Software Engineering I (02161)

Week 11

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2017

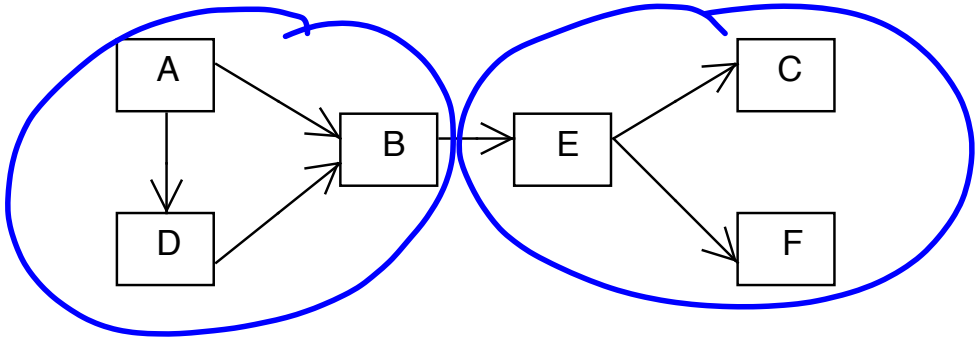
Recap

- Software Development Processes (cont.)
- Project Planning
- Design by Contract

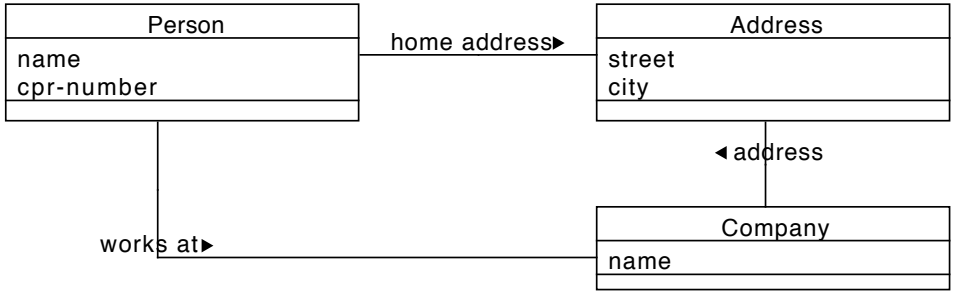
Contents

Low Copuling and High Cohesion

Low coupling



High Cohesion



→ Corner stones of good design

→ Layered Architecture

Law of Demeter

Law of Demeter

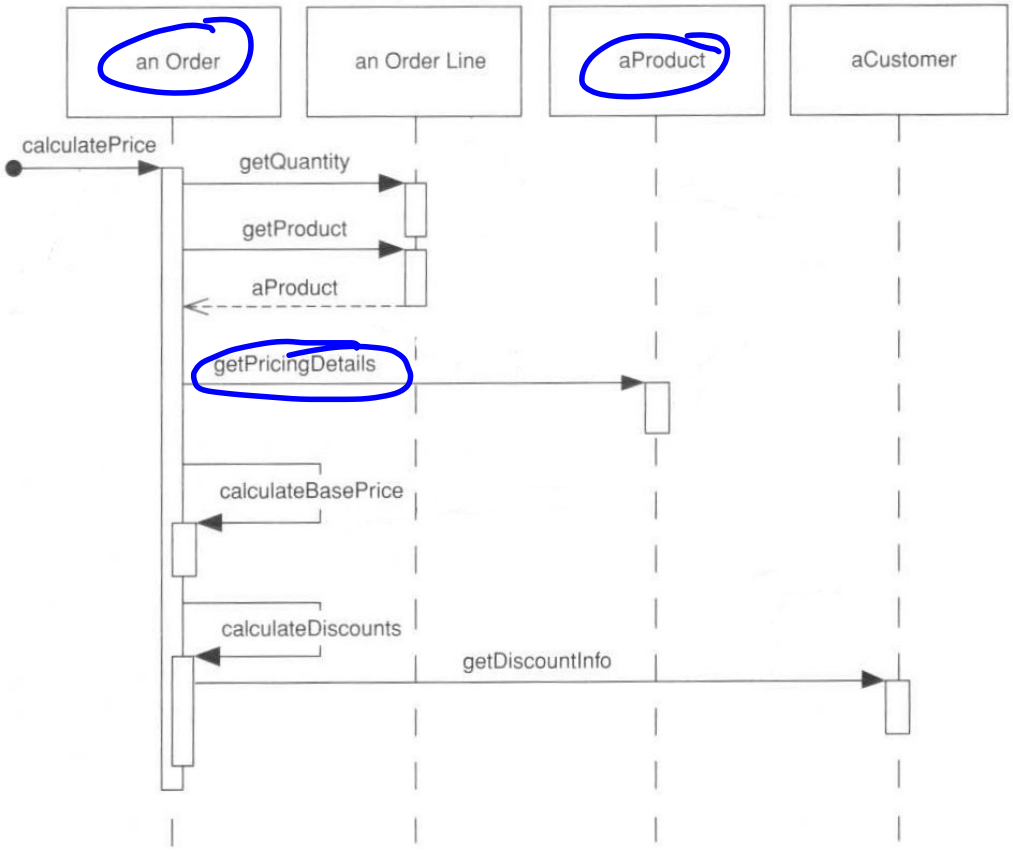
- ”Only talk to your immediate friends”
- Only method calls to the following objects are allowed
  - the object itself
  - its components
  - objects created by that object
  - parameters of methods

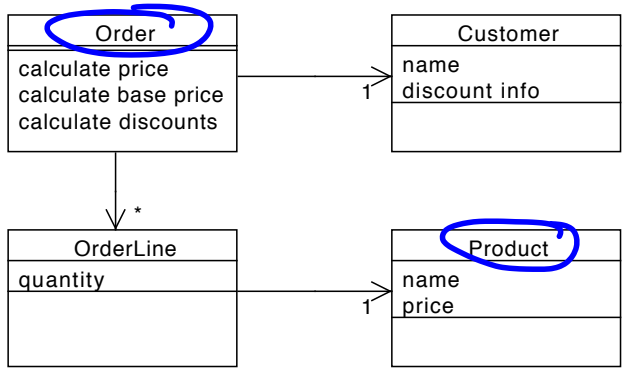
- Also known as: **Principle of Least Knowledge**
- Law of Demeter = **low coupling**

$\rightarrow$ **delegate functionality**

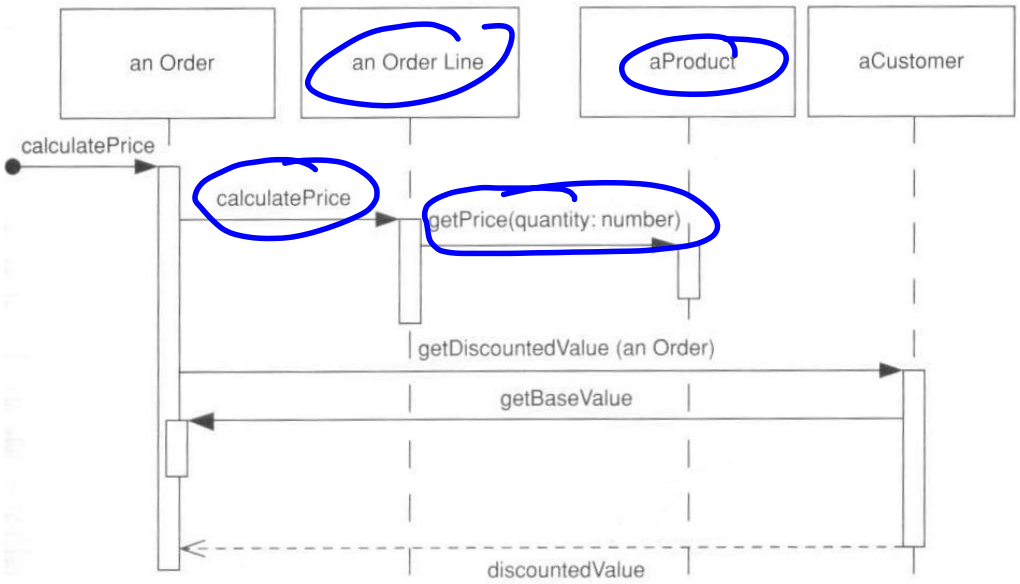
$\rightarrow$ decentralised control

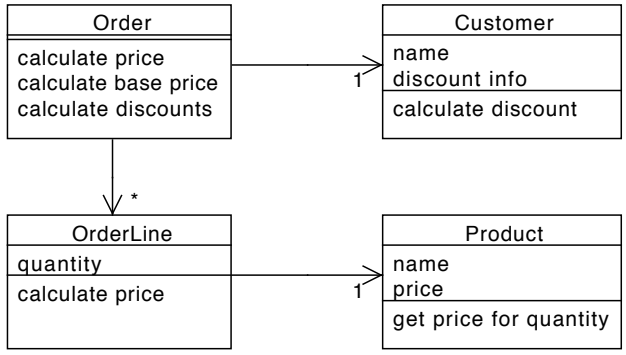
Computing the price of an order





Computing the price of an order





DRY principle

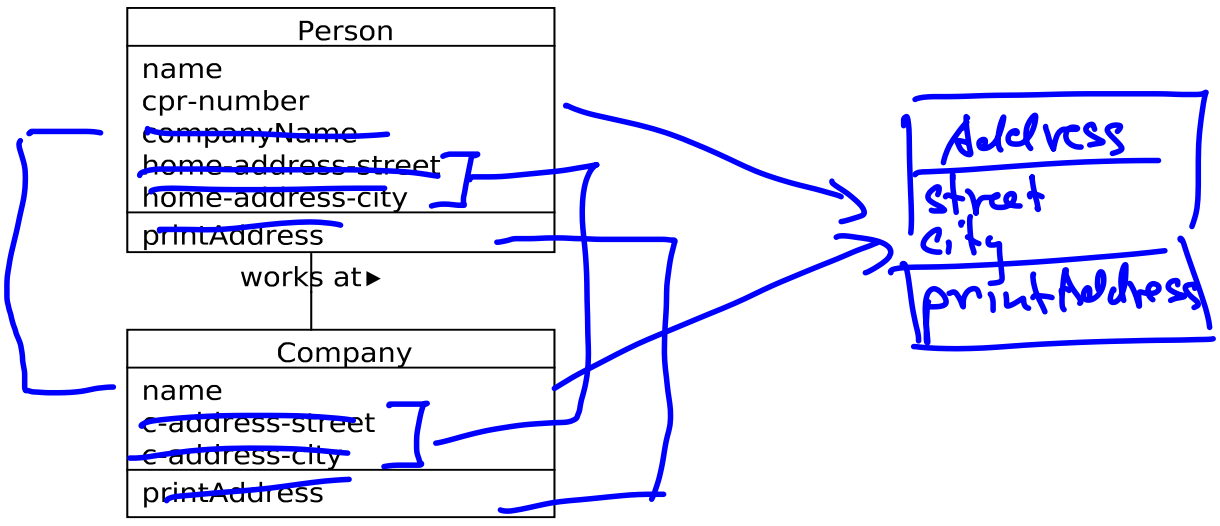
DRY principle

Don’t repeat yourself

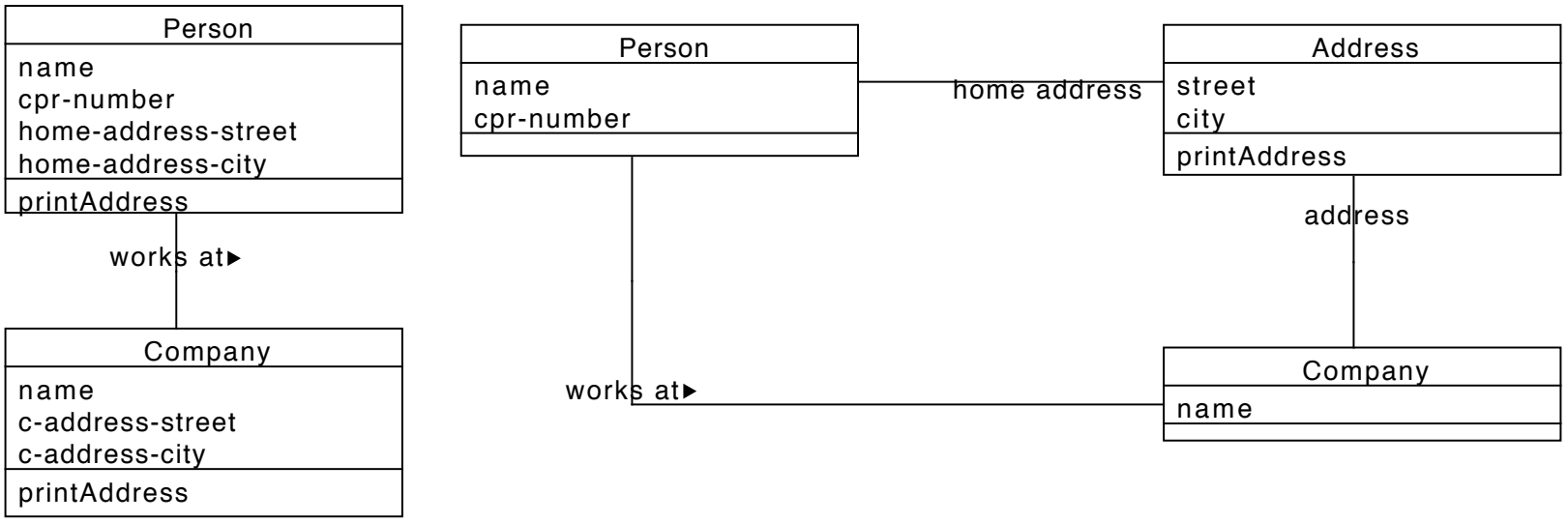
”Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.” The Pragmatic Programmer, Andrew Hunt and David Thomas

- code
- documentation
- build stystem

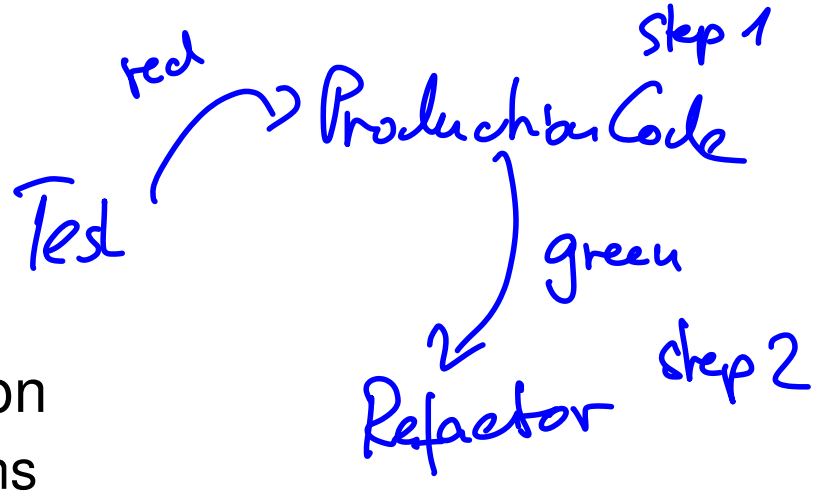
Example: Code Duplication



Example: Code Duplication



DRY principle



- Techniques to avoid duplication
  - Use appropriate abstractions
  - Inheritance
  - Classes with instance variables
  - Methods with parameters
- Refactor to remove duplication
- Generate artefacts from a common source. Eg. Javadoc

KISS principle

KISS principle

Keep it short and simple (sometimes also: Keep it simple, stupid)

- **simplest solution** *first*
- **Strive** for **simplicity**
  - **Takes time**!!
  - *refactor* for **simplicity**

Antoine de Saint Exupéry

”It seems that perfection is reached not when there is nothing left to add, but when there is nothing left to take away”.

Contents

Patterns in Architecture

number → Name of the pattern

182 EATING ATMOSPHERE

Context

. . . we have already pointed out how vitally important all kinds of communal eating are in helping to maintain a bond among a group of people—COMMUNAL EATING (147); and we have given some idea of how the common eating may be placed as part of the kitchen itself—FARMHOUSE KITCHEN (139). This pattern gives some details of the eating atmosphere.

❖ ❖ ❖

Problem

When people eat together, they may actually be together in spirit—or they may be far apart. Some rooms invite people to eat leisurely and comfortably and feel together, while others force people to eat as quickly as possible so they can go somewhere else to relax.

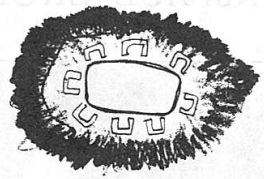
Forces

Above all, when the table has the same light all over it, and has the same light level on the walls around it, the light does nothing to hold people together; the intensity of feeling is quite likely to dissolve; there is little sense that there is any special kind of gathering. But when there is a soft light, hung low over the table, with dark walls around so that this one point of light lights up people's faces and is a focal point for the whole group, then a meal can become a special thing indeed, a bond, communion.

Therefore:

Solution

Put a heavy table in the center of the eating space—large enough for the whole family or the group of people using it. Put a light over the table to create a pool of light over the group, and enclose the space with walls or with contrasting darkness. Make the space large enough so the chairs can be pulled back comfortably, and provide shelves and counters close at hand for things related to the meal.



light in the middle

❖ ❖ ❖

Get the details of the light from POOLS OF LIGHT (252); and choose the colors to make the place warm and dark and comfortable at night—WARM COLORS (250); put a few soft chairs nearby—DIFFERENT CHAIRS (251); or put BUILT-IN SEATS (202) with big cushions against one wall; and for the storage space—OPEN SHELVES (200) and WAIST-HIGH SHELF (201). . . .

↑ Related Pattern

A Pattern Language, Christopher Alexander, 1977

Pattern and pattern language

- Christopher Alexander: Architecture (1977/1978)
  - Pattern: a *solution* to a *problem* in a context
  - Pattern language: set of related patterns
- Kent Beck and Ward Cunningham: Patterns for Smalltalk applications (1987)

Pattern: ”Objects from the User’s World”

Problem: What are the best objects to start a design with?

Constraints: The way the user sees the world should have a profound impact on the way the system presents information. Sometimes a computer program can be a user’s bridge to a deeper understanding of a domain. However, having a a software engineer second guess the user is a chancy proposition at best.

Kent Beck: ”Birds, Bees, and Browsers—Obvious sources of Objects” 1994 `http://bit.ly/2q4h0GC`

Pattern: ”Objects from the User’s World”

Forces:

- Some people say, ”I can structure the internals of my system any way I want to. What I present to the user is just a function of the user interface.” In my experience, this is simply not so. The structure of the internals of the system will find its way into the thoughts and vocabulary of the user in the most insidious way. Even if it is communicated only in what you tell the user is easy and what is difficult to implement, the user will build a mental model of what is inside the system.
- Unfortunately, the way the user thinks about the world isn’t necessarily the best way to model the world computationally. In spite of the difficulties, though, it is more important to present the best possible interface to the user than to make the system simpler to implement.

Therefore:

Pattern: ”Objects from the User’s World”

Solution: Begin the system with objects from the user’s world. Plan to decouple these objects from the way you format them on the screen, leaving only the computational model.

History of Patterns

- Christopher Alexander: Architecture (1977/1978)
- Kent Beck and Ward Cunningham: Patterns for Smalltalk applications (1987)
- Ward Cunningham: Portland Pattern Repository
  `http://c2.com/ppr`
  - the Wiki Wiki Web was invented for this purpose
- Gang of four: Design Patterns book (1994) (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)

Design Patterns

- Defined in the Design Pattern Book (1994)
- Best practices for object-oriented software
  - → use of *distributed control*
- Creational Patterns
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton
- Structural Patterns
  - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
- Behavioral Patterns
  - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor
- There are more: Implementation Patterns, Architectural Patterns, Analysis Patterns, Domain Patterns ...

Anti Patterns → "Don't do"

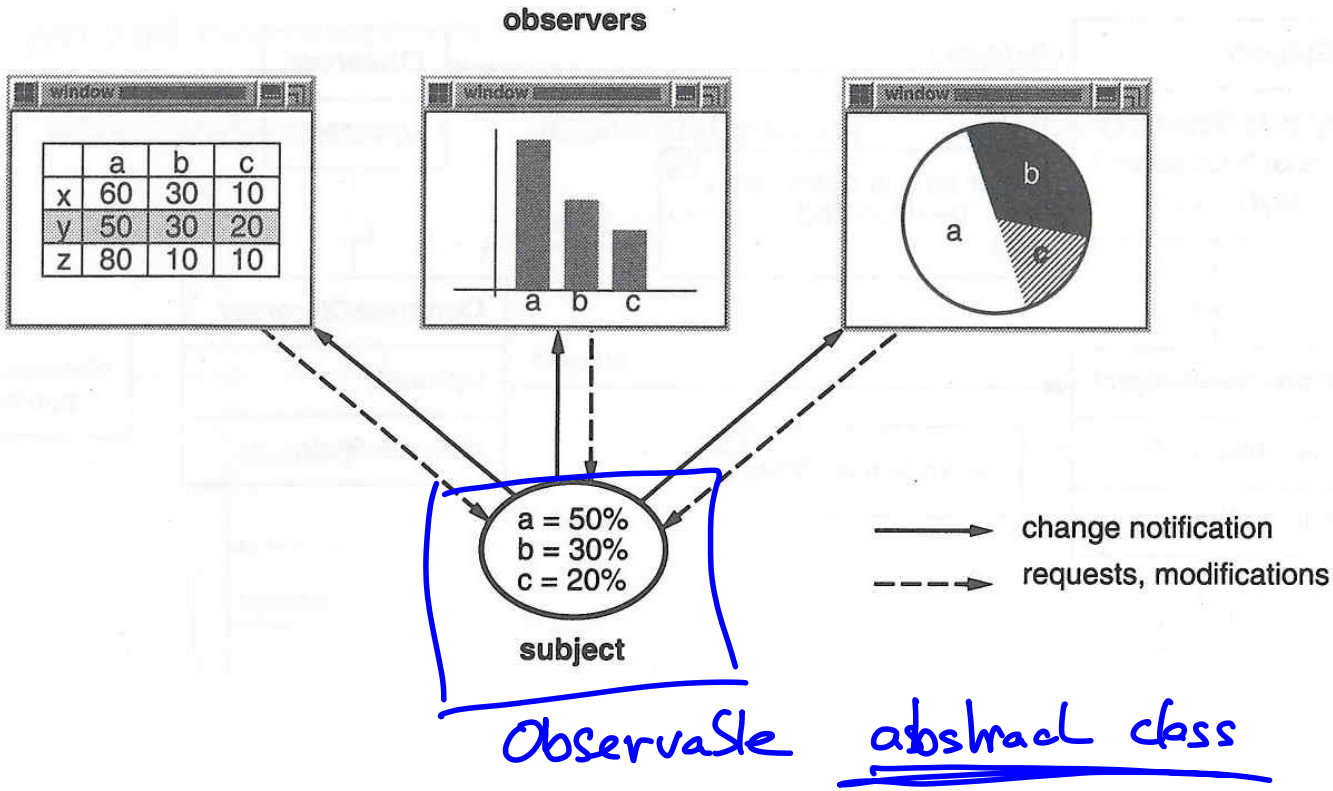
Places to find design patterns:

- Portland Pattern repository `http://c2.com/cgi/wiki?PeopleProjectsAndPatterns` (since 1995)
- Wikipedia `http://en.wikipedia.org/wiki/Design_pattern_(computer_science)`
- Wikipedia `http://en.wikipedia.org/wiki/Category:Software_design_patterns`

Observer Pattern

Observer Pattern

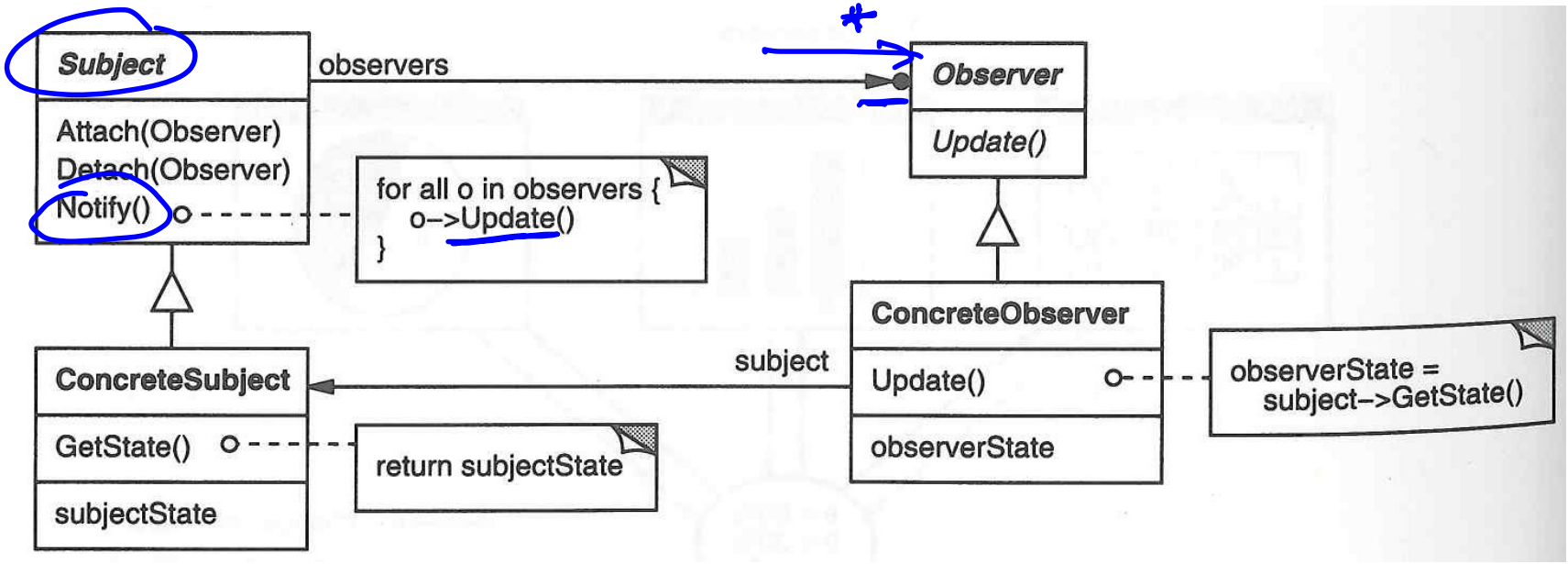
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



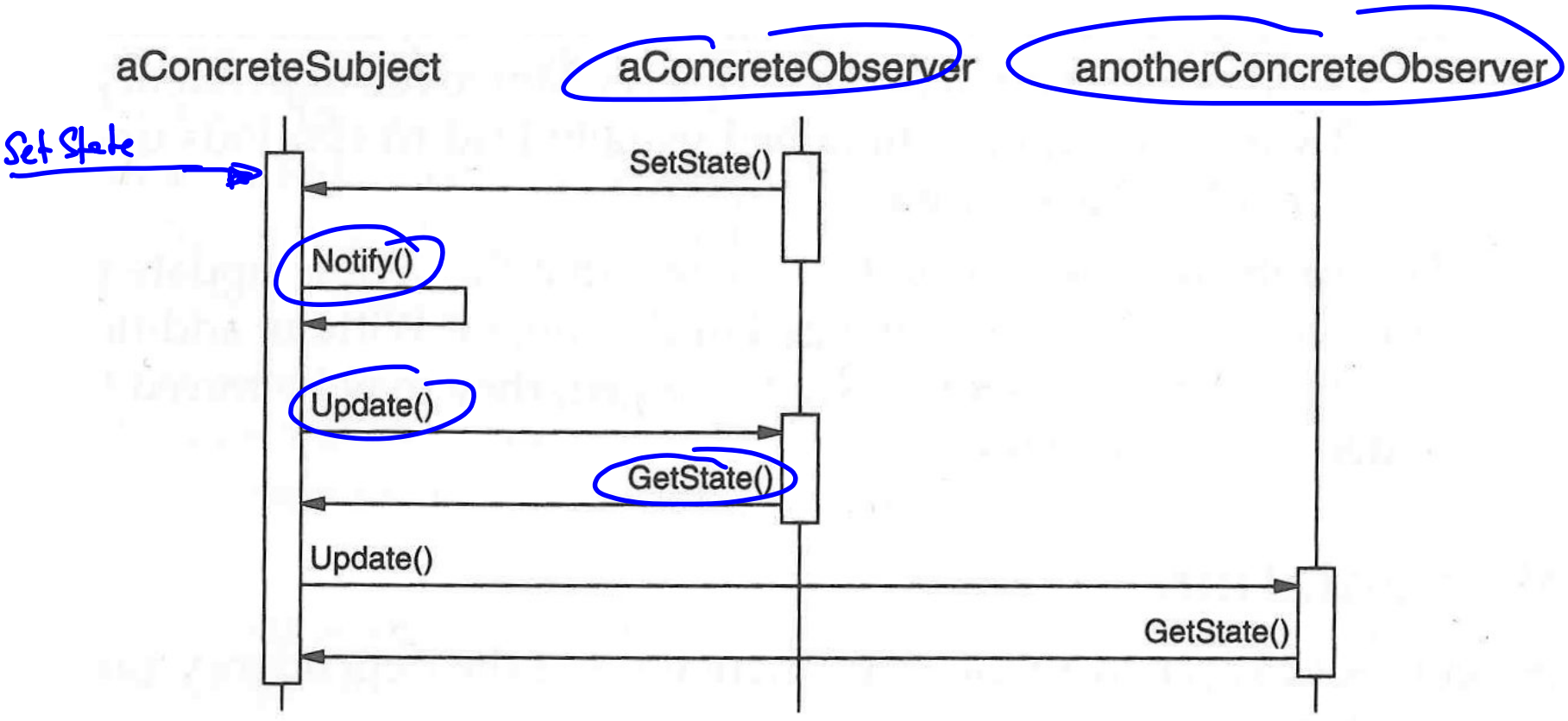
Observer Pattern

OMT pre UML class diagram notacha

Observable



Observer Pattern



Implementation in Java

- java.util.Observer: Interface
  - update(Observable o, Object aspect)
- java.util.Observable: Abstract class
  - addObserver, deleteObserver
  - setChanged
  - notifyObservers(Object aspect)

Example: Stack with observers

```
public class MyStack<E> extends Observable {
   List<E> data = new ArrayList<E>();

   void push(Type o) {
      data.add(o);
      setChanged();
      notifyObserver("data elements");
   }

   E pop() {
      E top = data.remove(data.size())'
      setChanged();
      notifyObserver("data elements");
   }

   E top() {
      return data.get(data.size());
   }

   int size() {
      return data.size();
   }
   String toString() {
      System.out.print("[");
      for (E d : data) { System.out.print(" "+d);
      System.out.print(" ]");
   }
   ...
```

Example: Stack observer

- Observe the number of elements that are on the stack.
- Each time the stack changes its size, a message is printed on the console.

```
class NumberOfElementsObserver() implements Observer {

   public void update(Observable o, Object aspect) {
      System.out.println(((MyStack)o).size()+
            " elements on the stack");
   }
}
```

- Observe the elements on the stack.
- Each time the stack changes, print the elements of the stack on the console.

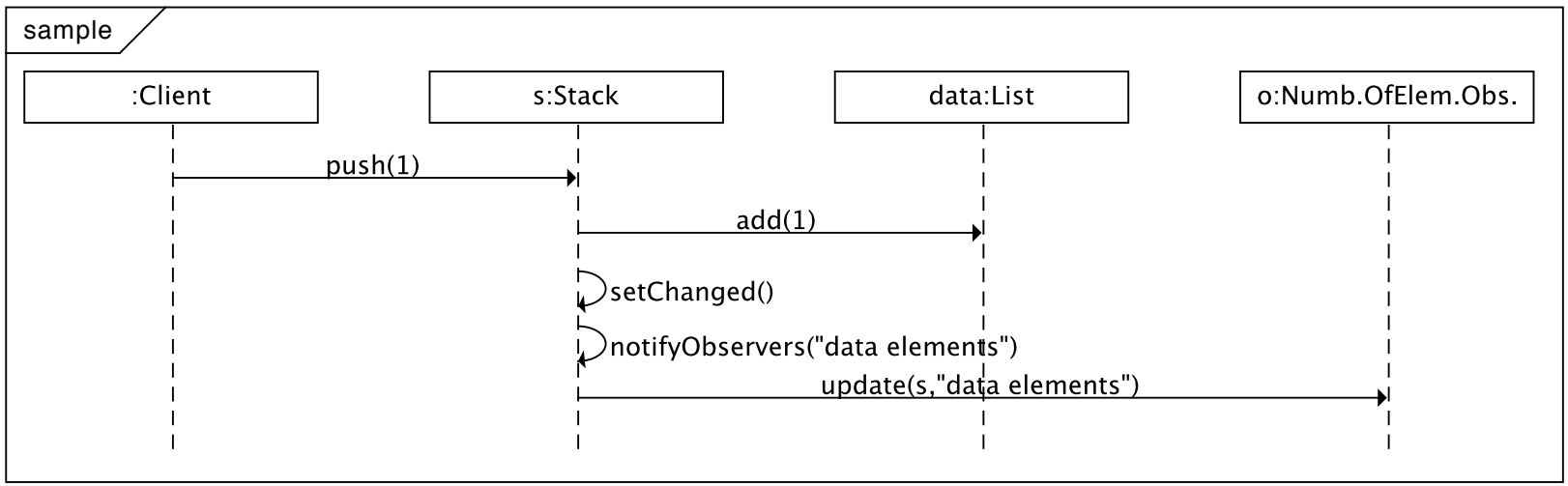
```
class StackObserver() implements Observer {

   public void update(Observable o, Object aspect) {
      System.out.println(o);
   }
}
```

Example: Stack observer

Adding an observer

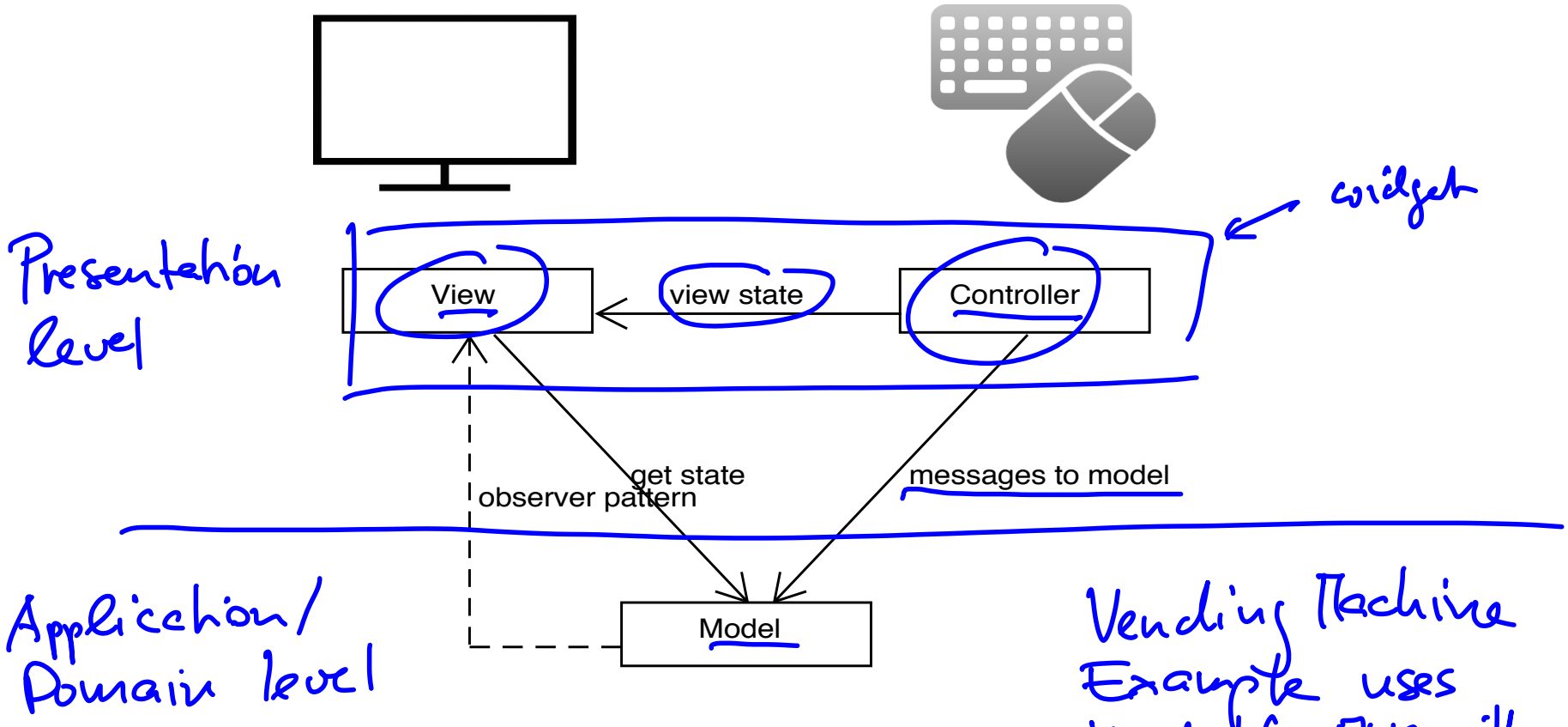
```
....
MyStack<Integer> stack = new MyStack<Integer>;
NumberOfElementsObserver obs1 =
            new NumberOfElementsObserver();
NumberOfElementsObserver obs2 =
            new StackObserver();
stack.addObserver(obs1);
stack.push(10);
stack.addObserver(obs2);
stack.pop();
...
stack.deleteObserver(obs1)
...
```

Sequence diagram for the stack

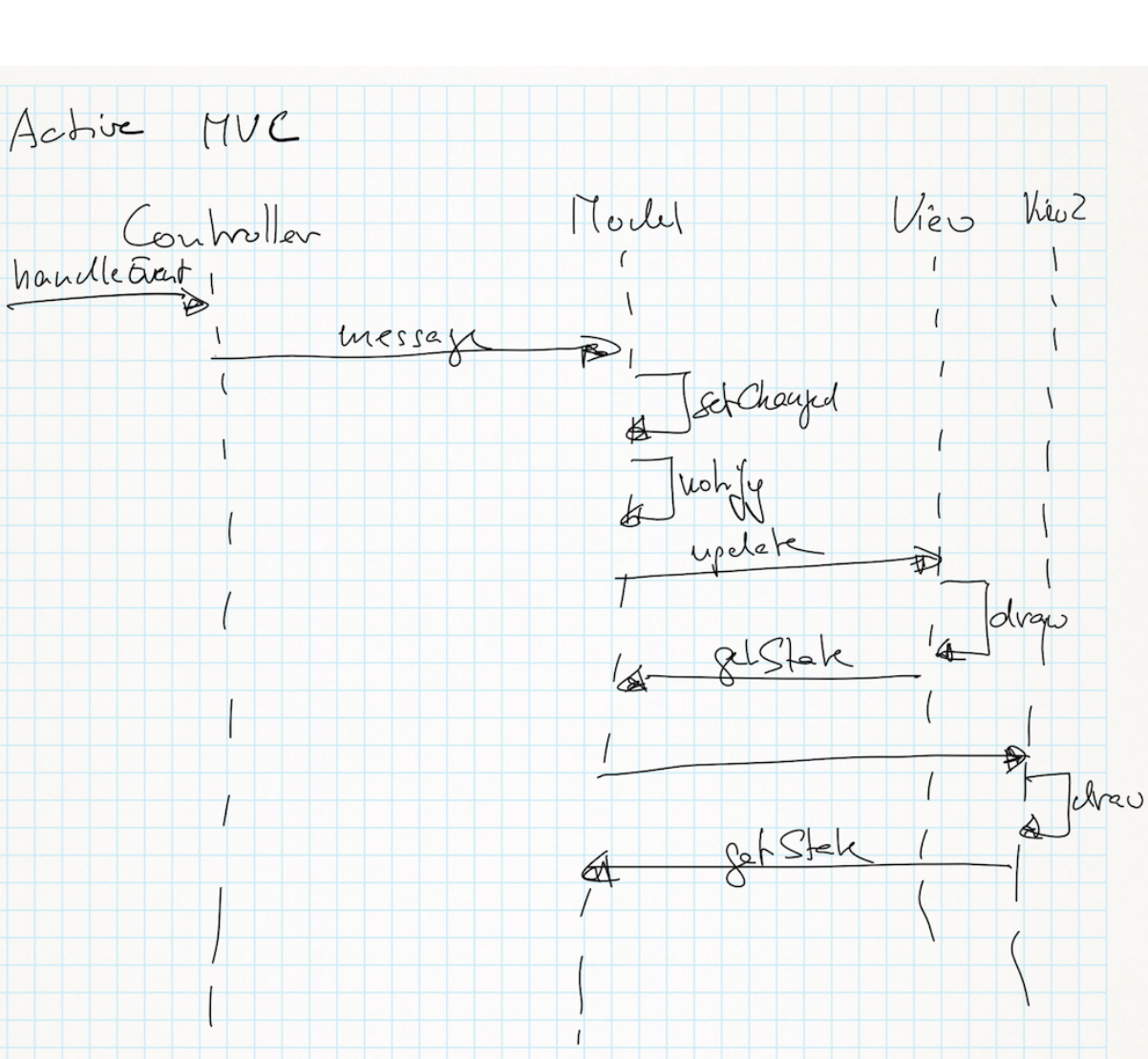


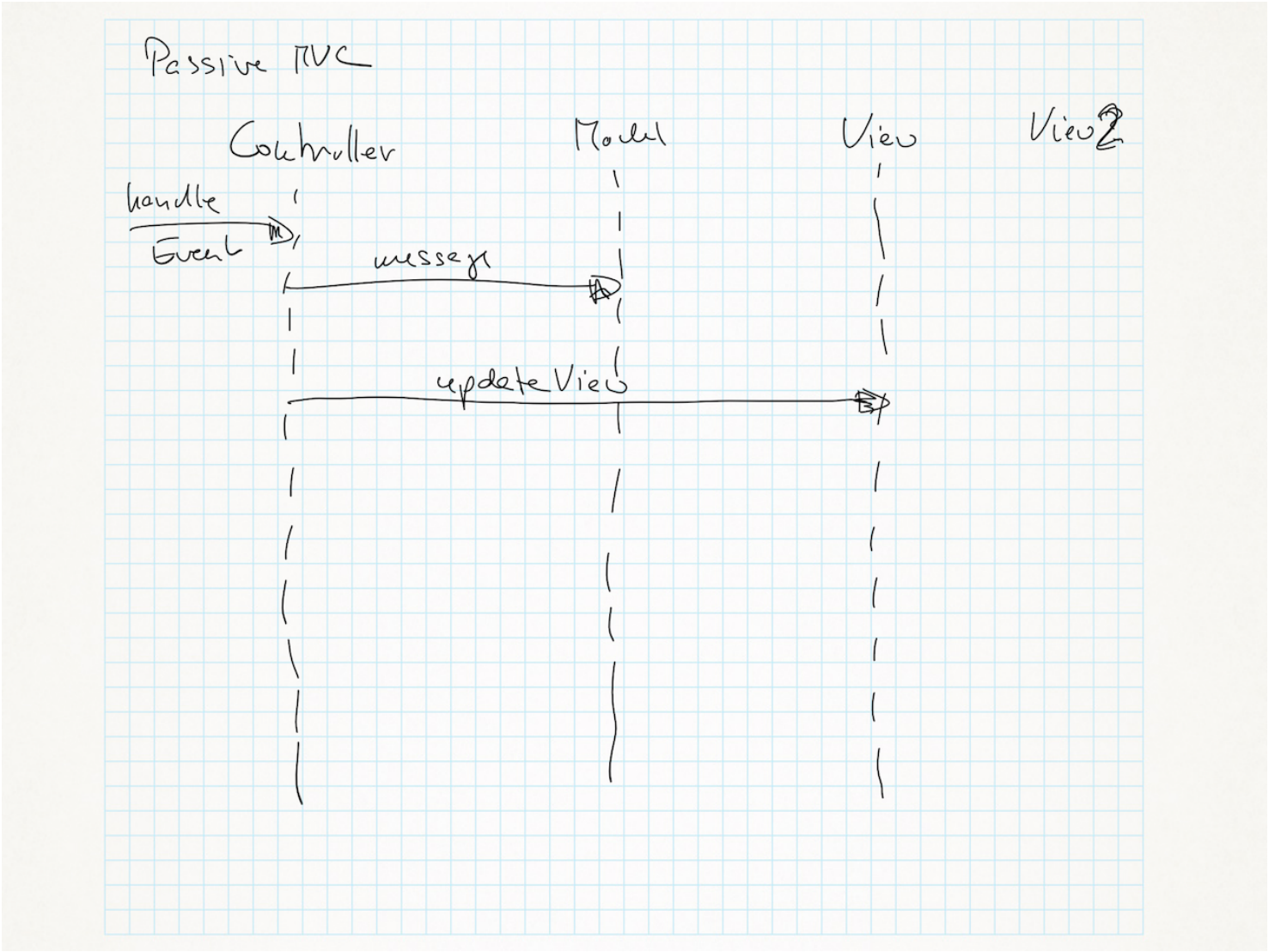
Model View Controller (MVC)

- Invented by Trygve Reenskaug for Smalltalk-76/80



- View: translate model state to display graphics
- Controller: translates mouse and keyboard events to application/domain specific messages to model

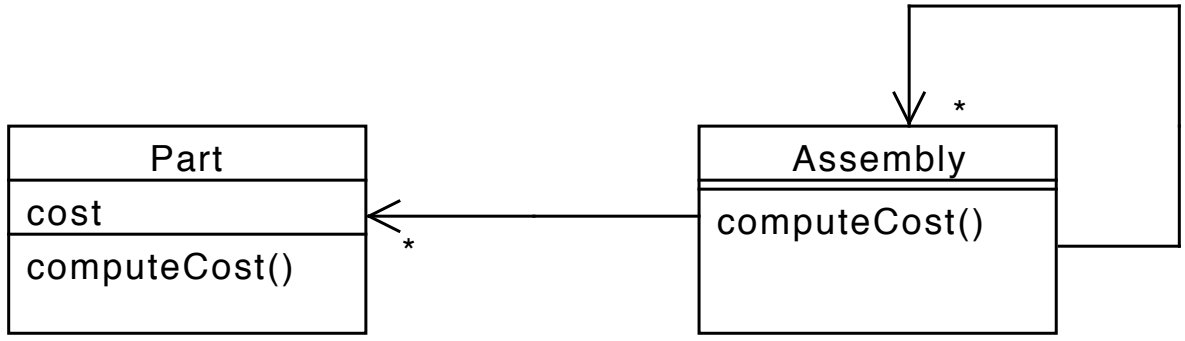

Active MVC
Controller
Model
View
View2
handleEvent
message
setChanged
notify
update
draw
getState
draw
getState

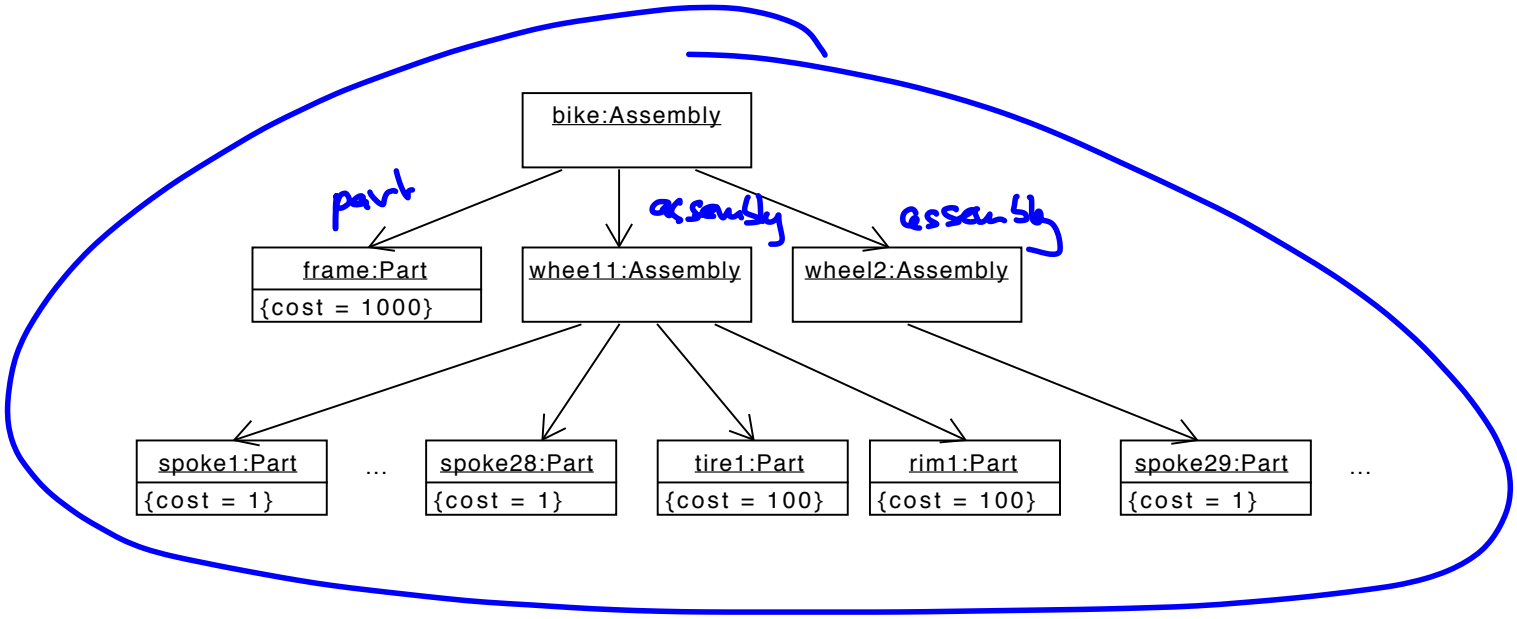

Passive MVC
Controller
Model
View
View2
handle
Event
message
updateView

Example: compute costs for components

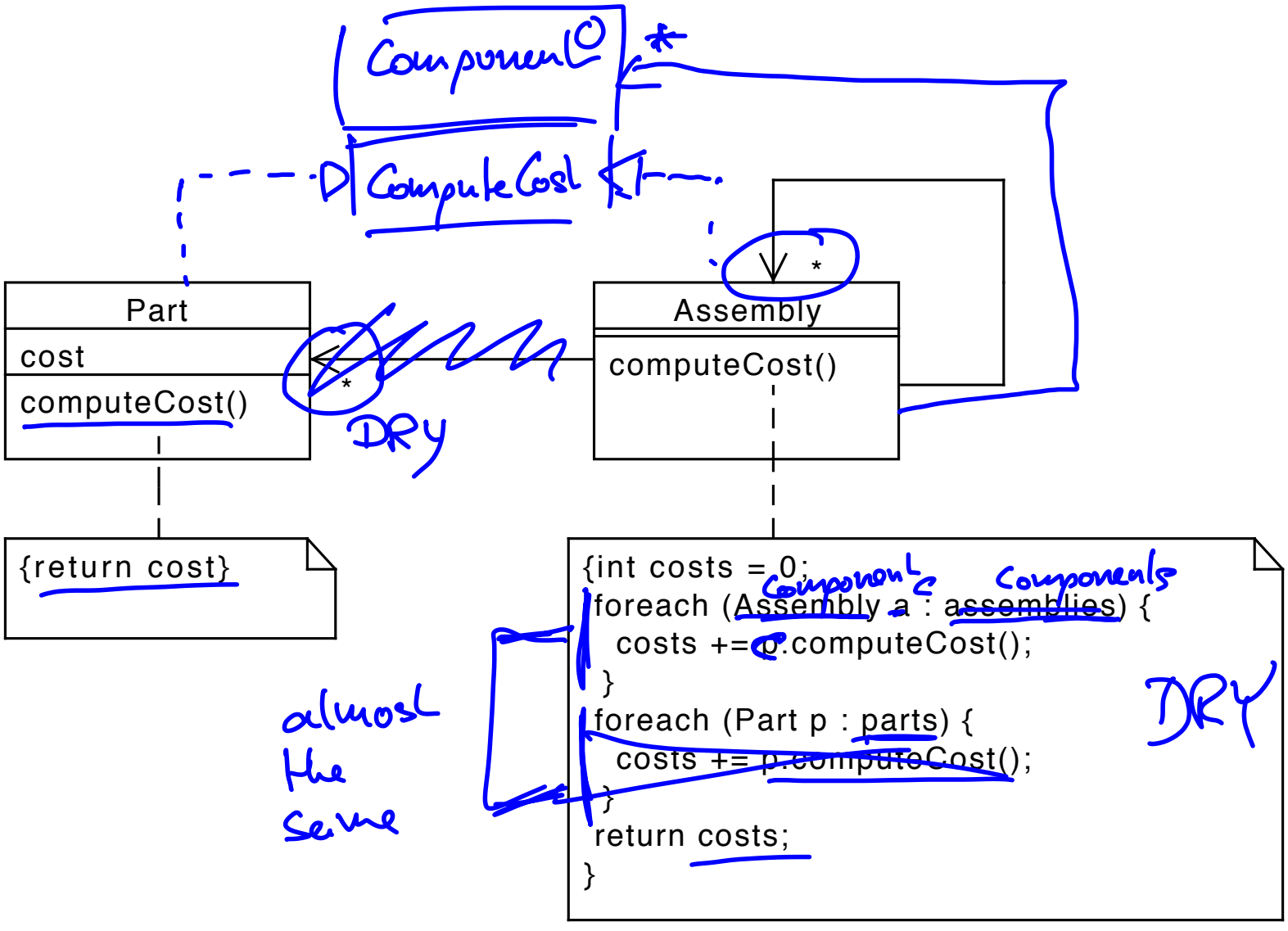
- Task: compute the overall costs of a bike
- Bike
  - Frame (1000 kr)
  - Wheel: 28 spokes (1 kr), rim (100 kr), tire (100 kr)
  - Wheel: 28 spokes (1 kr), rim (100 kr), tire (100 kr)

Example: compute costs for components

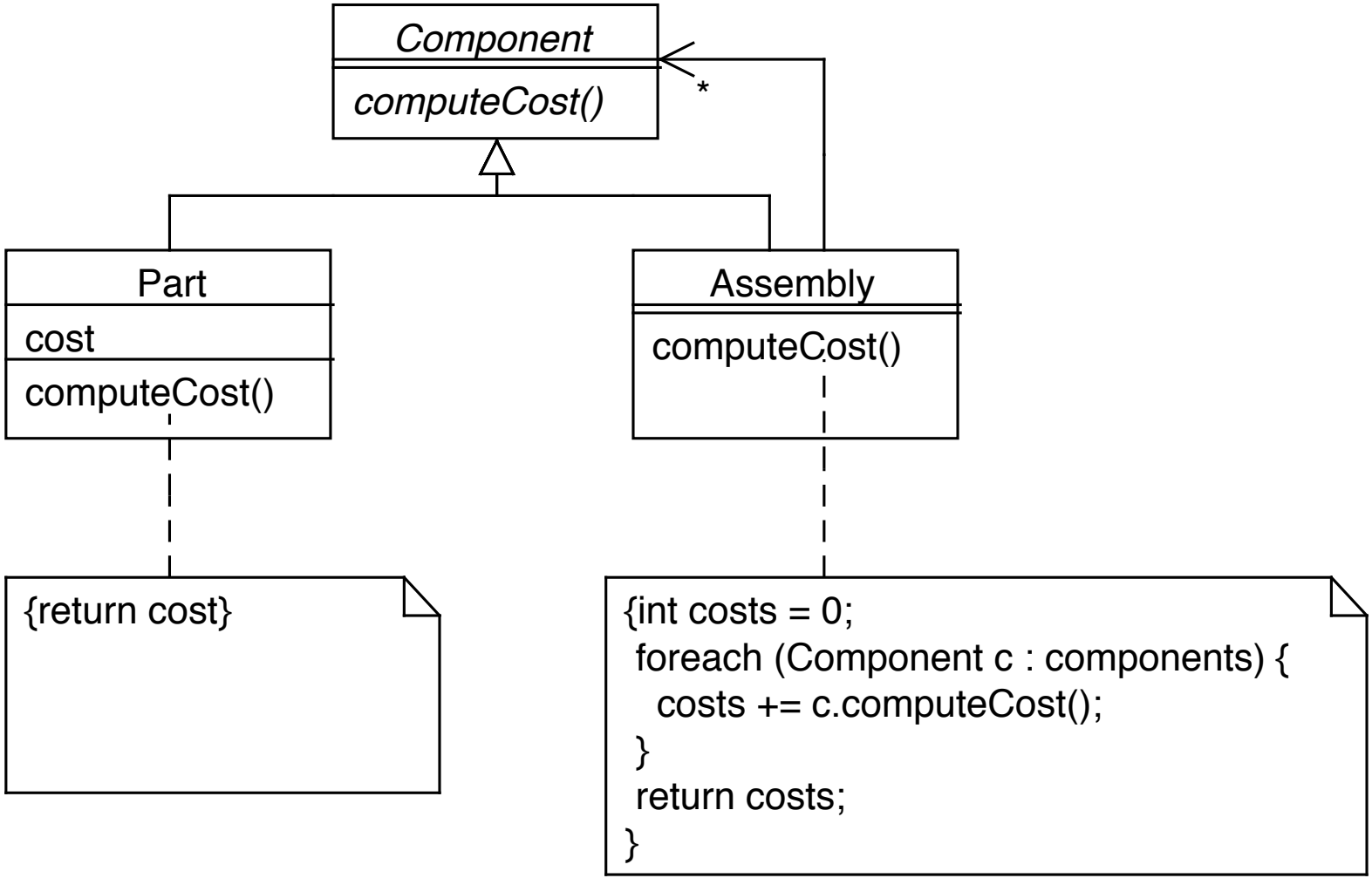




Example: compute costs for components



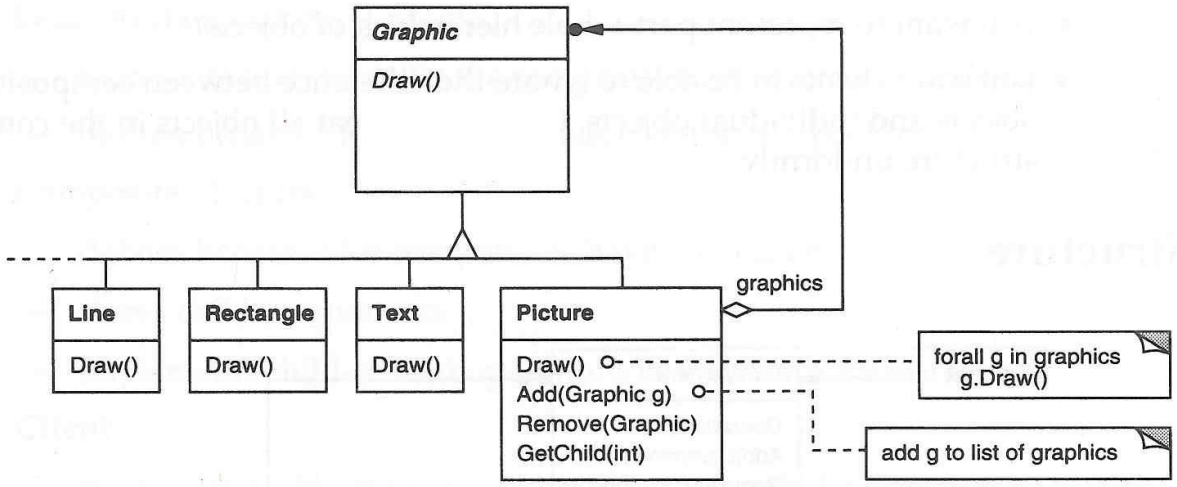
Example: compute costs for components



Composite Pattern

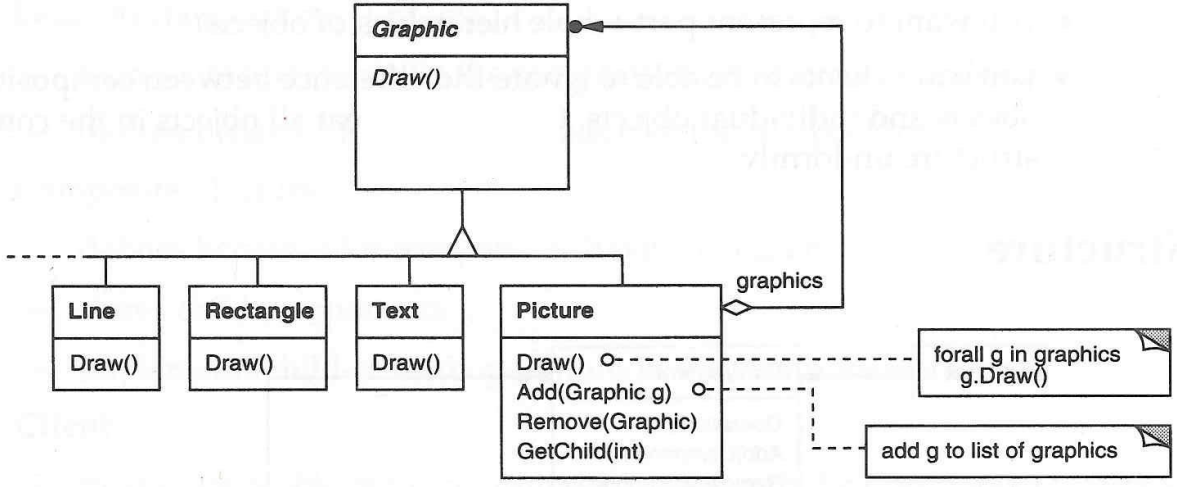
Composite Pattern

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

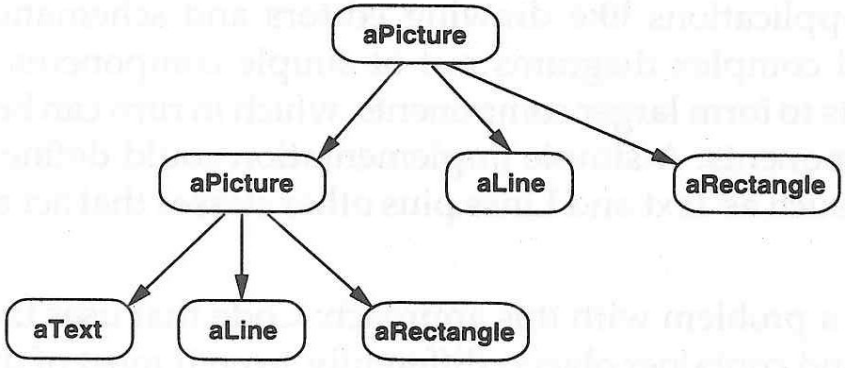


Composite Pattern: Graphics

- Class Diagram



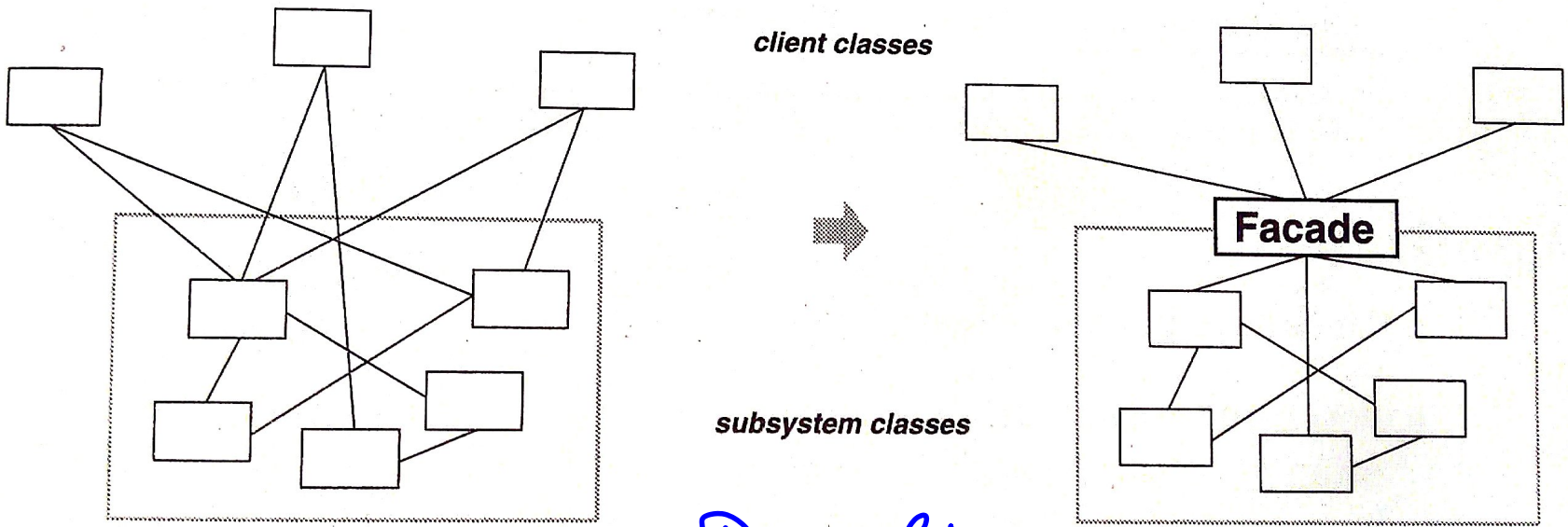
- Instance diagram



Facade

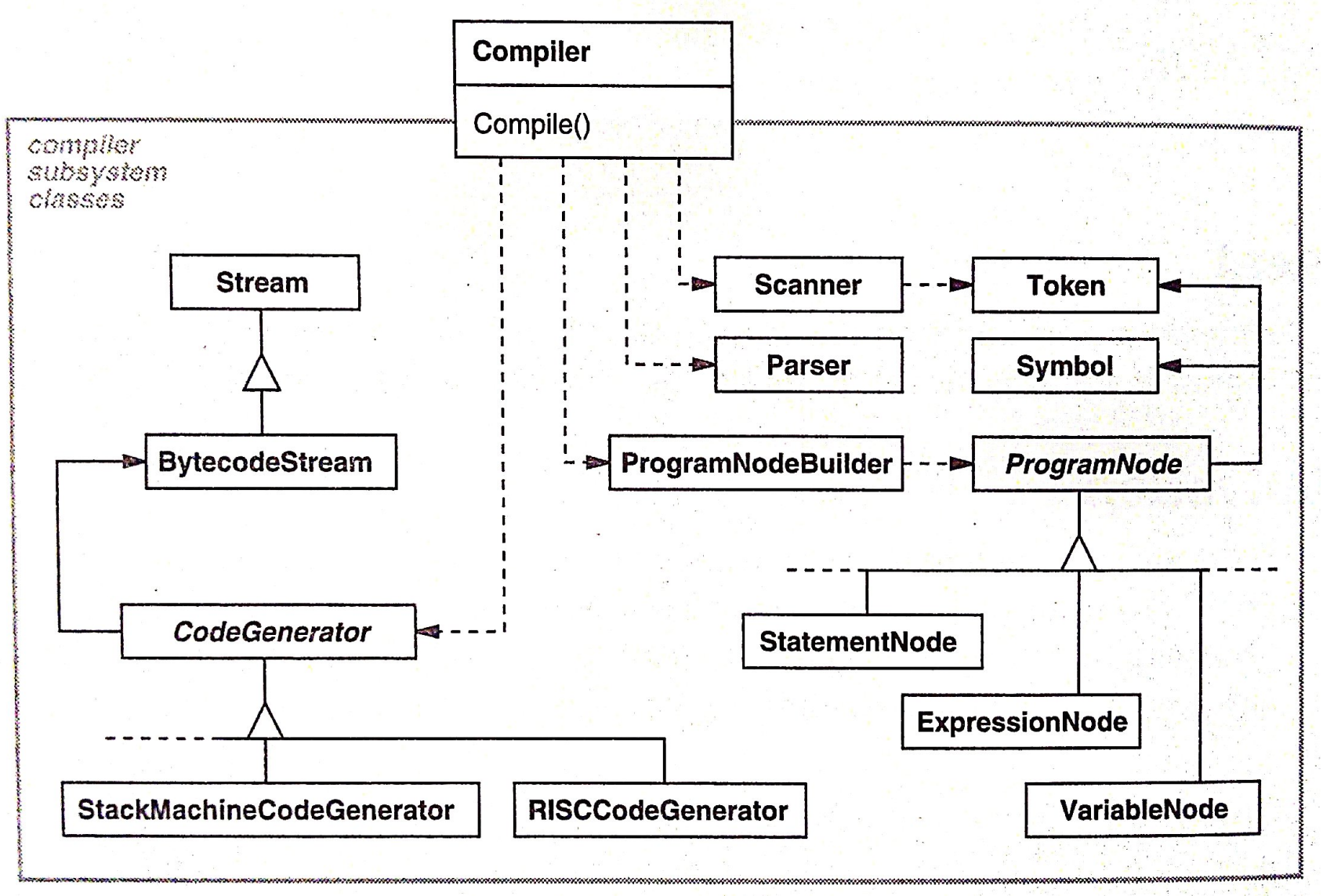
Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystems easier to use.



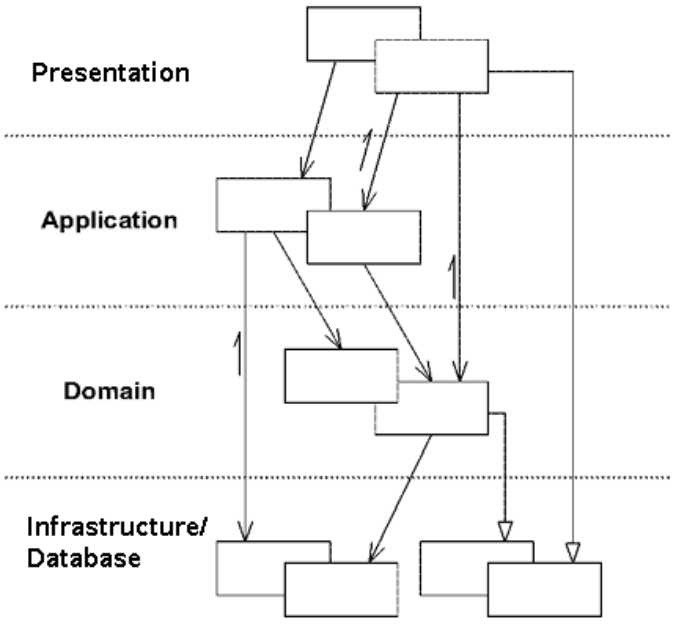
Design Patterns, Addison-Wesley, 1994

Example Compiler



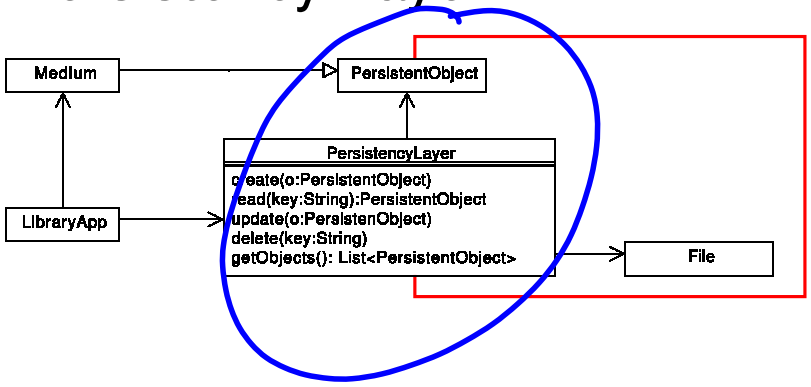
Design Patterns, Addison-Wesley, 1994

Example: Library Application



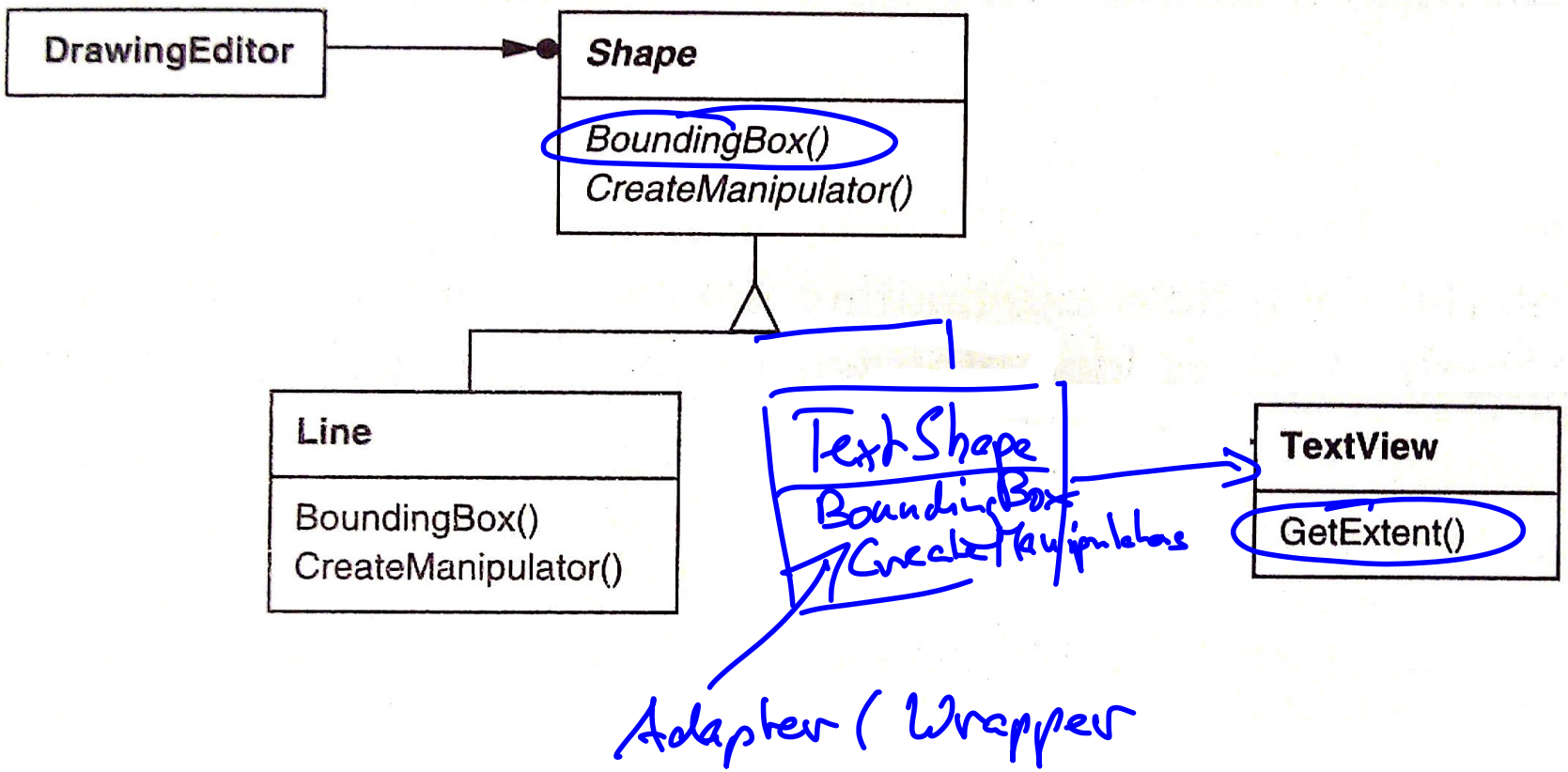
Eric Evans, Domain Driven Design, Addison-Wesley, 2004

- LibApp is the application facade
- Persistency Layer

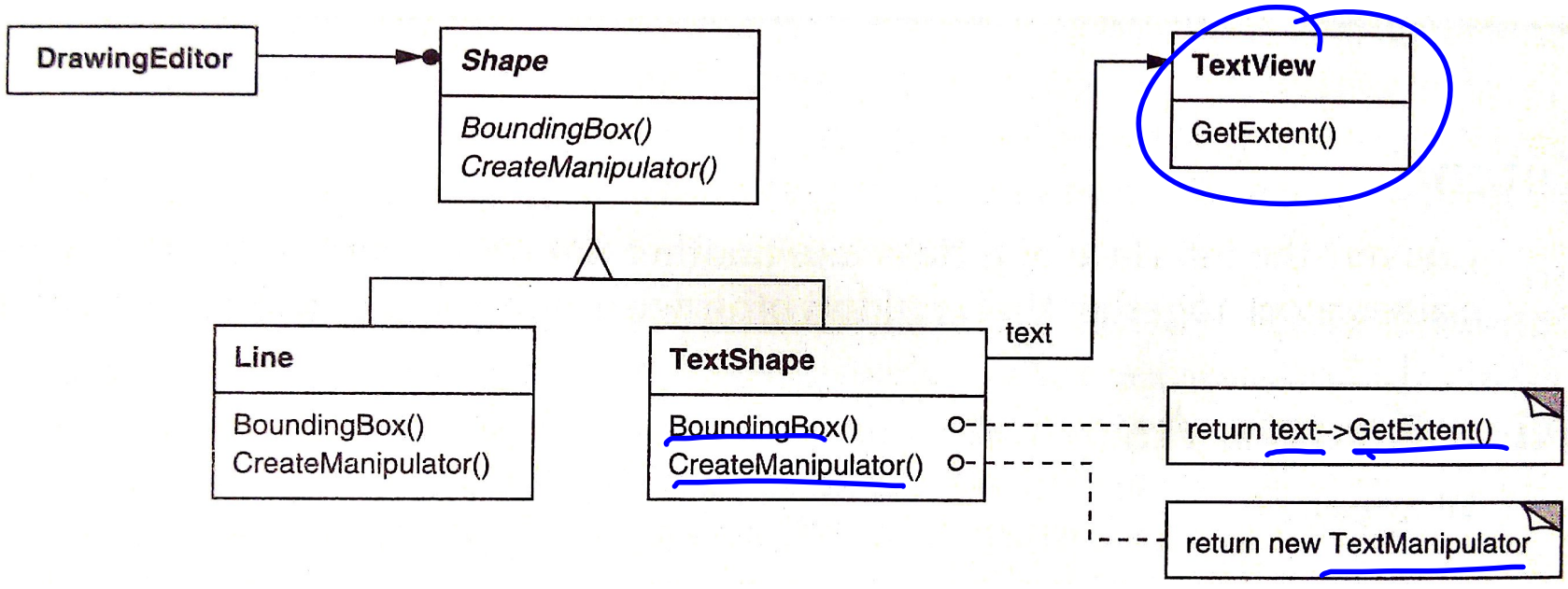


Adapter / Wrapper: Problem

- I want to include a text view as part of my graphic shapes
  - Shapes have a bounding box
  - But text views only have an method GetExtent()



Example: Using text views in a graphics editor

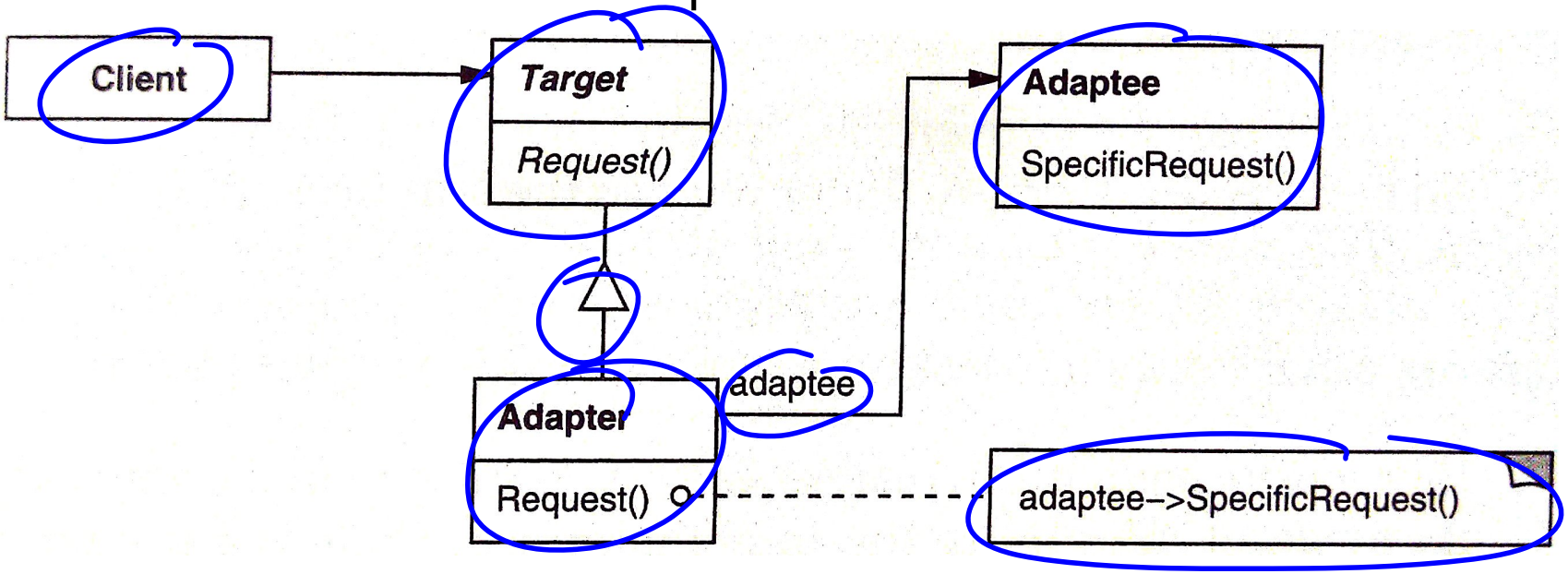


Design Patterns, Addison-Wesley, 1994

Adapter / Wrapper

Adapter / Wrapper

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



Design Patterns, Addison-Wesley, 1994

Contents

What did you learn?

- Requirements: Use Cases, User Stories, Use Case Diagrams, Domain modelling
- Testing: Systematic Tests, Test-Driven Development, Automated vs Manual Tests
- System Modelling: Class Diagram, Sequence Diagrams, State Machines, Activity Diagrams
- Design: CRC cards, Refactoring, Layered Architecture, Design Principles, Design Patterns, Design by Contract
- Software Development Process: Agile Processes, Project Planning

What did you learn?

- Requirements: Use Cases, User Stories, Use Case Diagrams, Domain modelling
- Testing: Systematic Tests, Test-Driven Development, Automated vs Manual Tests
- System Modelling: Class Diagram, Sequence Diagrams, State Machines, Activity Diagrams
- Design: CRC cards, Refactoring, Layered Architecture, Design Principles, Design Patterns, Design by Contract
- Software Development Process: Agile Processes, Project Planning

---

- Don’t forget the course evaluation

Plan for next weeks

- Week 12: Guest lecture my Motorola Solutions about agile software development in industry
  - Exercises as usual from 13:00 – 15:00
- Week 13: 12.5., 13:00 – 17:00: 10 min demonstrations of the software
  1. Show that all automatic tests run
  2. TA chooses one use case
     - 2.a Show the systematic tests for that use case
     - 2.b Execute the systematic test **manually**
  - Schedule will be published this week