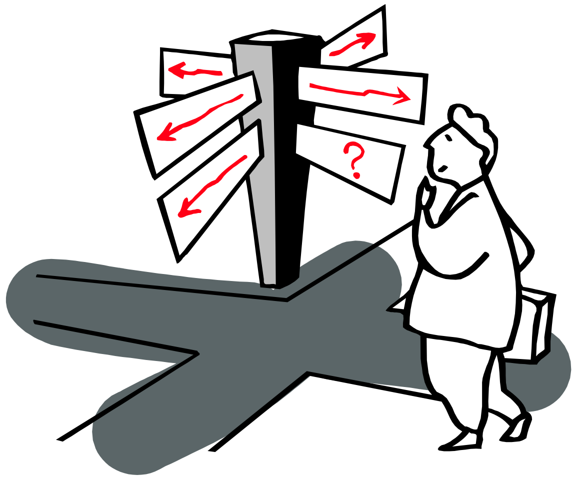
Software Reengineering P3: OO Design Principles and Violations

Martin Pinzger
Delft University of Technology

Slides adapted from the presentation by Steve Zhang

Outline

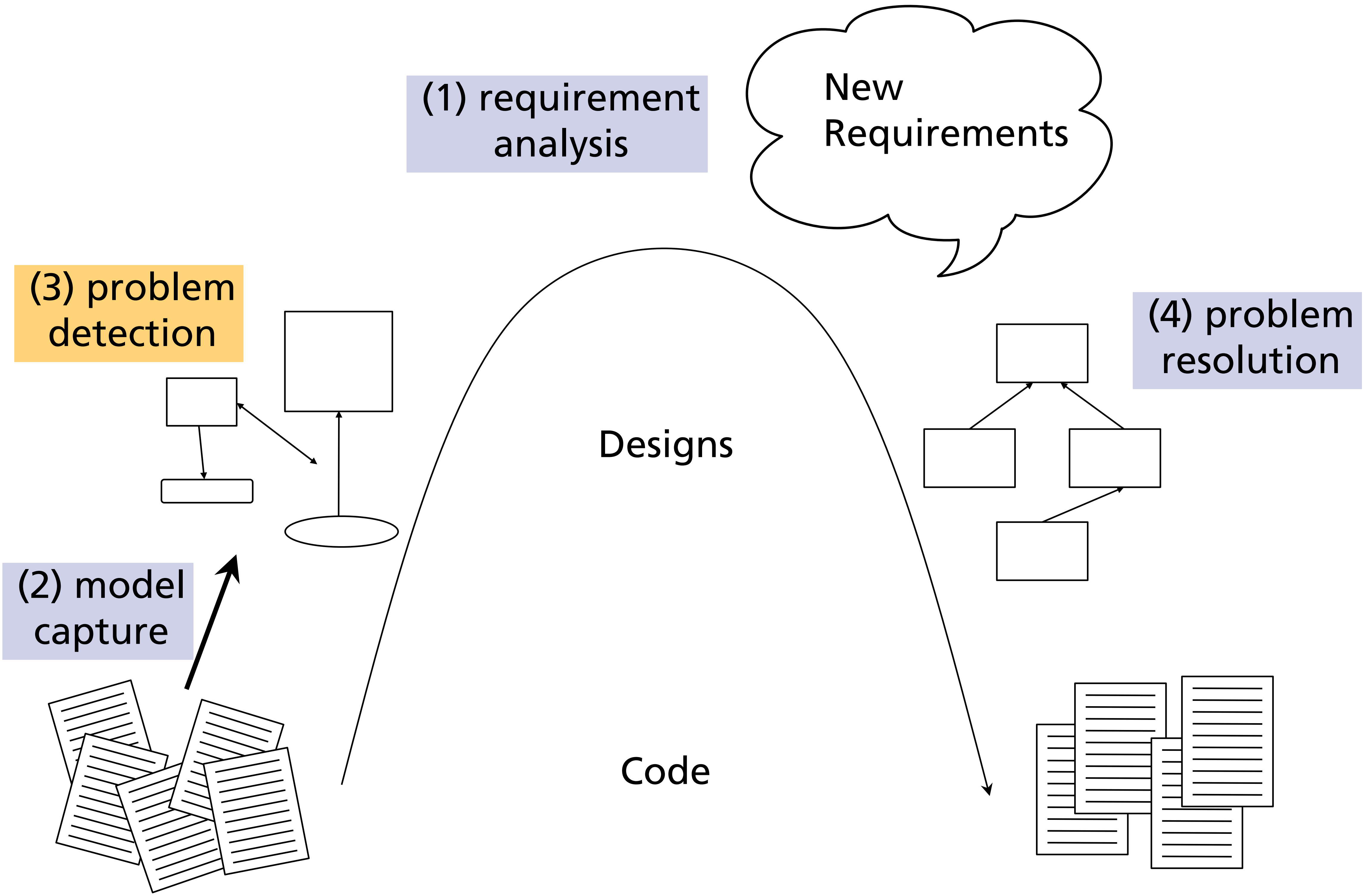


Design Smells

Object-Oriented Design Principles

Conclusions

The Reengineering Life-Cycle



(1) requirement analysis

New Requirements

(3) problem detection

(4) problem resolution

Designs

(2) model capture

Code

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

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

SRP 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

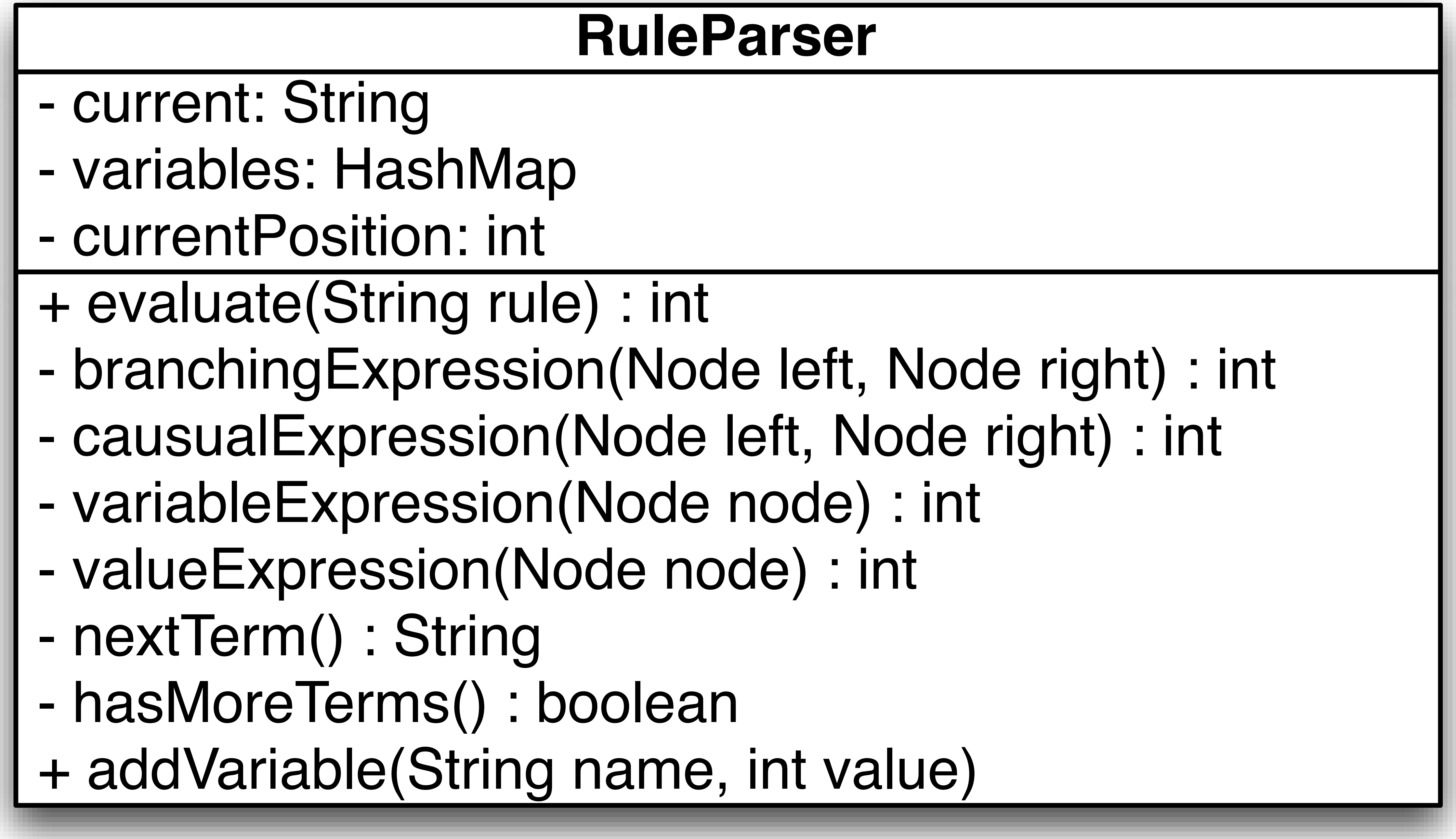
Look for decisions that can change (not “if-statements”)

- They should go into separated classes

Look for internal relationships

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

Exercise: SRP

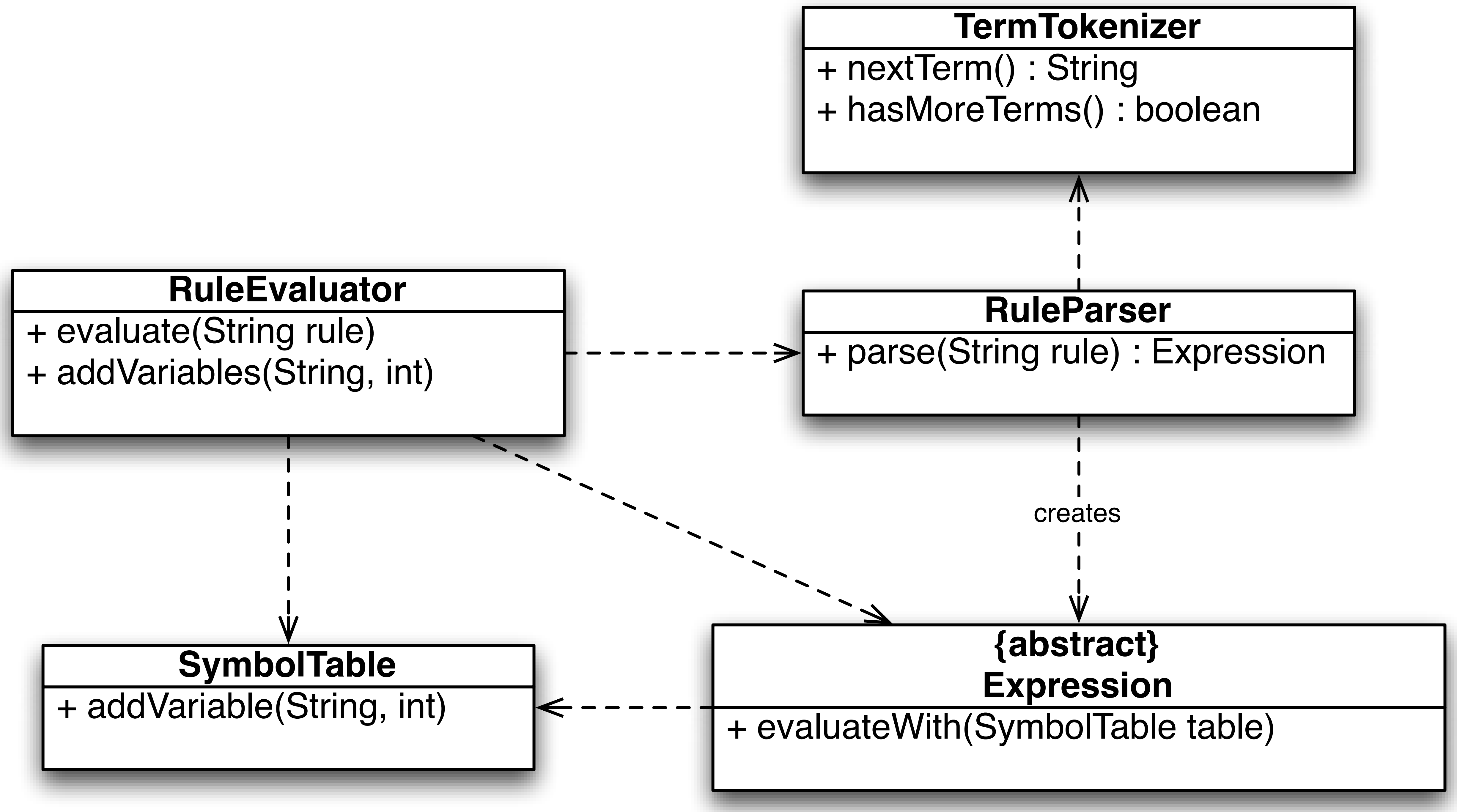


RuleParser

- 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)

Example: SRP (possible) solution

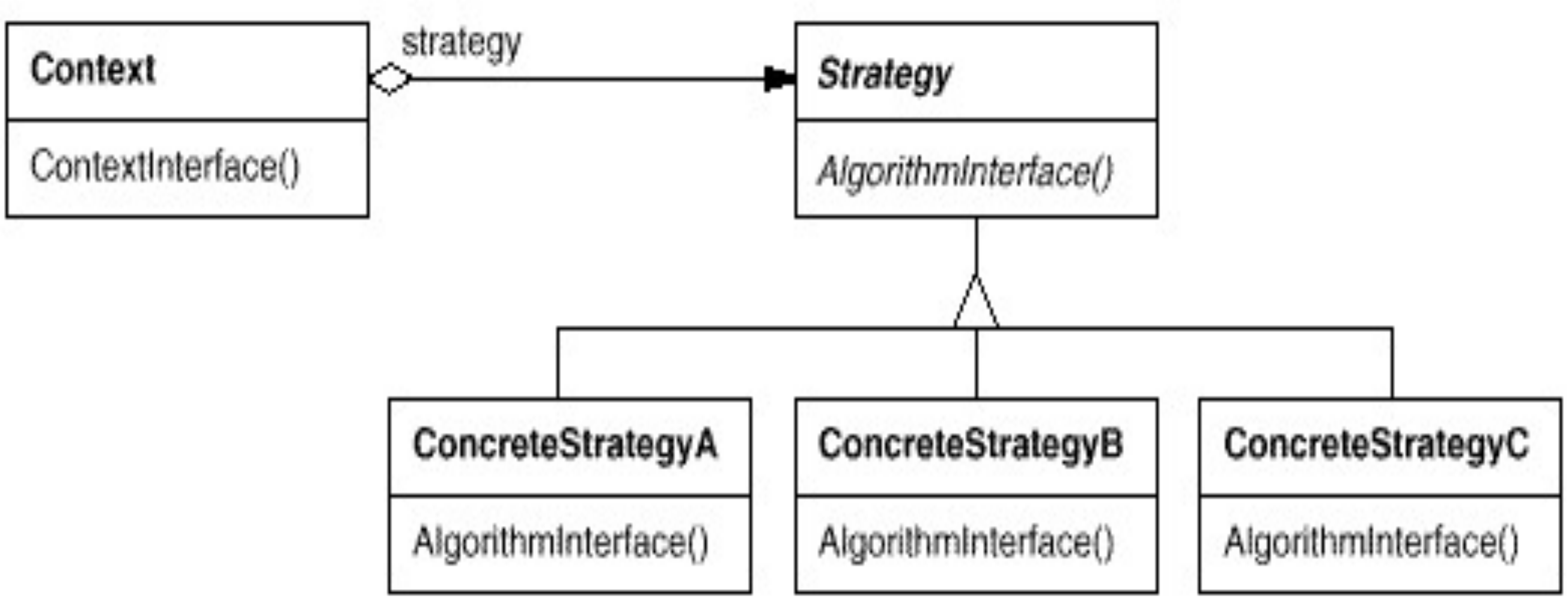

TermTokenizer
+ nextTerm() : String
+ hasMoreTerms() : boolean
RuleEvaluator
+ evaluate(String rule)
+ addVariables(String, int)
RuleParser
+ parse(String rule) : Expression
creates
SymbolTable
+ addVariable(String, int)
{abstract}
Expression
+ evaluateWith(SymbolTable table)

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



OCP cannot be fully achieved

- E.g.,

OCP heuristics

Look for duplicated code

Look at the change history

- Classes that frequently change together

Apply potential change scenarios

- Which classes would be affected by the change?

LSP: Liskov Substitution Principle

Subtypes must be substitutable for their base types

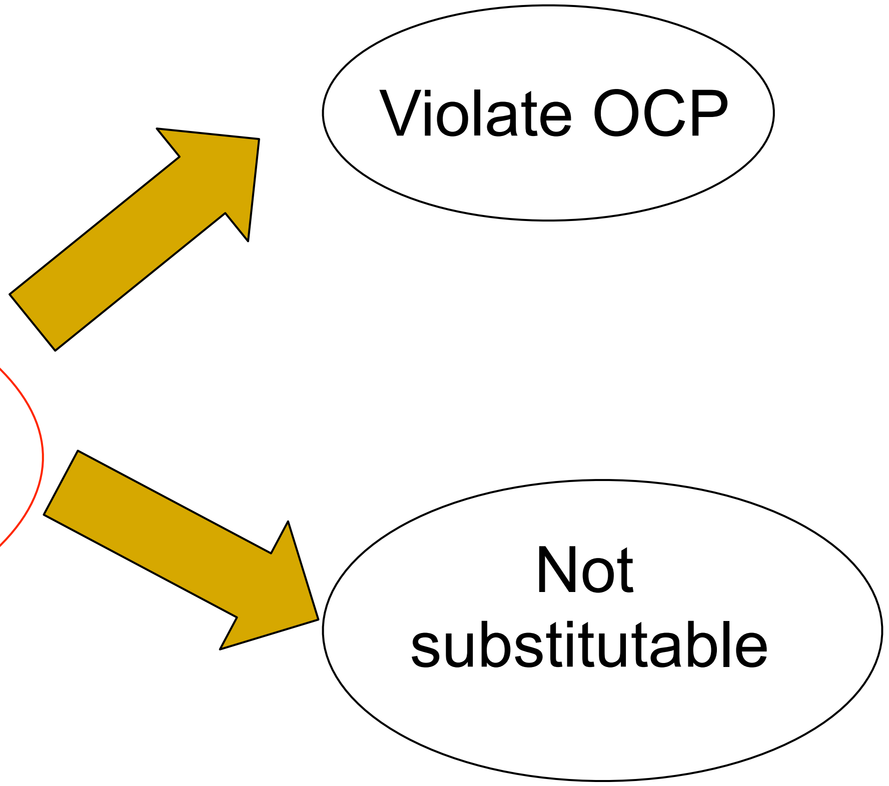
- LSP defines the OO inheritance principle
- If a client uses a base class, then it should not differentiate the base class from derived class

In terms of design by contract

- Precondition equal or weaker
  - Must accept anything the base class could accept
- Postcondition equal or stronger
  - Must not violate the post-condition of the base class

LSP violation example

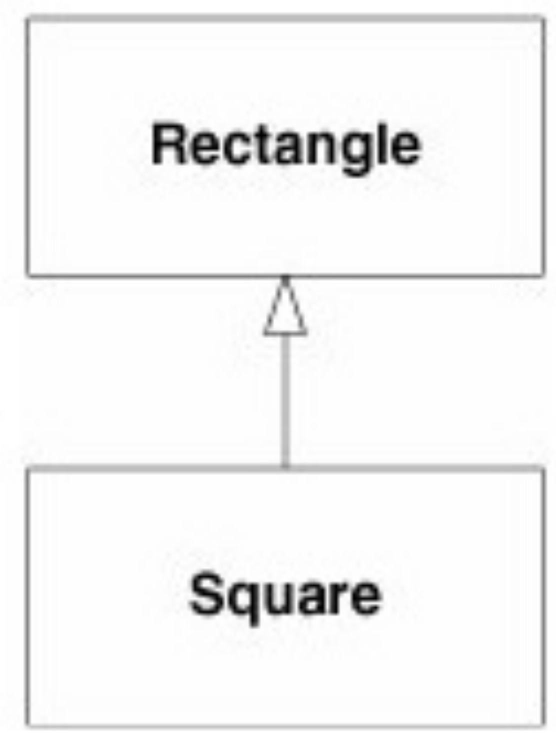
```
public enum ShapeType {square, circle};
public class Shape {
    public static void DrawShape(Shape s)  {
            if(s.type == ShapeType.square)
                (s as Square).Draw();
           else if(s.type == ShapeType.circle)
                (s as Circle).Draw();
    }
}
public class Circle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



Violate OCP

Not substitutable

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

LSP heuristics

Check the contracts of base and sub classes

- Every LSP violation is a violation of OCP but not vice versa

DIP: The Dependency Inversion Principle

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

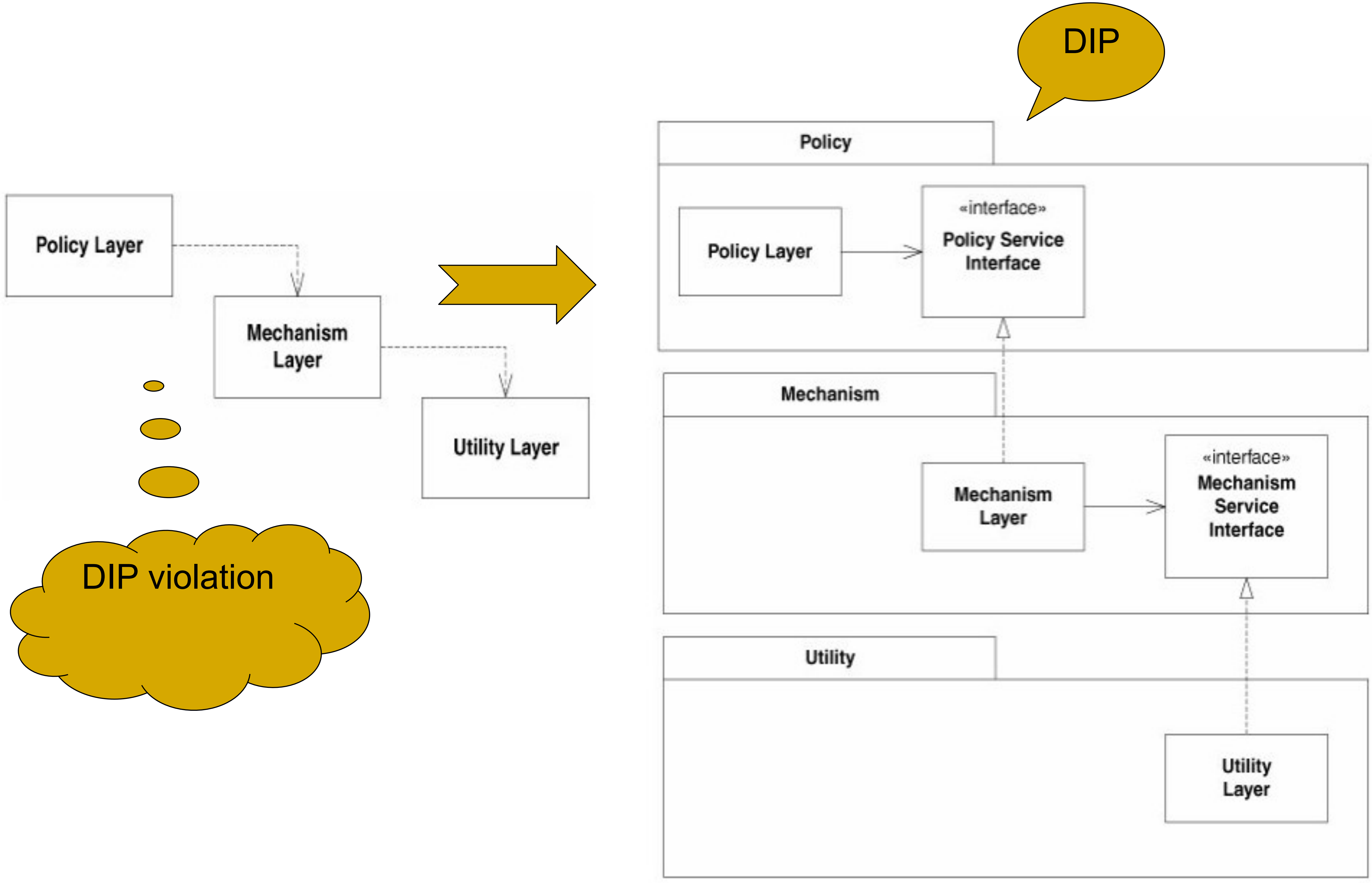
- Both should depend on abstractions

Abstractions should not depend on details

- Details should depend on abstractions

DIP is at the very heart of framework design

A DIP example



DIP heuristics

Depend on abstractions

- No variable should hold a reference to a concrete class
- No class should derive from a concrete class
- No method should override an implemented method of any of its base classes

Heuristic is typically violated at least once

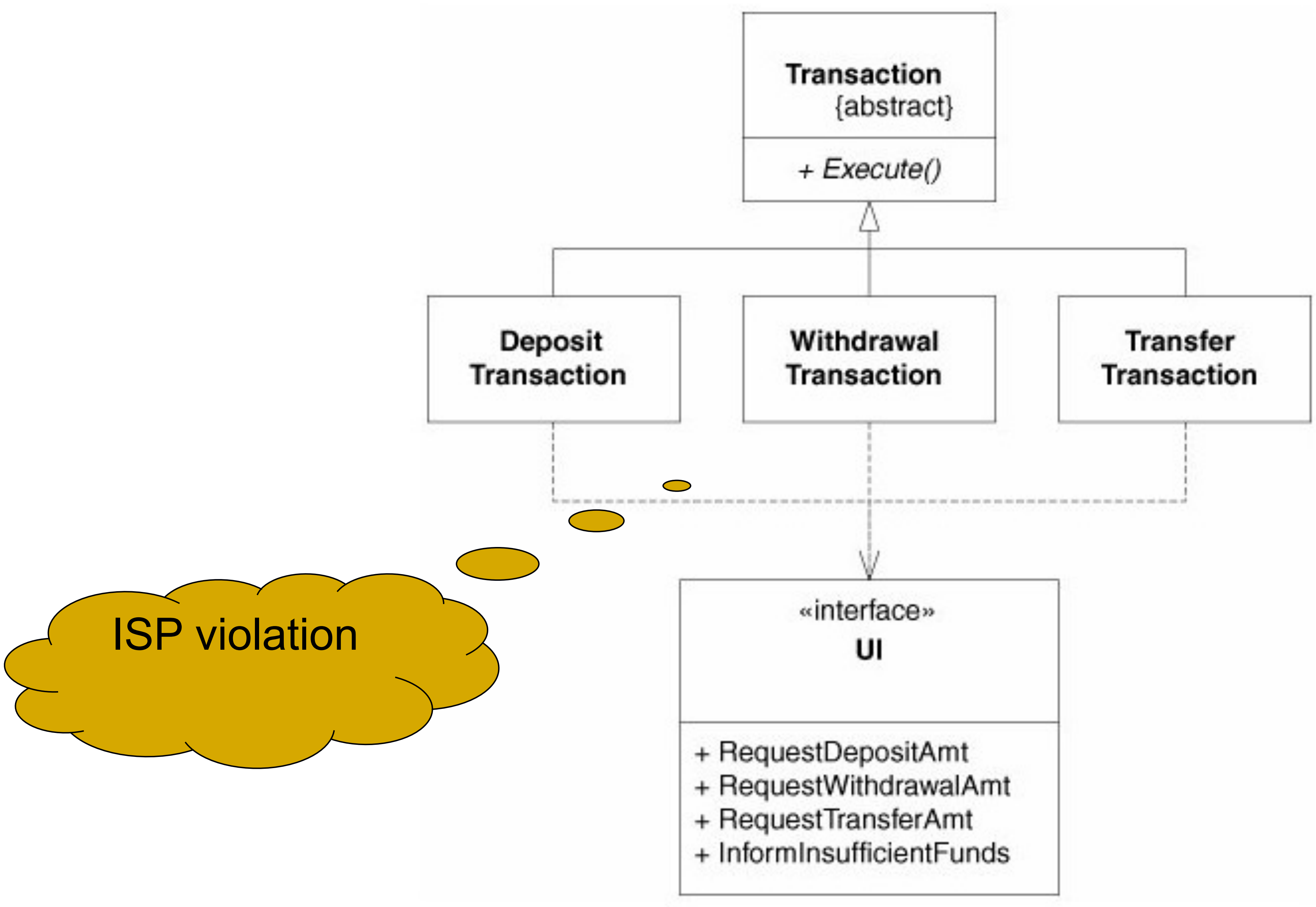
- Somebody has to create the instances of the concrete classes
- -> No reason to strictly follow this heuristic for classes that are concrete but non-volatile

ISP: The Interface Segregation Principle

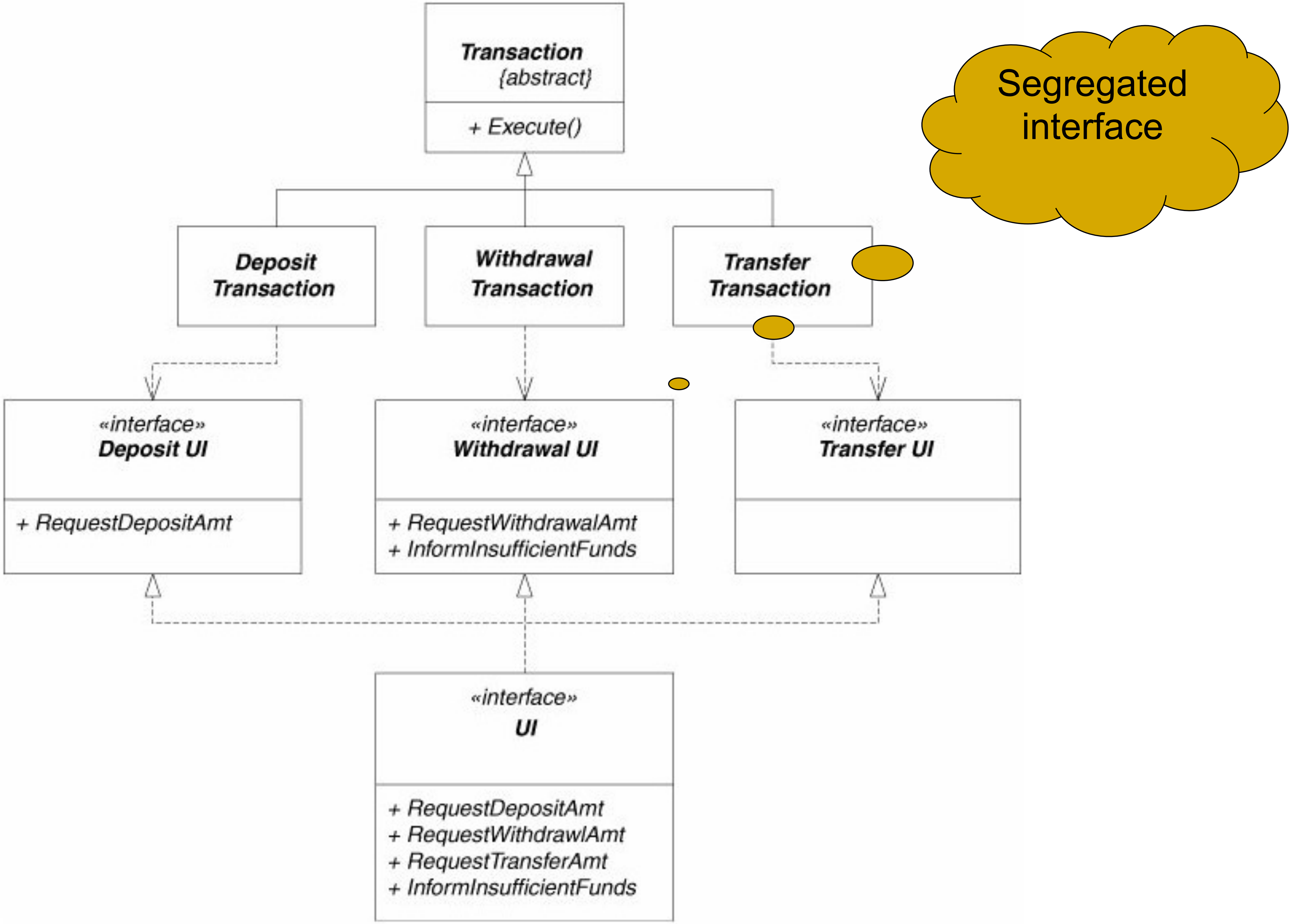
Clients should not be forced to depend on methods they do not use

- 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

ISP heuristics

Check classes with a high number of public methods

- Group clients according to their calls of the public methods
- Check for methods that frequently change together

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()
```

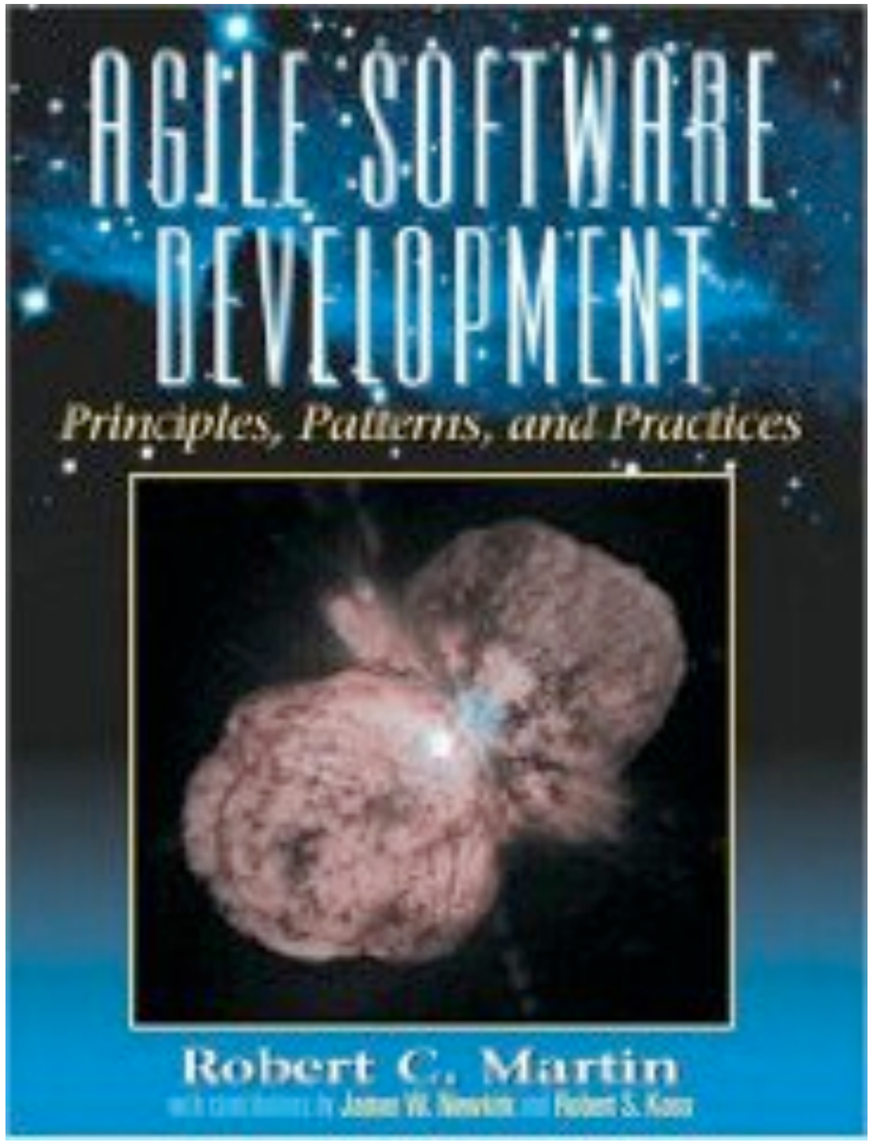
DRY – Don’t Repeat Yourself

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

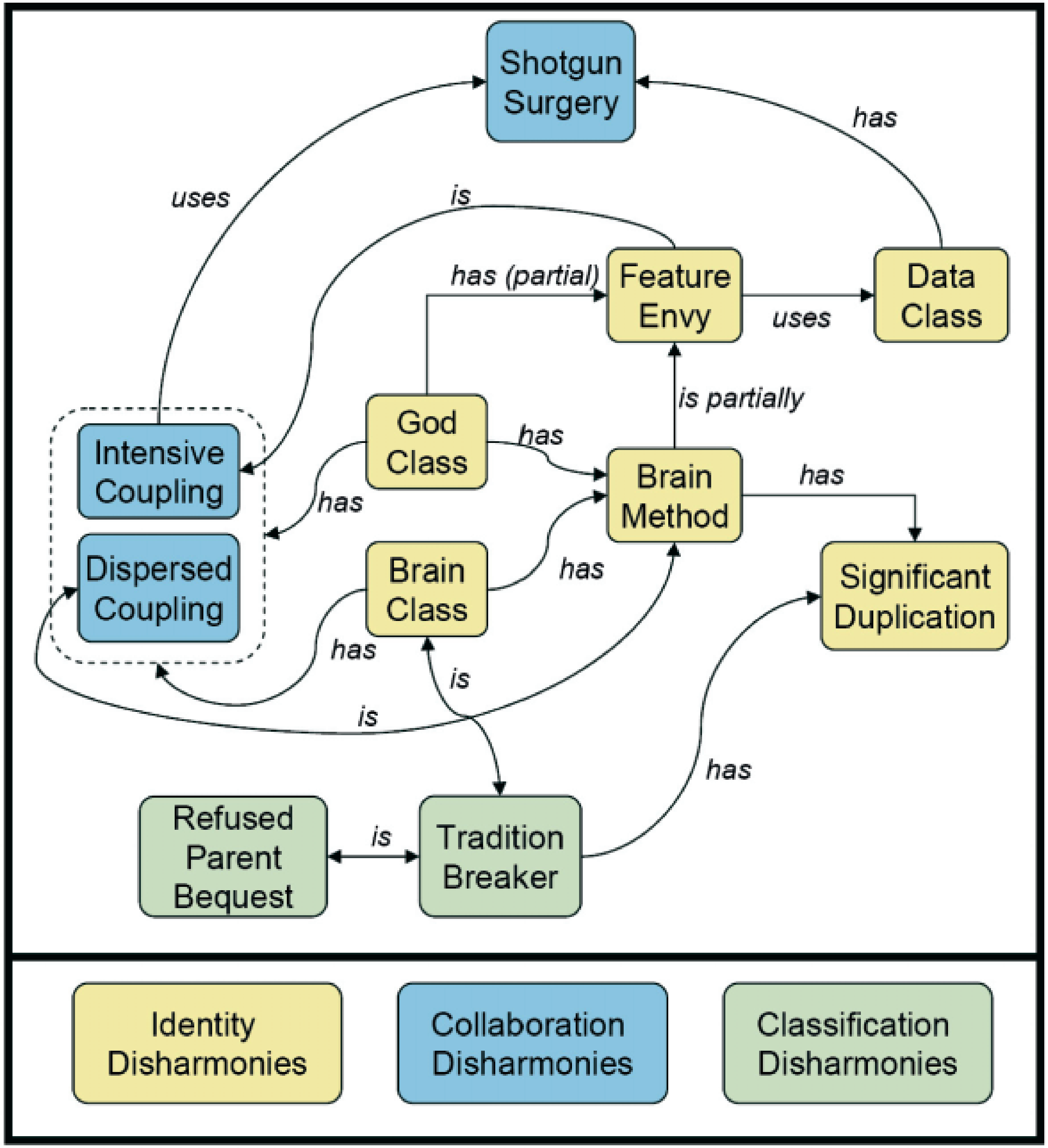
Following DRY will make software systems easier to understand and maintain

More information on Design Principles

Agile Software Development: Principles Patterns, and Practices
Robert C. Martin, Prentice Hall, 2002


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

Design Disharmonies



Collaboration Disharmonies

Collaboration Disharmonies

Limit collaboration intensity

Operations should collaborate (mainly unidirectional) with a limited number of services provided by other classes

Limit collaboration extent

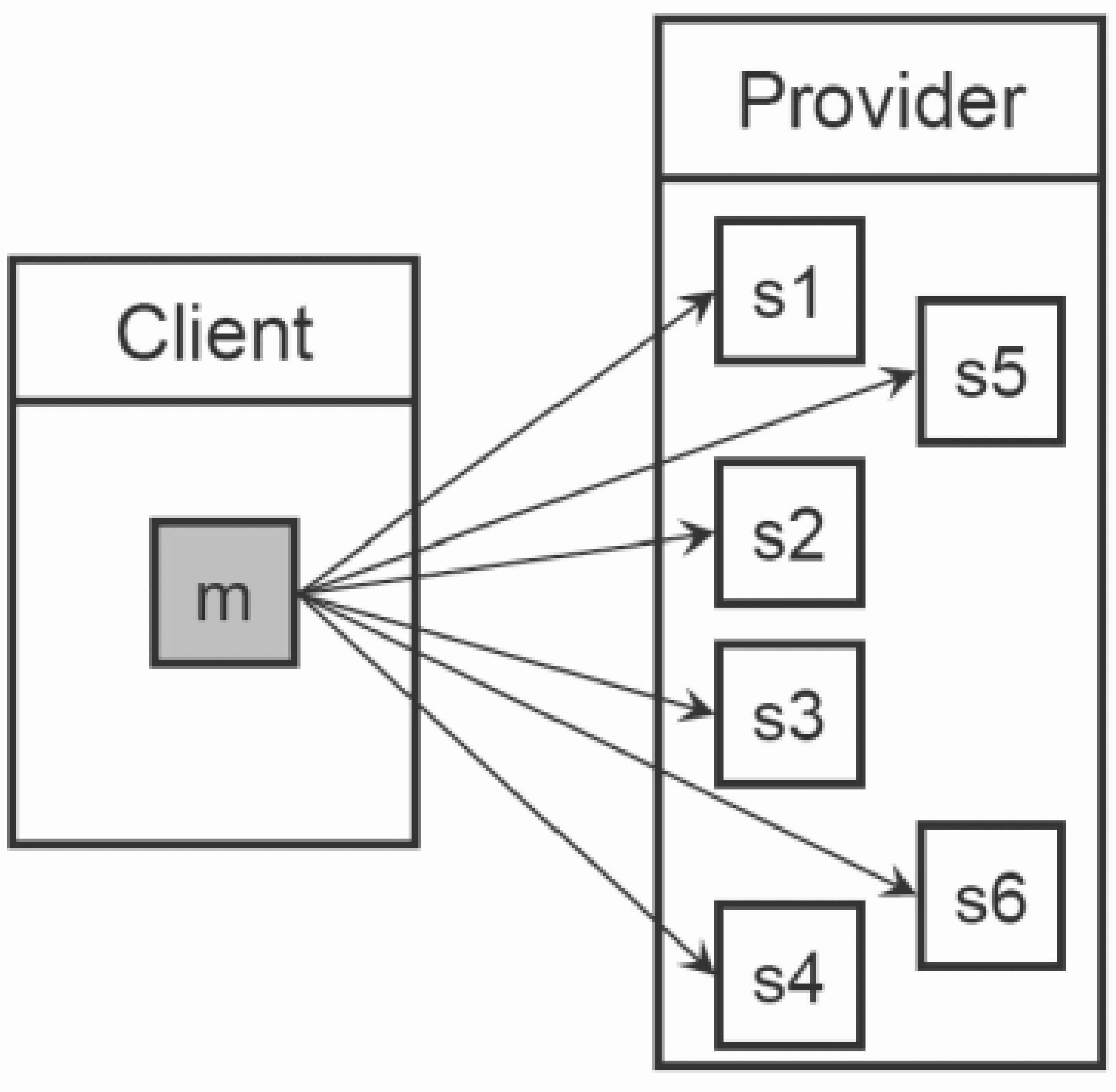
Operations (and consequently their classes) should collaborate with operations from a limited number of other classes

Limit collaboration dispersion

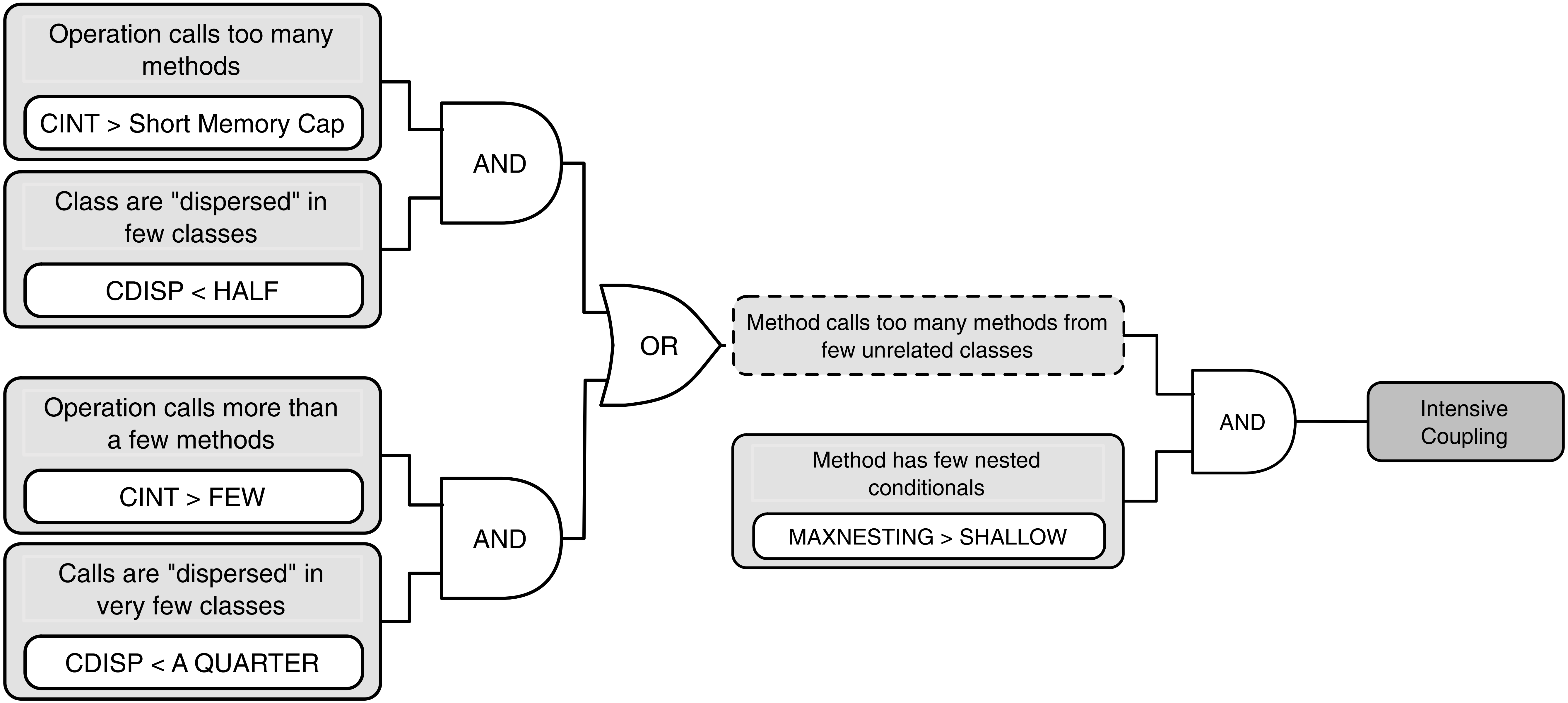
An entity should collaborate closely only with a selected set of entities, preferable located in the

- same abstraction
- same hierarchy
- same package (or sub- system)

Intensive Coupling



Intensive Coupling: Detection Strategy



Operation calls too many methods

CINT > Short Memory Cap

Class are "dispersed" in few classes

CDISP < HALF

Operation calls more than a few methods

CINT > FEW

Calls are "dispersed" in very few classes

CDISP < A QUARTER

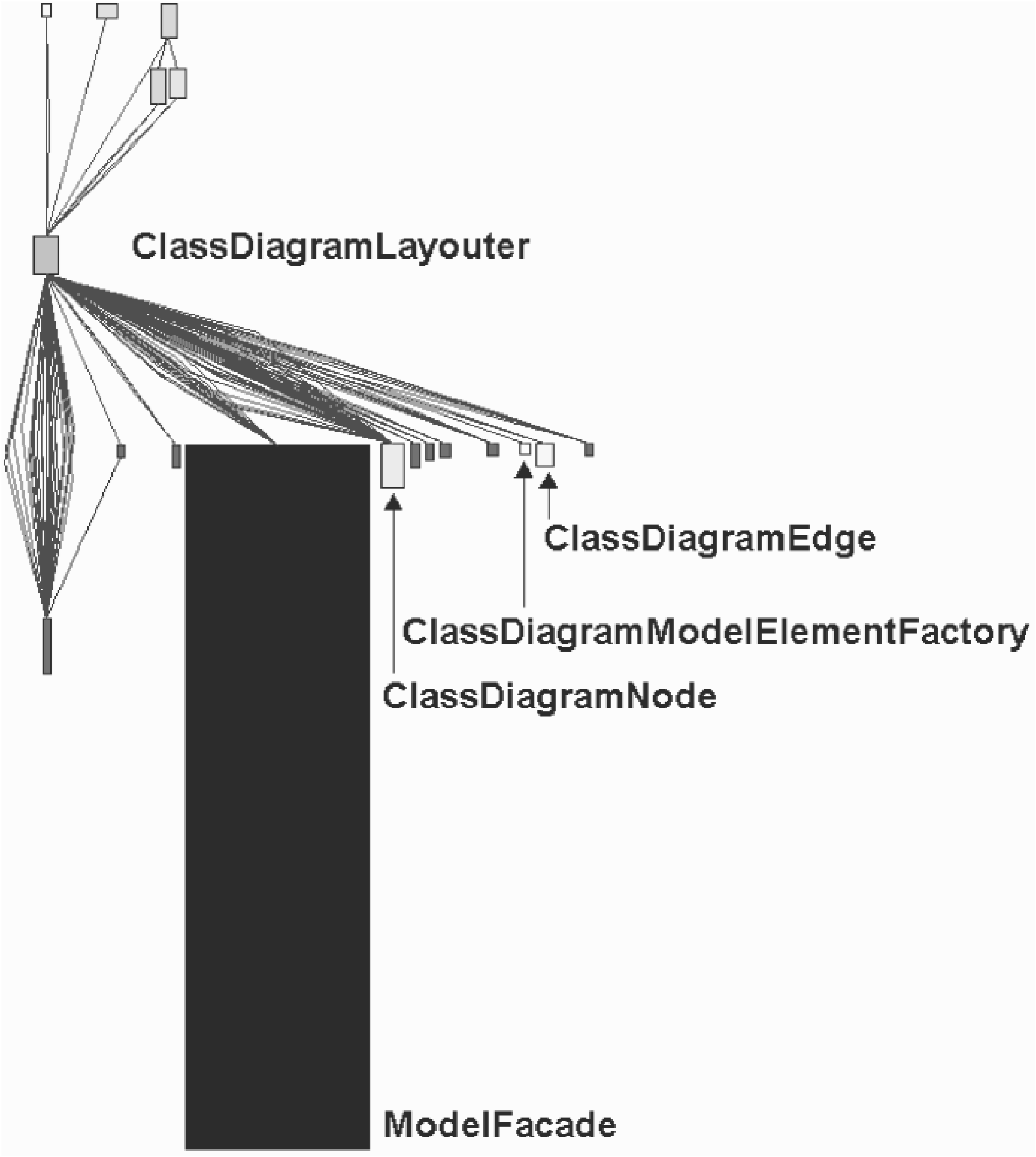
Method calls too many methods from few unrelated classes

Method has few nested conditionals

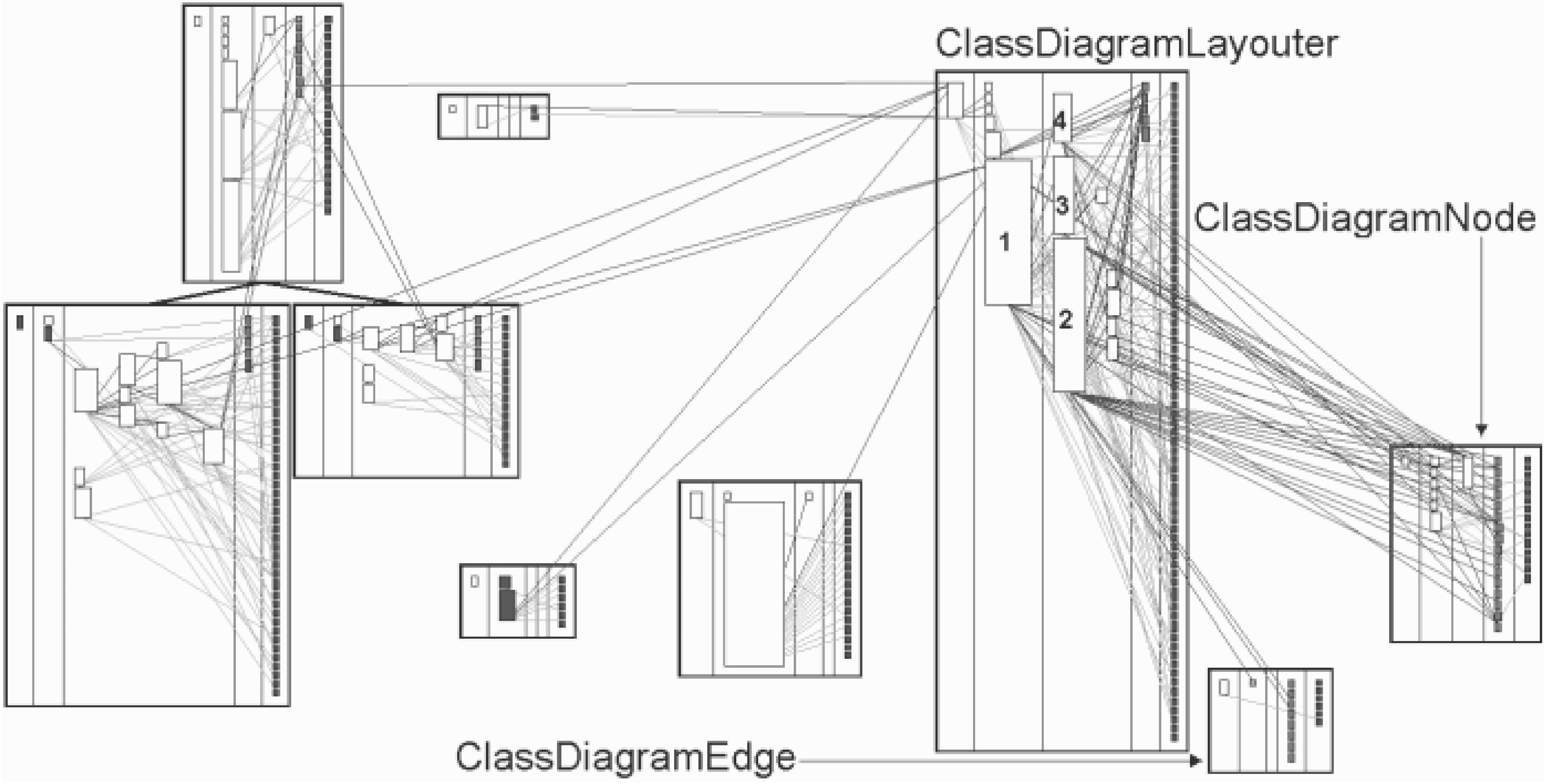
MAXNESTING > SHALLOW

Intensive Coupling

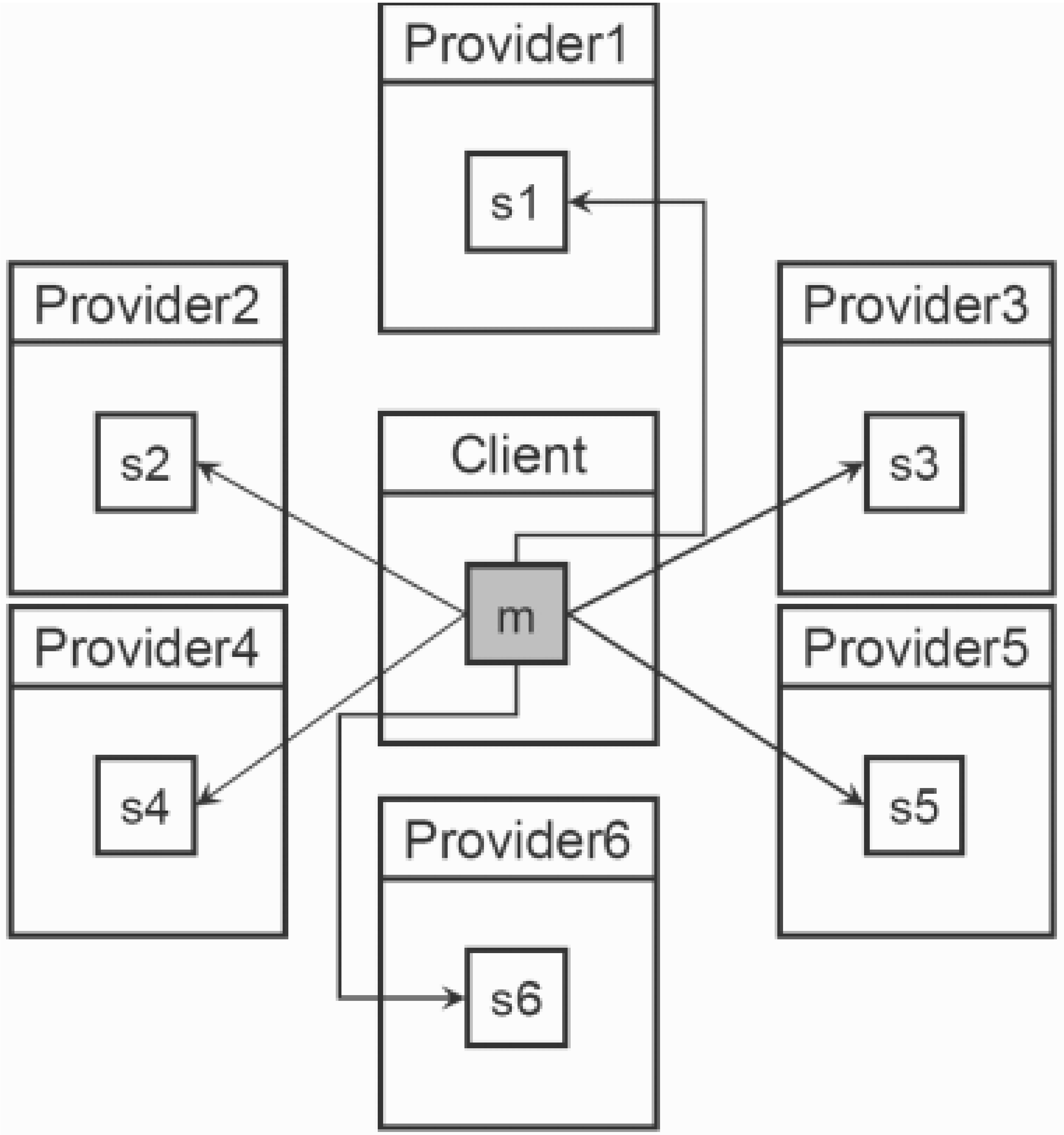
Intensive Coupling: Example



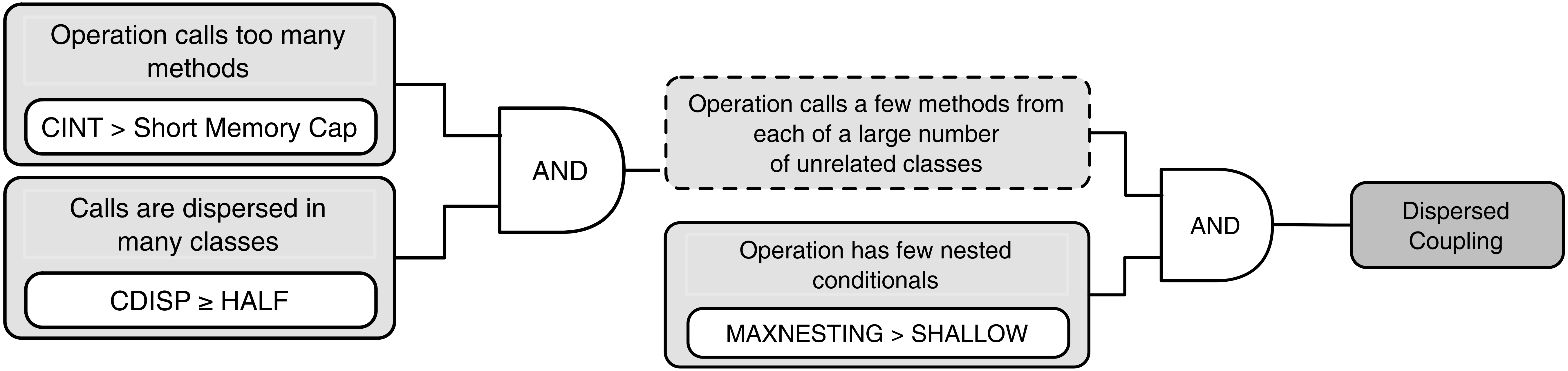
Intensive Coupling: Class Blueprint



Dispersed Coupling



Dispersed Coupling: Detection Strategy



Operation calls too many methods

CINT > Short Memory Cap

Calls are dispersed in many classes

CDISP ≥ HALF

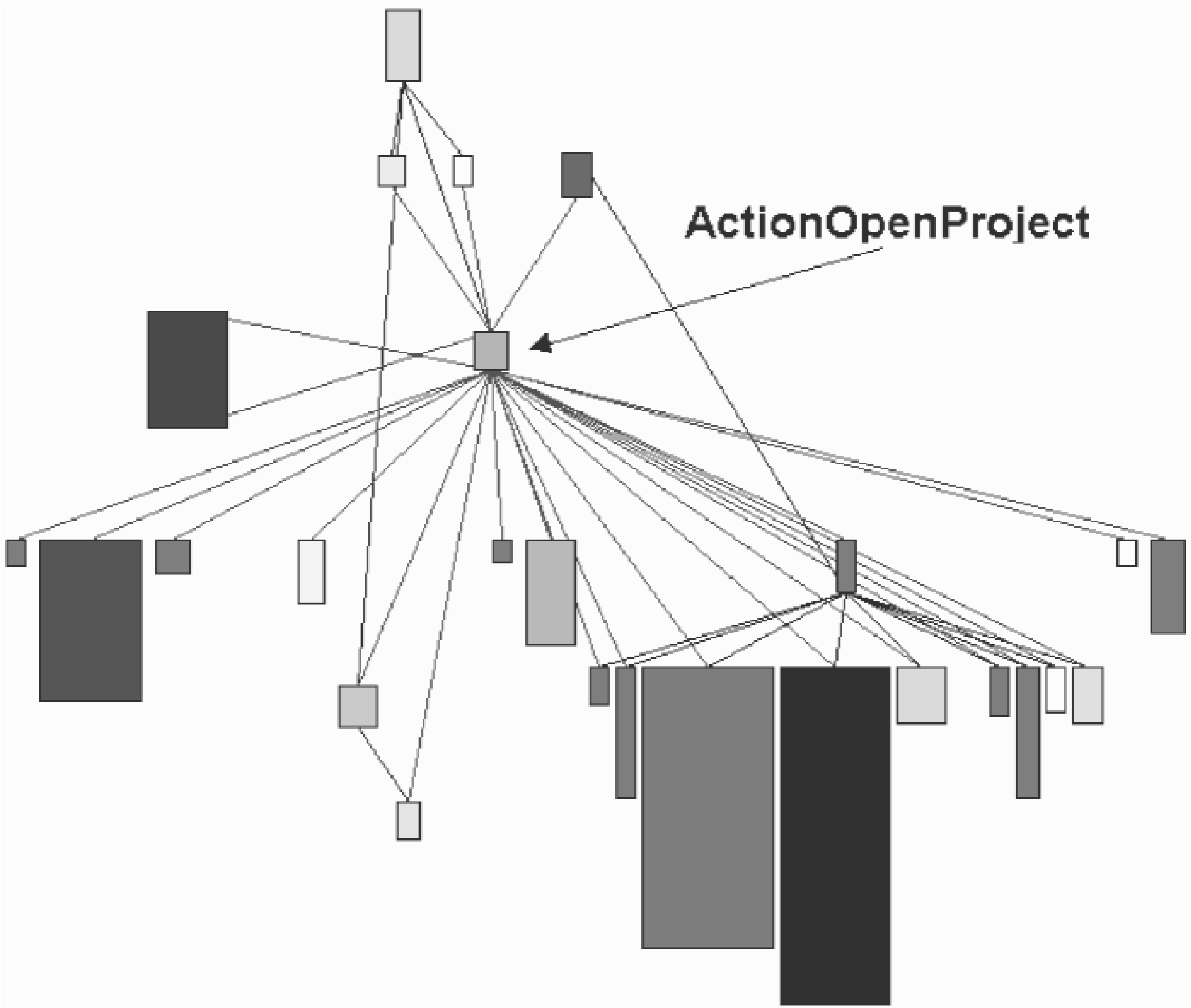
Operation calls a few methods from each of a large number of unrelated classes

Operation has few nested conditionals

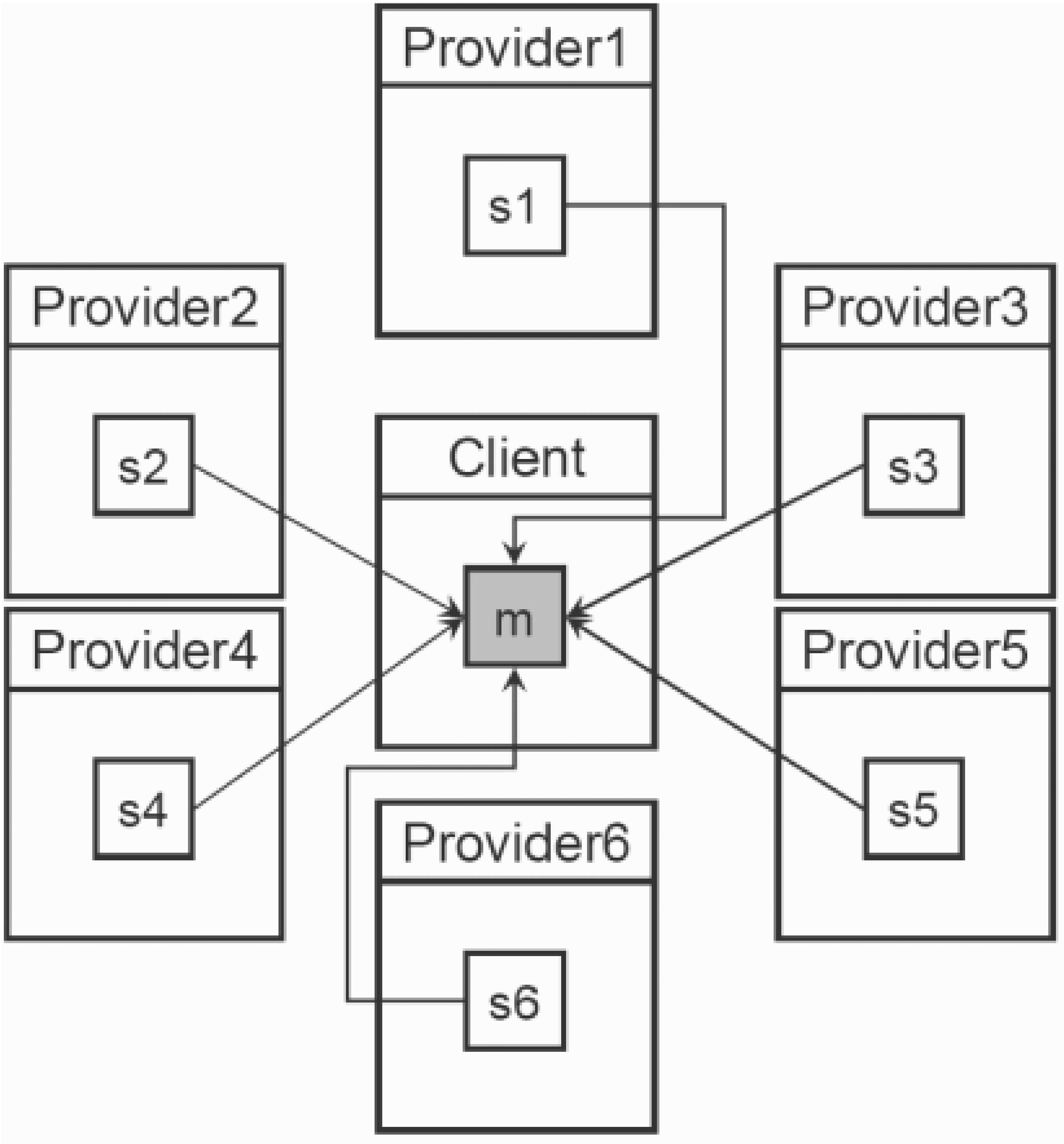
MAXNESTING > SHALLOW

Dispersed Coupling

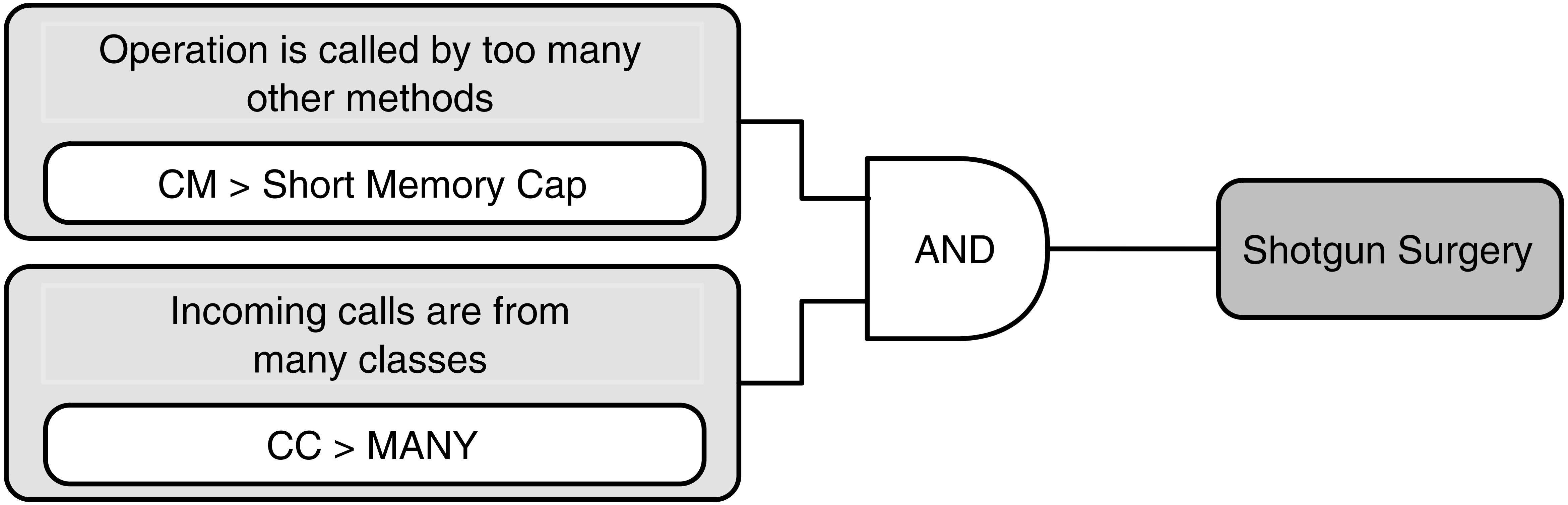
Dispersed Coupling: Example



Shotgun Surgery



Shotgun Surgery: Detection Strategy



Operation is called by too many other methods

CM > Short Memory Cap

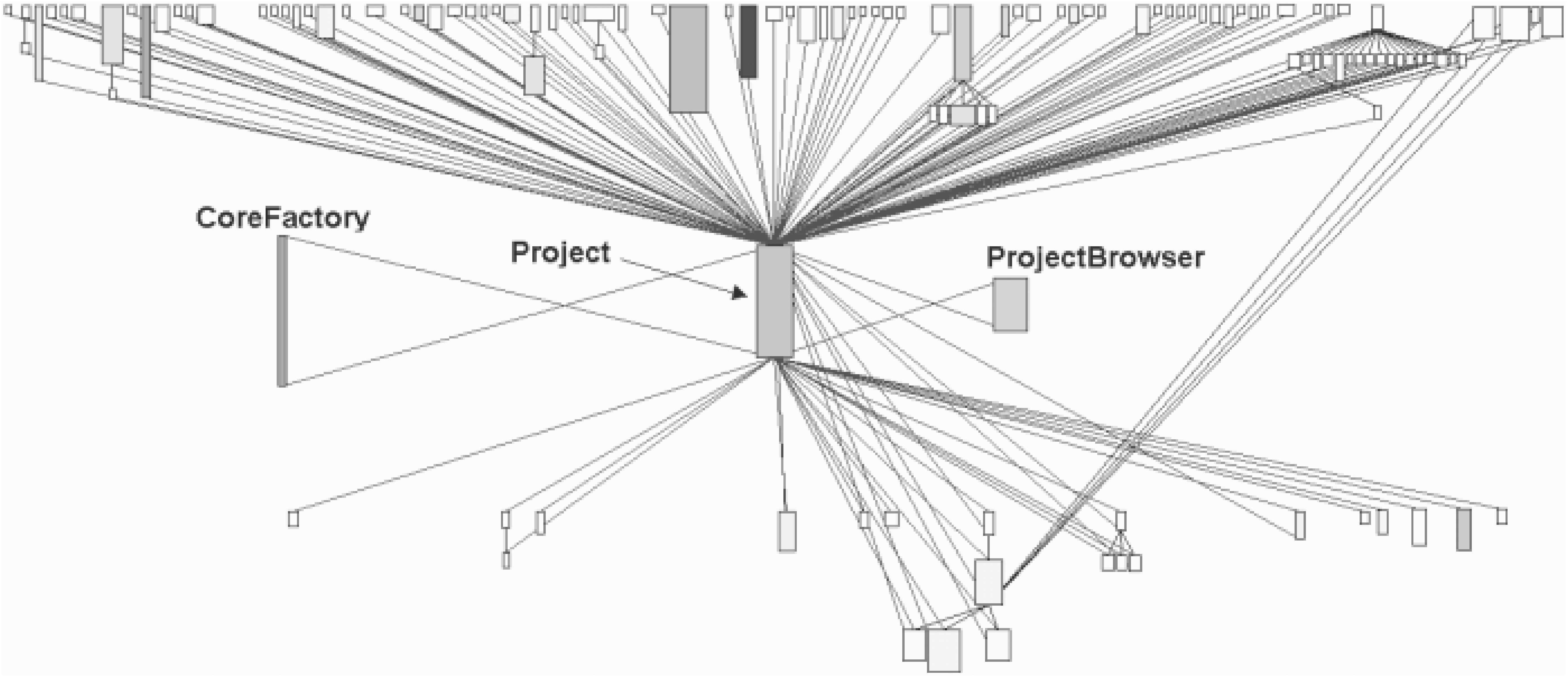
Incoming calls are from many classes

CC > MANY

AND

Shotgun Surgery

Shotgun Surgery: Example



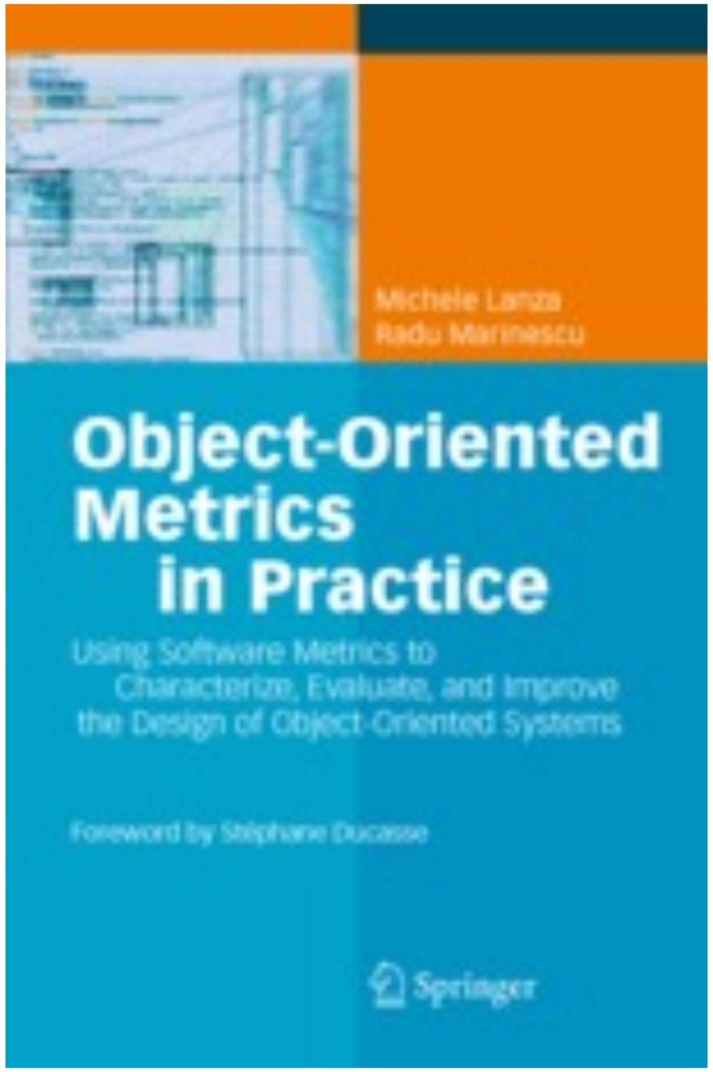
Tool to Detect Collaboration Disharmonies

inCode

- http://loose.upt.ro/incode/pmwiki.php/

More info on Detection Strategies

Object-Oriented Metrics in Practice
Michele Lanza and Radu Marinescu, Springer 2006
http://www.springer.com/computer/swe/book/978-3-540-24429-5



Michele Lanza
Radu Marinescu

Object-Oriented Metrics in Practice

Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems

Foreword by Stéphane Ducasse

Springer

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

Keep the design of a system as simple, clean, and expressive as possible

- Don’t allow broken windows
- Apply them in iterations (*not* to a big, up-front design)
- Sometimes you have to make trade-offs