Software Reengineering
OO Design Principles

Martin Pinzger
Delft University of Technology

Slides adapted from the presentation by Steve Zhang

TU Delft

SERG

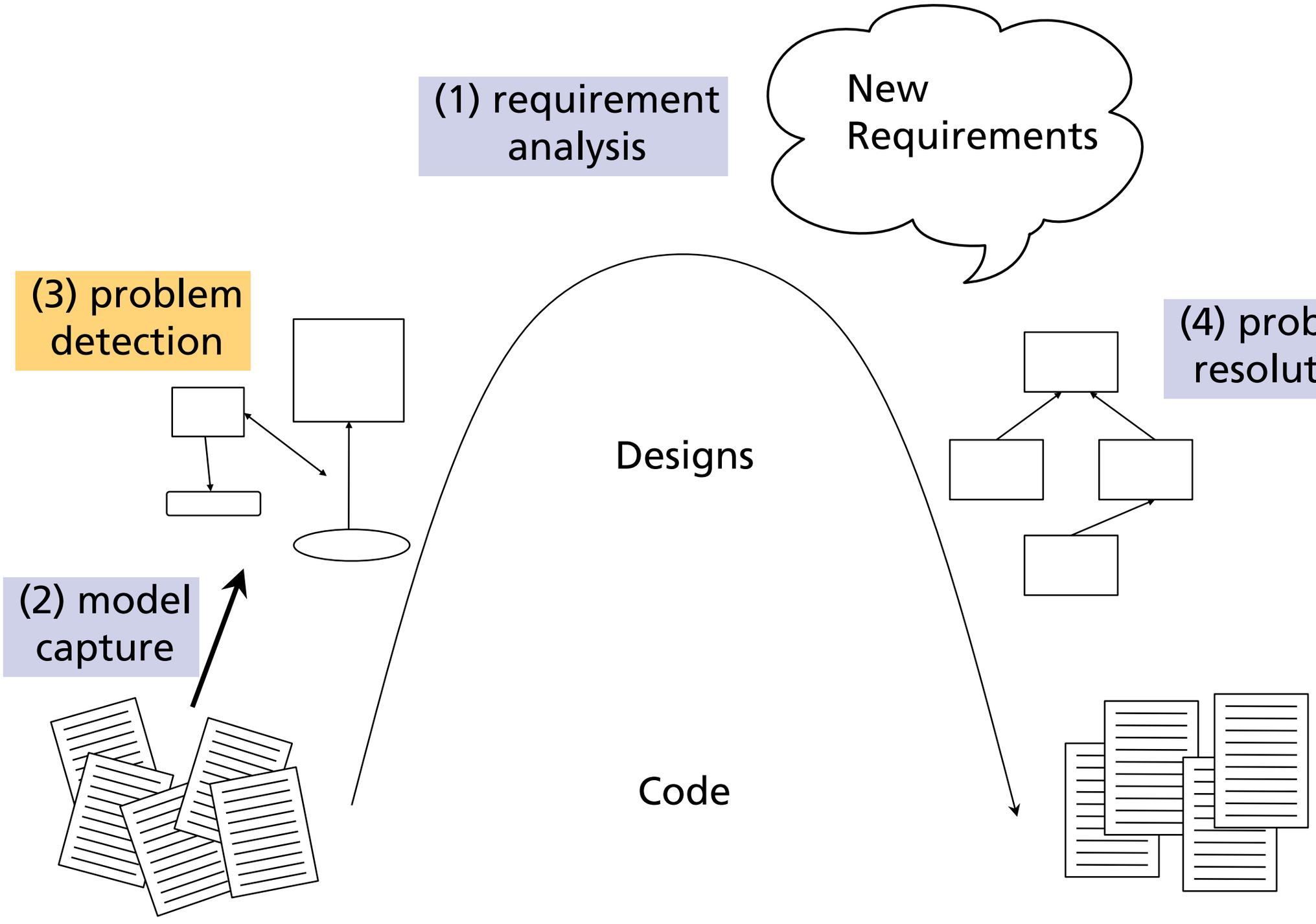
Outline

Design Smells

Object-Oriented Design Principles

Conclusion

The Reengineering Life-Cycle



Design Smells

The Odors of Rotting Software

- Rigidity – The design is hard to change
- Fragility – The design is easy to break
- Immobility – The design is hard to reuse
- Viscosity – It is hard to do the right thing
- Needless complexity – Overdesign
- Needless Repetition – Copy/paste
- Opacity – Disorganized expression

Rigidity

The tendency for software to be difficult to change

Single change causes cascade of subsequent changes in dependent modules

The more modules must be changed, the more rigid the design

Fragility

The tendency for a program to break in many places when a single changes is made

The new problems in area that have no conceptual relationship with the area that was changed

As the fragility of a module increases, the likelihood that a change will introduce unexpected problems approaches certainty.

Immobility

Difficult to reuse

A design is immobile when it contains parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too great.

This is an unfortunate but very common occurrence.

Viscosity

It is easy to do the wrong thing, but hard to do the right thing.

When the design-preserving methods are more difficult to use than the hacks, the viscosity of the design is high

When development environment is slow and inefficient, developers will be tempted to do wrong things

Needless complexity

Overdesign

A design smells of needless complexity when it contains elements that aren't currently useful

The design becomes littered with constructs that are never used

Makes the software complex and difficult to understand.

Needless Repetition

The design contains repeating structures that could be unified under a single abstraction

The problem is due to developer's abuse of cut and paste.

It is really hard to maintain and understand the system with duplicated code.

Duplication is Evil!

DRY – DON'T REPEAT YOURSELF

Opacity

Opacity is the tendency of a module to be difficult to read and understand

The code does not express its intent well

The code is written in an opaque and convoluted manner

The Broken Window Theory

A broken window will trigger a building into a smashed and abandoned derelict

So does the software

Don’t live with the Broken window

S.O.L.I.D. Design Principles

S.O.L.I.D Design Principles

SRP – The Single Responsibility Principle

OCP – The Open-Closed Principle

LSP – The Liskov Substitution Principle

ISP – The Interface Segregation Principle

DIP – The Dependency Inversion Principle

SRP: The Single-Responsibility Principle

A class should have a single purpose and only one reason to change

- If a class has more than one responsibility, then the responsibilities becomes coupled

SRP is one of the simplest of the principles, and the one of the hardest to get right

Heuristics

Describe the primary responsibility in a single sentence

Group similar methods

Look at hidden methods (private, protected)

- Many of them indicate that there is another class in the class tying to get out

Look for decisions that can change

- They should go into a separate class

Look for internal relationships

- Are certain variables used by some methods and not others?

Exercise: SRP

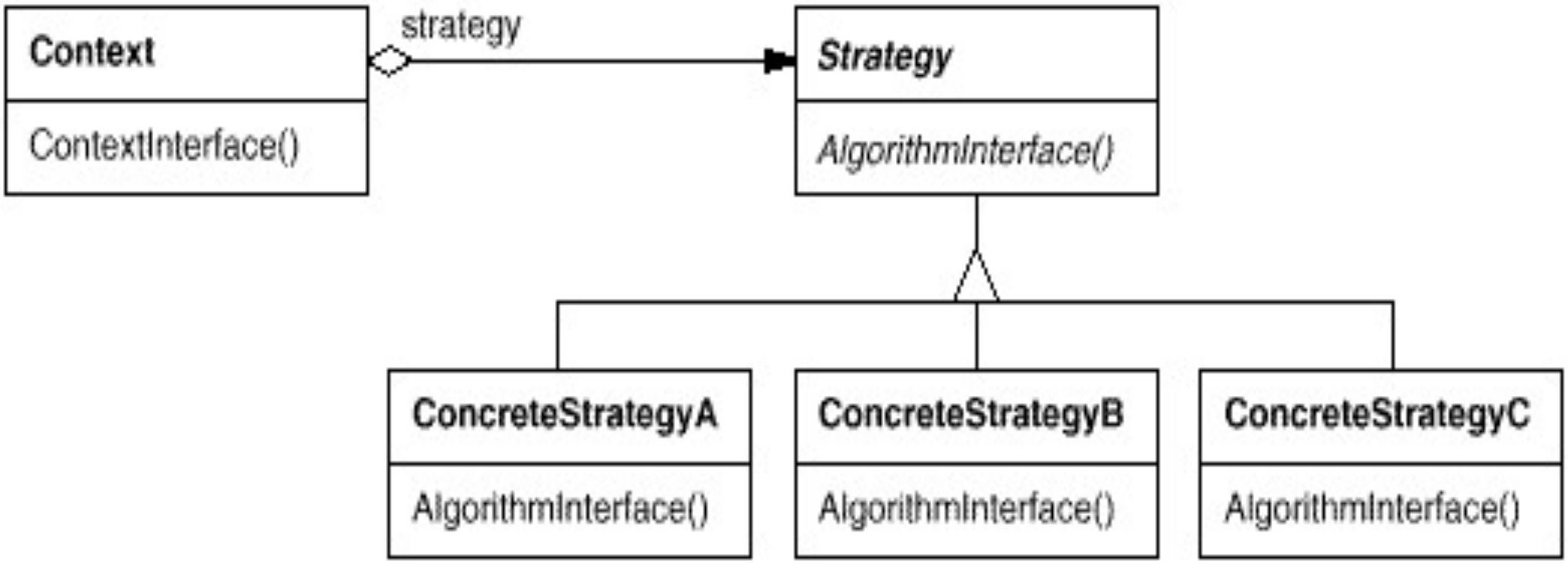
RuleParser
- current: String - variables: HashMap - currentPosition: int
+ evaluate(String rule) : int - branchingExpression(Node left, Node right) : int - causualExpression(Node left, Node right) : int - variableExpression(Node node) : int - valueExpression(Node node) : int - nextTerm() : String - hasMoreTerms() : boolean + addVariable(String name, int value)

OCP: The Open-Closed Principle

Software entities(classes, modules, functions, etc.) should be open for extension, but closed for modification

- “Open for extension”
  - The behavior of the module can be extended (e.g., by subclassing)
- “Closed for modification”
  - Extending the behavior of a module does not result in changes to the existing source code or binary code of the module

Example: OCP – Strategy Pattern



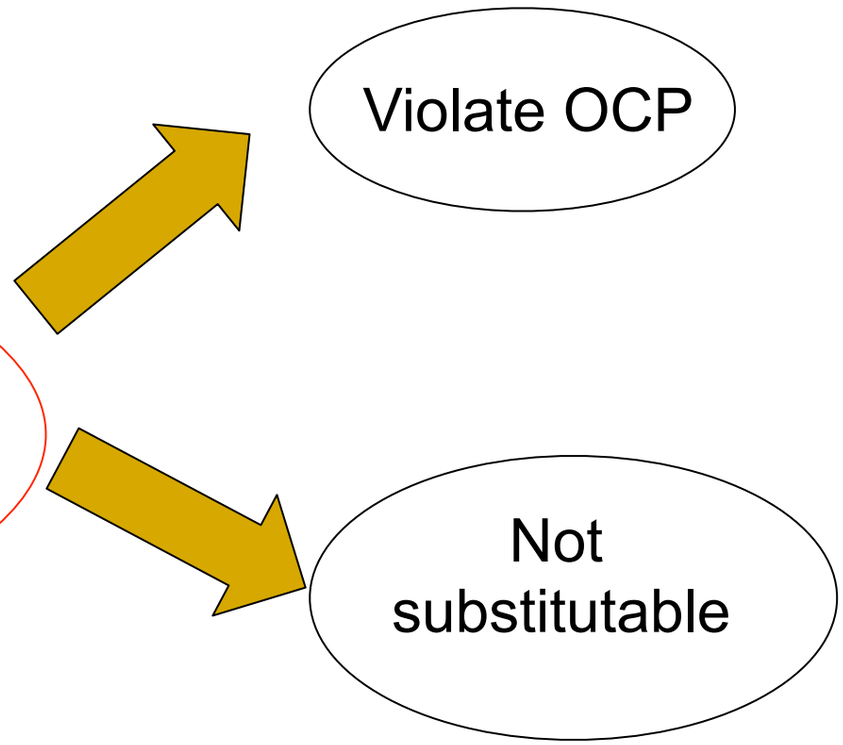
LSP: Liskov Substitution Principle

Subtypes must be substitutable for their base types

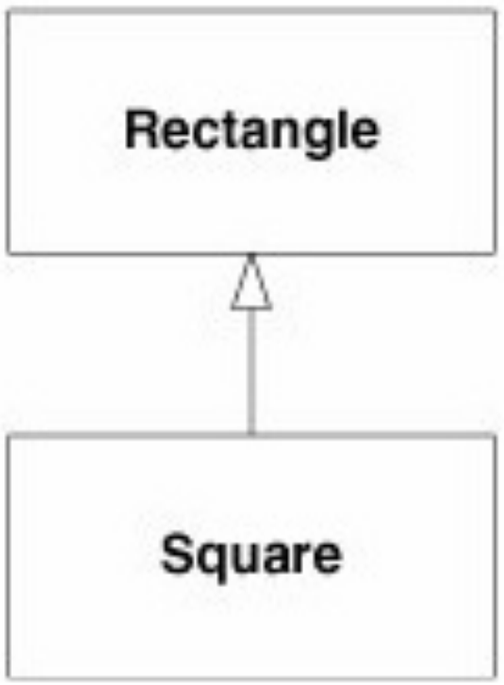
- LSP defines the OO inheritance principle
- If a client uses a base class, then it should not differentiate the base class from derived class, which means the derived class can substitute the base class

LSP violation example

```
public enum ShapeType {square, circle};
public class Shape {
    public static void DrawShape(Shape s)  {
        if(s.type == ShapeType.square)
            (s as Square).Draw();
        else if(s.type == ShapeType.circle)
            (s as Circle).Draw();
    }
}
public class Circle : Shape {
     public void Draw() {/* draws the circle */}
}
public class Square : Shape{
        public void Draw() {/* draws the square */}
}
```



Another LSP violation example




```
void g(Rectangle r)
{
    r.setWidth(5);
    r.setHeight(4);
    if(r.getArea() != 20)
        throw new Exception("Bad area!");
}
```

Square is not Rectangle!

IS-A Relationship

Square's behavior is changed, so it is not substitutable to Rectangle

DIP: The Dependency Inversion Principle

High-level modules should not depend on low-level modules

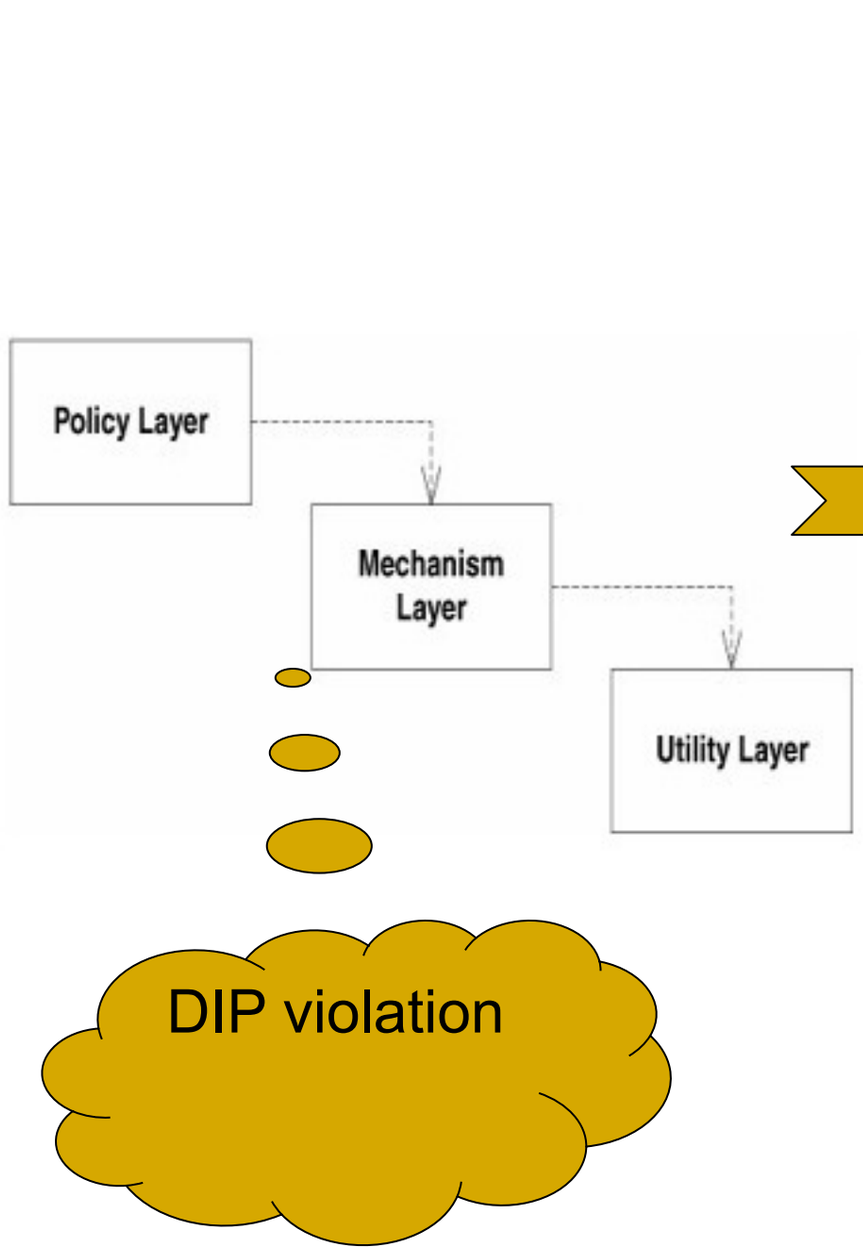
- Both should depend on abstractions

Abstractions should not depend on details

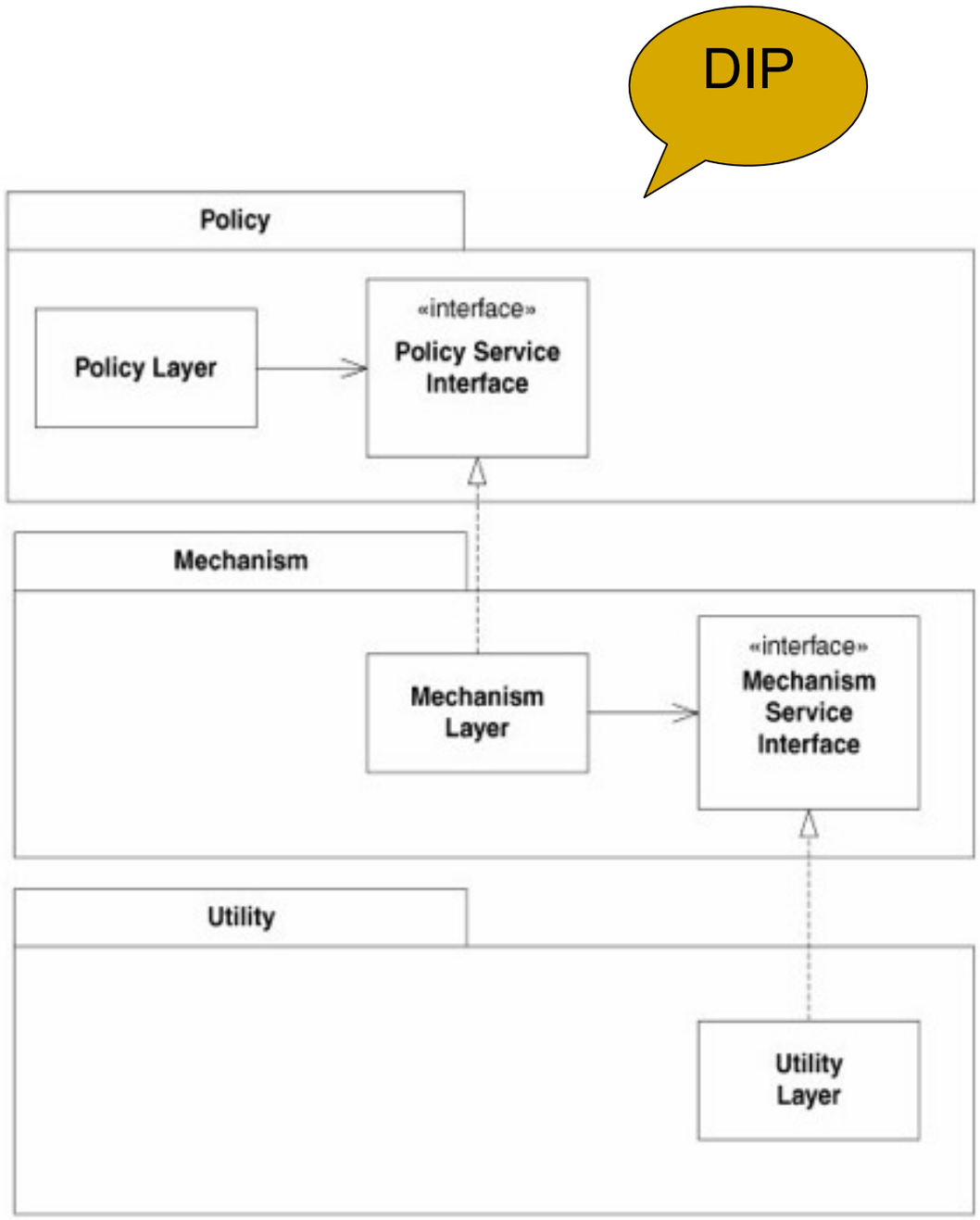
- Details should depend on abstractions

DIP is at the very heart of framework design

A DIP example



DIP violation



ISP: The Interface Segregation Principle

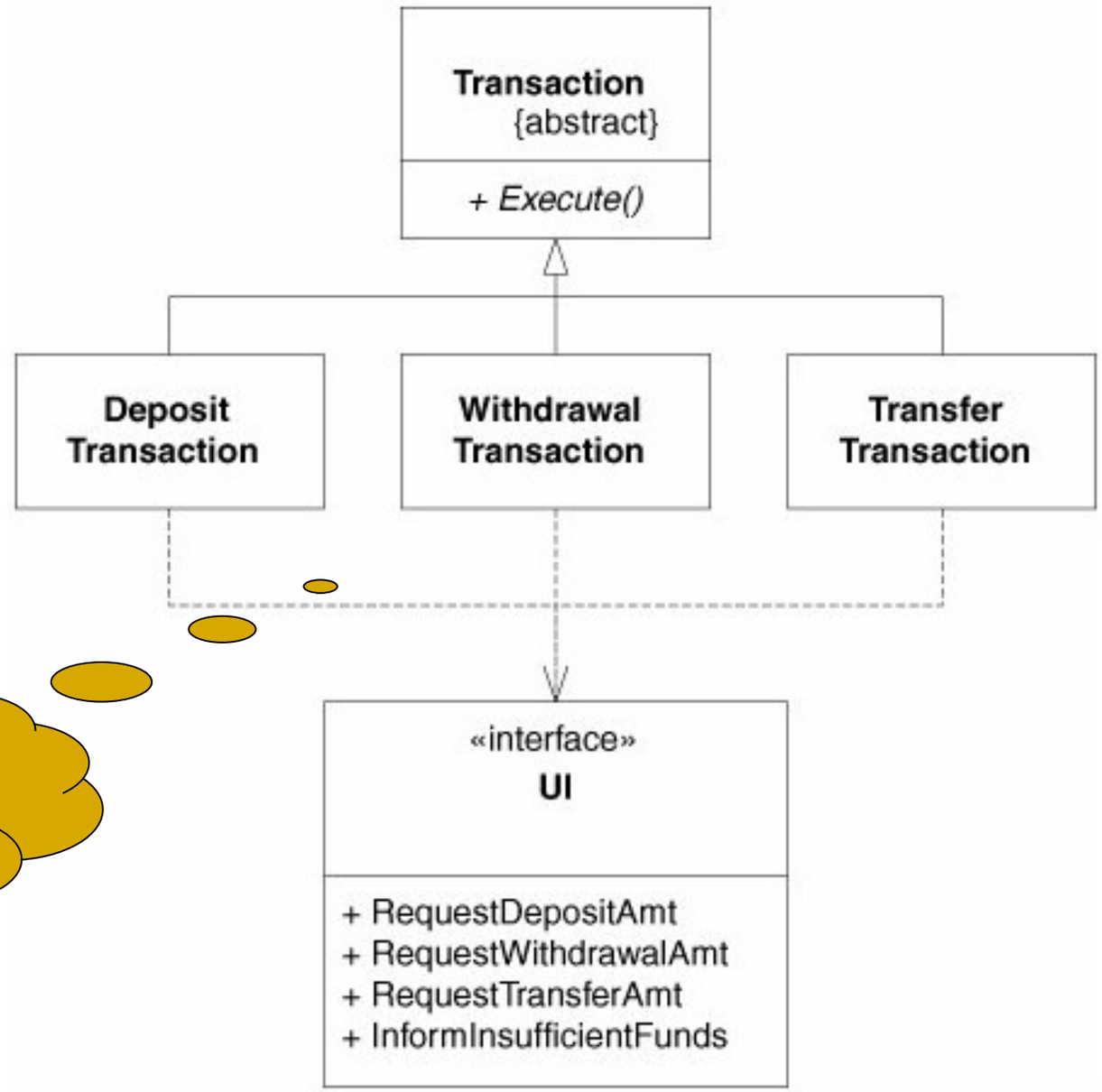
Clients should not be forced to depend on methods they do not use

- Design cohesive interfaces and avoid "fat" interfaces
- The dependency of one class to another one should depend on the smallest possible interface

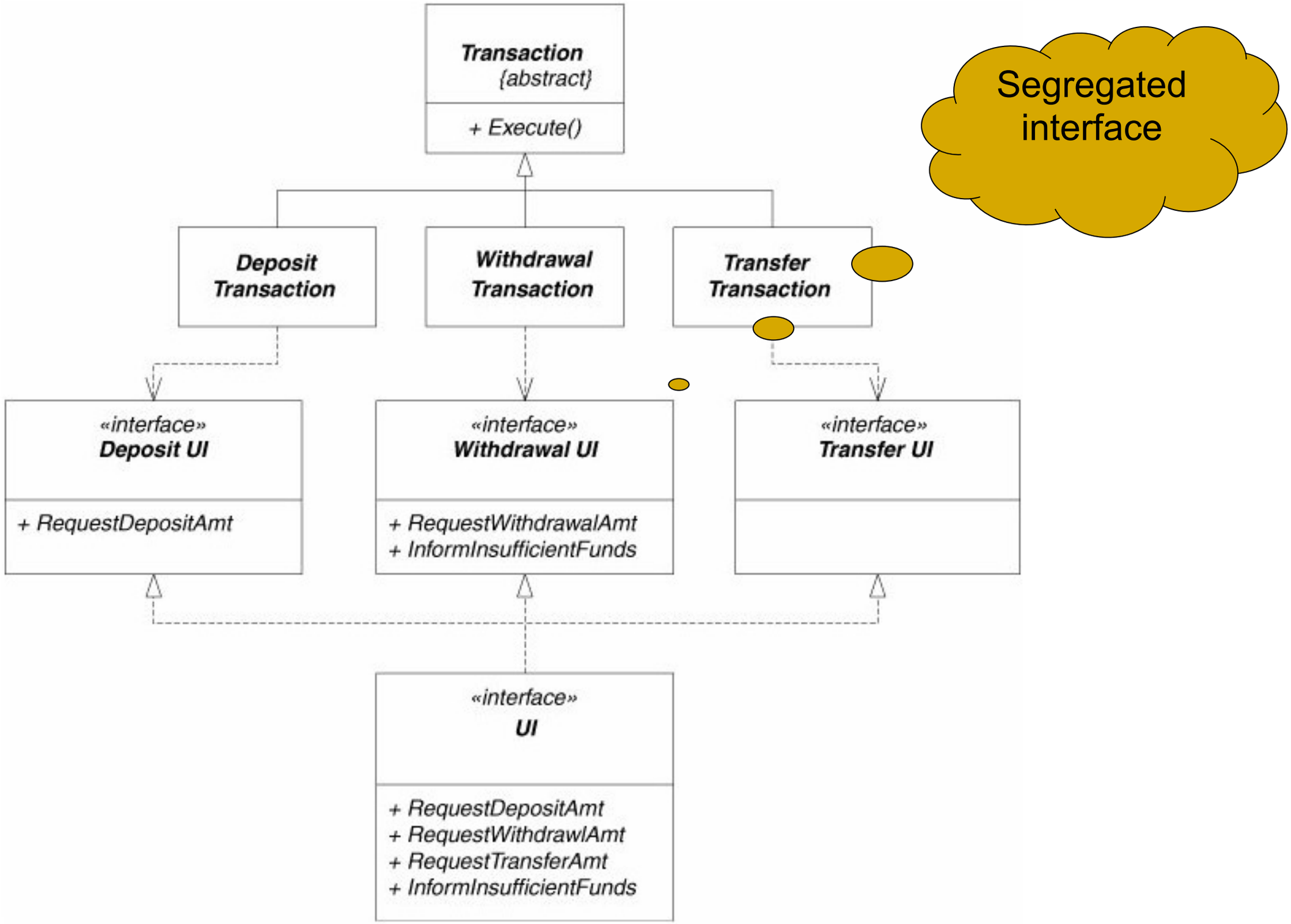
The interfaces of the class can be broken up into groups of methods

- Each group serves a different set of clients

An violation of ISP example



An ISP Violation example: solution



Segregated interface

LoD - Law of Demeter

Principle of Least Knowledge

Only talk to your immediate friends

Don’t talk to strangers

Write “shy” codes

Minimize coupling

LoD formal definition

A method M of an object O may only invoke the methods of the following kinds of objects

- O itself
- M's parameters
- Any objects created/instantiated within M
- O's direct component objects

Example LoD

```
class Demeter {
  public A a;
  public int func() {
     // do something
  }
  public void example(Arg arg) {
      C c = new C();
      int f = func();      // functions belonging to itself
      arg.invert();        // to passed parameters
      a = new A();
      a.setActive();     // to any objects it has created
      c.print();           // to any held objects
  }
}
```

LoD violation example

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

```
a.getB().getC().doSomething()
```

DRY – Don’t Repeat Yourself

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

Following DRY will make software developments easier to understand and maintain

Duplication is Evil

Summary

The OO design principles help us:

- As guidelines when designing flexible, maintainable and reusable software
- As standards when identifying the bad design
- As laws to argue when doing code review