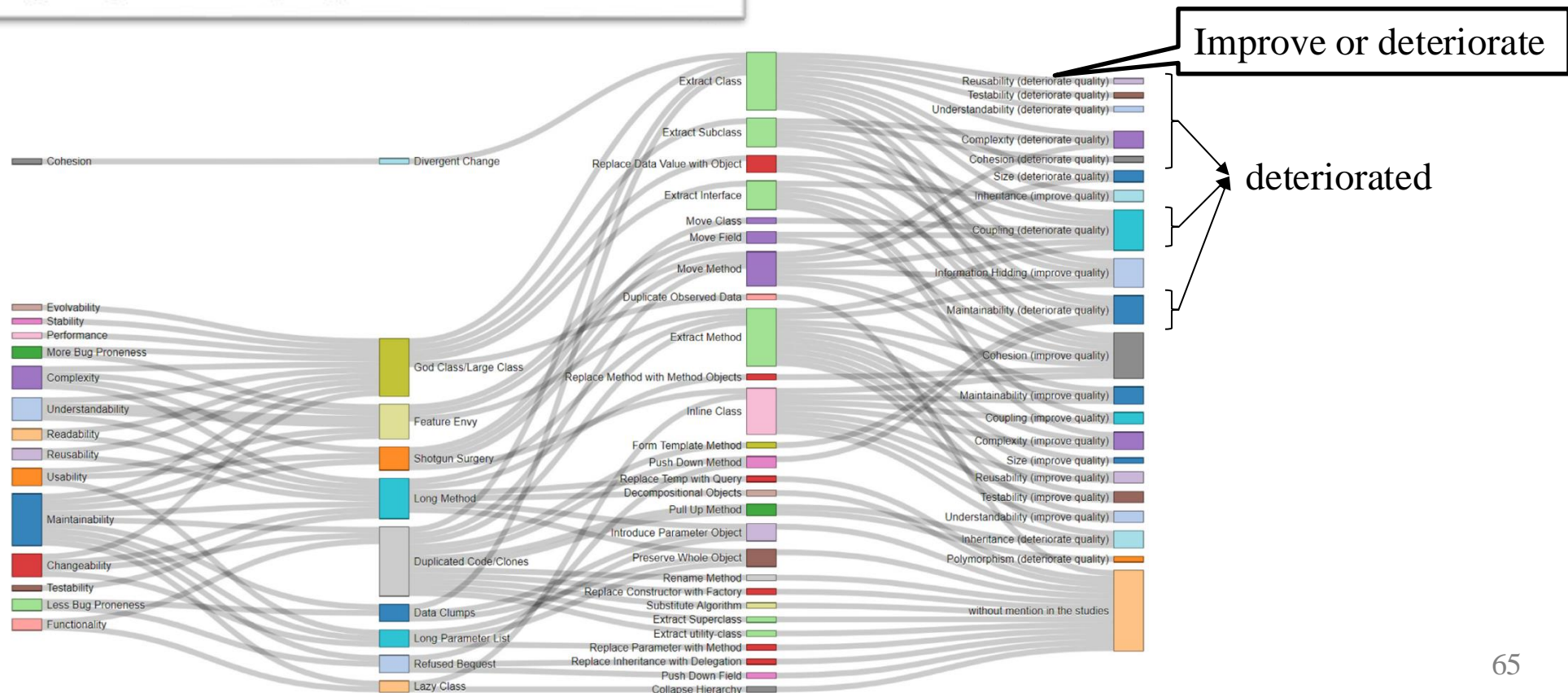


Fig. 23. The relationship among (external and internal) quality attributes and their impact on code smells and refactoring.

# Summary

---

- ◆ Maintaining code structure is important
- ◆ Accumulating problems in the code will lead to have more code changes, more latent bugs, and less comprehensible code “in the future”
  - Manage the amount of code smells in code
- ◆ Refactor the code to improve code readability and maintainability, either manually or with tool assistance, if developers are confident about validation of changes and able to handle code changes across different branches in version control system.
  - Note that removing all of them is *never* an option

# References and Resources

---

1. Bertrand Meyer, 1988. *Object-Oriented Software Construction*. Prentice Hall. ISBN 0-13-629049-3.
  - Introduce both design by contract and open-closed principle
2. Barbara H. Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (November 1994), 1811-1841.  
DOI=<http://dx.doi.org/10.1145/197320.197383>
  - Introduce the substitution principle
3. Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. 2009. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering (WCRE '09)*. IEEE Computer Society, Washington, DC, USA, 75-84. DOI=<http://dx.doi.org/10.1109/WCRE.2009.28>
  - <http://swat.polymtl.ca/~foutsekh/docs/TechReport.pdf>
4. Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Softw. Engg.* 17, 3 (June 2012), 243-275.  
DOI=<http://dx.doi.org/10.1007/s10664-011-9171-y>
5. Visual Studio Analyzer: <https://docs.microsoft.com/en-us/visualstudio/code-quality/?view=vs-2017>

# References and Resources

---

6. Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering(CSMR '11). IEEE Computer Society, Washington, DC, USA, 181-190. DOI=<http://dx.doi.org/10.1109/CSMR.2011.24>
7. Martin Fowler. *Refactoring: Improving the Design of Existing Code*. <https://martinfowler.com/books/refactoring.html>
8. Martin Fowler. <https://refactoring.com/catalog/>
9. A long list of possible static analysis for code smell detection: <https://github.com/mre/awesome-static-analysis>
10. Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. IEEE Trans. Softw. Eng. 30, 2 (February 2004), 126-139. DOI: <https://doi.org/10.1109/TSE.2004.1265817>
11. Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12). ACM, New York, NY, USA, , Article 50 , 11 pages. DOI=<http://dx.doi.org/10.1145/2393596.2393655>
  - A longer version: <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/kim-tse-2014.pdf>

# References and Resources

---

12. Guilherme Lacerda, Fabio Petrillo , Marcelo Pimenta, Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. Journal of systems and software 67 (2020) 110610.  
<https://doi.org/10.1016/j.jss.2020.110610>
13. Eman Abdullah AlOmar, Hussein Alrubaye, Mohamed Wiem Mkaouer, Ali Ouni, Marouane Kessentini: Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox. ICSE (SEIP) 2021: 348-357