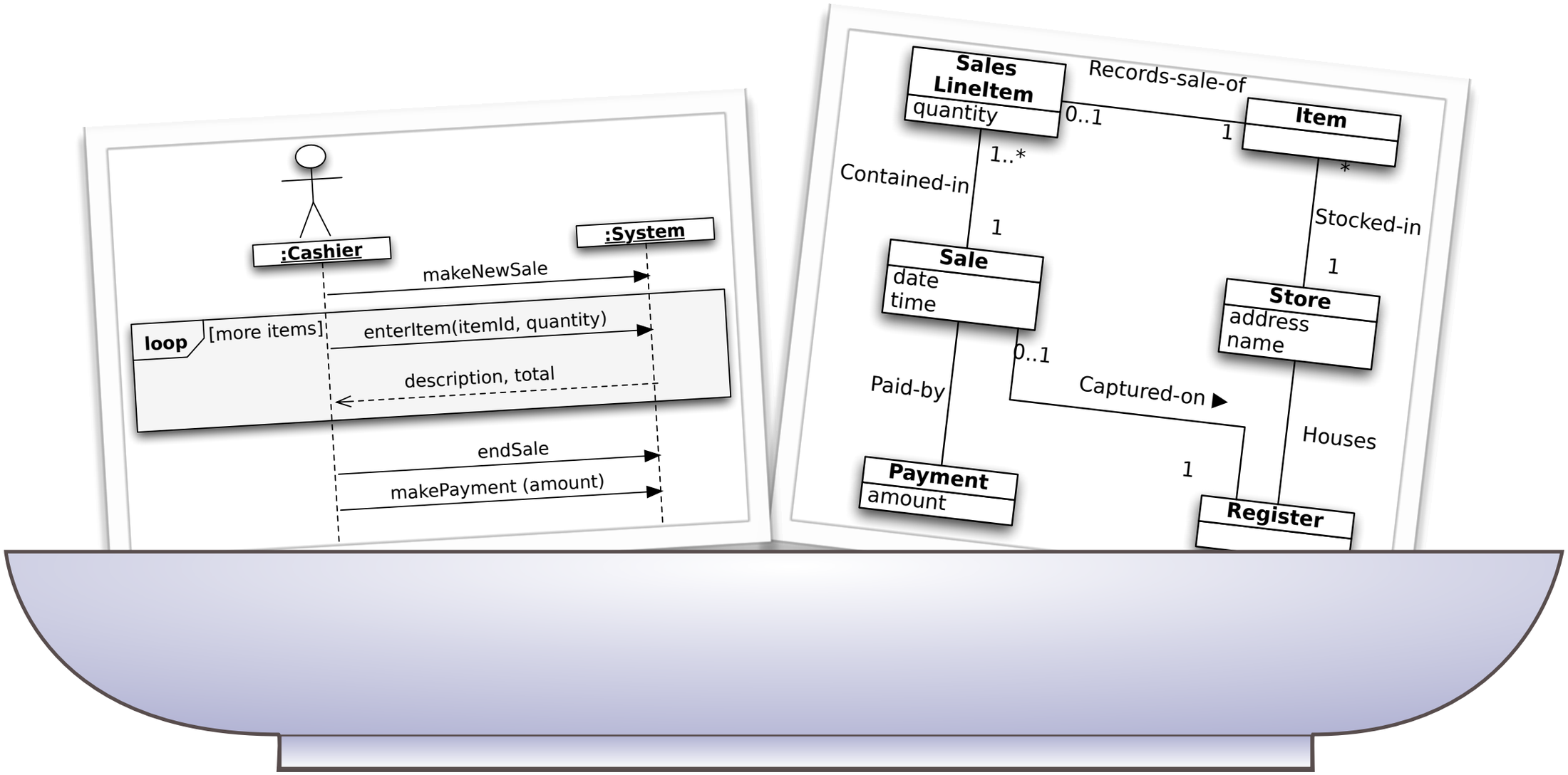
Dr. Michael Eichberg
Software Engineering
Department of Computer Science
Technische Universität Darmstadt

Software Engineering

On to Object-oriented Design

TECHNISCHE UNIVERSITÄT DARMSTADT

A popular way of thinking about the design of software objects and also large scale components is in terms of **responsibilities, roles and collaborations**.



Which class / object should have which responsibility?

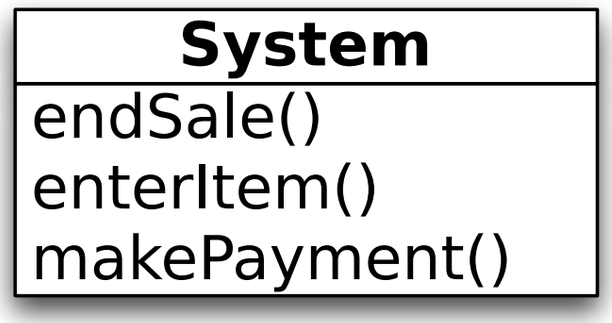
Object-oriented Design

- **Artifacts that are/can be used as input for the object-oriented design:**
  - a domain (analysis / conceptual) model
  - descriptions of use-cases (user stories) which are under development in the current iterative step

- **Next steps:**
  Build interaction diagrams for system operations of the use-cases at hand by applying guidelines and principles for assigning responsibilities

Responsibility for System Operations

- During system behavior analysis (e.g. of the POS system), system operations are assigned to a conceptual class (e.g. System)
  Does not necessarily imply that there will be a class System in the design.
- A **controller** class is assigned to perform the system operations



Responsibility for System Operations

- During system behavior analysis (e.g. of the POS system), system operations are assigned to a conceptual class (e.g. System)
  Does not nece in the design.
- A controller operations

Who should be responsible for handling system operations? What first object beyond the UI layer receives and coordinates a system operation?

System
endSale() enterItem() makePayment()

Responsibility for System Operations

- During system behavior analysis (e.g. of the POS system), system operations are assigned to a conceptual class (e.g. System)
  Does not imply the ... gn.
- A controller ... operations

Who should be responsible for handling system operations? What first object beyond the UI layer receives and coordinates a system ...tion?

The system operations become the starting messages entering the controllers for domain layer interaction diagrams.

endSale()
enterItem()
makePayment()

Interaction Diagrams for System Operations

- Create a separate diagram for each system operation in the current development cycle
- Use the system operation, e.g., enterItem(), as starting message
- If a diagram gets complex, split it into smaller diagrams
- Distribute responsibilities among classes:
  - from the conceptual model and may be others added during object design
    The classes will collaborate for performing the system operation.
  - based on the description of the behavior of system operations

Foundations of Object-oriented Design

TECHNISCHE UNIVERSITÄT DARMSTADT

Responsibility

R. Martin

Each responsibility is an axis of change.

When the requirements change, a change will manifest through a change in responsibility amongst the classes.

If a class has multiple responsibilities, it has multiple reasons to change.

Assigning **Responsibility** to classes is one of the most important activities during the design. Patterns, idioms, principles etc. help in assigning the responsibilities.

In Responsibility-driven Design (RDD) we think of software objects as having responsibilities.

The responsibilities are assigned to classes of objects during object-design.

How does one determine the assignment of responsibilities to various objects?

How does one determine the assignment of responsibilities to various objects?

There is a great variability in responsibility assignment :

- Hence, "good" and "poor" designs, "beautiful" and "ugly" designs, "efficient" and "inefficient" designs.
- Poor choices lead to systems which are fragile and hard to maintain, understand, reuse, or extend!

Coupling

Coupling measures the strength of dependence between classes and packages.

- Class C1 is coupled to class C2 if C1 requires C2 directly or indirectly.
- A class that depends on 2 other classes has a lower coupling than a class that depends on 8 other classes.

Coupling is an *evaluative principle*!

Common Forms of Coupling in Java

- Type X has an attribute that refers to a type Y instance or type Y itself

```
class X{ private Y y = …}
class X{ private Object o = new Y(); }
```

- A type X object calls methods of a type Y object

```
class Y{f(){;}}
class X{ X(){new Y.f();}}
```

- Type X has a method that references an instance of type Y
  (E.g. by means of a parameter, local variable, return type,…)

```
class Y{}
class X{ X(y Y){…}}
class X{ Y f(){…}}
class X{ void f(){Object y = new Y();}}
```

- Type X is a subtype of type Y

```
class Y{}
class X extends Y{}
```

- …

Coupling in Java - Exemplified

Class `QuitAction` is coupled with:

- …`ActionListener`
- …`ActionEvent`
- `java.lang.Override`
- `java.lang.System`
- `java.lang.Object`

```
package de.tud.simpletexteditor;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class QuitAction implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
}
```

Example Source Code

Coupling

- **High Coupling**

  A class with high coupling is undesirable, because...

  - changes in related classes may force local changes
  - harder to understand in isolation
  - harder to reuse because its use requires the inclusion of all classes it is dependent upon
  - ...

Coupling

- ...

- **Low Coupling**
  Low coupling supports design of relatively independent, hence more reusable, classes.
  - Generic classes, with high probability for reuse, should have especially low coupling
  - Very little or no coupling at all is also not desirable
  - Central metaphor of OO: a system of connected objects that communicate via messages
  - Low coupling taken to excess results in active objects that do all the work

Coupling

- ...

- **Low Coupling**
  Low coupling supports design of relatively independent, hence more reusable, classes.
  - Generic classes, with high probability for reuse, should have especially low
  - Very little or n
  - Central metap
  - Low coupling

High coupling to stable elements and to pervasive elements is seldom a problem.

objects that communicate

results in active objects that do all the work

Coupling

- ...

- **Low Coupling**
  Low coupling supports design of relatively independent, hence more reusable, classes.
  - Generic classe… high probability for reuse, should have especially low…
  - Very little or n…
  - Central metap… objects that communicate…
  - Low coupling… cts that do all the work

Beware: the quest for low coupling to achieve reusability in a future (mythical!) project may lead to needless complexity and increased project cost.

Different kinds of Coupling

- Coupling to “stable classes” (e.g., the JDK) is typically no problem
  - they don’t change
  - they don’t hamper reuse
- Coupling related to interface is better than coupling related to classes
  - testability is improved
  - reuse is facilitated
  - they are (often) more stable

Cohesion

Cohesion measures the strength of the relationship amongst elements of a class.

All operations and data within a class should "naturally belong" to the concept that the class models.

Cohesion is an evaluative principle!

Cohesion in Java - Exemplified

Analysis of the cohesion of `SimpleLinkedList`

- the constructor uses both fields
- head uses only the field `value`
- tail uses only `next`
- `head` and `tail` are simple getters; they do not mutate the state

```
public class SimpleLinkedList {

    private final Object value;
    private final SimpleLinkedList next;

    public SimpleLinkedList(
            Object value, SimpleLinkedList next
    ) {
        this.value = value; this.next = next;
    }

    public Object head() {
        return value;
    }

    public SimpleLinkedList tail() {
        return next;
    }
}
```

Example Source Code

Cohesion in Java - Exemplified

Analysis of the cohesion of `ColorableFigure`

- `lineColor` is used only by its getter and setter
- `fillColor` is used only by its getter and setter
- `lineColor` and `fillColor` have no interdependency

```
import java.awt.Color;

abstract class ColorableFigure implements Figure {

    private Color lineColor = Color.BLACK;
    private Color fillColor = Color.BLACK;

    public Color getLineColor() { return lineColor; }
    public void setLineColor(Color c) {
        lineColor = c;
    }

    public Color getFillColor() { return fillColor; }
    public void setFillColor(Color c) {
        this.fillColor = c;
    }

}
```

Example Source Code

Types of Cohesion

- Coincidental
  No meaningful relationship amongst elements of a class.
- Logical cohesion (functional cohesion)
  Elements of a class perform one kind of a logical function.
  E.g., interfacing with the POST hardware.
- Temporal cohesion
  All elements of a class are executed “together”.

Responsibility

To keep design complexity manageable, assign responsibilities while maintaining high cohesion.

Cohesion

Low Cohesion

- Classes with low cohesion are undesirable, because they are...
  - hard to comprehend,
  - hard to reuse,
  - hard to maintain - easily affected by change
  - ...

Classes with high cohesion can often be described by a simple sentence.

Low Cohesion

- Classes with low cohesion...
  - often represent a very large-grain abstraction
  - have taken responsibility that should have been delegated to other objects

Classes with high cohesion can often be described by a simple sentence.

Measuring Cohesion

Lack of Cohesion in Methods (LCOM) measures the disparateness among methods in a class.

- a high value indicates “poor” cohesion
- a low value indicates “high” cohesiveness

(Multiple definitions exist!)

Container classes generally exhibit low cohesion.

Measuring Cohesion

Li, W., & Henry, S., Maintenance metrics for the object oriented paradigm, 1993, IEEE

LCOM = Number of disjoint sets of local methods.

Each set has one or more local methods of the class, and any two methods in the set access at least one attribute of the class in common; the number of common attributes ranging from 0 to N (where N is a positive integer)

Variants exist that (do not) consider inheritance and constructors.

Measuring Cohesion

B. Henderson-sellers, Object-oriented metrics: measures of complexity, 1996

Let:

$M_i(i = 1,...,m)$ be the set of methods defined by the class

$A_j(j = 1,...,a)$ be the set of attributes defined by the class

$\mu(A_j)$ be the number of methods that access the attribute

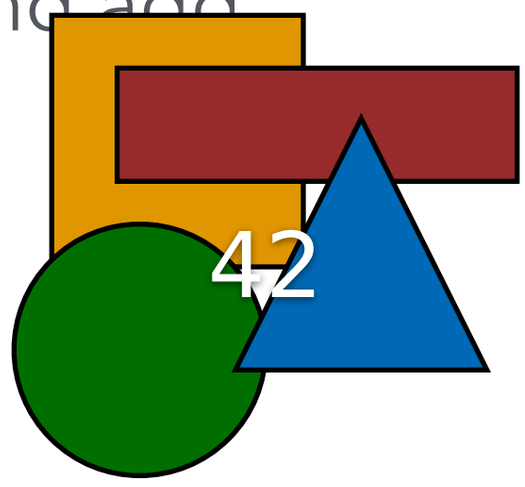
Then:

$$\text{LCOM}^* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1 - m}$$

Design needs principles.

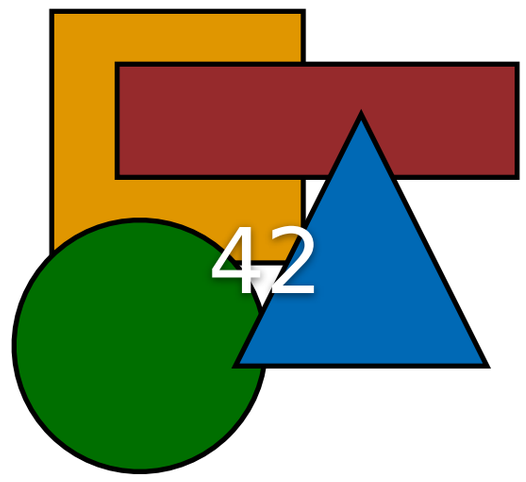
A common pitfall in object-oriented design is the inheritance relation.

- Let’s assume that we want to extend our library for vector graphic applications and our library already defines classes for Circles and Squares.
- Let’s assume we want to further evolve our library and add support for Rectangles…



A common pitfall in object-oriented design is the inheritance relation.

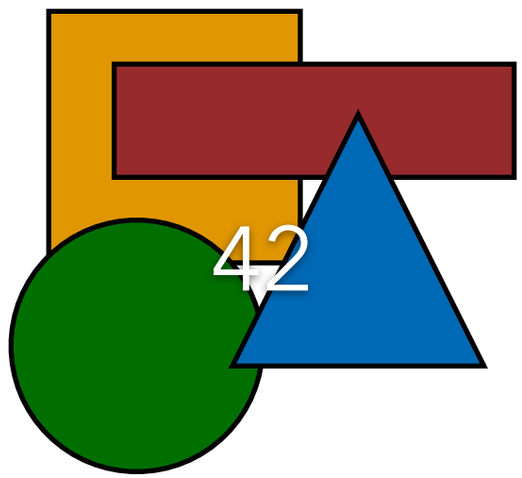
- Now let's assume we want to further evolve our library and add support for Rectangles…
- Should Rectangle inherit from Square?
- Should Square inherit from Rectangle?
- Is there some other solution?



A common pitfall in object-oriented design is the inheritance relation.

- Now let's assume we want to further evolve our library and add support for Rectangles…
- Should Rectangle inherit from Square?
- Should Square
- Is there some

A first test:
"Is a Rectangle a Square?"

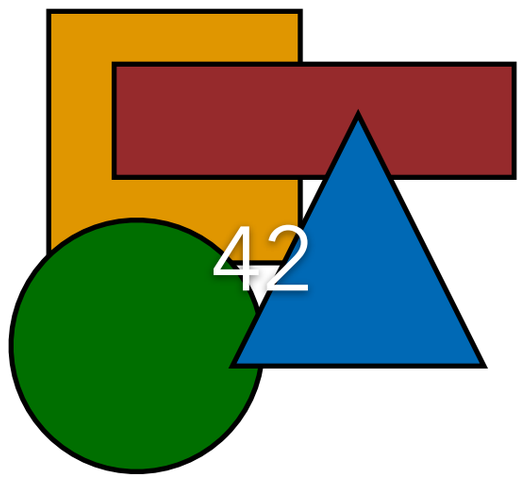


A common pitfall in object-oriented design is the inheritance relation.

- Now let's assume we want to further evolve our library and add support for Rectangles…
- ~~Should Rectangle inherit from Square?~~
- Should Square
- Is there some

A first test:
"Is a Rectangle a Square?"

No.

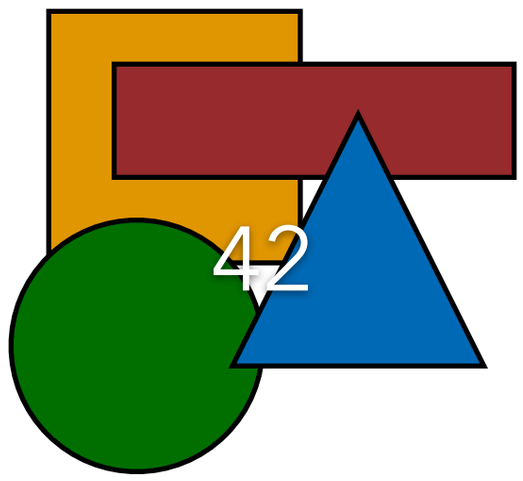


A common pitfall in object-oriented design is the inheritance relation.

- Now let's assume we want to further evolve our library and add support for Rectangles…
- ~~Should Rectangle inherit from Square?~~
- Should Square inherit from Rectangle?
- Is there some

A first test:
"Is a Square a Rectangle"?

Well… yes, but … how about a Square's behavior?

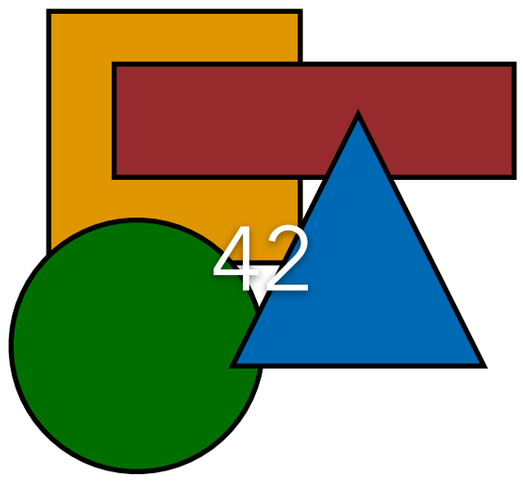


A common pitfall in object-oriented design is the inheritance relation.

- Now let's assume we want to further evolve our library and add support for Rectangles…
- ~~Should Rectangle inherit from Square?~~
- ~~Should Square inherit from Rectangle?~~
- Is there some

A first test:
"Is a Square a Rectangle"?

Well… yes, but … how about a Square's behavior?



A large number of Design Heuristics and Design Principles exists that help you to design “better” programs.

- Low Coupling
- High Cohesion
- Single Responsibility Principle
- Don’t repeat yourself
- No cyclic dependencies
- Liskov Substitution Principle
- Open-Closed Principle
- …

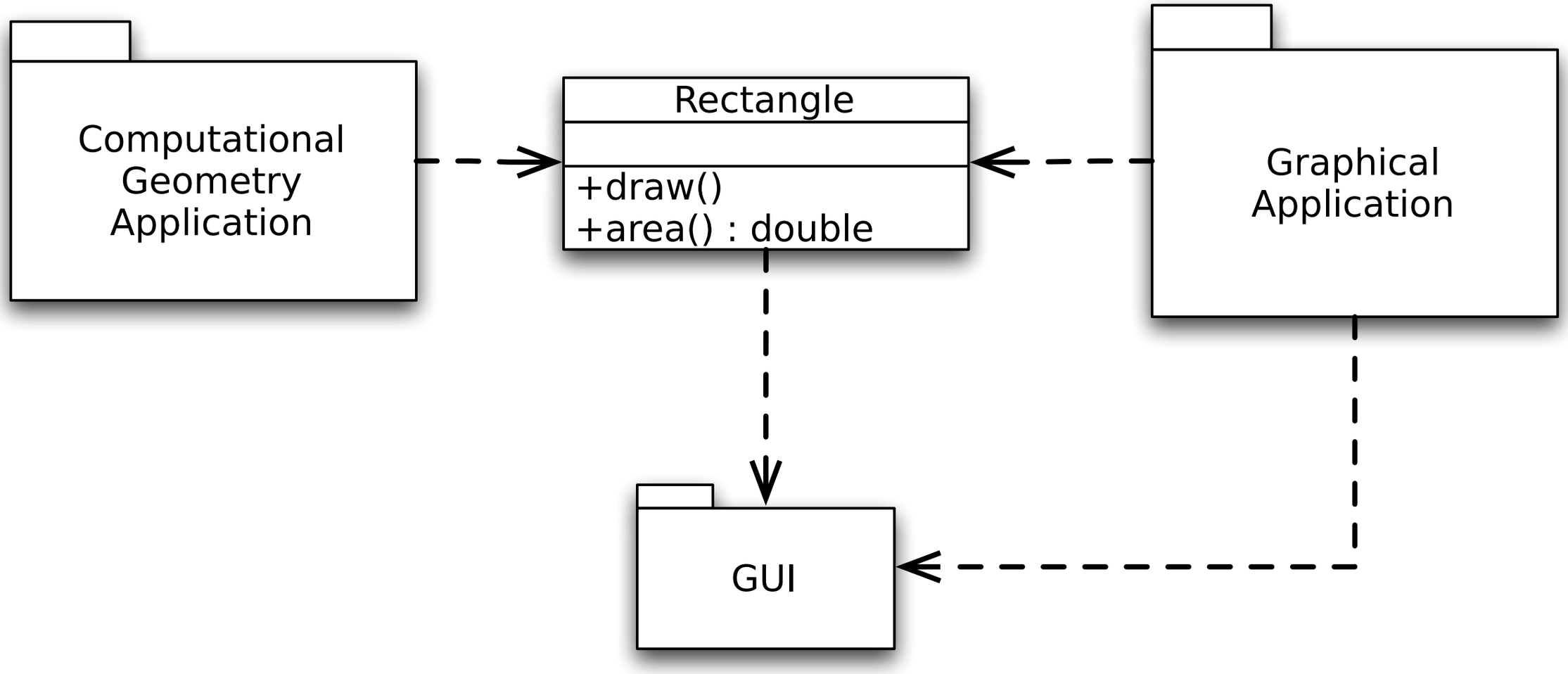
> "A class should have only one reason to change."
>
> I.e. a responsibility is primarily a reason for change.

Agile Software Development; Robert C. Martin; Prentice Hall, 2003

The Single Responsibility Principle

Example: a Rectangle Class

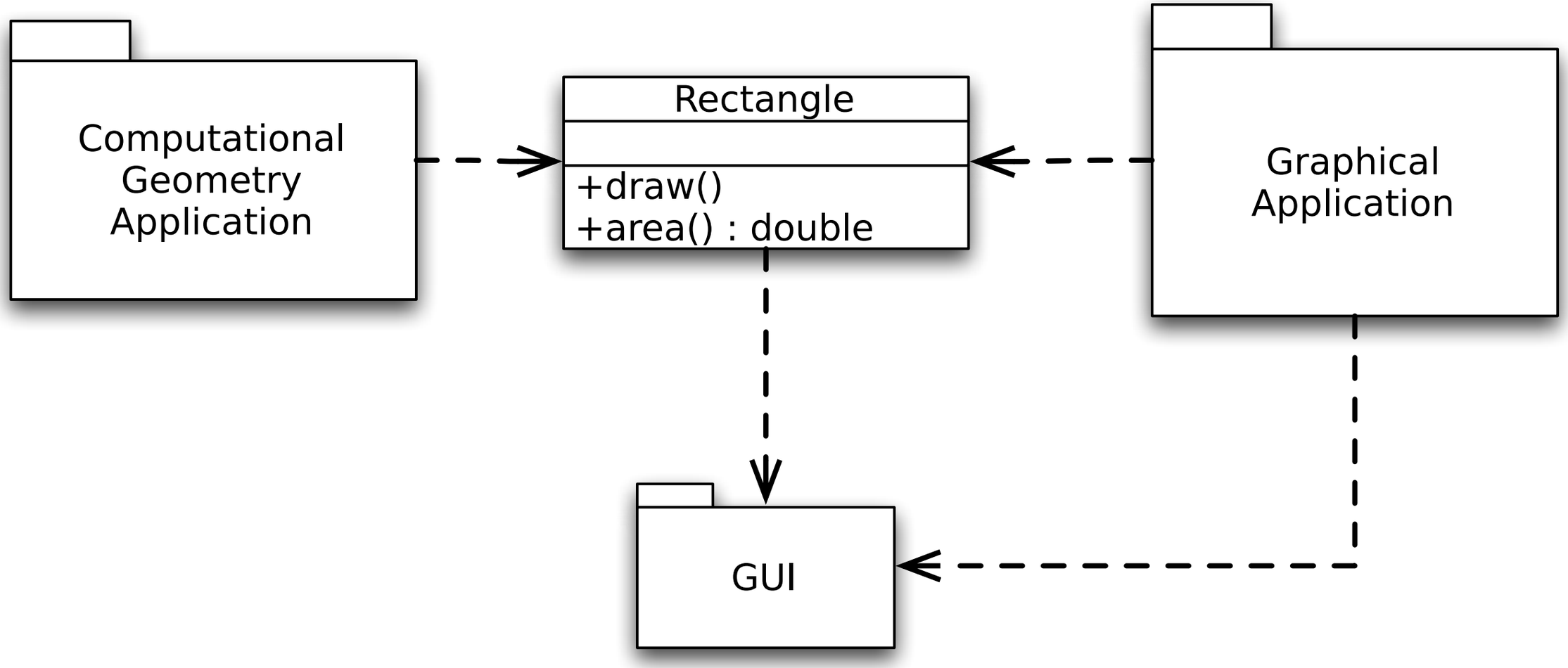
The Single Responsibility Principle



Does the Rectangle class have a single responsibility or does it have multiple responsibilities?

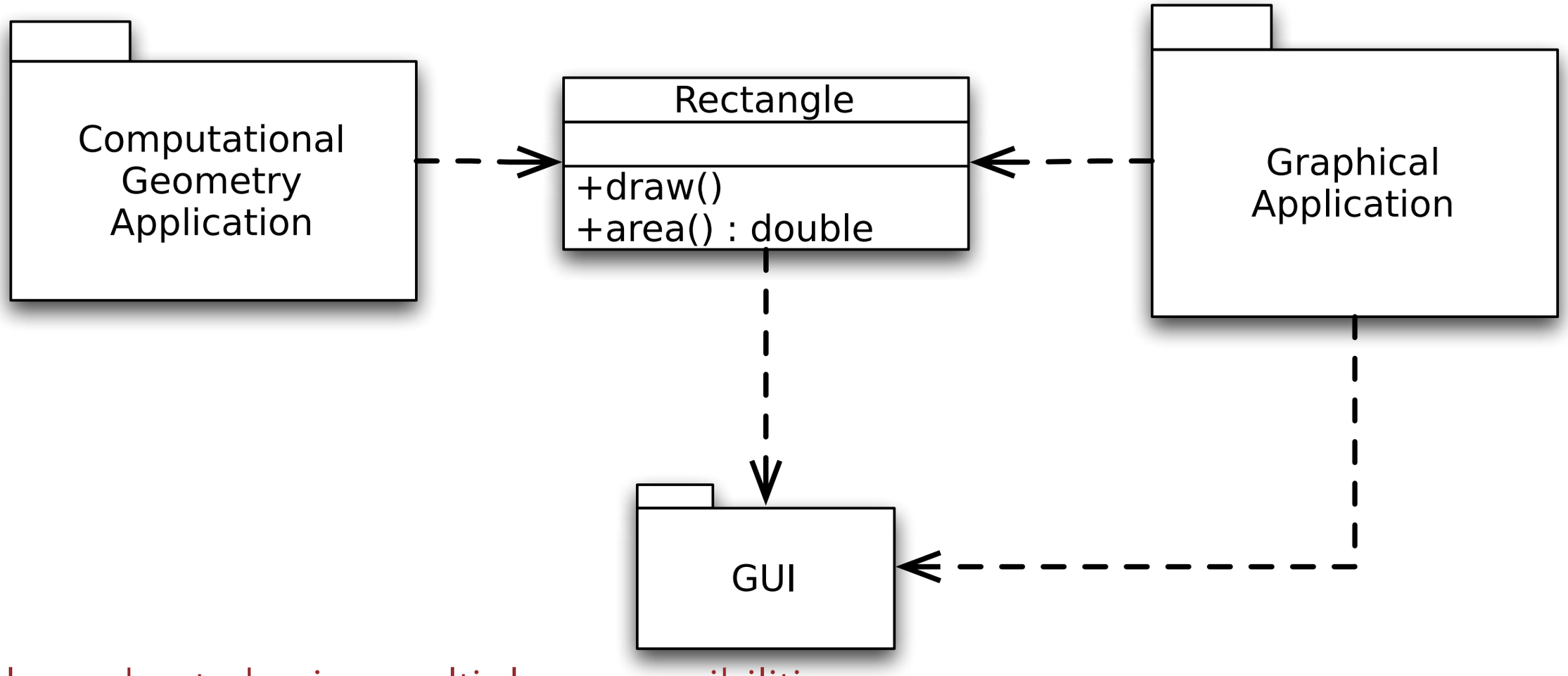
Example: a Rectangle Class

- The Rectangle class has multiple responsibilities:
- Calculating the size of a rectangle; a mathematical model
- To render a rectangle on the screen; a GUI related functionality
- Do you see any problems?



Example: a Rectangle Class

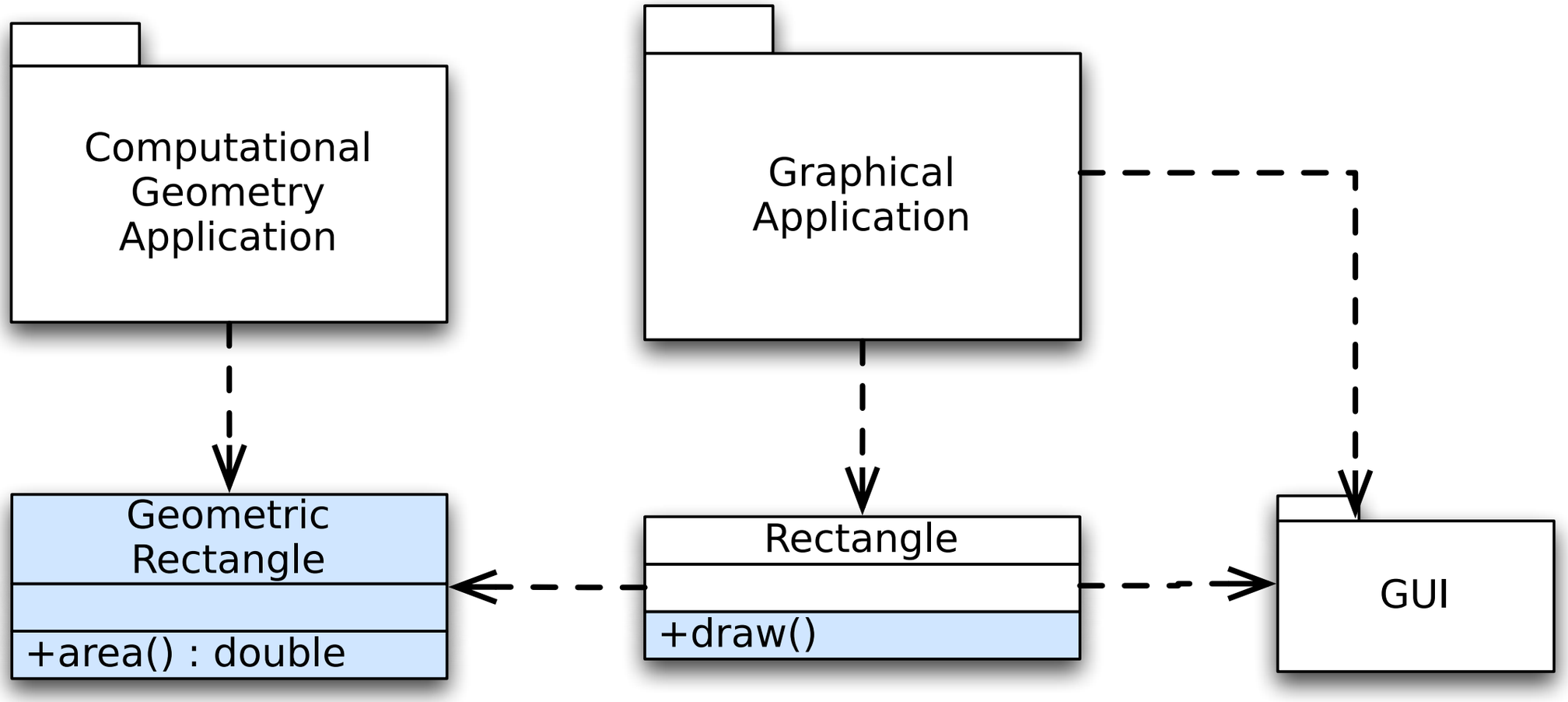
The Single Responsibility Principle



Problems due to having multiple responsibilities:

- Reuse of the Rectangle class (e.g. in a math package) is hindered due to the dependency on the GUI package
  (GUI classes have to be deployed along with the Rectangle class)
- A change in the Graphical Application that results in a change of Rectangle requires that we retest and redeploy the Rectangle class in the context of the Computational Geometry Application

Example: Rectangle classes with single responsibilities

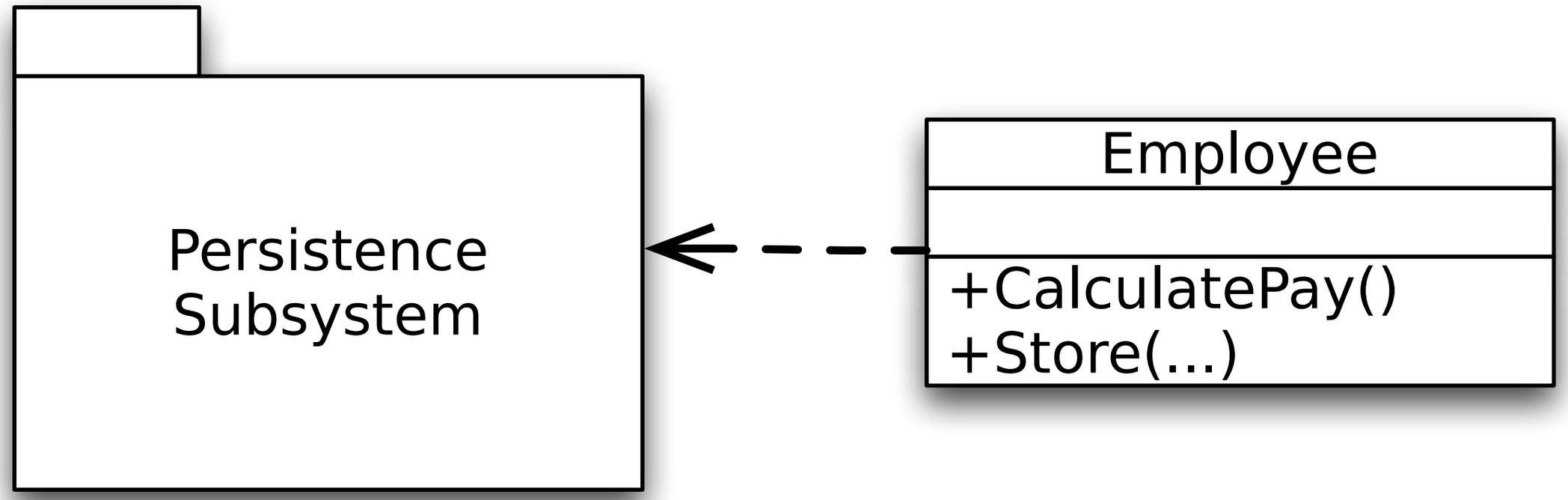


The solution is to separate the functionality for drawing a rectangle and the functionality for doing calculations are separated.

Coupling? Cohesion?

Example: Handling Persistence

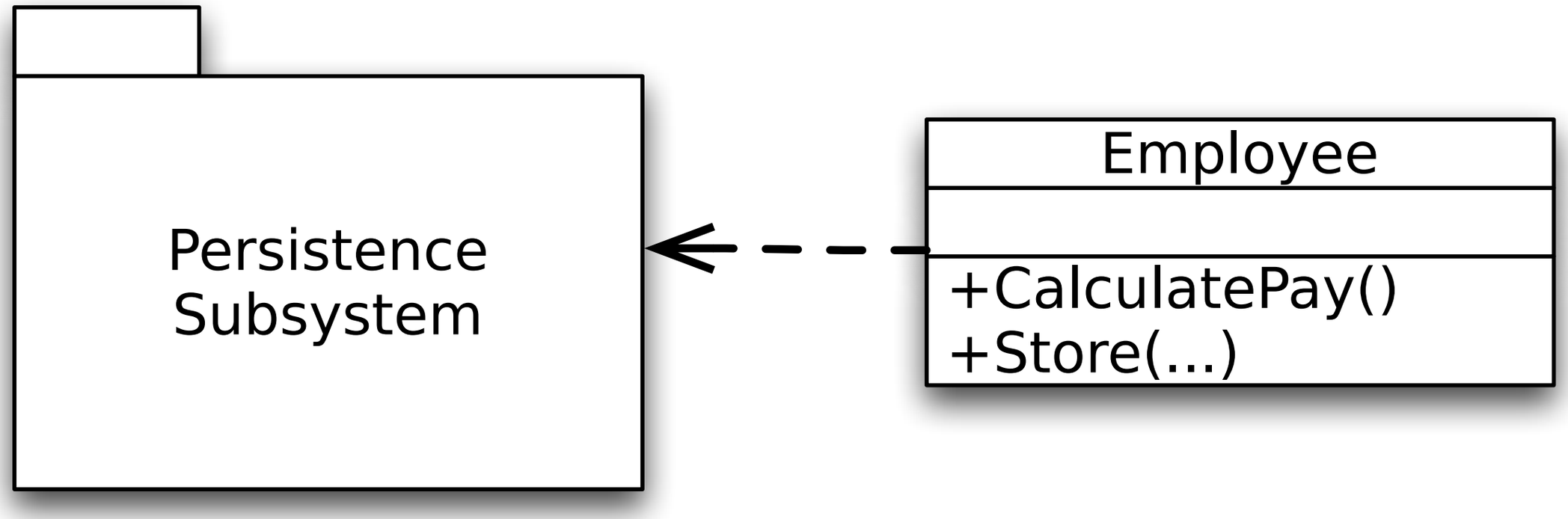
The functionality for drawing a rectangle and the functionality for doing calculations are separated.



Do we need to change the Employee class?

Example: Handling Persistence

The functionality for drawing a rectangle and the functionality for doing calculations are separated.



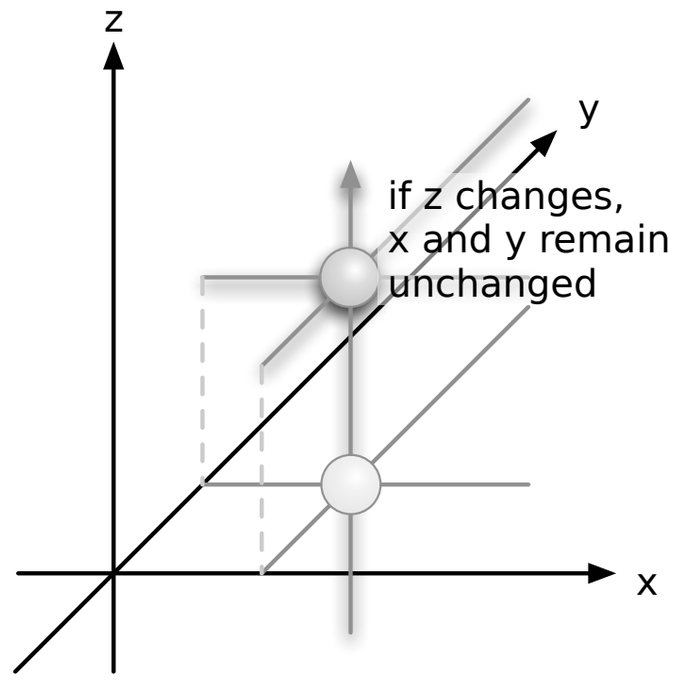
Two responsibilities:

- Business functionality
- Persistence related functionality

Do we need to change the Employee class?

Orthogonality

Two or more things are orthogonal if changes in one do not affect any of the others; e.g. if a change to the database code does not affect your GUI code, both are said to be orthogonal.



Andrew Hunt and David Thomas; The Pragmatic Programmer;

Addison-Wesley, 2000

Design Heuristics

- J. Riel; Object-Oriented Design Heuristics; Addison-Wesley, 1996

Design Heuristics

- Design Heuristics help to answer the question:
  "Is it good, bad, or somewhere in between?"
- Object-Oriented Design Heuristics *offer insights into object-oriented design improvement*
- The following *guidelines are language-independent* and allow to rate the integrity of a software design
- *Heuristics are not hard and fast rules*; they are meant to serve as warning mechanisms which allows the flexibility of ignoring the heuristic as necessary
- *Many heuristics are small tweakings on a design* and are local in nature

  A single violation rarely causes major ramifications on the entire application.

Two areas where the object-oriented paradigm can drive design in dangerous directions...

- ...poorly distributed systems intelligence

  The God Class Problem

- ...creation of too many classes for the size of the design problem

  Proliferation of Classes

  (Proliferation =dt. starke Vermehrung)

A Very Basic Heuristic

All data in a base class should be private;
do not use non-private data.
Define (protected) accessor methods instead.

If you violate this heuristic your design tends to be more fragile.

A Very Basic Heuristic

All data in a base class should be private;
do not use non-private data.
Define (protected) accessor methods instead.

```
public class Line {
   // a "very smart developer" decided:
   // p and v are package visible to enable efficient access
   /*package visible*/ Point p;
   /*package visible*/ Vector v;
   public boolean intersects(Line l) {…}
   public boolean contains(Point p) {…}
}
```

Implementation of a `Line` class as part of a math library.

```
Line l1 = …;
Line l2 = …;

if (l1.v.equals(l2.v)) {…}
```

Some code in the same package that uses `Line` objects.

A Very Basic Heuristic

All data in a base class should be private;
do not use non-private data.
Define (protected) accessor methods instead.

```
public class Line {
   /*package visible*/ Point p1;
   /*package visible*/ Point p2;
   public boolean intersects(Line l) {…}
   public boolean contains(Point p) {…}
}
```

Now, assume the following change to the implementation of Line

The public interface remains stable - just implementation details are changed.

```
Line l1 = …;
Line l2 = …;
// check if both lines are parallel
if (l1.v.equals(l2.v)) {…}
```

The change breaks our code.

A Very Basic Heuristic

All data in a base class should be private;
do not use non-private data.
Define (protected) accessor methods instead.

```
public class Line {
   private Point p;
   private Vector v;
   public boolean intersects(Line l) {…}
   public boolean contains(Point p) {…}
   protected Vector getVector() { return v; };
}
```

"Better design."

```
Line l1 = …;
Line l2 = …;
// check if both lines are parallel
if (l1.getVector().equals(l2.getVector())) {…}
```

Some code in the same package that uses Line objects.

The God Class Problem

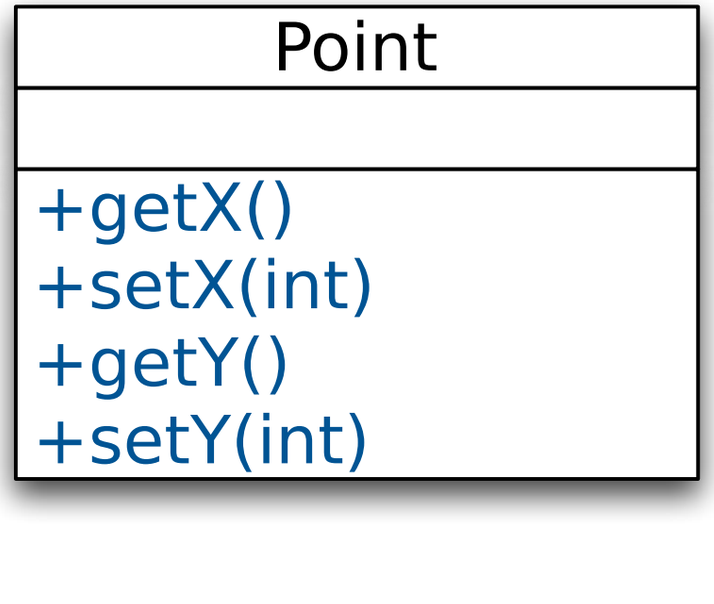
Distribute system intelligence as uniformly as possible, that is, the top-level classes in a design should share the work uniformly.

Beware of classes that have many accessor methods defined in their public interface. Having many implies that related data and behavior are not kept in one place.

Beware of classes that have too much noncommunicating behavior, that is, methods that operate on a proper subset of the data members of a class.

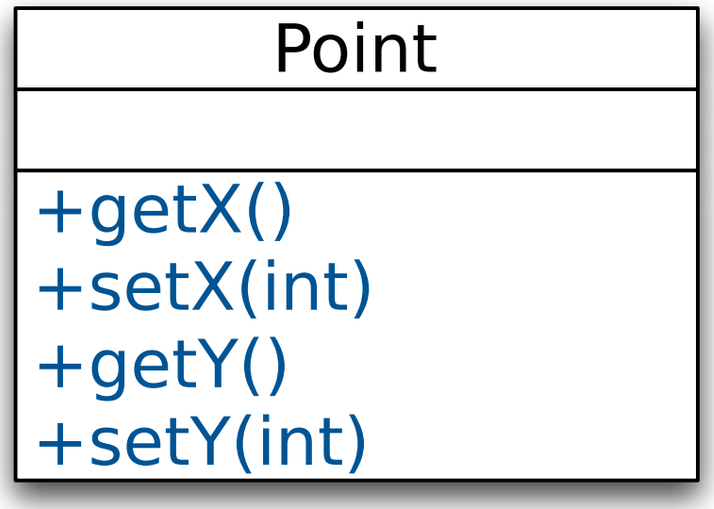
God classes often exhibit much noncommunicating behavior.

The Problem of Accessor Methods



- The class Point has accessor operations in the public interface. Are there any problems with this design of Point, you can think of?
- Is Point eventually giving too much implementation details away to clients?

The Problem of Accessor Methods

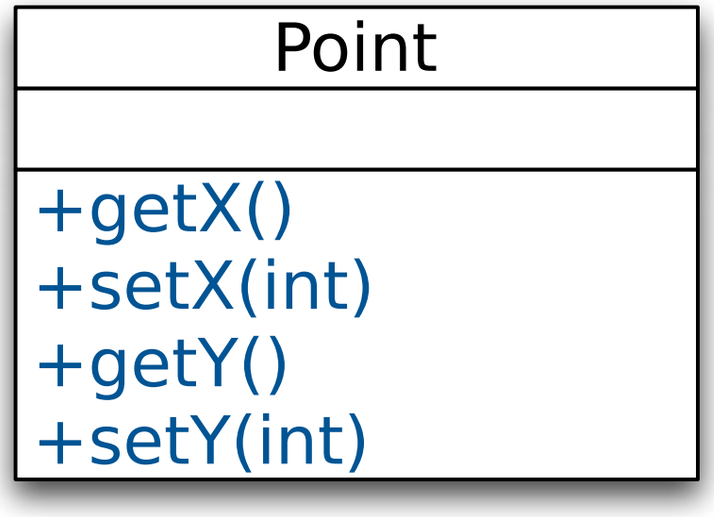


- The class Point has accessor operations in the public interface.
  Are there any problems with this design of Point, you can think of?
- Is Point eventually giving too much implementation details away to clients?

The answer to this question is:

"No, accessor methods do not necessarily expose implementation details."

The Problem of Accessor Methods



- Accessor methods indicate poor encapsulation of related data and behavior; someone is getting the x- and y-values of Point objects to do something with them – *executing behavior that is related to points* – that the class Point is not providing
- Often the client that is using accessor methods is a god class capturing centralized control that requires data from the mindless Point object

The Problem of Accessor Methods


```
public class Line {
   private Point p;
   private Vector v;
   public boolean intersects(Line l) {…}
   public boolean contains(Point p) {…}
   protected Vector getVector() {return v;};
   public boolean isParallel(Line l) {…};
}
```

```
Line l1 = …;
Line l2 = …;
// check if both lines are parallel
if (l1.isParallel(l2)) {…}
```

Reconsider the `Line` class.

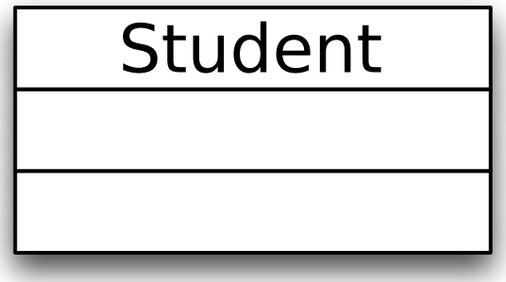
Some code in the same package that uses `Line` objects.

Two Reasonable Explanations For the Need of Accessor Methods...

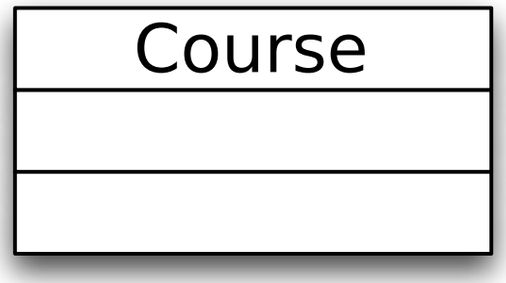
- … a class performing the gets and sets is implementing a policy
  (policy = dt. Verfahren(-sweise))
- … or it is in the interface portion of a system consisting of an object-oriented model and a user interface
  (The UI layer needs to be able to get the data to visualize it.)

Implementing Policies Between Two or More Classes

Example from the Course-scheduling Domain



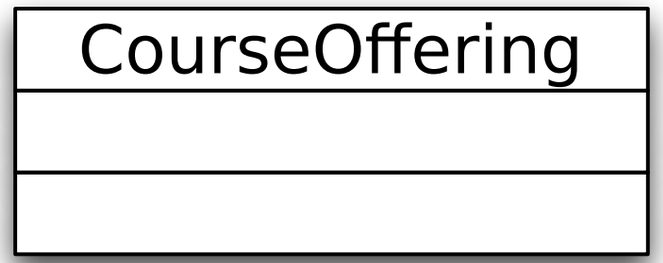
Captures **static information about students**, e.g., name, identification number, list of courses (s)he has taken, etc.



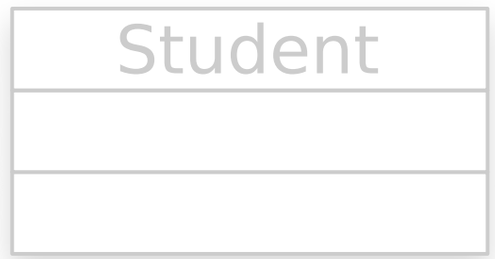
Captures **static information about the course objects**, e.g., the course number, description, duration, minimum and maximum number of students, list of prerequisites, etc.

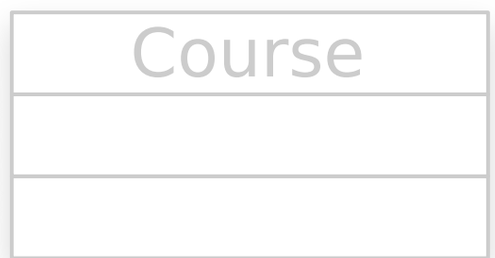
Implementing Policies Between Two or More Classes

Example from the Course-scheduling Domain



Captures **static and dynamic information related to a particular section of a given course**, e.g., the course being offered, the room and schedule, instructor, list of attendees, etc.

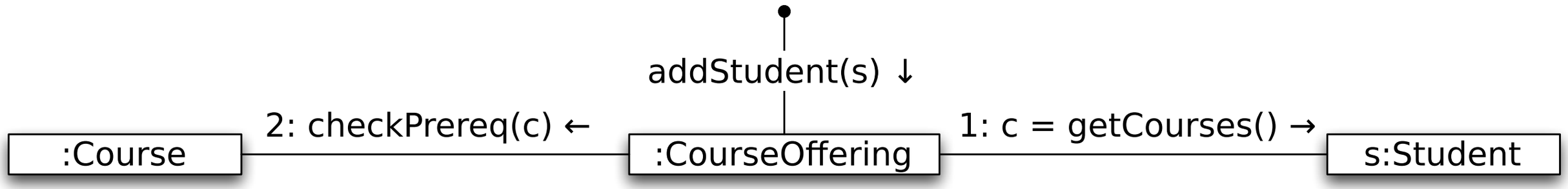




Implementing Policies Between Two or More Classes

(here: `addStudents`)

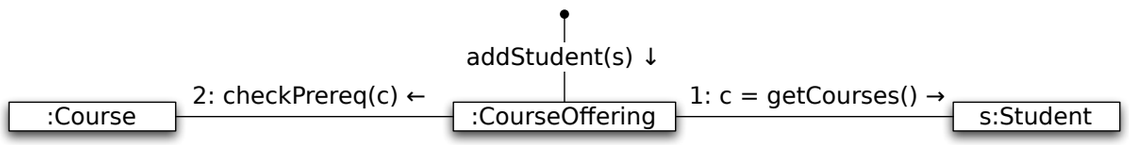
Example from the Course-scheduling Domain

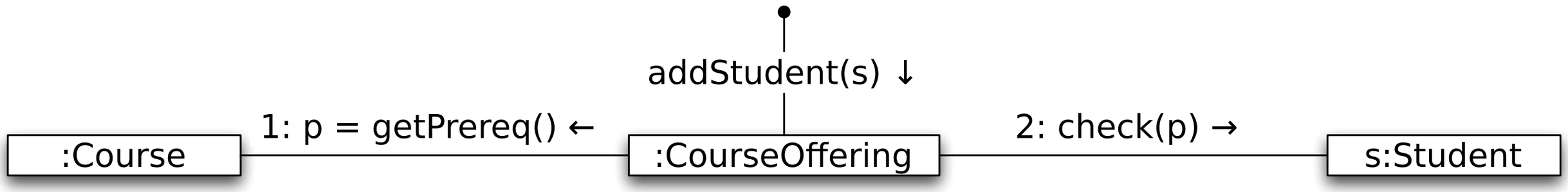


First design for checking the prerequisites of students

Implementing Policies Between Two or More Classes

Example from the Course-scheduling Domain

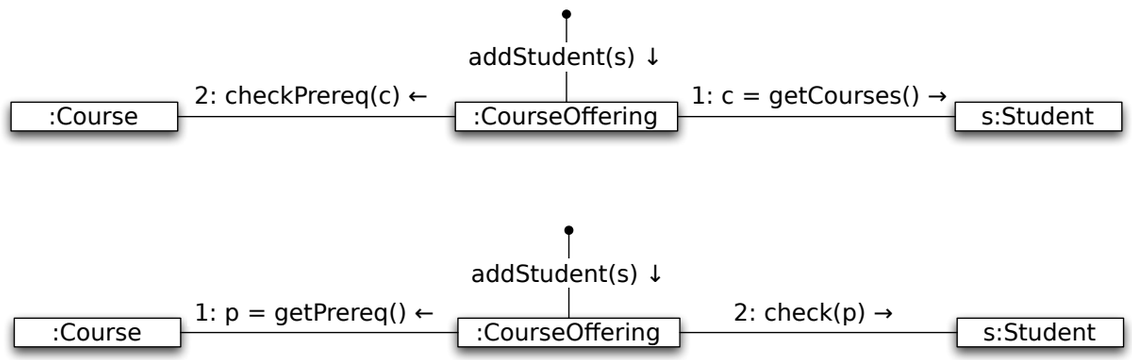


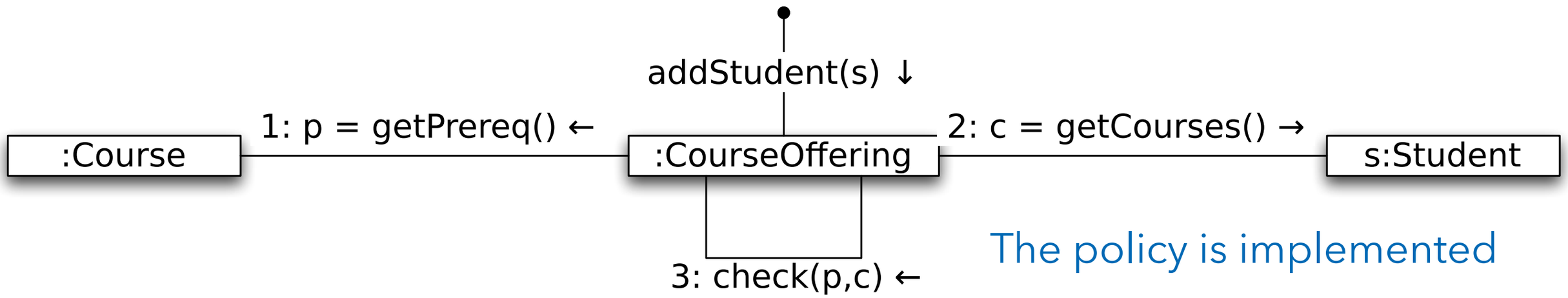


Second design for checking the prerequisites of students

Implementing Policies Between Two or More Classes

Example from the Course-scheduling Domain



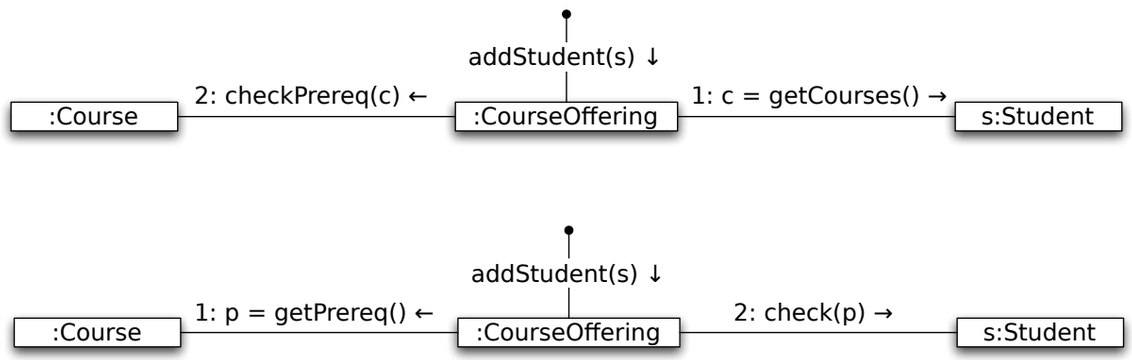


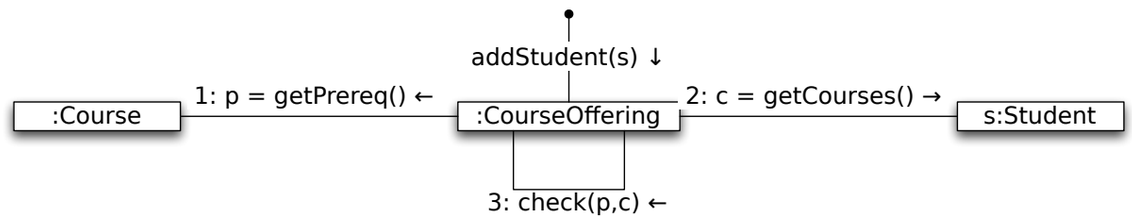
The policy is implemented by course offering.

Third design for checking the prerequisites of students

Implementing Policies Between Two or More Classes.

Example from the Course-scheduling Domain





- What do you think of these three designs?
  (Discuss the pros and cons - regarding the implementation of the policy - with your fellow students.)

The God Class Problem - Behavioral Form Summary

- In general, always try to model the real world
  (Low representational gap facilitates maintenance and evolution.)
  But modeling the real world is not as important as the other heuristics.
  (E.g., in the real world a room does not exhibit any behavior, but for a heating system it is imaginable to assign the responsibility for heating up or cooling down a room to a corresponding class.)
- Basically, a god class is a class that does too much
  (Behavioral Form)
- By systematically applying the principles that we have studied previously, the creation of god classes becomes less likely

Classes That Model the Roles an Object Plays

Be sure that the abstractions that you model are classes and not simply the roles objects play.

Classes That Model the Roles an Object Plays

Variant A	Variant B

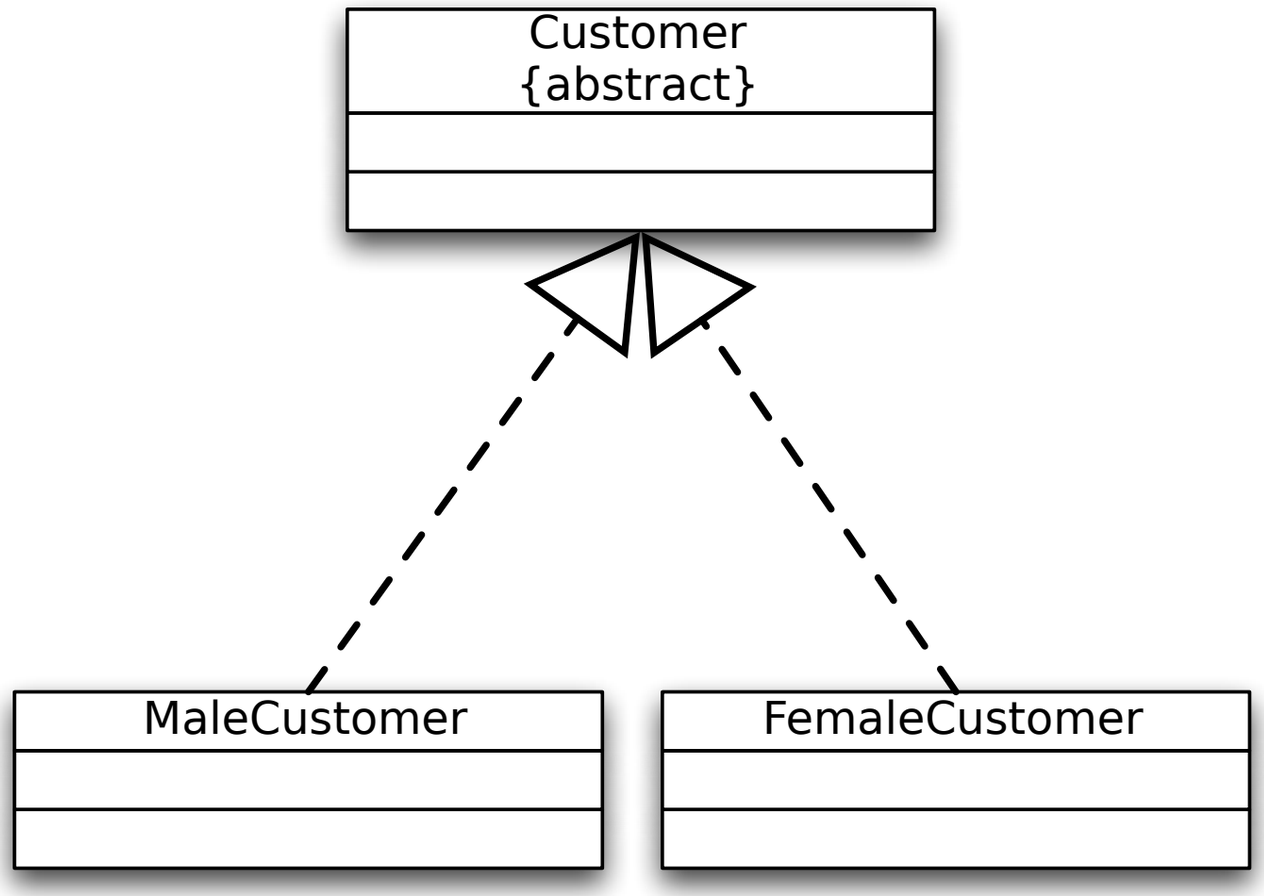
`class Person {…}`
`class Father extends Person {…}`
`class Mother extends Person {…}`
	`class Person {…}`

`main () {`
`  Father f = new Father(…);`
`  Mother m = new Mother(…);`
`}`
	`main () {`
	`  Person father = new Person(…);`
	`  Person mother = new Person(…);`
	`}`

- Whether to choose Variant A or B depends on the domain you are modeling; i.e. whether Mother and Father **exhibit different behavior**
- Before creating new classes, be sure the behavior is truly different and that you do not have a situation where each role is using a subset of Person functionality

Classes That Model the Roles an Object Plays

- What do you think of the following design?



Which question do you have to ask yourself to decide if such a design makes sense?

Summary

The goal of this lecture is to enable you to systematically carry out small(er) software projects that produce quality software.

- Always assign responsibilities to classes such that the coupling is as low as possible ↓, the cohesion is as high as possible ↑ and the representational gap is as minimal as possible ↓.
- Coupling and cohesion are evaluative principles to help you judge OO designs.
- Design heuristics are not hard rules, but help you to identify weaknesses in your code to become aware of potential (future) issues.

- The goal of this lecture is to enable you to systematically carry out small(er) commercial or open-source projects.