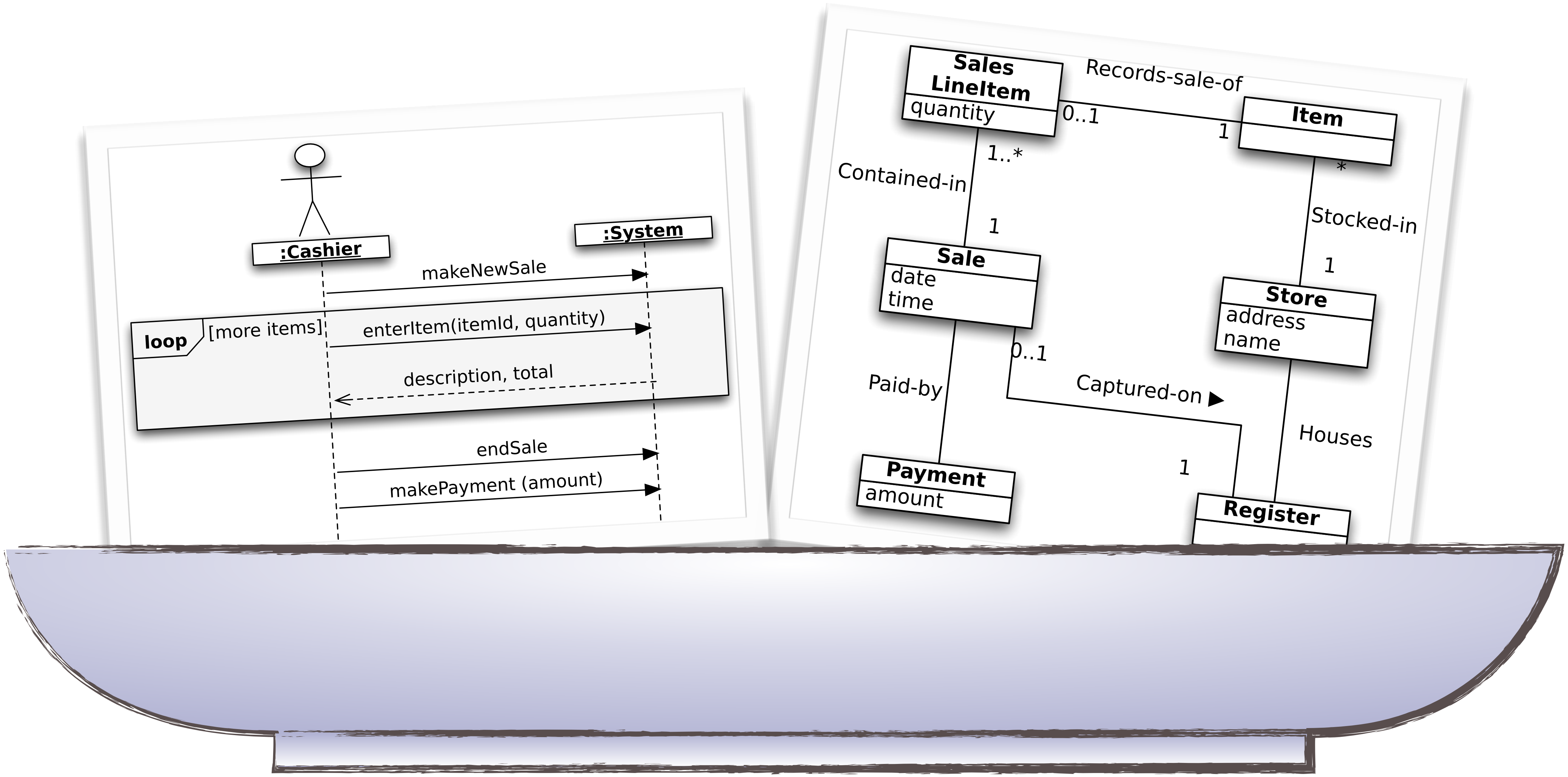
Dr. Michael Eichberg
Software Technology Group
Department of Computer Science
Technische Universität Darmstadt

Introduction to 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
  - a system sequence diagram

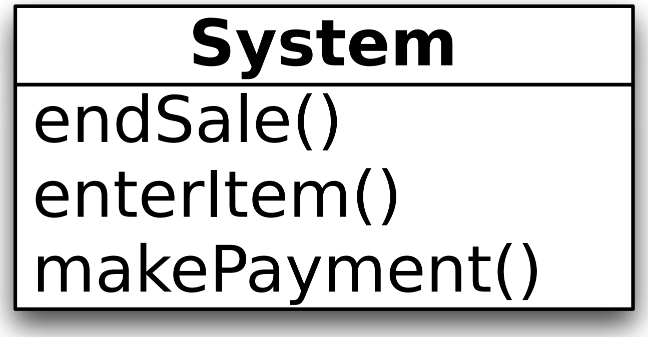
- **Next steps:**
  Build interaction diagrams for system operations of the use-cases at hand by applying guidelines and principles for assigning responsibilities

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 operati**… o a conceptu…
  Does not ne… System in the design.
- A controll… e system operation…

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 operati**… o a conceptu…
  Does not im… the design.
- A controll… e system operation…

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

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

System

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.

Responsibilities are related to the obligations or behavior of an object in terms of its role.
We can distinguish two basic types of responsibilities.

- Doing responsibilities
  - Doing something itself
    E.g. creating an object or doing a calculation.
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects
  - Example: a Sale object is responsible for creating SalesLineItem objects
- Knowing responsibilities
  - Knowing about private encapsulated data
  - Knowing about related objects
  - Knowing about things it can derive or calculate
  - Example: a Sale is responsible for knowing its total

Responsibilities are assigned to objects by using methods of classes to implement them.

- To implement a responsibility, methods act alone or collaborate with other methods (of other objects):
  - 1 method in 1 object,
  - 5 methods in 1 object,
  - 50 methods across 10 objects

depending on the granularity of the responsibility

A responsibility is not the same thing as a method.

Responsibilities are assigned to objects by using methods of classes to implement them.

Examples:

- Providing access to data bases may involve dozens of classes
- Print a sale may involve only a single or a few methods

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

A responsibility is not the same thing as a method.

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 probabilit… reuse, should have espe…
  - Very little … desirable
  - Central me… ected objects that comm…
  - Low coupli… to excess results in active objects that do all the work

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

Coupling

- …

- **Low Coupling**
  Low coupling supports design of relatively independent, hence more reusable, classes

  - Generic cla… probability for reuse, should have espec…
  - Very little … desirable
  - Central m… ected objects that comm…
  - Low coup… ctive objects 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.

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

```java
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.

Design needs principles.

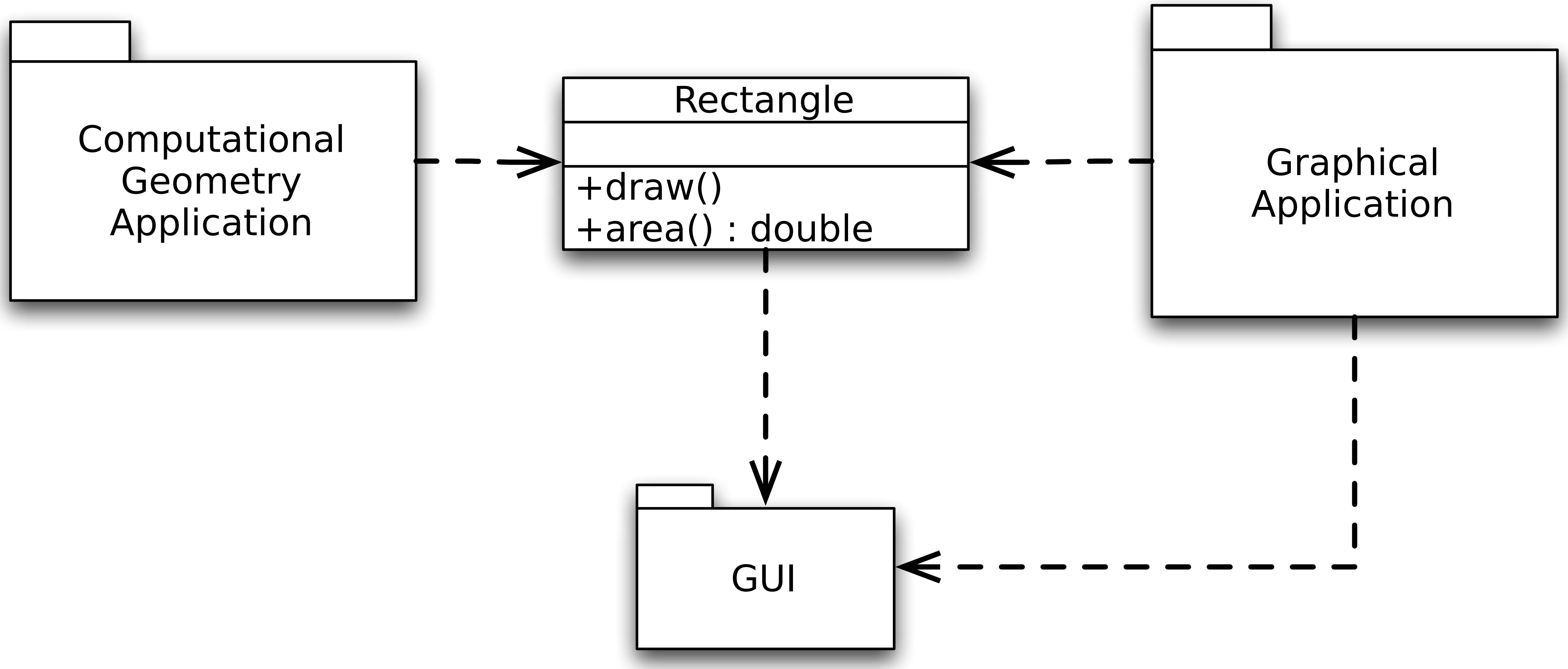
> "A class should have only one reason to change."
>
> I.e. a responsibility is primarily a reason for change.

The Single Responsibility Principle

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

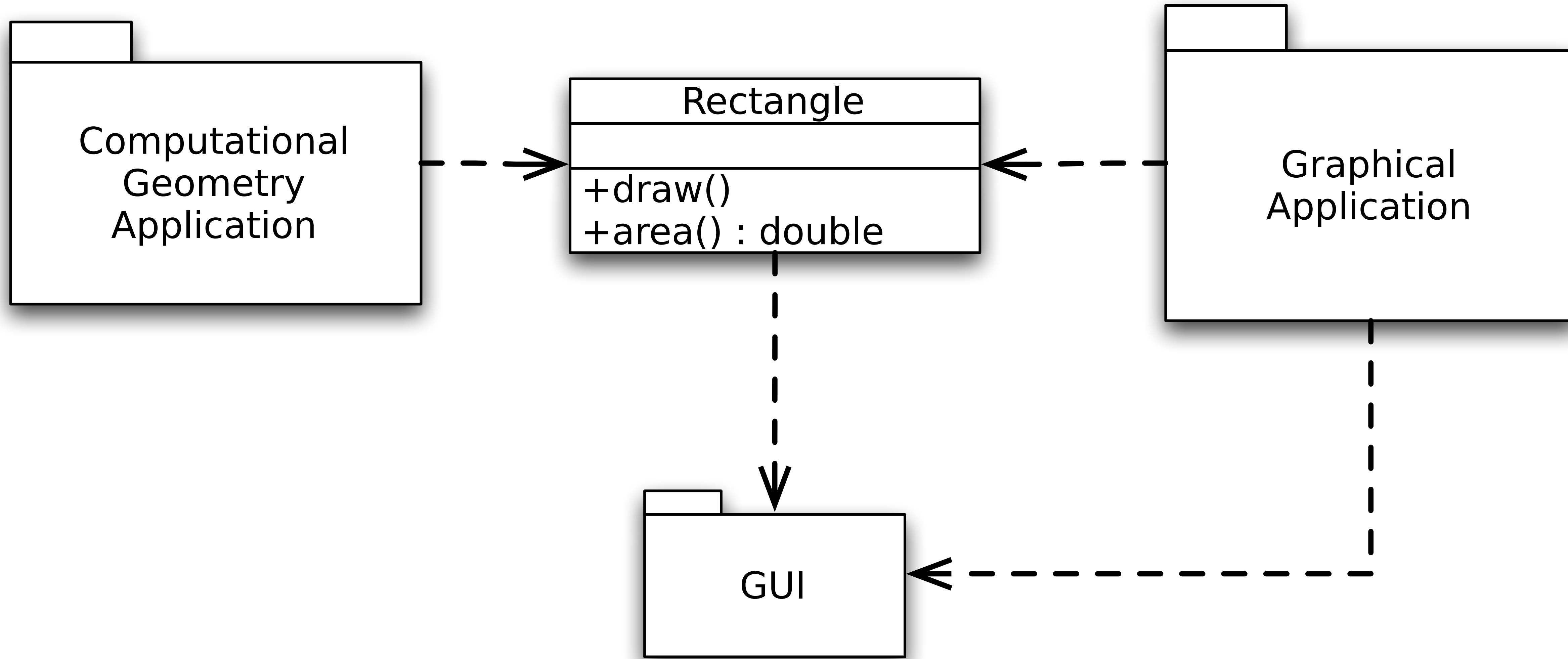
Example: a Rectangle Class

The Single Responsibility Principle



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

Example: a Rectangle Class



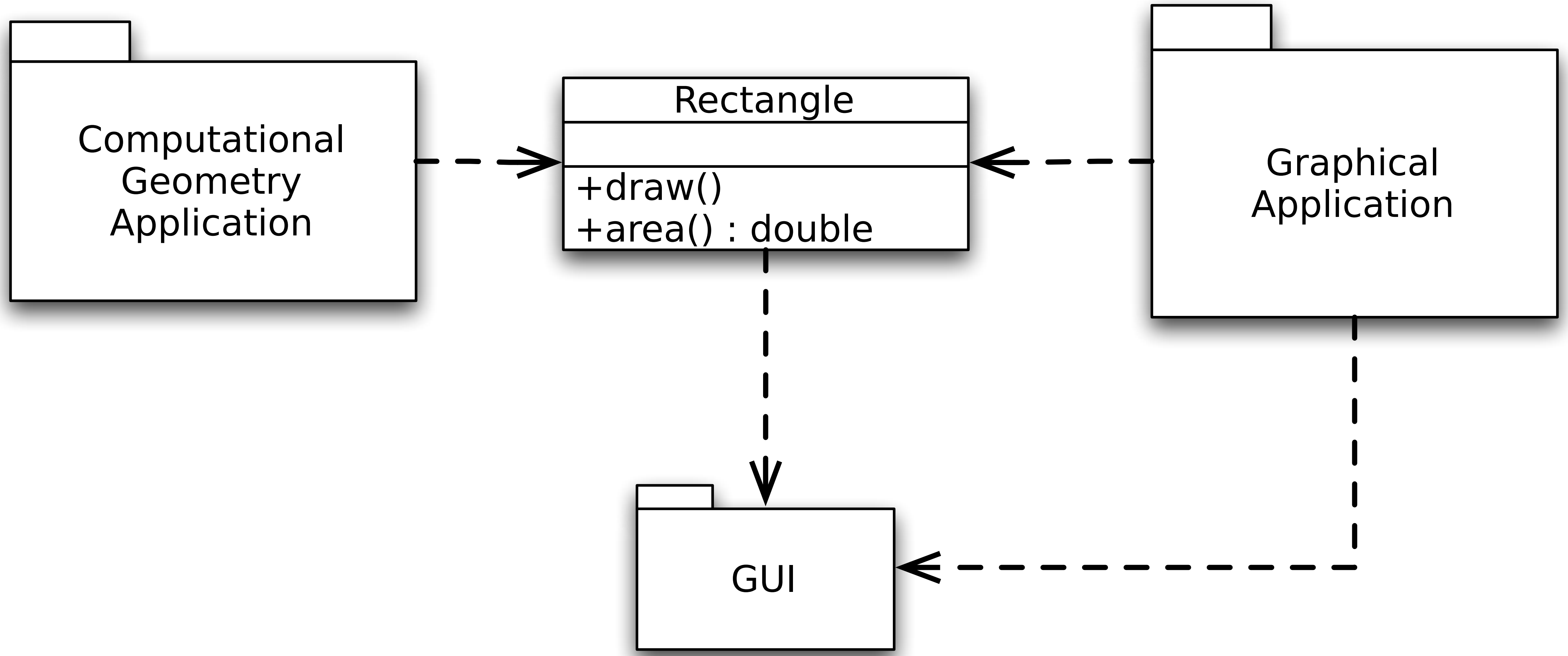
The Rectangle class has multiple responsibilities:

- 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

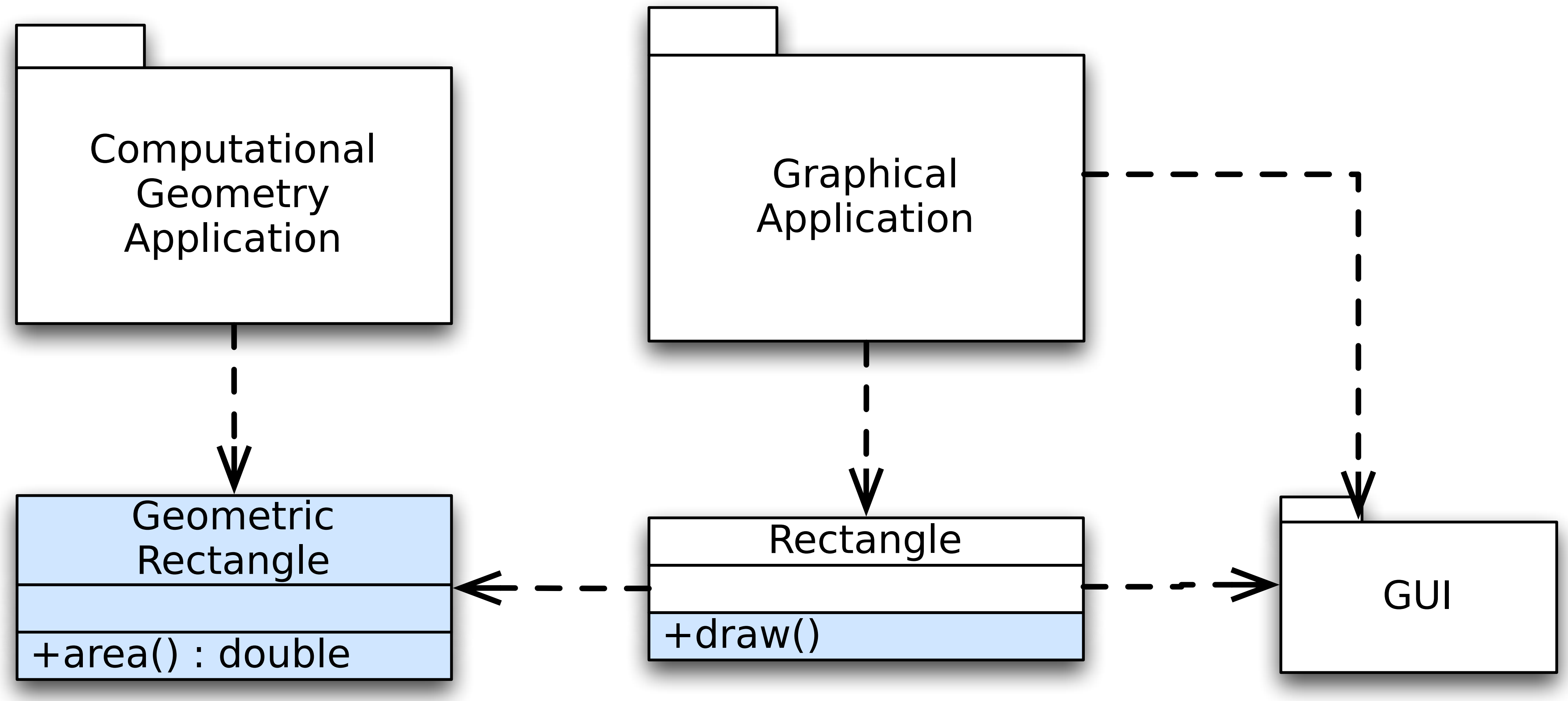
The Single Responsibility Principle



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

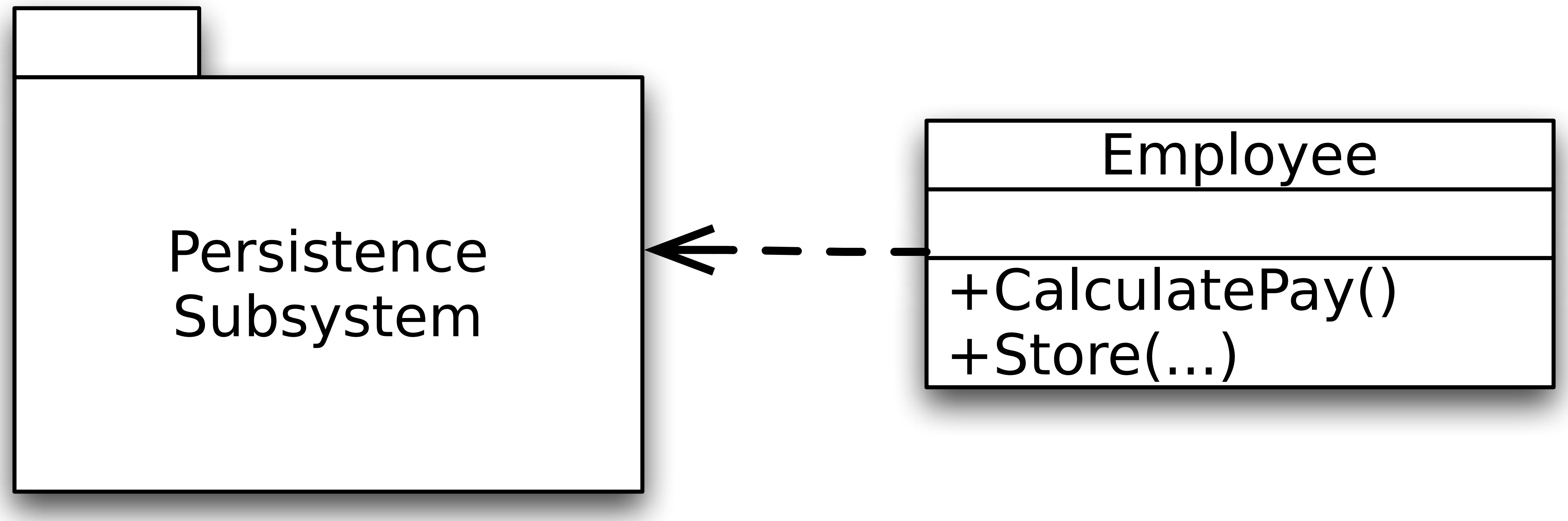


The solution is to separate the functionality for drawing a rectangle and the functionality for doing calculations are separated.

Coupling? Cohesion?

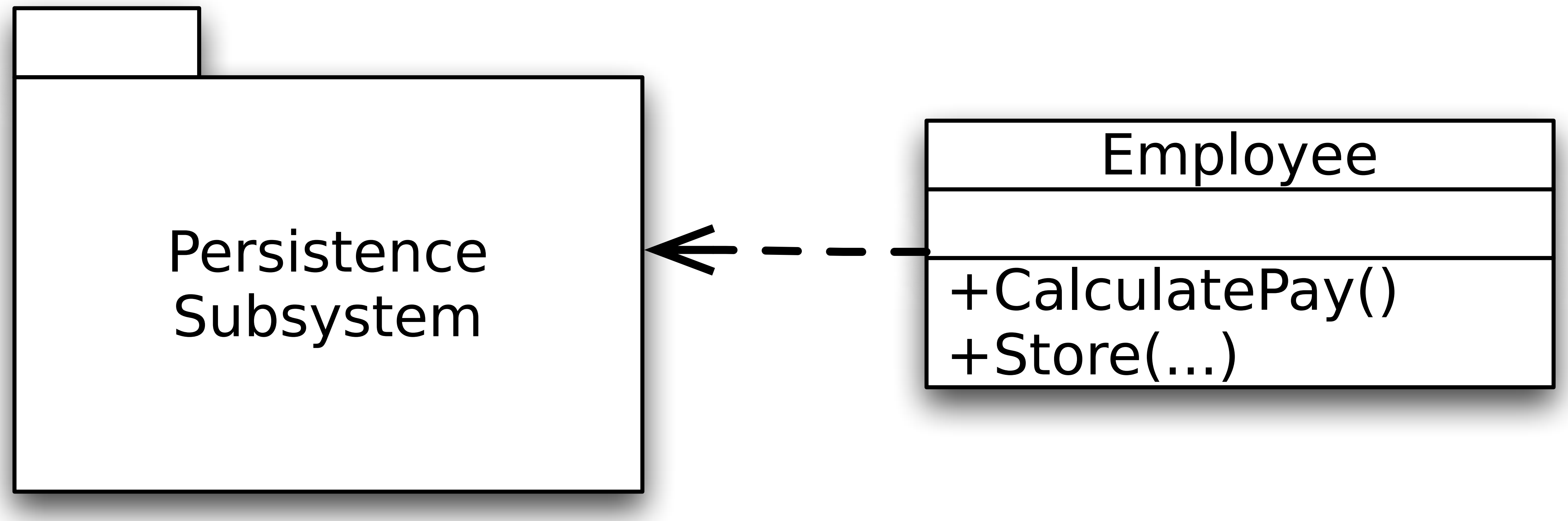
Example: Handling Persistence

The Single Responsibility Principle



Do we need to change the Employee class?

Example: Handling Persistence
The Single Responsibility Principle



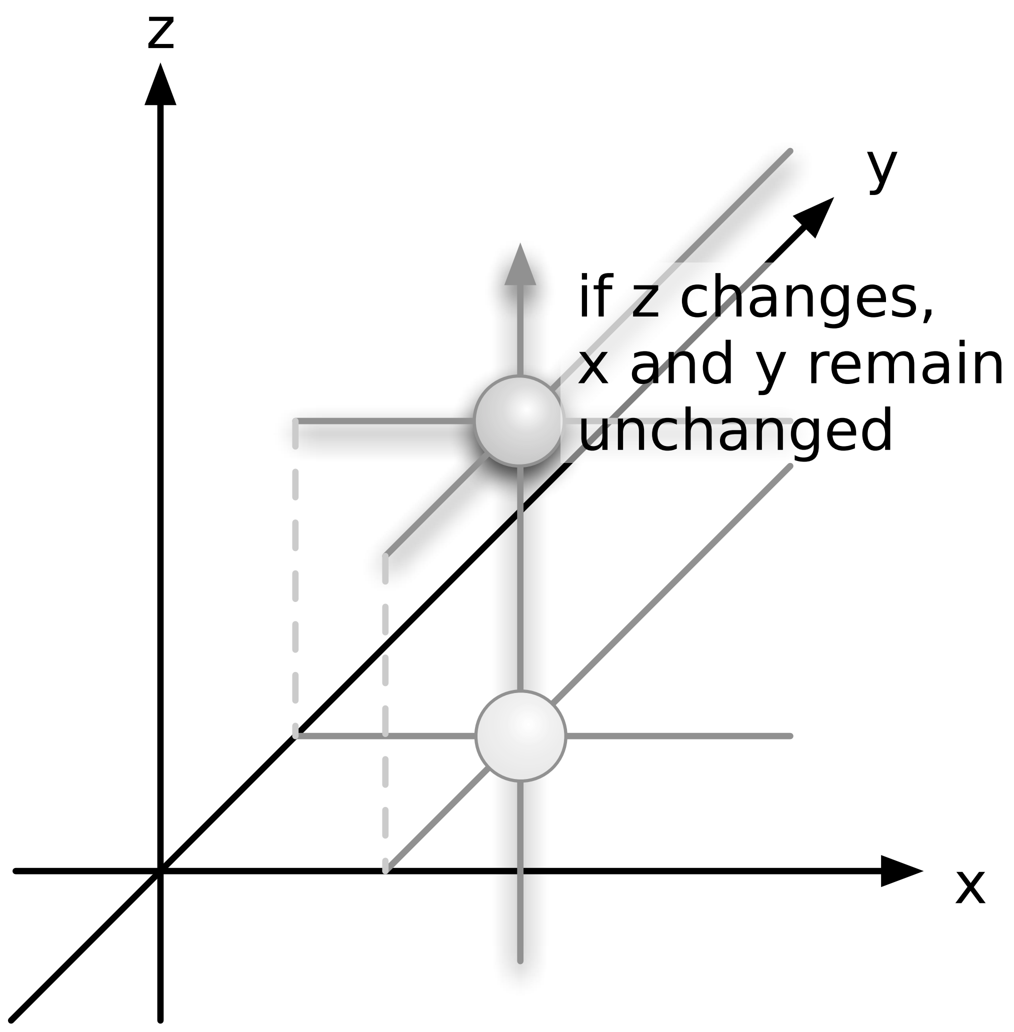
Two responsibilities:

- 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

GRASP

General Responsibilit nt Principles

DAS THEMA GRASP WIRD zum nächsten WS hin gestrichen.

- The following slides make ex aterial from:
  Applying UML and Patterns, 3rd Edition; Craig Larman; Prentice Hall

TECHNISCHE UNIVERSITÄT DARMSTADT

Fundamental GRASPrinciples...

- Controller
- Creator
- (Information)Expert
- ...

The GRASPrinciples are a learning aid.

GRASP - **Controller** - Candidates

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

System
endSale() enterItem() makePayment()

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

GRASP - **Controller** - Candidates

- **Façade controller**
  A class that represents the overall "system" or "business"

- **Use Case controller**
  A class that represents an artificial handler of all events of a use case

Candidates for assigning controller responsibility.

enterItem(itemId,quantity)→ :???

enterItem(itemId,quantity)→ :POST **Real world actor**

enterItem(itemId,quantity)→ :Register **Facade: overall system**

enterItem(itemId,quantity)→ :ProcessSaleHandler **Use-case handler**

GRASP - **Controllers and High Cohesion**

- **Façade controllers** are suitable when there are only a “few” system events
- **Use Case controller**
  These are not domain objects, these are artificial constructs to support the system.
  - Good when there are many system events across several processes
  - Possible to maintain state for the use case, e.g., to identify out-of-sequence system events: a makePayment before an endSale operation

GRASP - **Controllers and Responsibility**

- A controller should mostly coordinate activities
- Delegate to other objects work that needs to be done
- Signs of a **bloated controller**:
  - Receives all system events
  - Performs all tasks itself without delegating
  - Has many attributes and maintains significant information about the domain
  - Duplicates information found in other objects

Split a bloated controller into use case controllers - likely to help in maintaining low coupling and high cohesion.

GRASP - Controllers and Presentation Layer

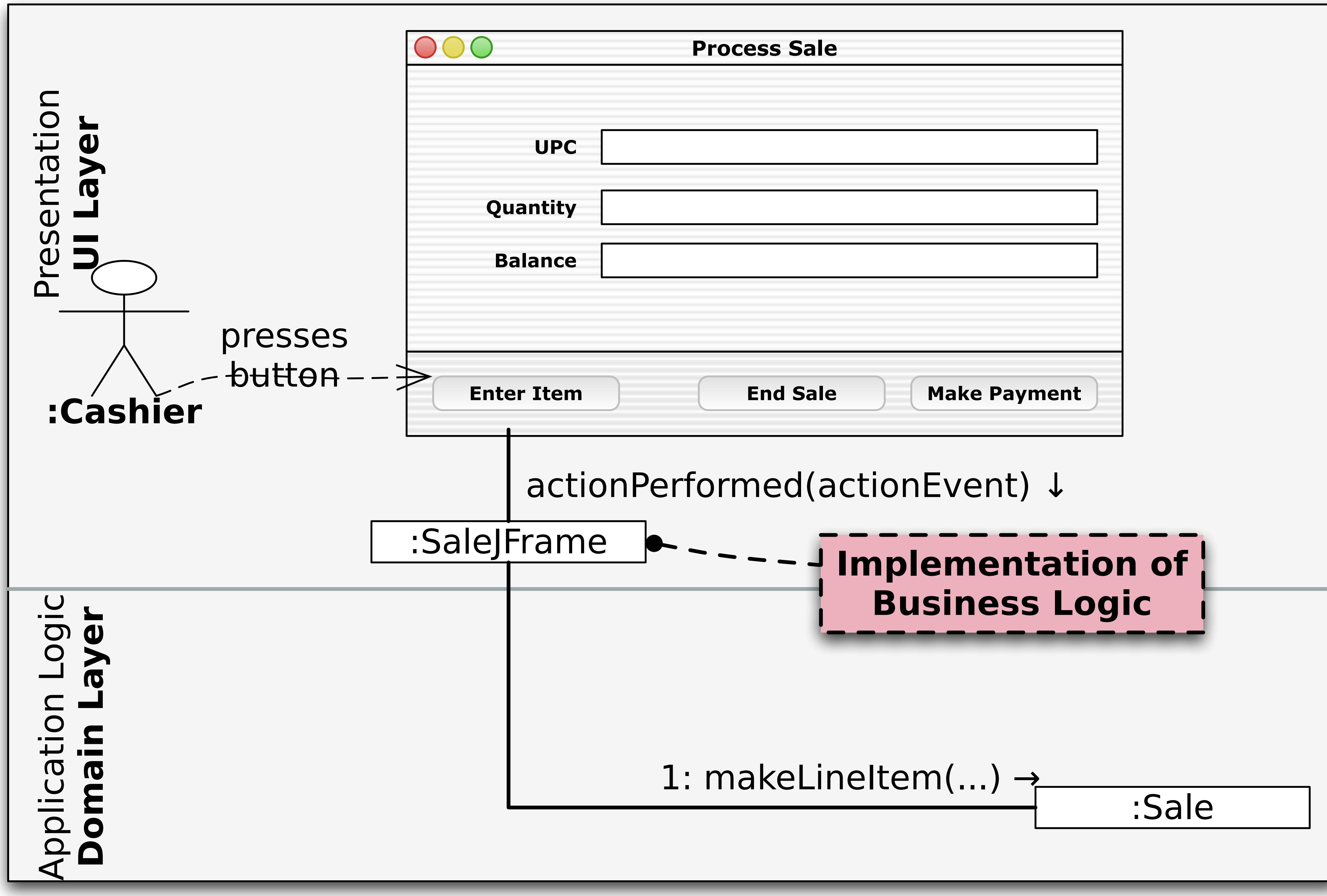
- UI objects and the UI layer should not have the responsibility for handling system events
  Examples that do not qualify as controllers: “Window”, “Menu Item”, “Sensor”,...
- **System operations should be handled by objects belonging to the domain layer**
  This increases the reuse potential; “encapsulation” of the business process.

GRASP - Controllers and Presentation Layer
Bad Design vs. Good Design

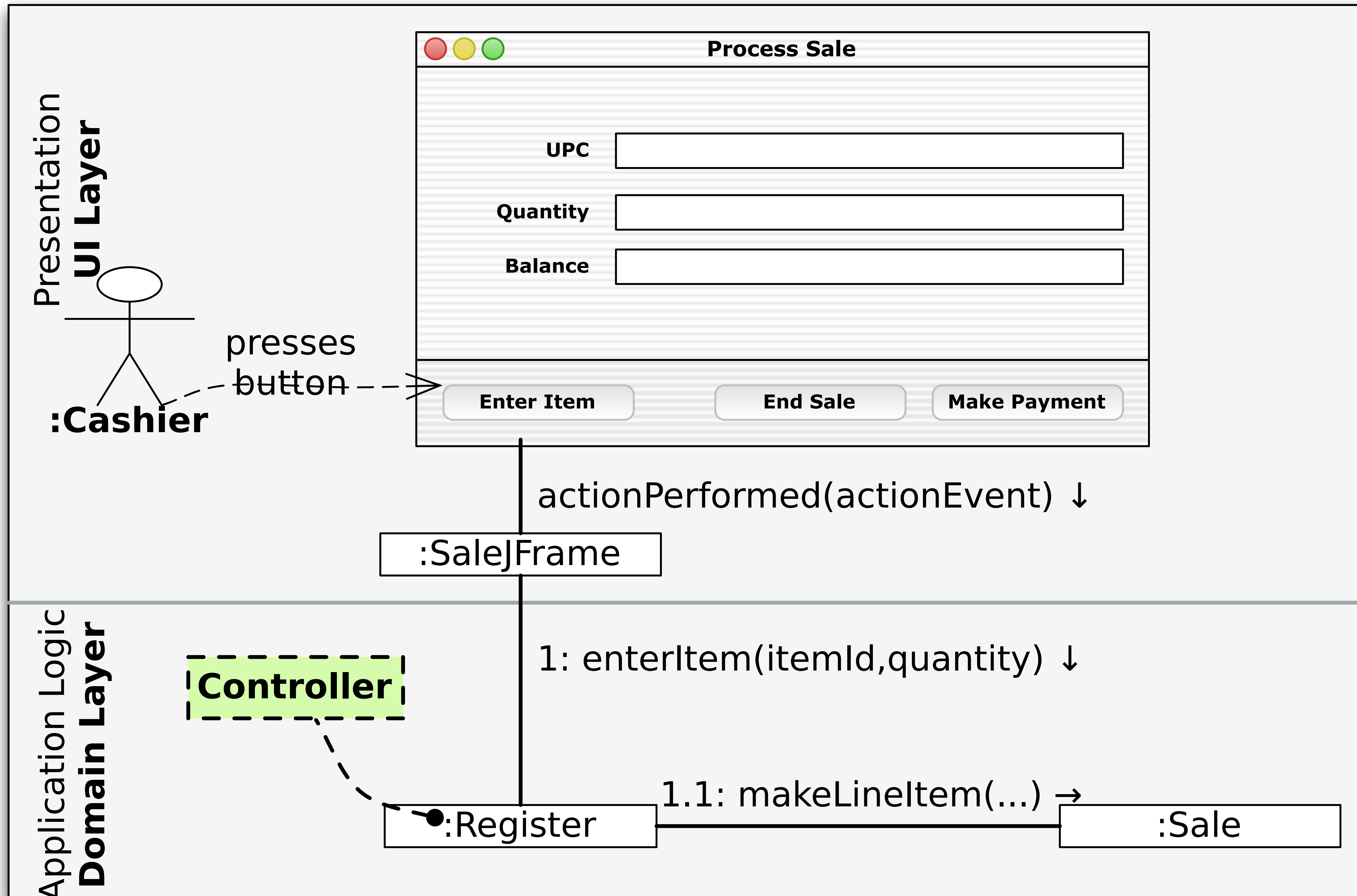
- A user-interface-as-controller design …
  - reduces the opportunity to reuse domain process logic in future applications
  - it is bound to a particular interface that is seldom applicable in other applications

- Placing system operation responsibility in a domain object controller makes it easier …
  - to unplug the interface layer and use a different interface technology
    E.g. in case of multi-channel application.
  - to run the system in an off-line "batch" mode

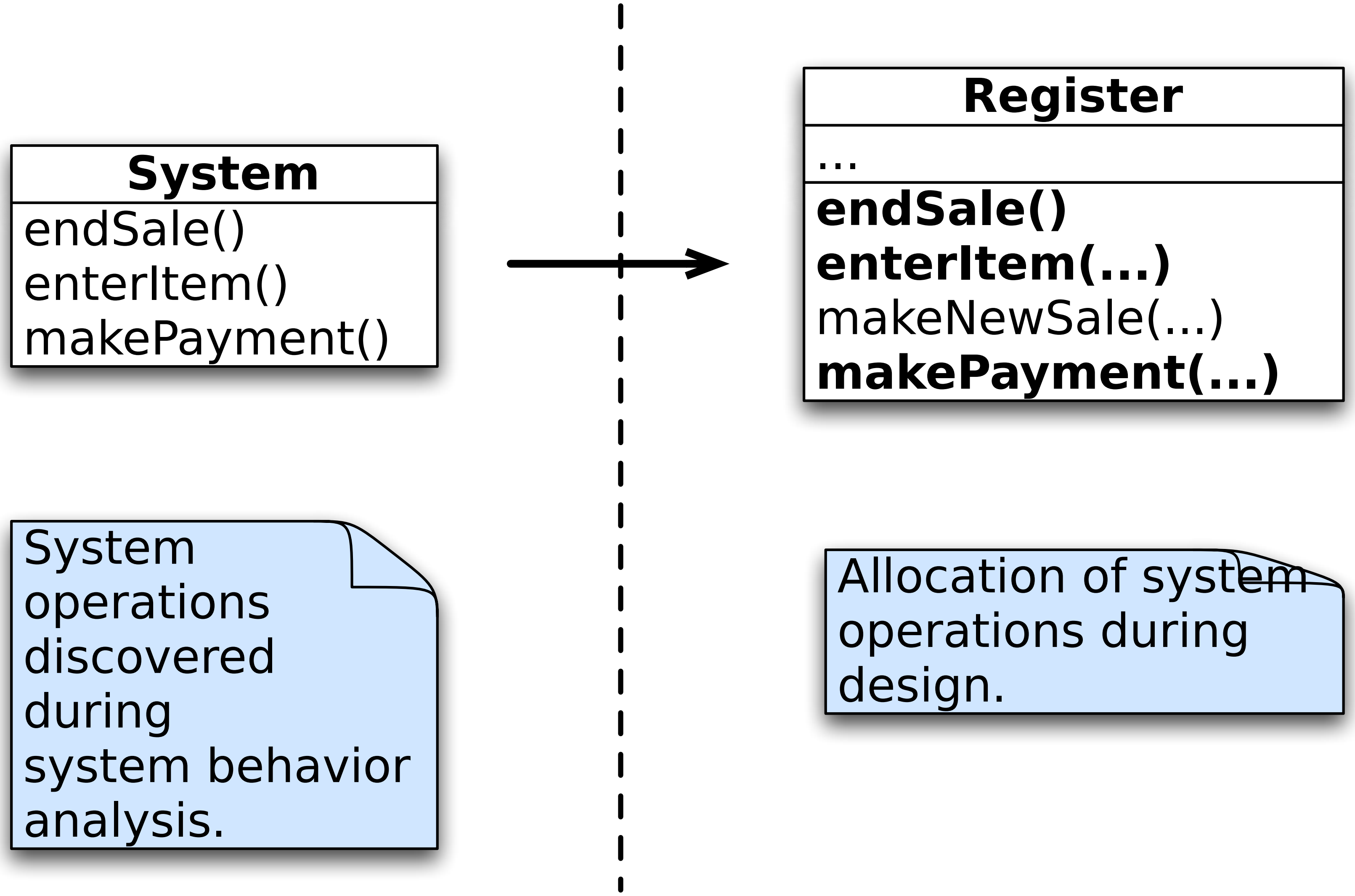
GRASP - Controllers and Presentation Layer
Bad Design



GRASP - Controllers and Presentation Layer
Good Design



GRASP - **Controllers** - Summary



System operations - identified during analysis - are assigned - during design - to one or more non-UI classes called controllers that define an operation for each system operation

Example

Designing **makeNewSale** of the ProcessSale Use Case

System Operation Contract

...	...
Preconditions	None
Postconditions	• a Sale instance s was created Instance creation • s was associated with the Register Association formed • the attributes of s are initialized

Example

Designing **makeNewSale** of the ProcessSale Use Case

Choosing the Controller for makeNewSale

- What first object beyond the UI layer receives and coordinates a system operation?

- A controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.

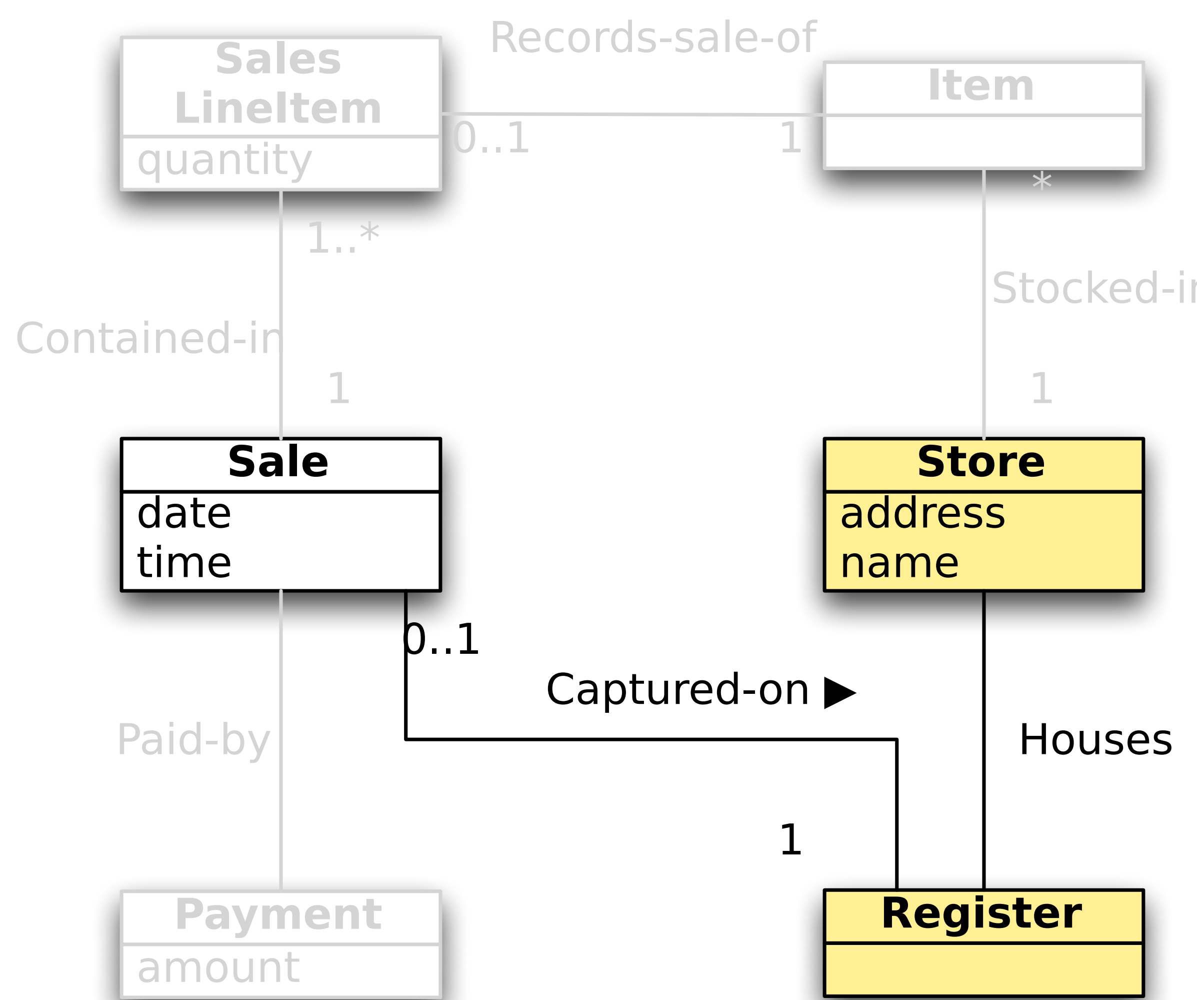
Controller

Example

Designing **makeNewSale** of the ProcessSale Use Case

Choosing the Controller for makeNewSale

- A class that represents the overall system, a root object, a specialized device, or a major subsystem:
  - a **Store** object representing the entire store
  - a **Register** object (a specialized device that the software runs on)
- Represents a receiver or handler of all system events of a use case (artificial object):
  - a **ProcessSaleHandler** object
  - a **ProcessSaleSession** object

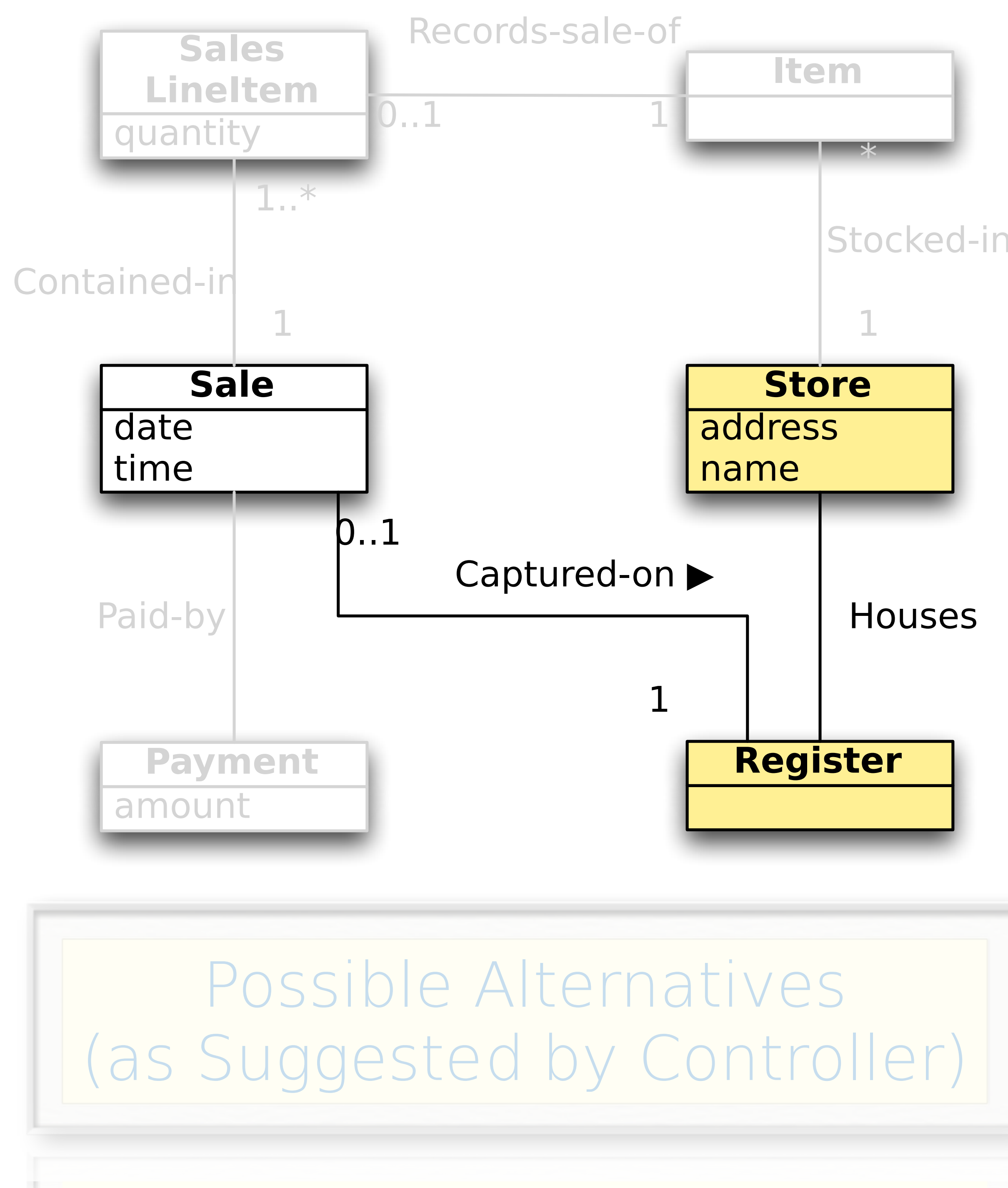


Possible Alternatives
(as Suggested by Controller)

Choosing the Controller for makeNewSale

Reasoning

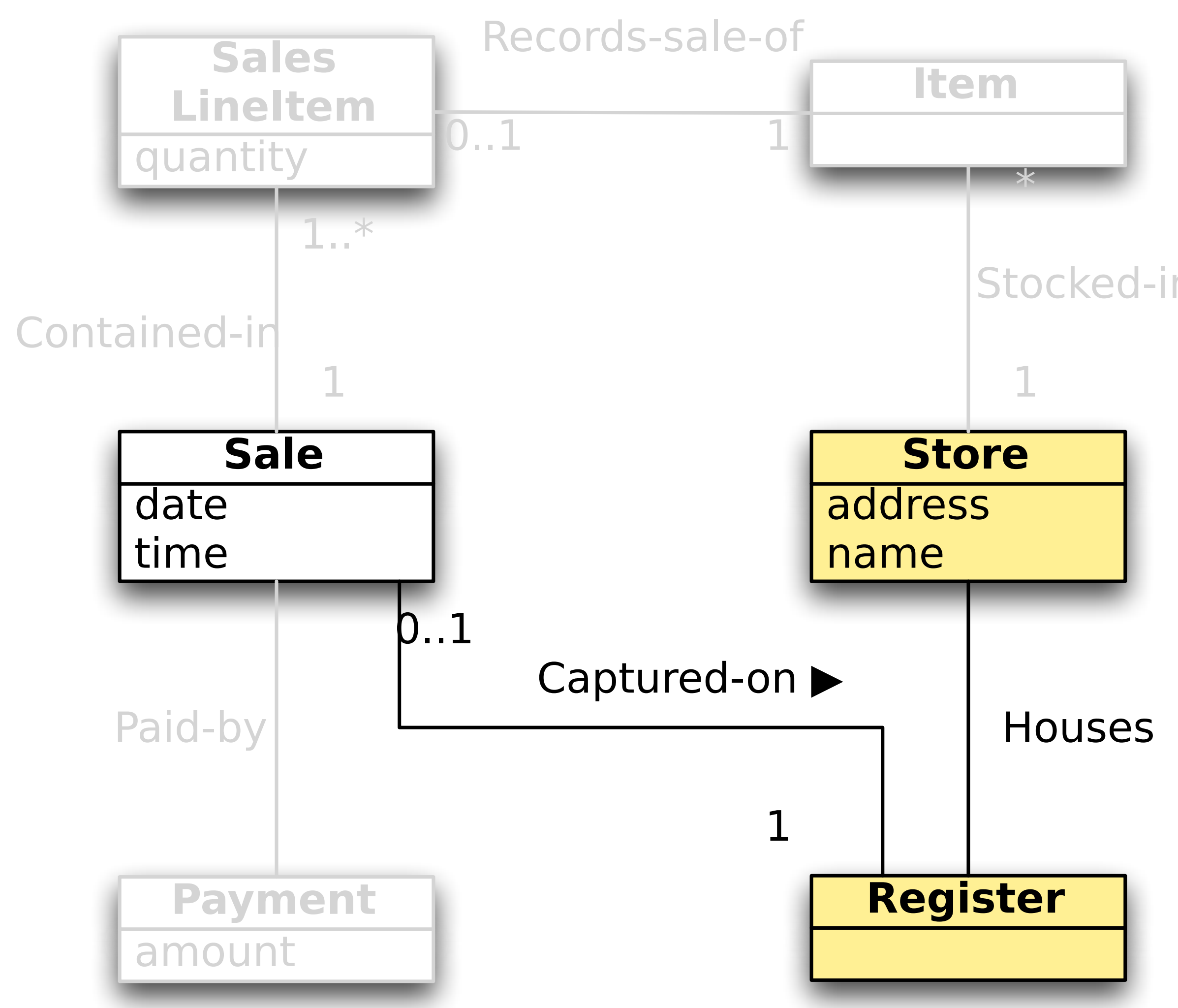
- **Register** would represent a device **façade controller**
- Recall from the discussion of Controller:
  ... Device façade controllers are suitable when there are only a "few" system events...



Choosing the Controller for makeNewSale

Reasoning

- Choosing a **Store** object would lead to low cohesion If we continue using Store for everything.
- Choosing **Store** results in a high representational gap



Possible Alternatives
(as Suggested by Controller)

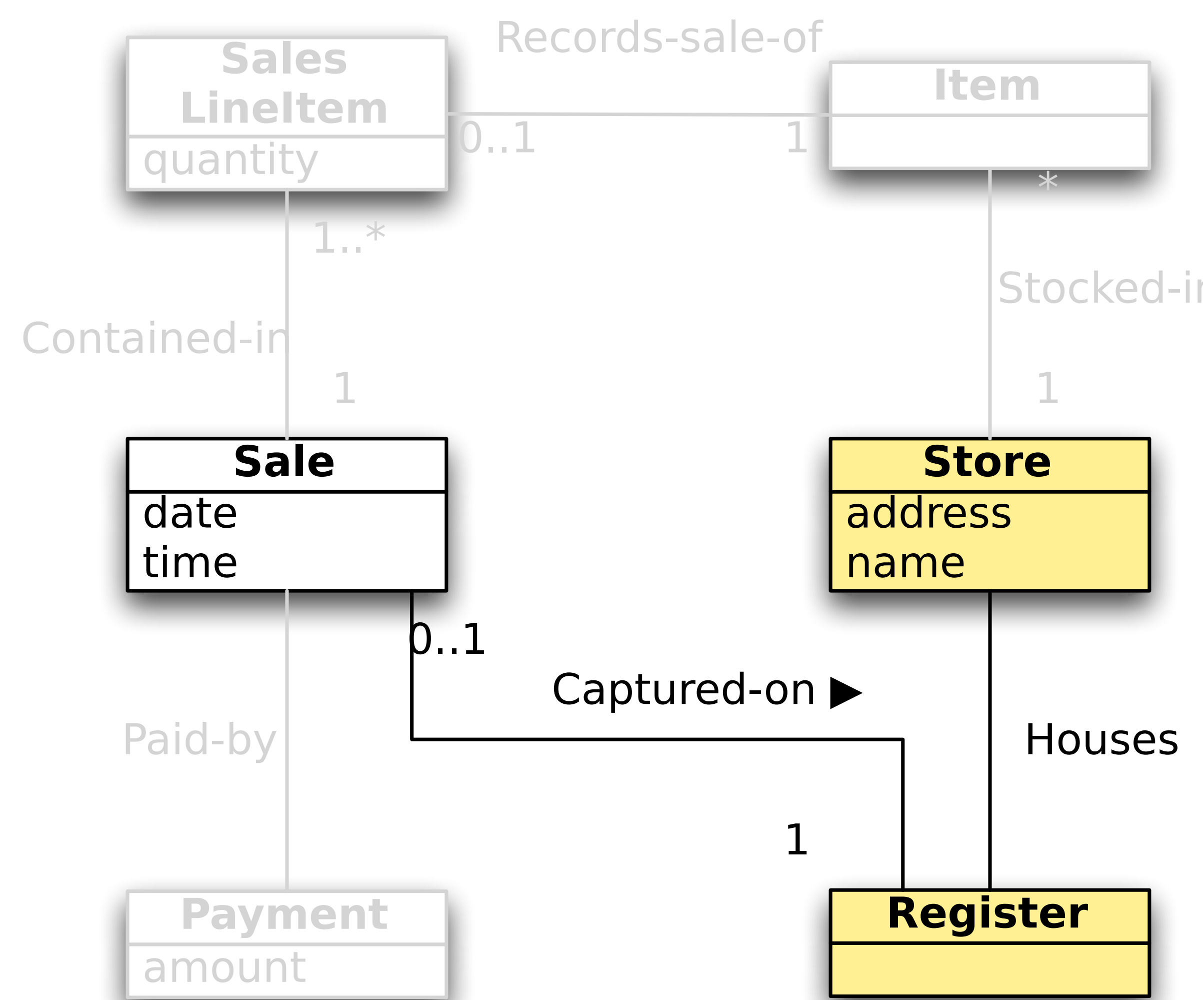
Example

Designing **makeNewSale** of the ProcessSale Use Case

Choosing the Controller for makeNewSale

Reasoning

- **Use-case controllers** (ProcessSaleHandler, ProcessSaleSession) are good when...
  - there are many system events across several processes,
  - it is necessary to identify out-of-sequence system events.



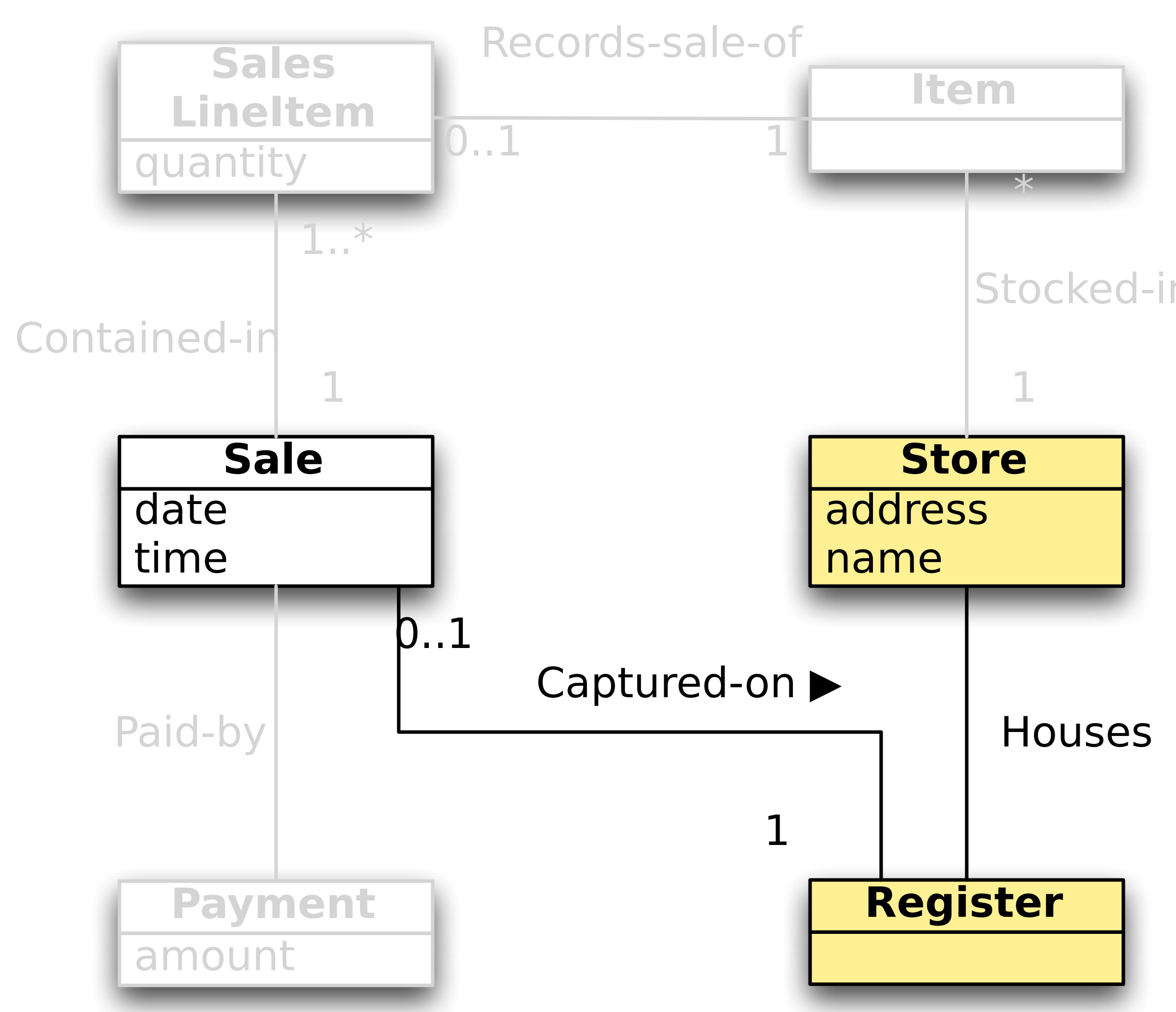
Possible Alternatives
(as Suggested by Controller)

Example
Designing **makeNewSale** of the ProcessSale Use Case

Choosing the Controller for makeNewSale

Conclusion

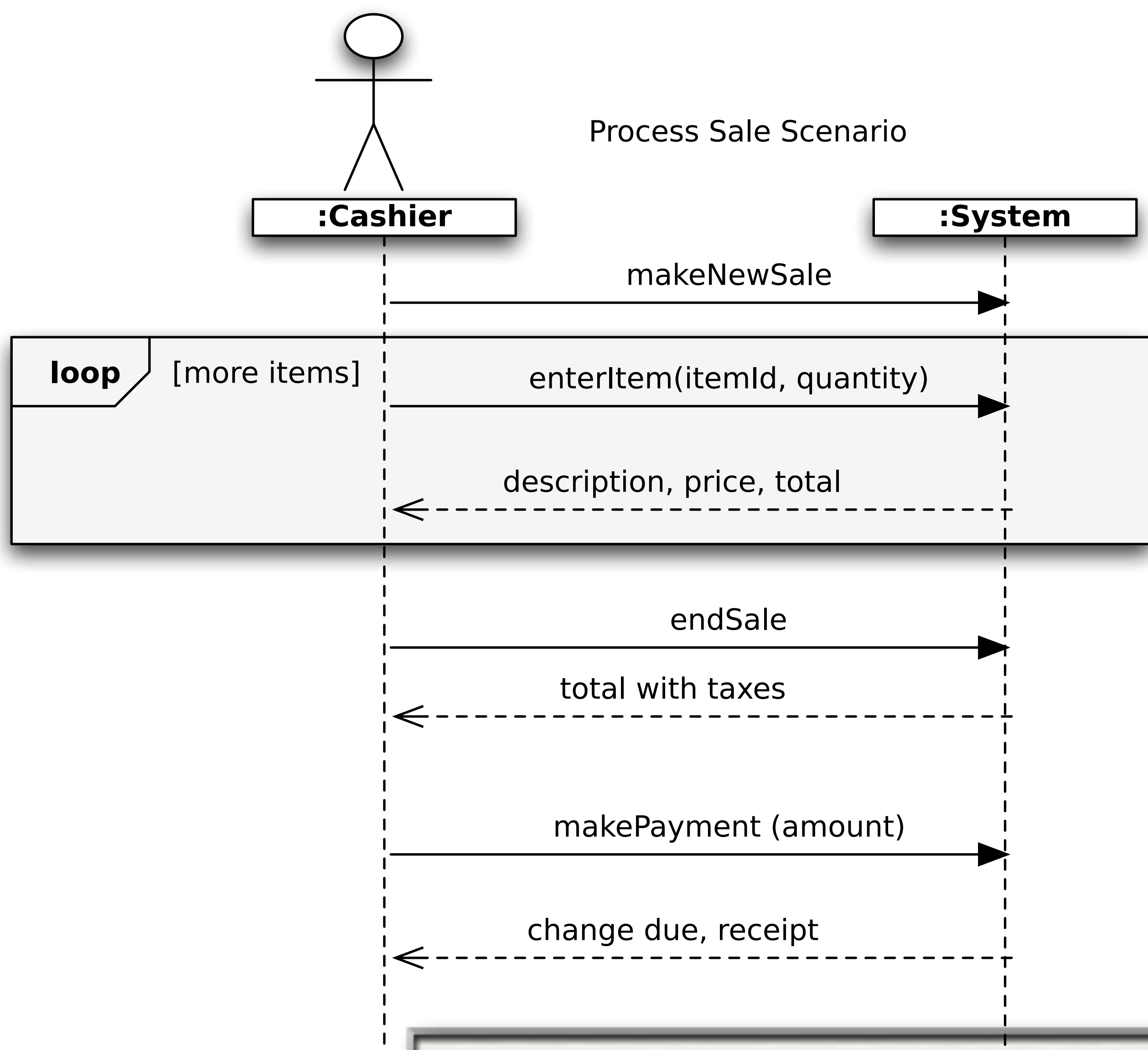
- Register would represent a device façade controller.
  ... Device façade controllers are suitable when there are only a “few” system events...
- ~~Choosing Store results in low cohesion and a high representational gap.~~
- ~~Use case controller (e.g. ProcessSaleHandler, ProcessSaleSesion)~~



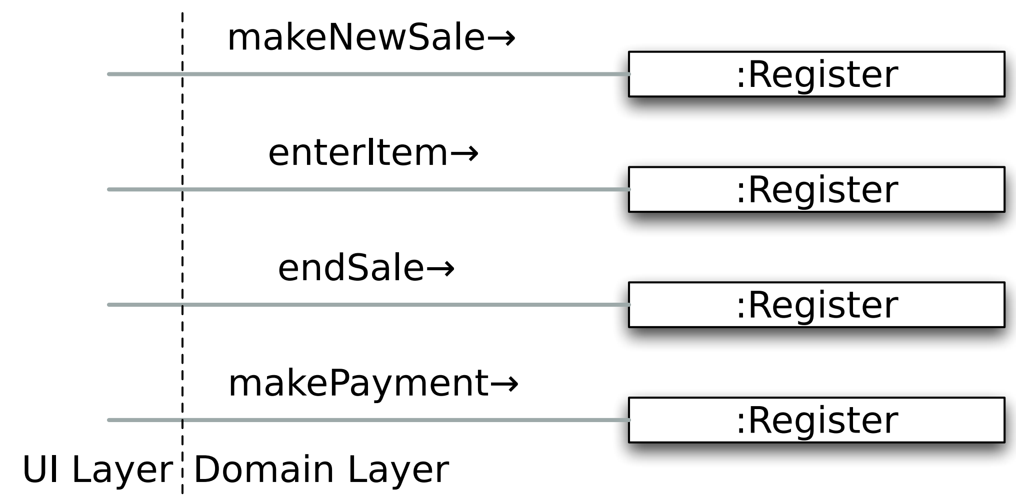
Possible Alternatives
(as Suggested by Controller)

Example
Choosing the Controller for the other System Operations

System Sequence Diagram



Interaction with the domain layer object Register (as suggested by the Controller pattern)



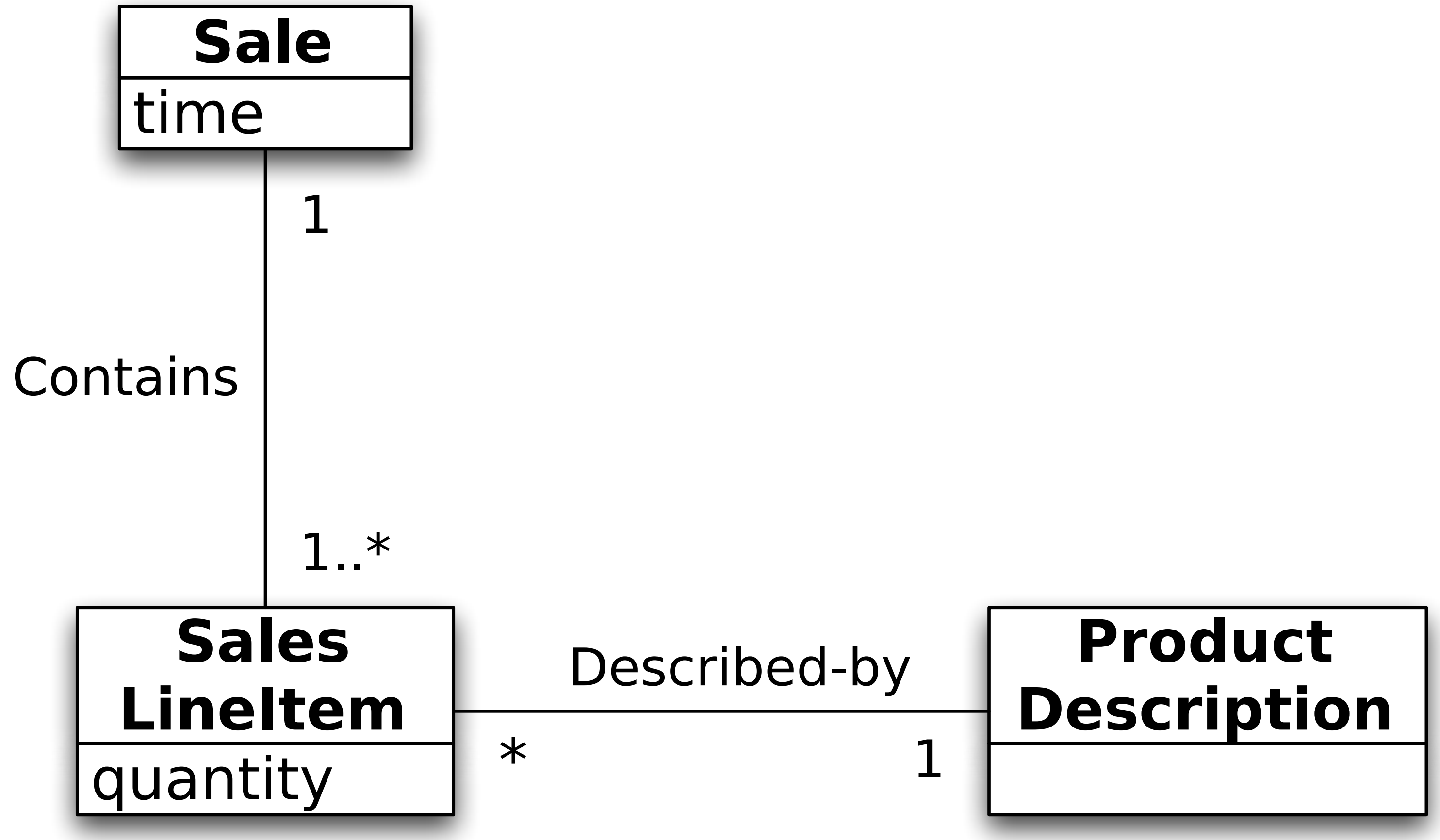
Apply the same reasoning!

GRASP - **Information Expert**

- What is the most basic, general principle of responsibility assign?
- Assign a responsibility to an information expert, i.e., to a class that has the information needed to fulfill that responsibility.

GRASP - **Information Expert** - Example

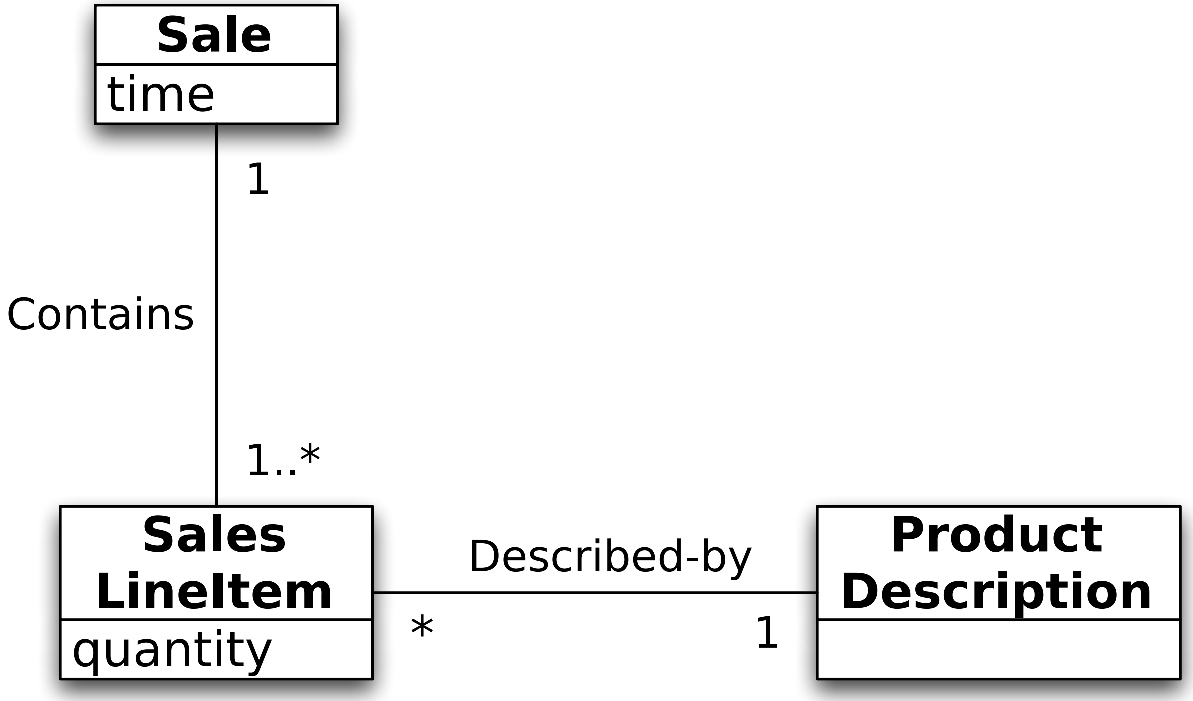
Calculating the Grand Total



Given this conceptual model, who should be responsible for calculating the grand total of a sale?

GRASP - **Information Expert** - Example

Calculating the Grand Total



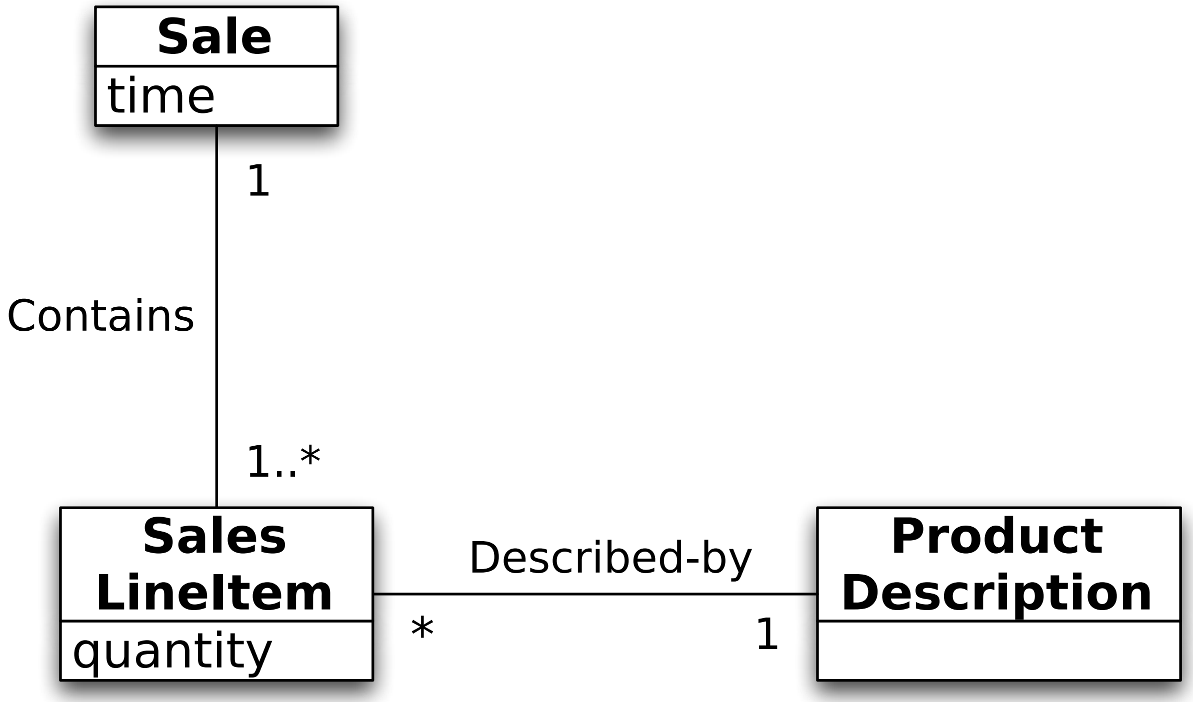
Given this conceptual model, who should be responsible for calculating the grand total of a sale?

Which class has the information needed for calculating the grand total, i.e.,

- knowledge of all SalesLineItems, and
- their subtotals?

GRASP - **Information Expert** - Example

Calculating the Grand Total



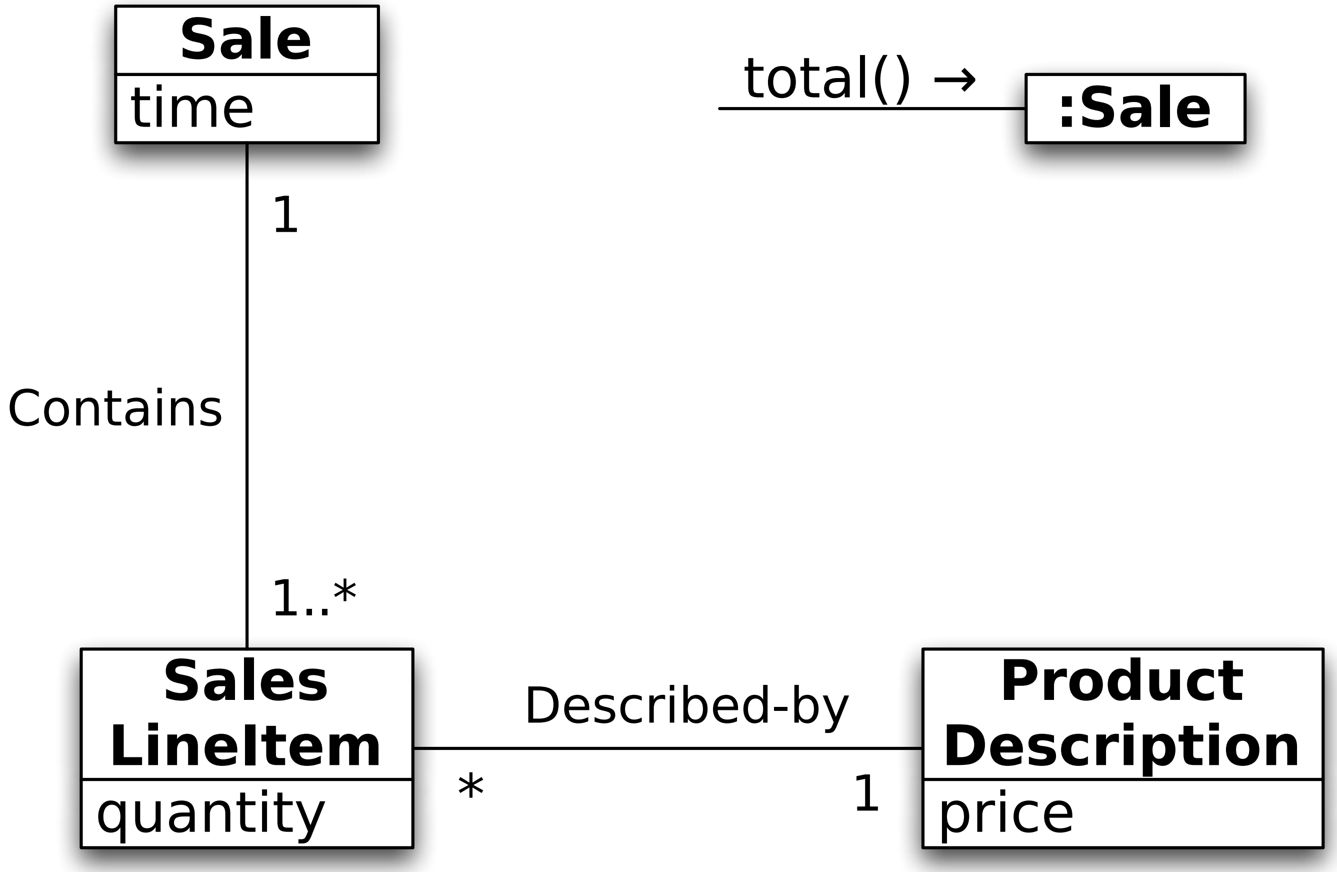
Given this conceptual model, who should be responsible for calculating the grand total of a sale?

Which class has the information needed for calculating the grand total, i.e., knowledge of all SalesLineItems, and their subtotals?

The Sale object possesses the knowledge about all SaleLineItems. Hence, Sale will be assigned the responsibility.

GRASP - **Information Expert** - Example

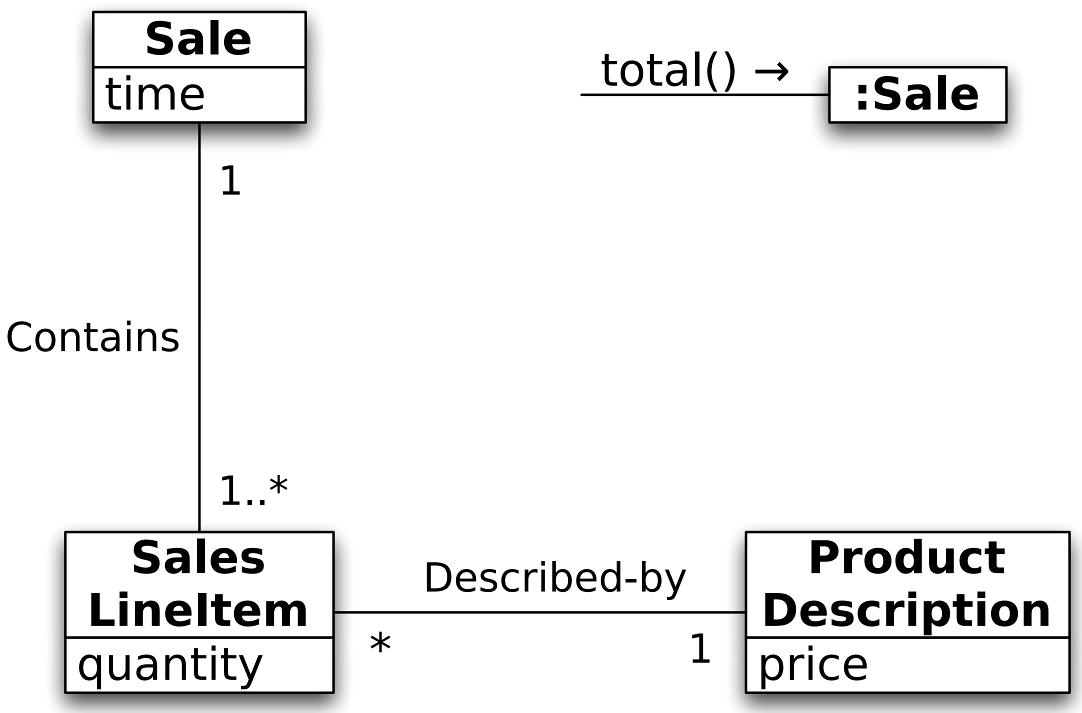
Calculating the Sub Total



Which class has the information needed for calculating the subtotals?

GRASP - **Information Expert** - Example

Calculating the Sub Total



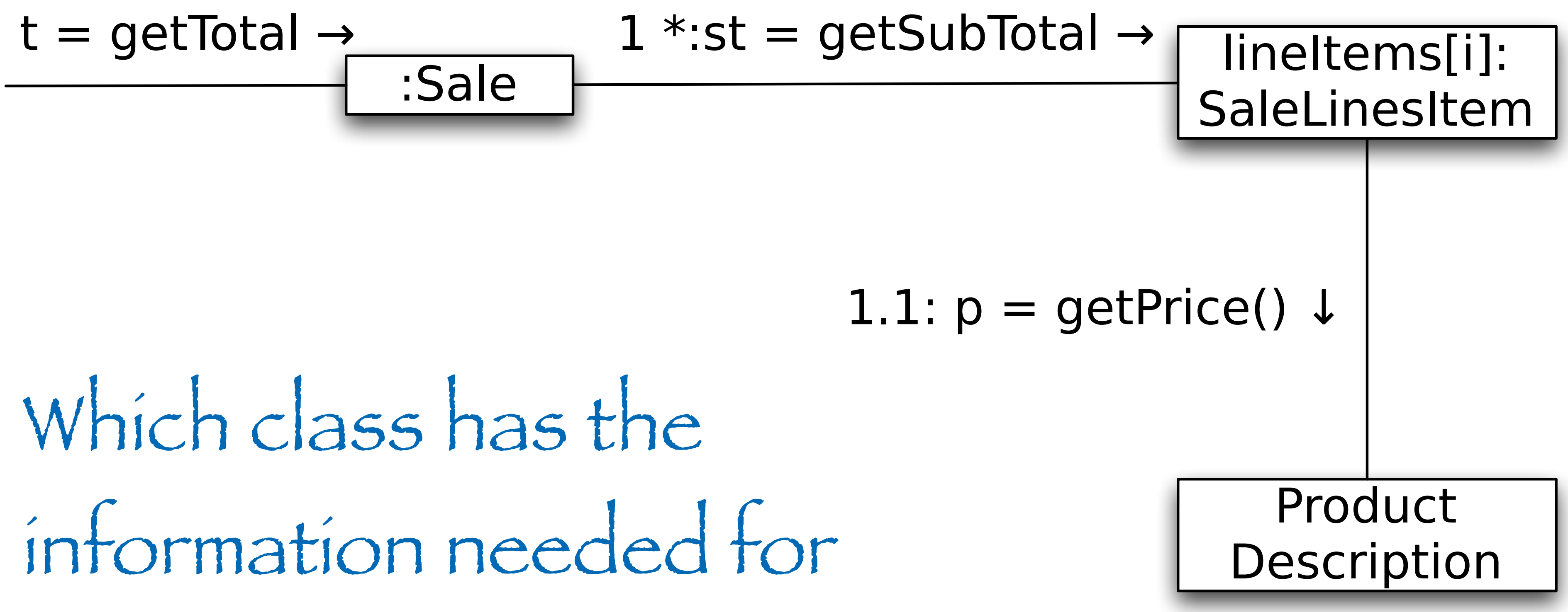
Which class has the information needed for calculating the subtotals?

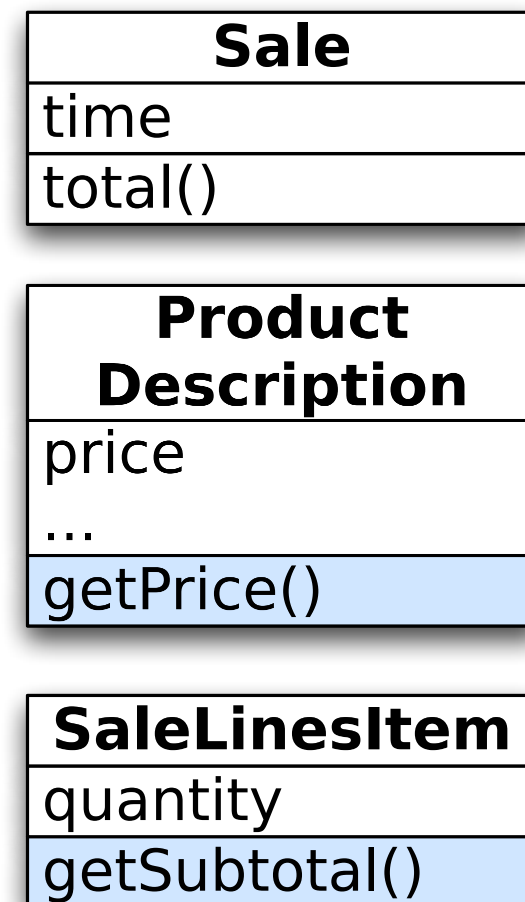
Required information: **quantity and price of each SalesLineItem**

- quantity is available with SalesLineItem
- price is available with ProductDescription

GRASP - **Information Expert** - Example

Calculating the Sub Total





Which class has the information needed for calculating the subtotals?

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductDescription	knows product price

GRASP - **Information Expert** - Summary

- Fulfillment of a responsibility often requires interaction amongst several objects (4 in our example)
  There are many semi-experts who collaborate in performing a task.
- Use of *(Information) Expert* guideline allows us to retain encapsulation of information
  Information hiding
- It often leads to “lightweight” classes collaborating to fulfill a responsibility

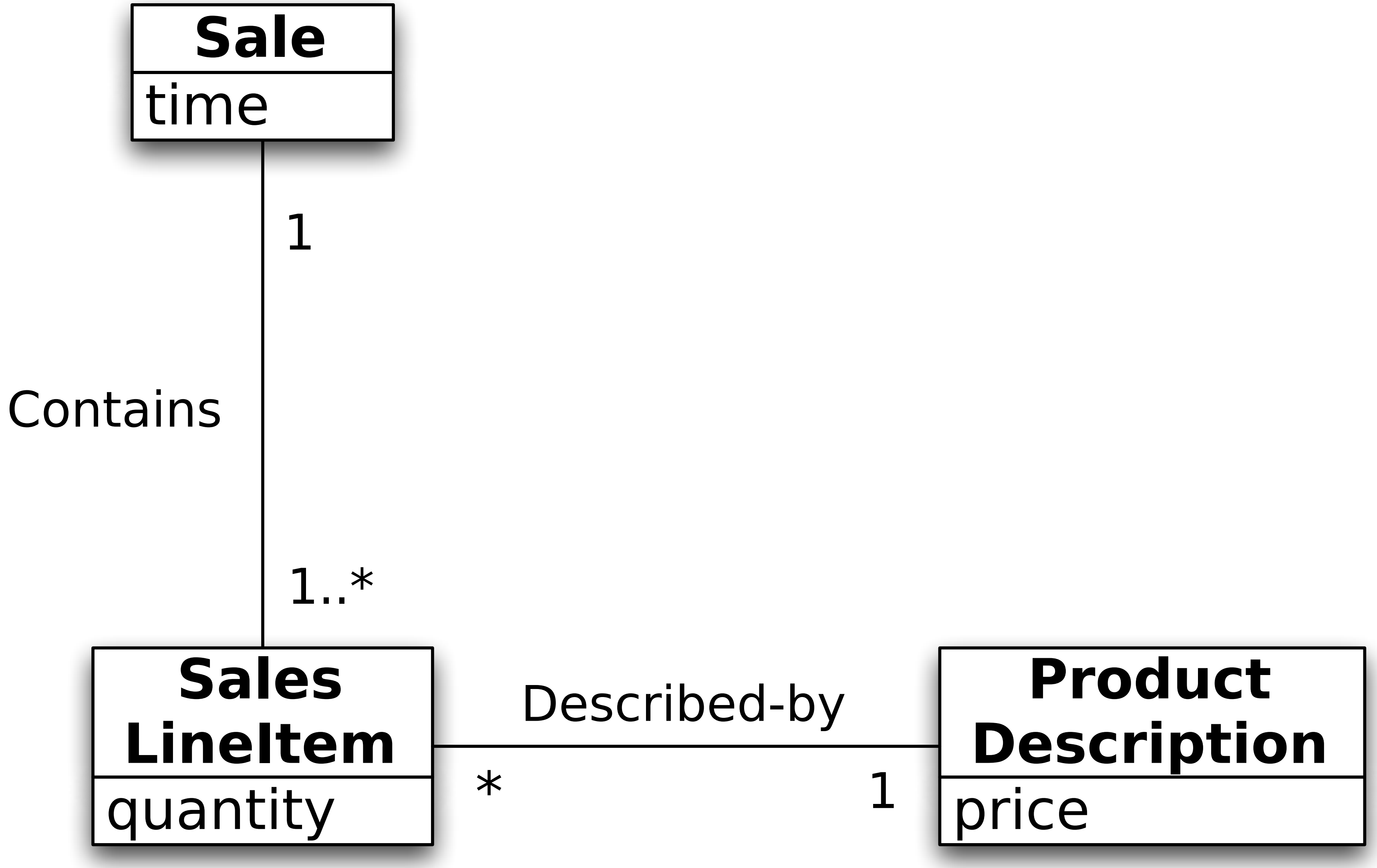
Who should be responsible for creating an instance of a class ?

Who should be responsible for creating an instance of a class ?

Assign to class B the responsibility to create an object of class A if the following is true:

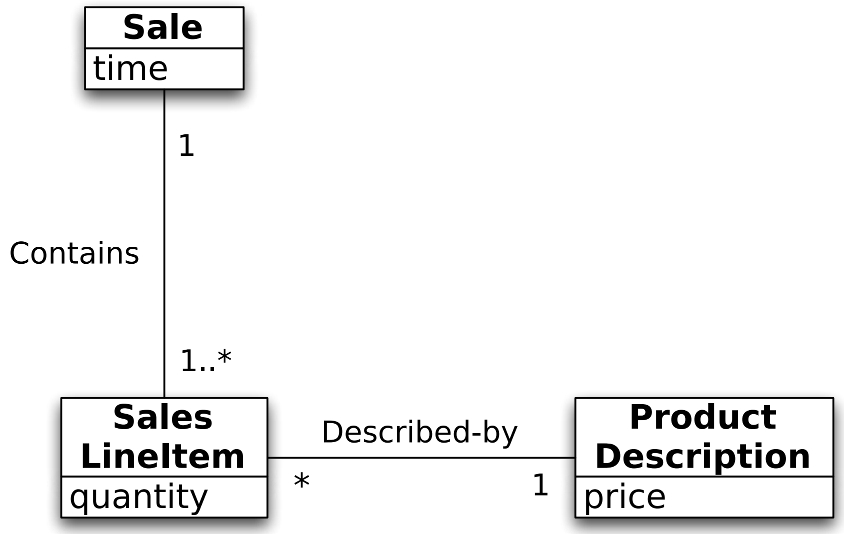
- B aggregates or (closely) uses objects of type A
- B records A
- B has the data to be passed to A when A is created
  B is an expert in the creation of A

GRASP - **Creator**



Who should be responsible for creating a SalesLineItem?

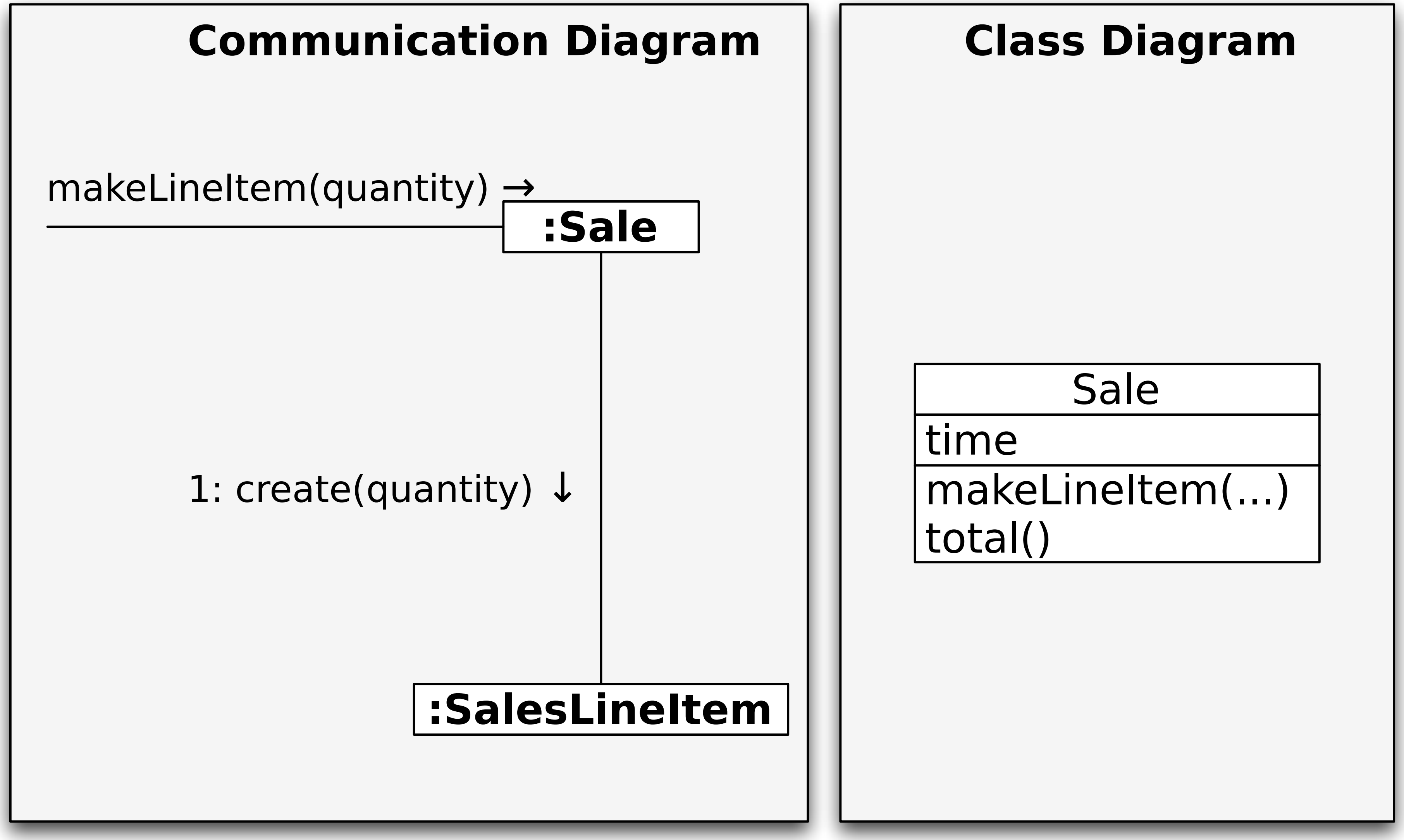
GRASP - **Creator**



Who should be responsible for creating a SalesLineItem?

- Sale contains SalesLineItem objects; hence, Sale is a good candidate for creating a SalesLineItem

GRASP - **Creator**

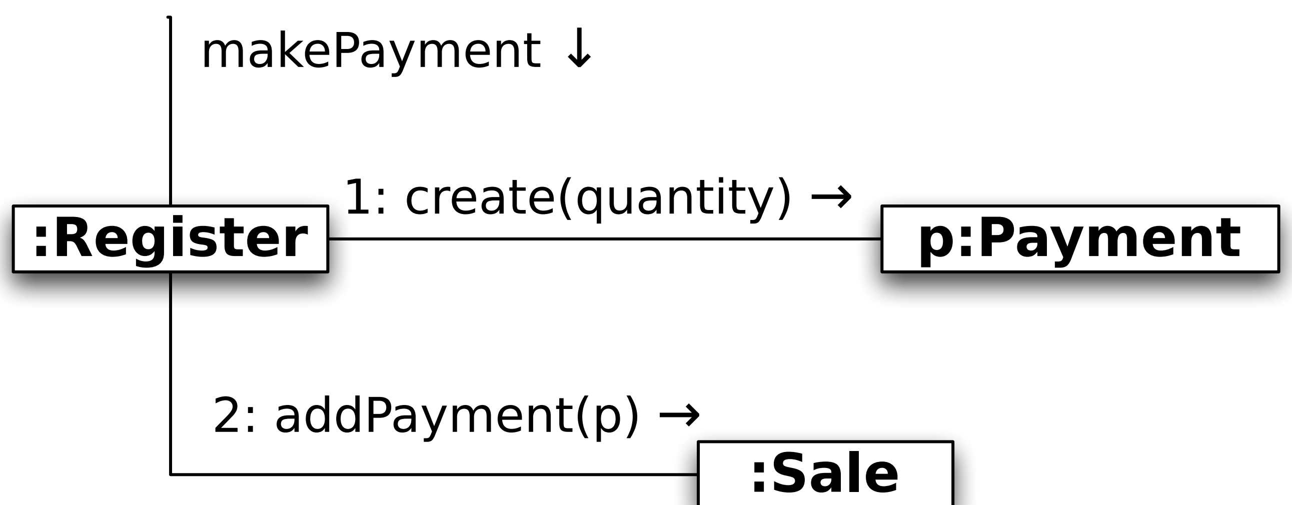


Communication diagram after assigning the responsibility for creating SalesLineItems to Sale.

GRASP - **Creator**

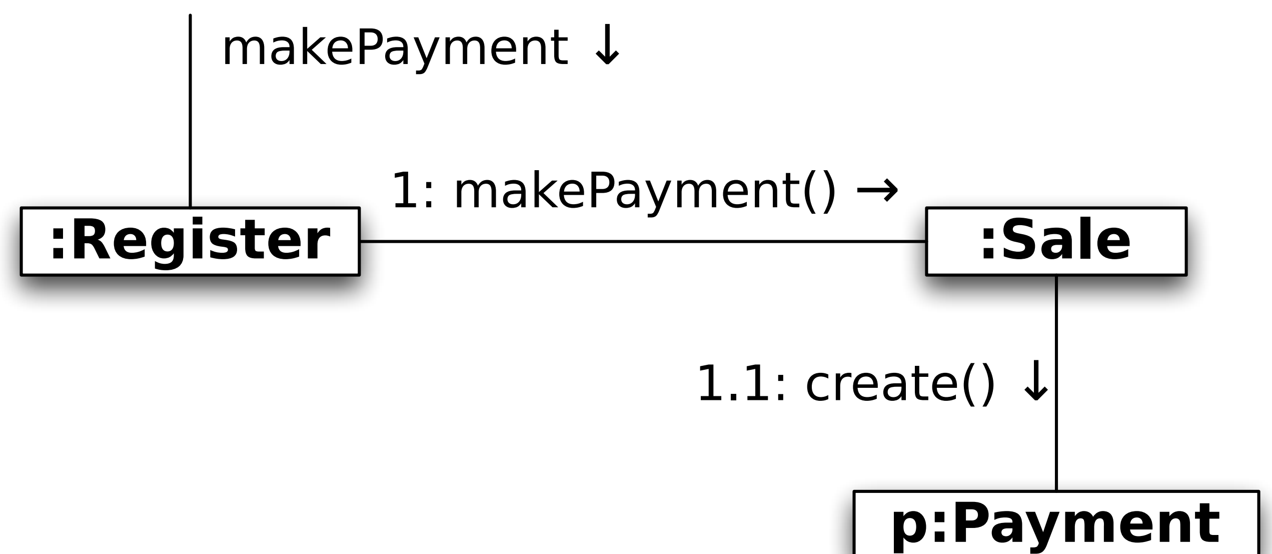
Variant A

Register creates an instance of Payment and passes it to Sale.
(Suggested by Creator as Register records Payments.)



Variant B

Sale creates an instance of Payment.
(Suggested by Creator as Sale uses Payment.)

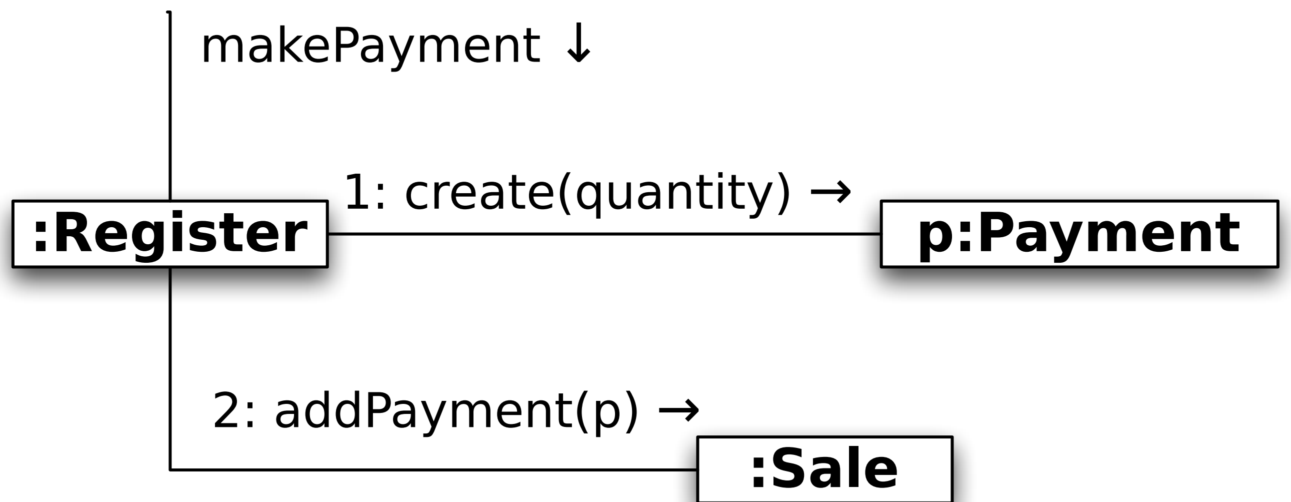


Which class should be responsible for creating a Payment?

GRASP - **Creator**

Variant A

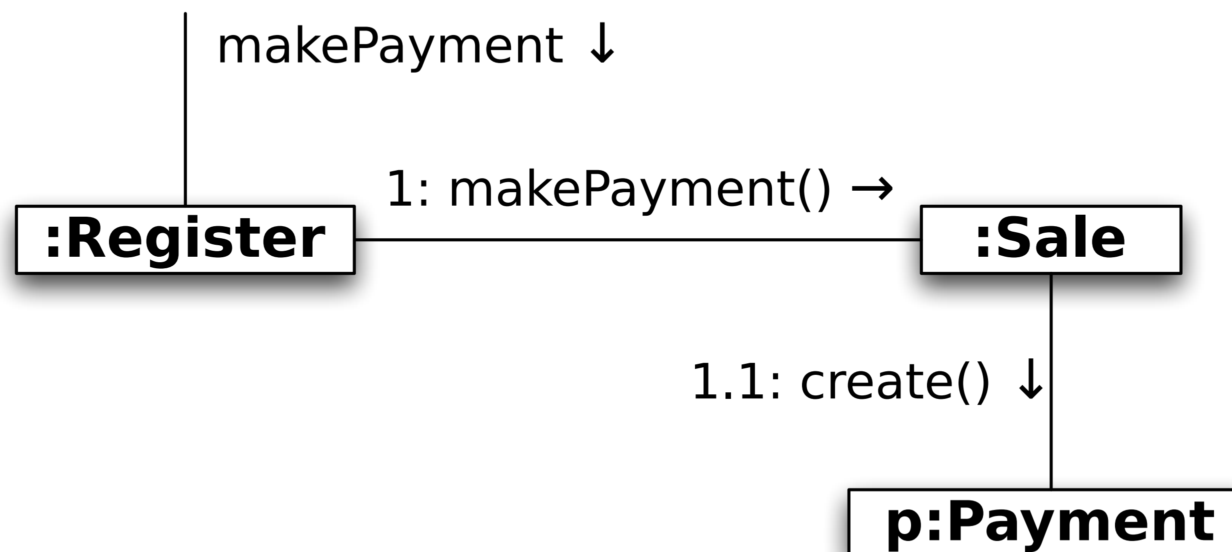
Register creates an instance of Payment and passes it to Sale.



Using this variant might lead to a non-cohesive class. If there are several system operations, and Register does some work related to each, it will be a large non-cohesive class.

Variant B

Sale creates an instance of Payment.



This variant supports both: high cohesion and low coupling.

Example

Designing **makeNewSale** of the ProcessSale Use Case

System Operation Contract

...	...
Preconditions	None
Postconditions	• a Sale instance s was created Instance creation • s was associated with the Register Association formed • the attributes of s are initialized

Example

Designing **makeNewSale** of the ProcessSale Use Case

Creating a New Sale Object

- Who should be responsible for creating a new instance of some class?

Creator

Example

Designing **makeNewSale** of the ProcessSale Use Case

Creating a New Sale Object

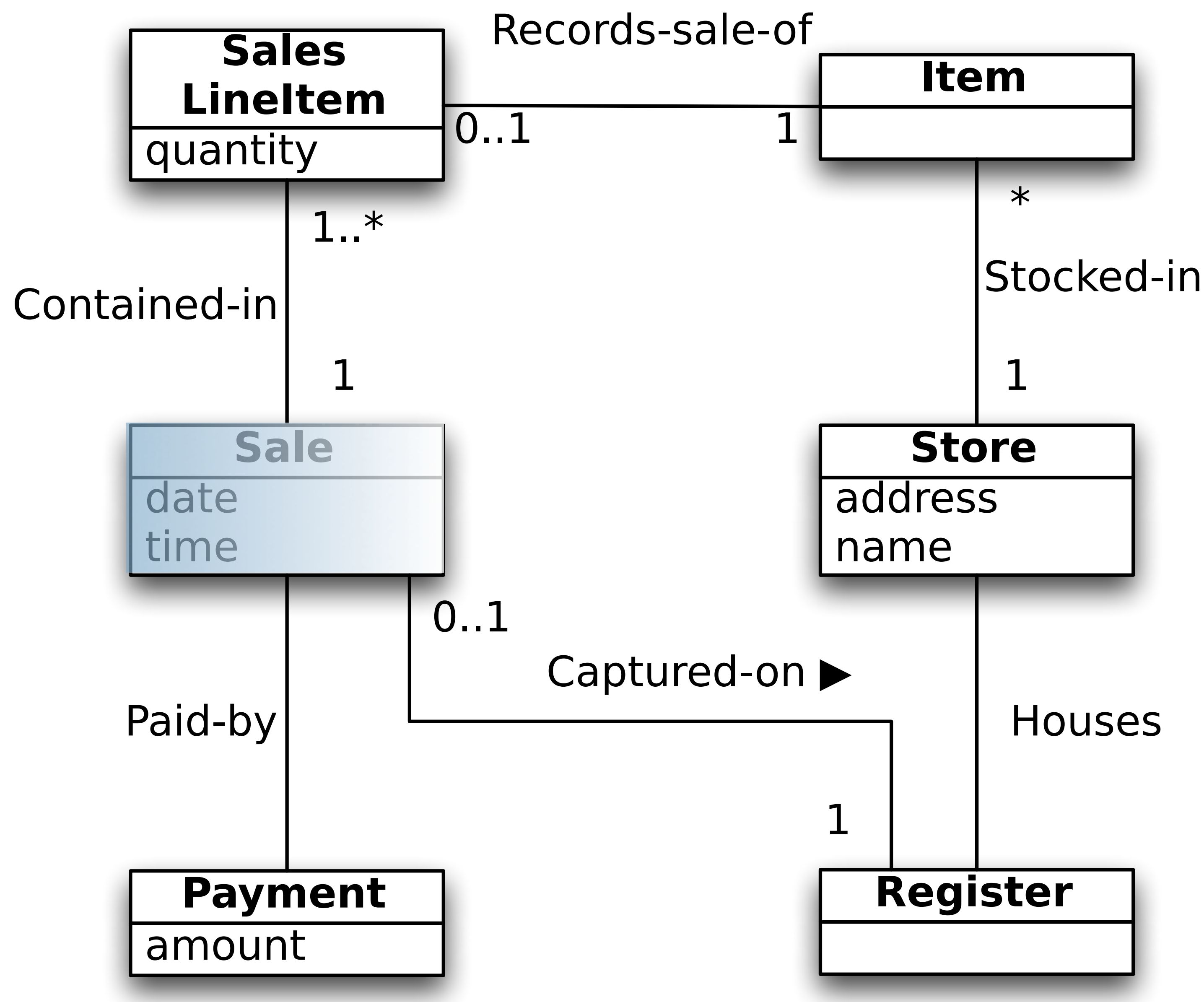
From the contract:

"... a Sale instance was created".

Creator suggests a class that...

- aggregates,
- contains or
- records

the object (Sale) to be created.



Example

Designing **makeNewSale** of the ProcessSale Use Case

Creating a New Sale Object

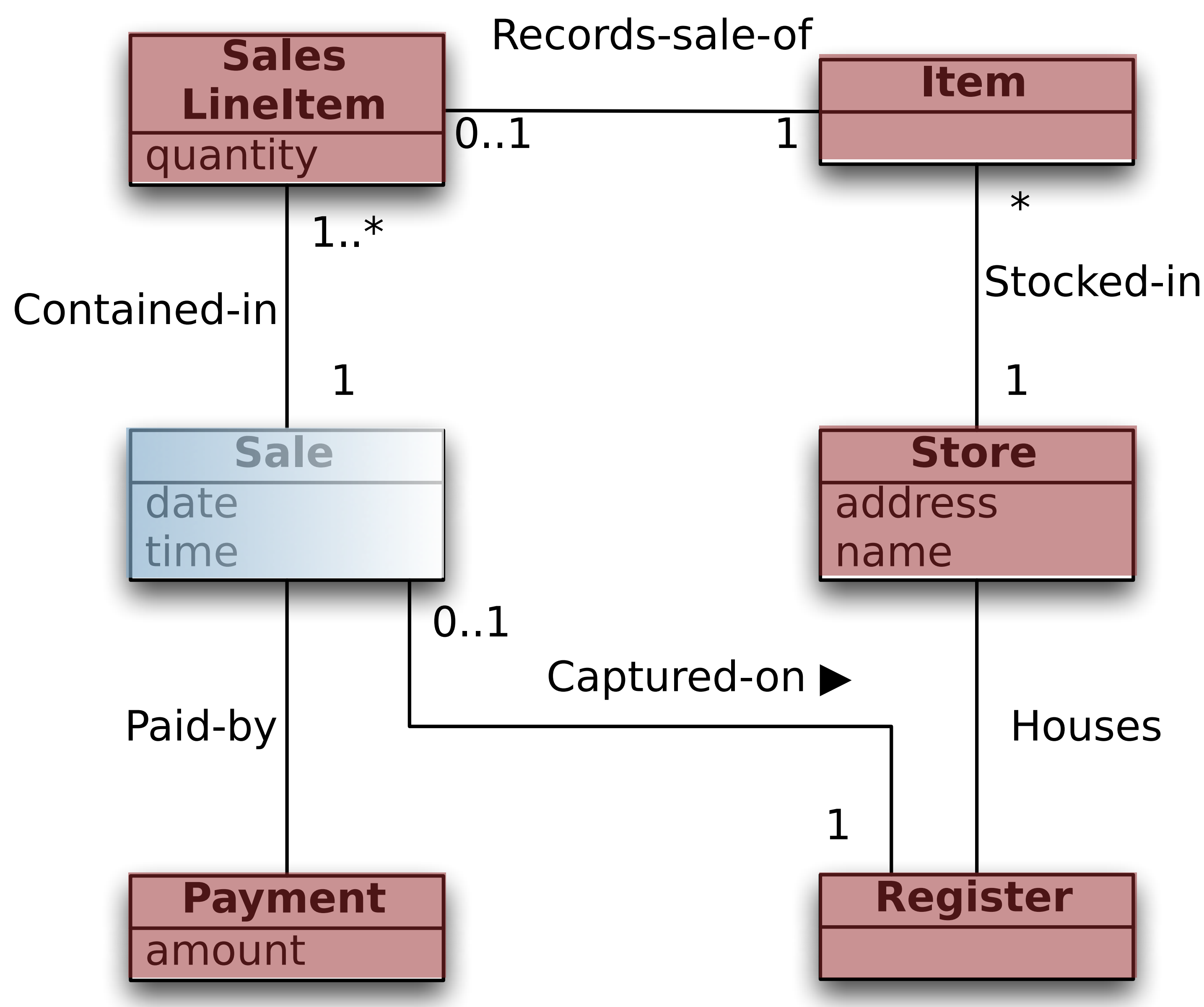
From the contract:

"… a Sale instance was created".

Creator suggests a class that…

- **aggregates**,
- contains or
- records

the object (Sale) to be created.



No Candidate

Example

Designing **makeNewSale** of the ProcessSale Use Case

Creating a New Sale Object

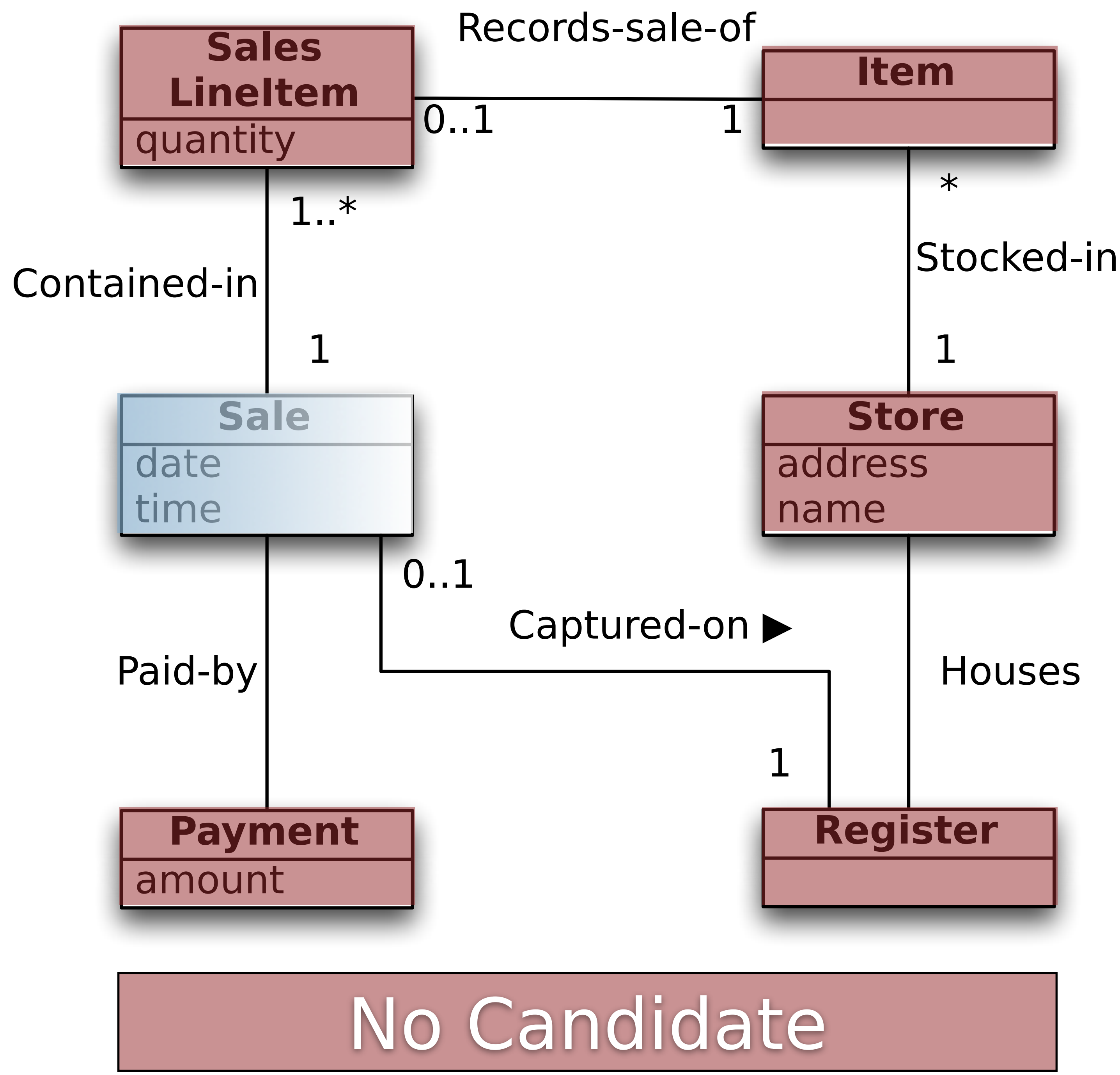
From the contract:

"… a Sale instance was created".

Creator suggests a class that…

- aggregates,
- **contains** or
- records

the object (Sale) to be created.



Example

Designing **makeNewSale** of the ProcessSale Use Case

Creating a New Sale Object

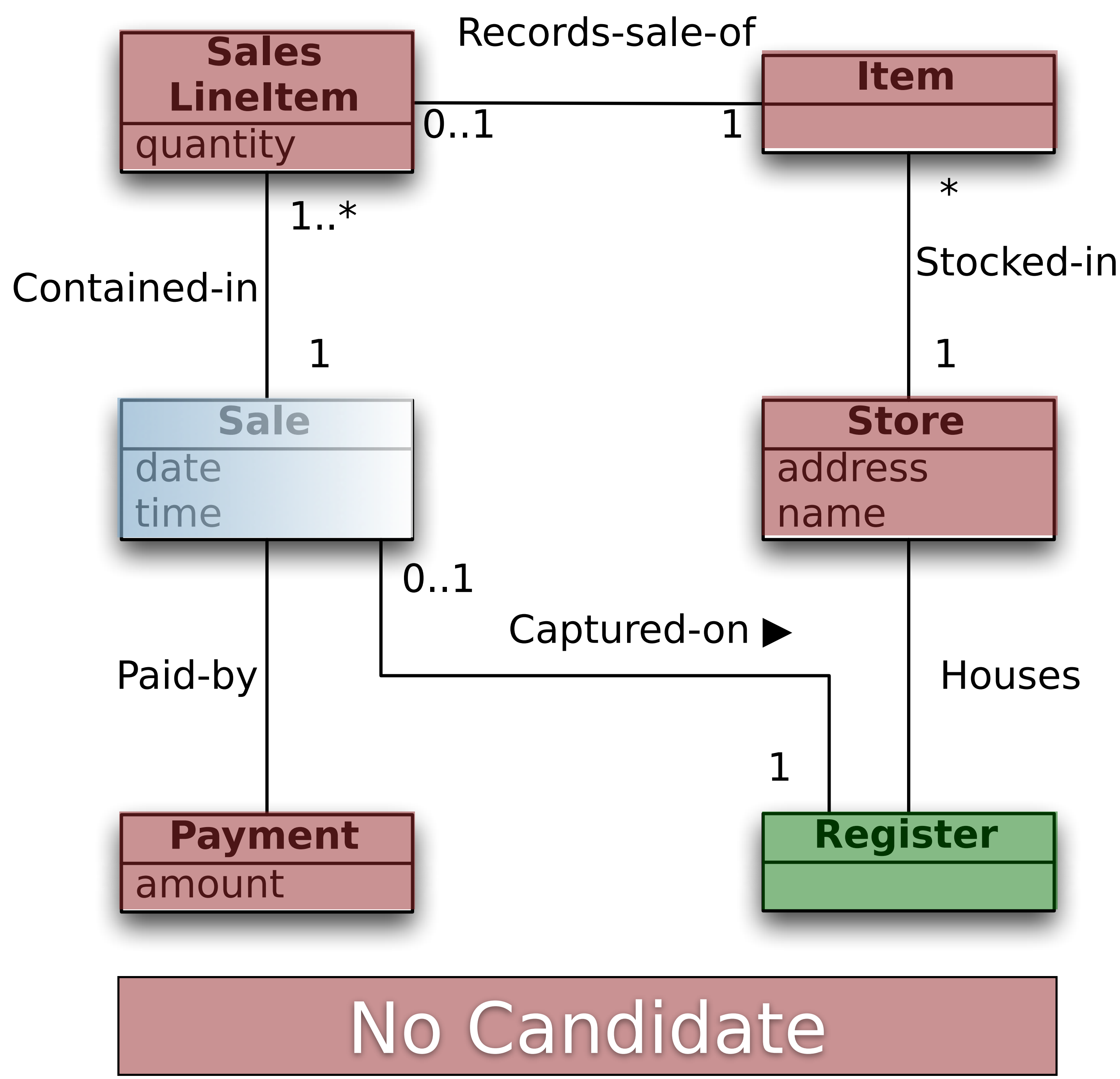
From the contract:

"… a Sale instance was created".

Creator suggests a class that…

- aggregates,
- contains or
- **records**

the object (Sale) to be created.

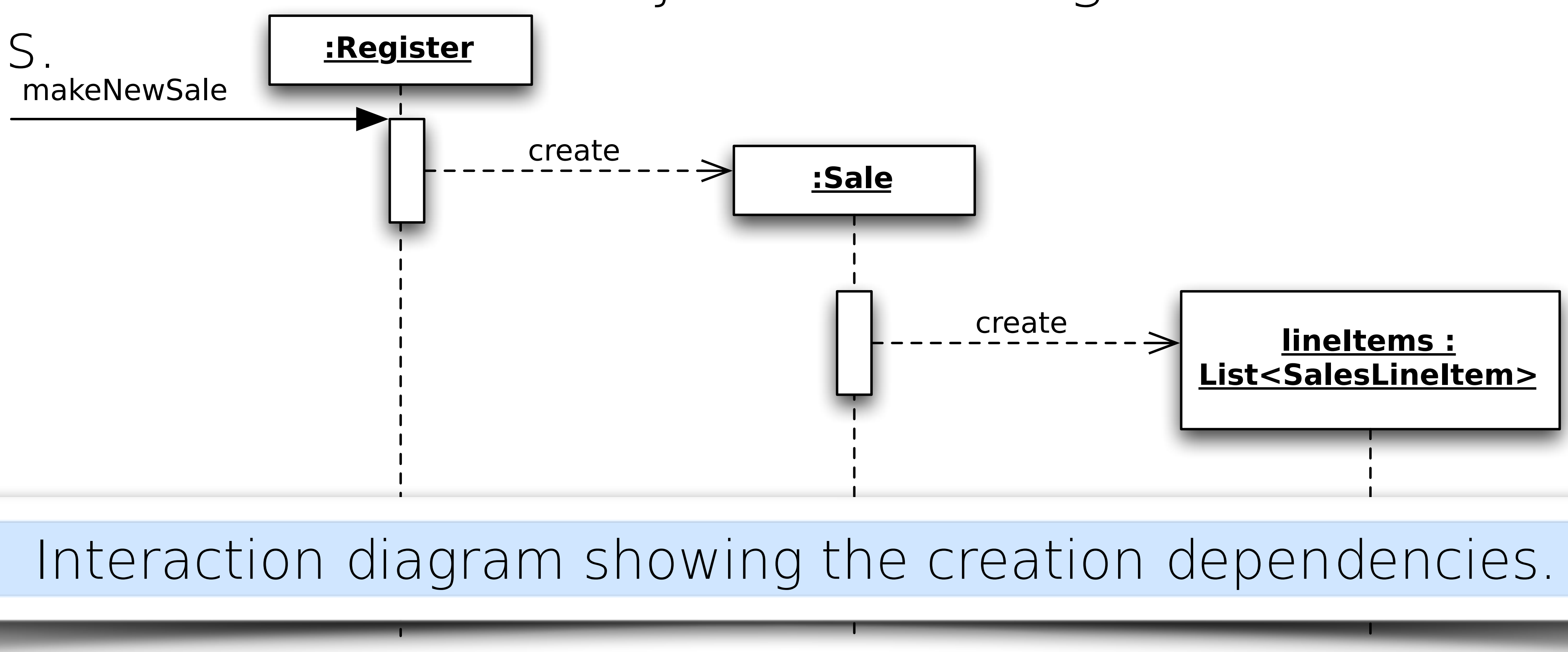


Example

Designing **makeNewSale** of the ProcessSale Use Case

From the contract:
"...the attributes of [the newly created Sale instance] are initialized."

Since a **Sale** will also contain **SalesLineItems** it is necessary to further create a **List** object for storing the sale line items.



Interaction diagram showing the creation dependencies.

Design Heuristi

AB HIER (jan. 2012)

- J. Riel; Object-Oriented Desig dison-Wesley, 1996

TECHNISCHE UNIVERSITÄT DARMSTADT

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 = …;
// check if both lines are parallel
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.

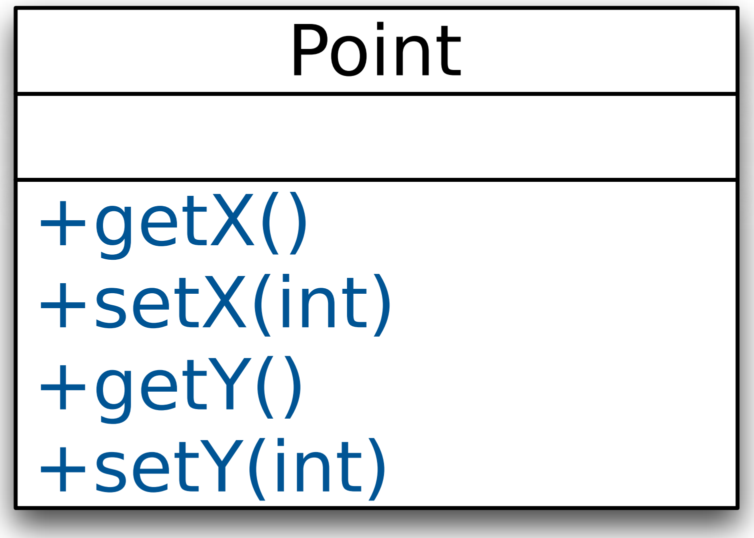
The God Class Problem

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.

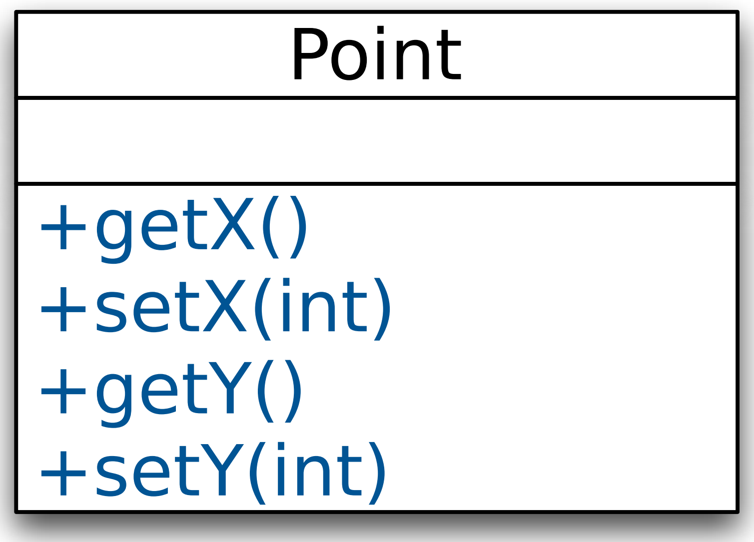
Beware of classes that have too much noncommunicating behavior, that is, methods that operate on a proper subset of the data members of a class. God classes often exhibit much noncommunicating behavior.

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

Point
+getX() +setX(int) +getY() +setY(int)

But, still there is an issue. What is it?

- **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.

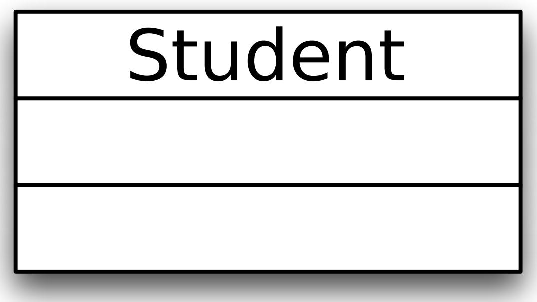
Two Reasonable Explanations For the Need of Accessor Methods...

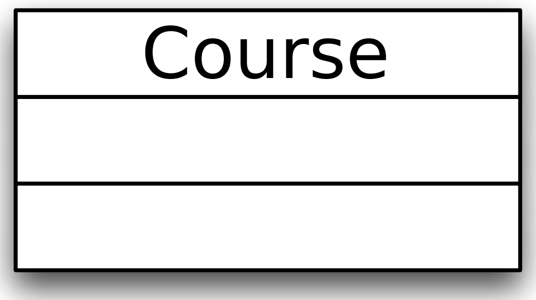
- ... a **class performing the gets and sets is implementing a policy**
  (policy = dt. Verfahren(-sweise))

- ... 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



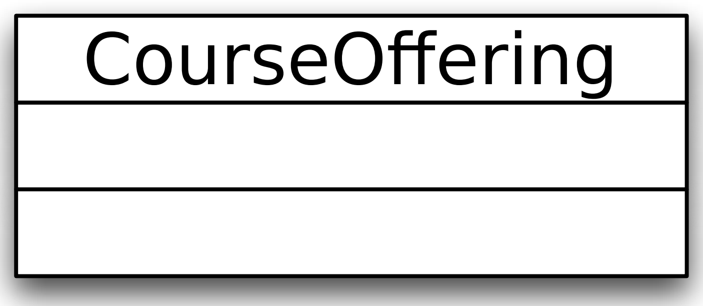


Captures **static information about students**, e.g., name, identification number, list of courses (s)he has taken, etc.

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

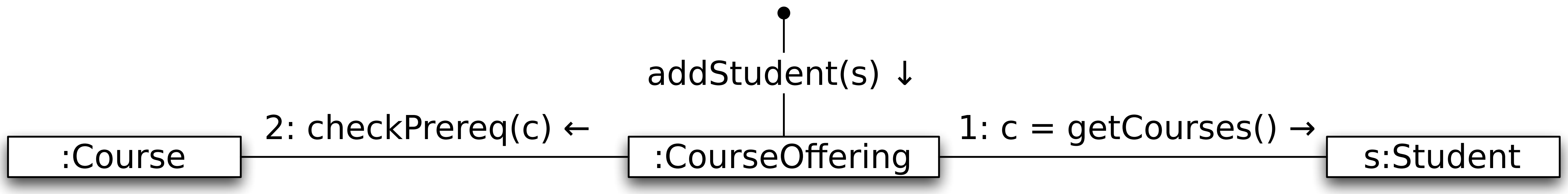
Example from the Course-scheduling Domain



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

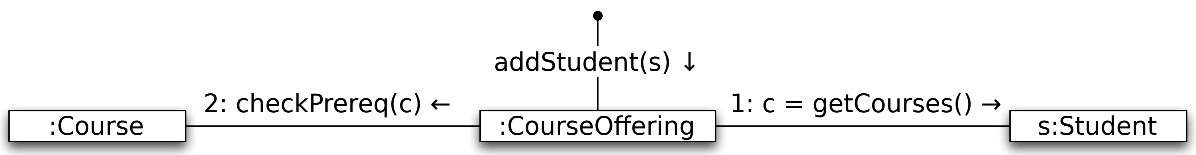
Example from the Course-scheduling Domain

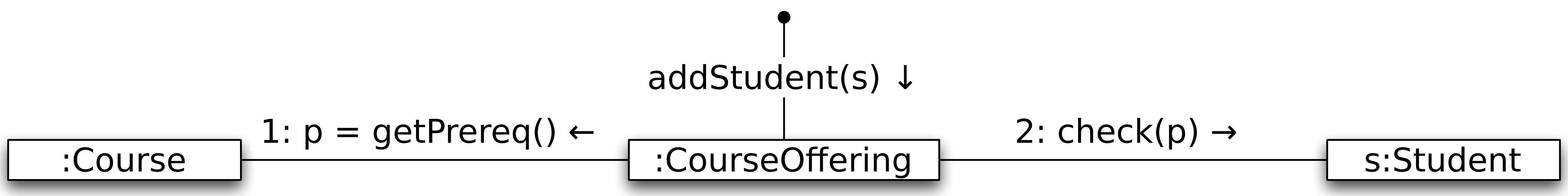


First design for checking the prerequisites of students

Implementing Policies Between Two or More Classes

Example from the Course-scheduling Domain

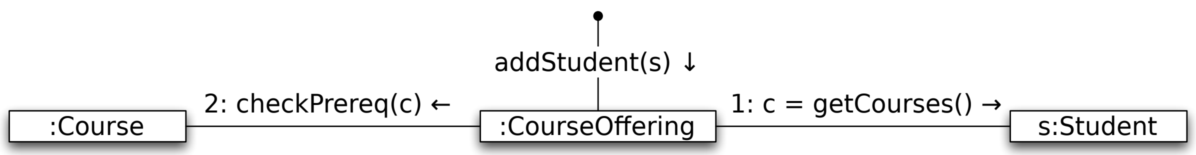


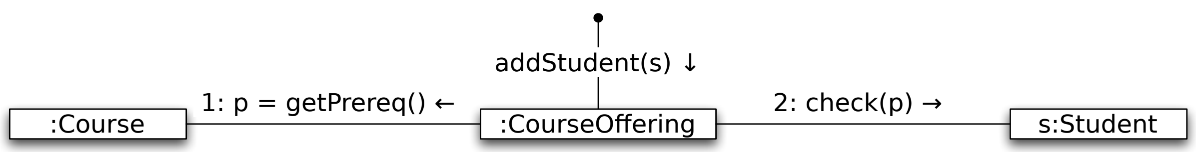


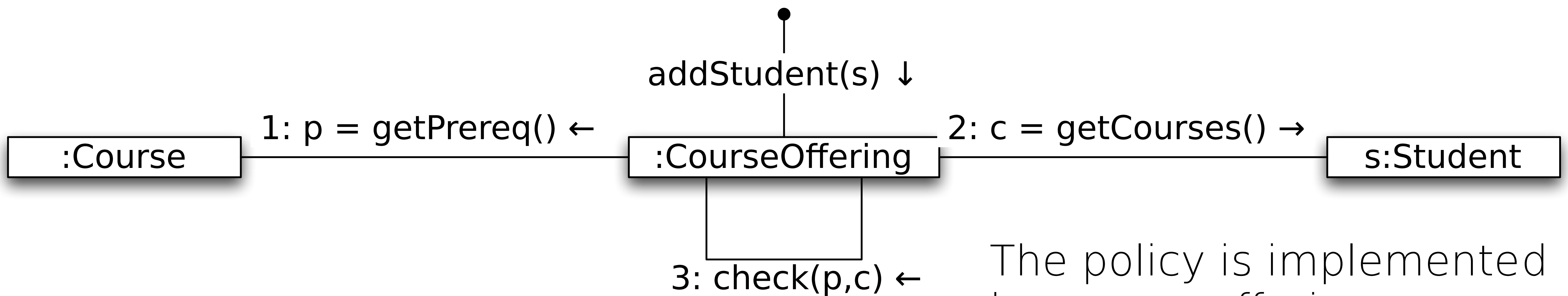
Second design for checking the prerequisites of students

Implementing Policies Between Two or More Classes

Example from the Course-scheduling Domain





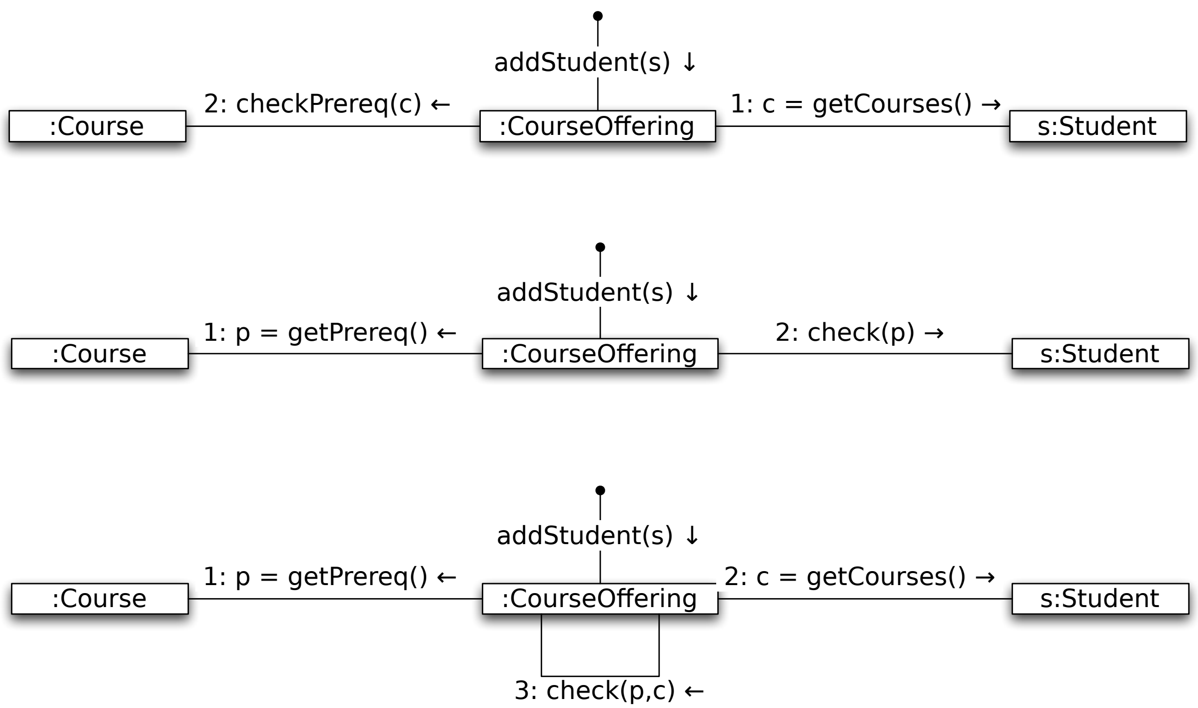


The policy is implemented by course offering.

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

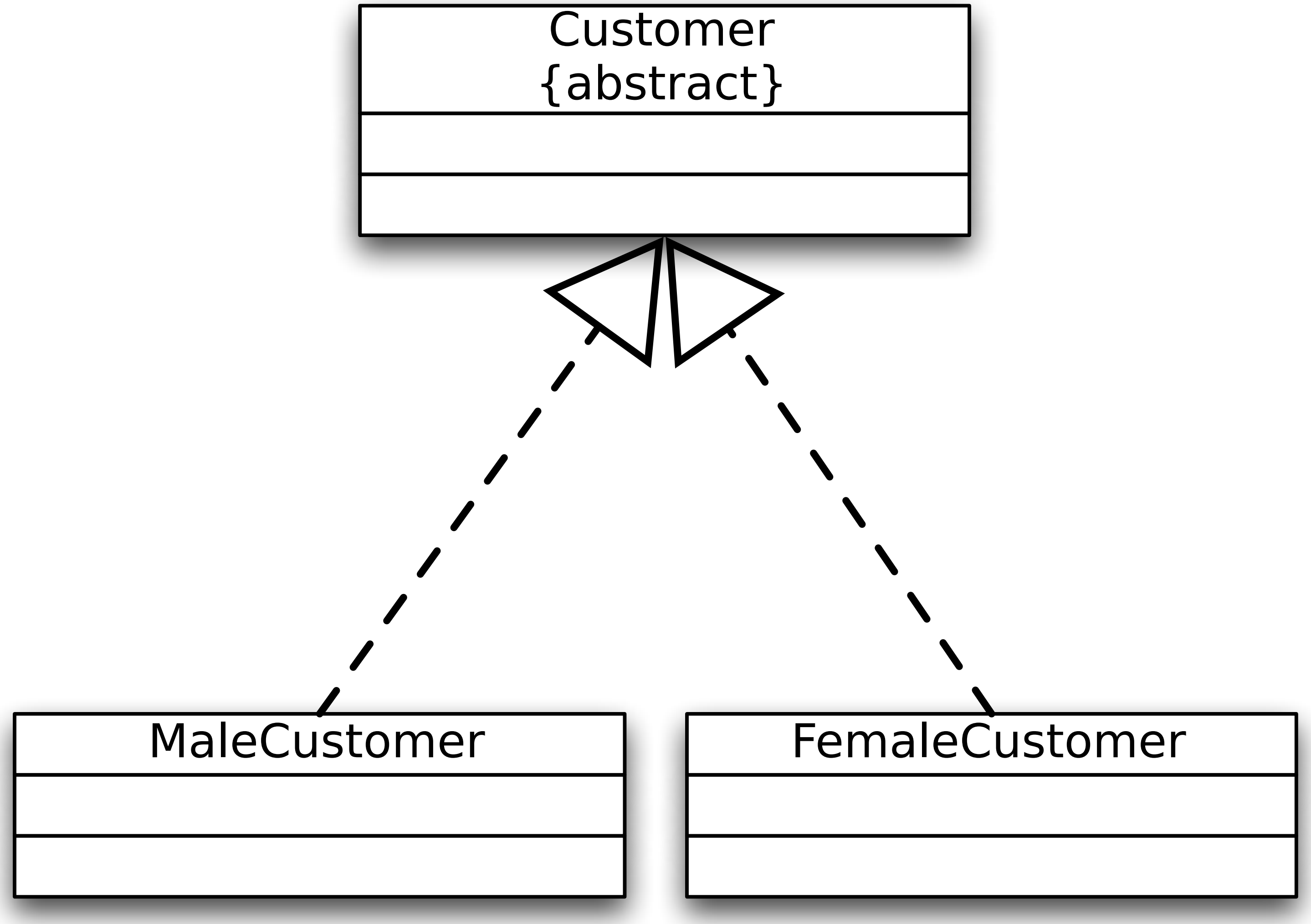
`class Person {…}`
`class Father extends Person {…}`
`class Mother extends Person {…}`
	`class Person {…}`

`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

TECHNISCHE
UNIVERSITÄT
DARMSTADT

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.