Software Engineering I (02161)

Week 10

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2015

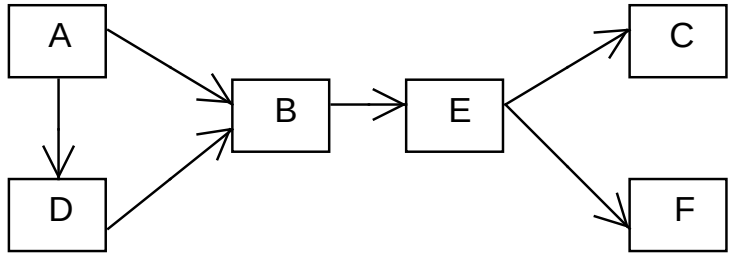
Last Time

- Project Planning
  - Non-agile
  - Agile
- Refactoring

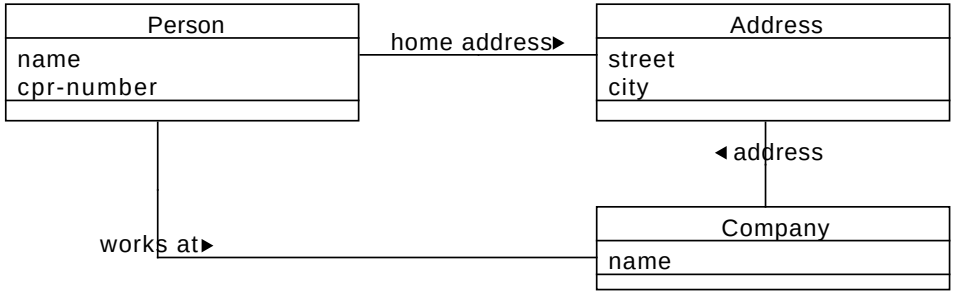
Contents

Low Copuling and High Cohesion

Low coupling



High Cohesion



→ Corner stones of good design

→ Layered Architecture

Law of Demeter

Law of Demeter

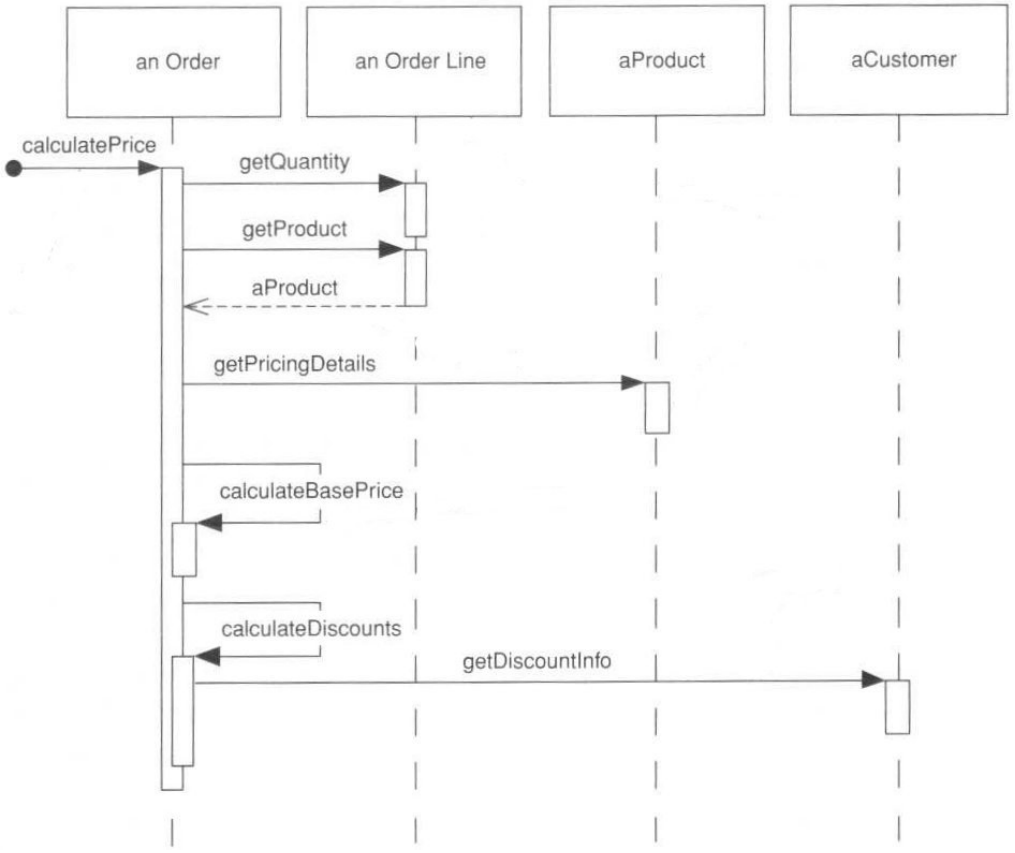
- ”Only talk to your immediate friends”
- Only method calls to the following objects are allowed
  - the object itself
  - its components
  - objects created by that object
  - parameters of methods

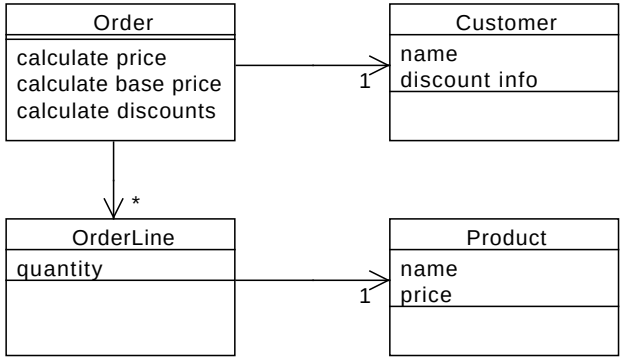
- Also known as: **Principle of Least Knowledge**
- Law of Demeter = **low coupling**

$\rightarrow$ **delegate functionality**

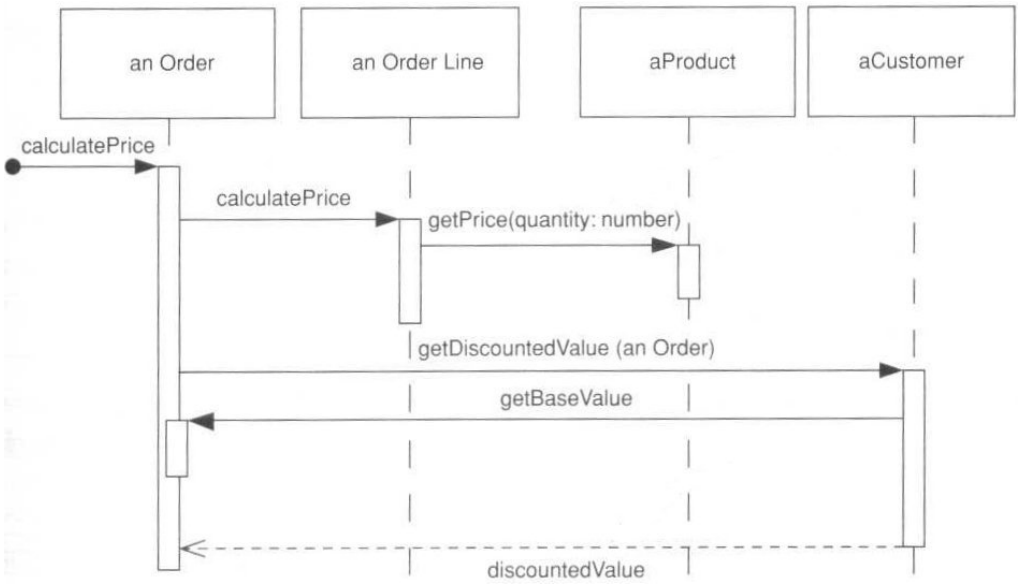
$\rightarrow$ decentralised control

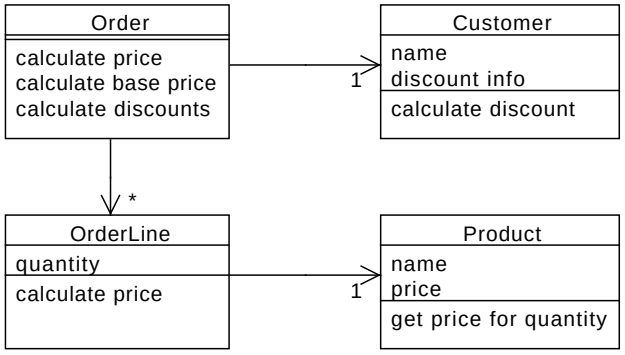
Computing the price of an order





Computing the price of an order





DRY principle

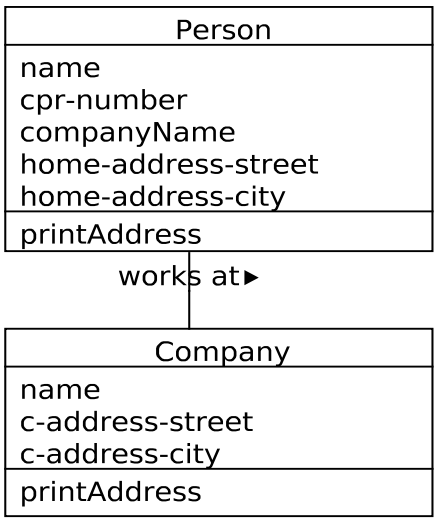
DRY principle

Don’t repeat yourself

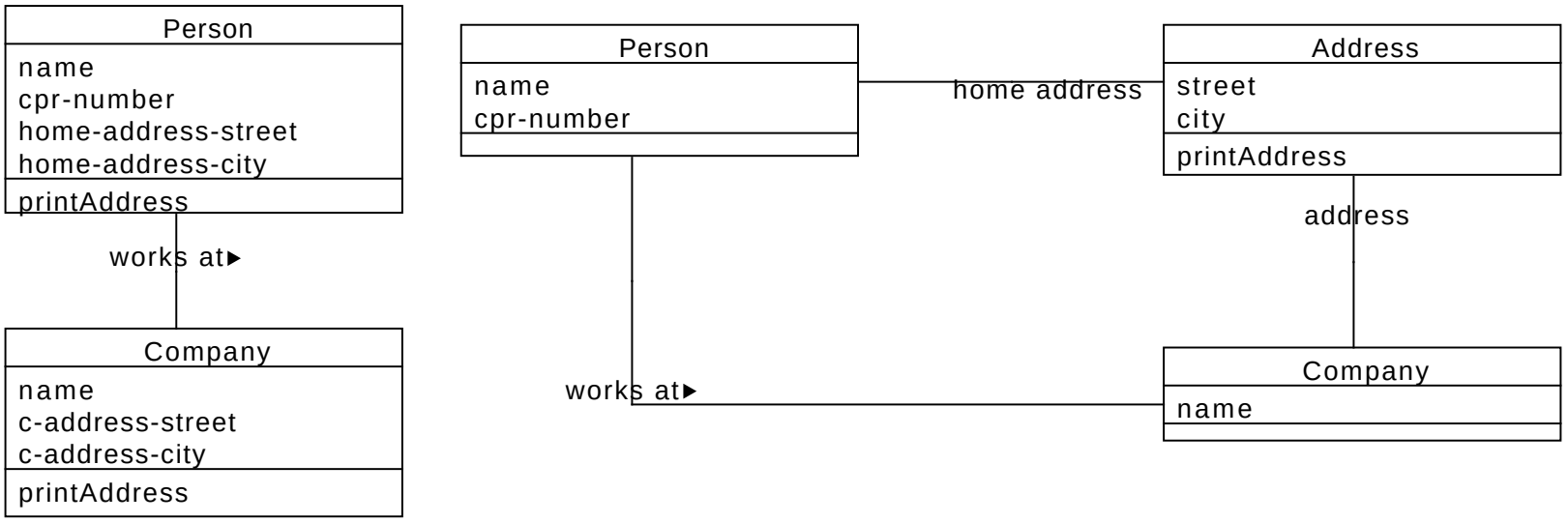
”Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.” The Pragmatic Programmer, Andrew Hunt and David Thomas

- 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

182 EATING ATMOSPHERE

. . . 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.

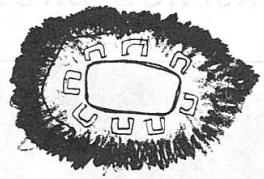
✣ ✣ ✣

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.

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:

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). . . .

A Pattern Language, Christopher Alexander, 1977

Pattern and pattern language

- Pattern: a *solution* to a *problem* in a context
- Pattern language: set of related patterns

History of Patterns

- Chstiopher Alexander: Architecture (1977/1978)
- Kent Beck and Ward Cunningham: Patterns for Smalltalk applications (1987)
- Portland Pattern Repository `http://c2.com/ppr`
  - $\rightarrow$ origin of *wikis*
- Design Patterns book (1994)

Design Patterns

- Defined in the Design Pattern Book
  - Authors: Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm
- Best practices for object-oriented software

  $\rightarrow$ 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 ...

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`

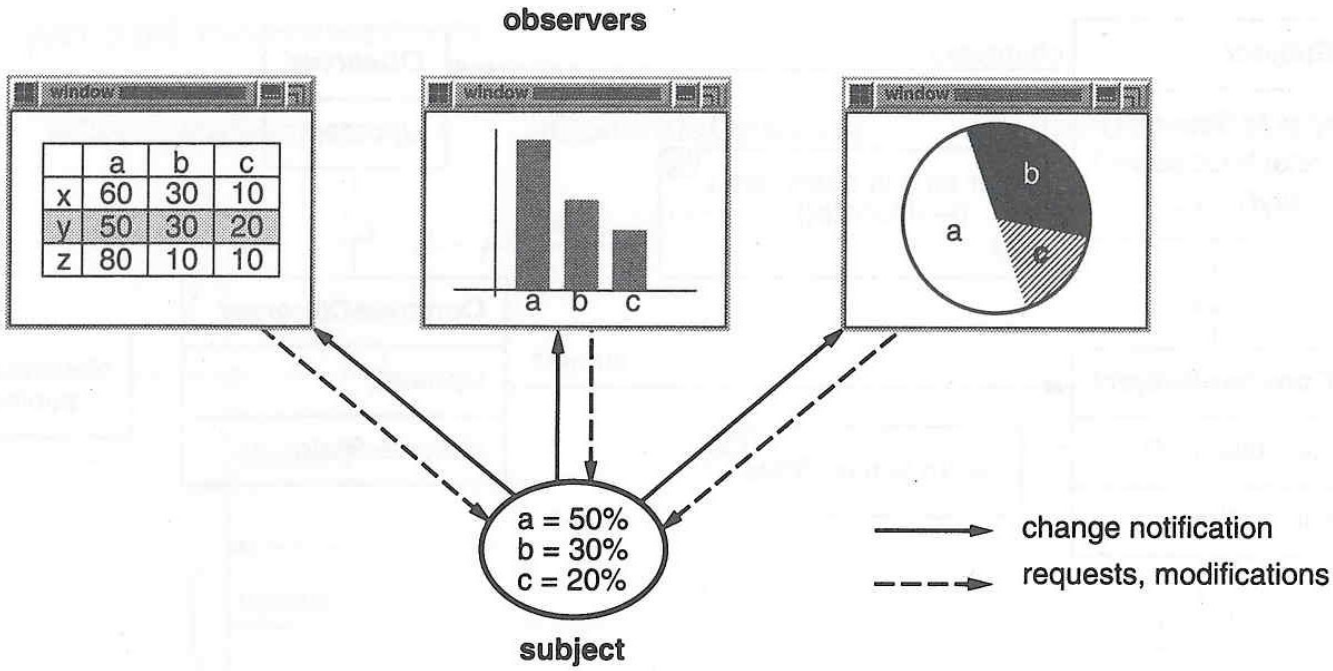
Design Pattern structure

- Alexander: Context, Problem, Forces, Solution, Related Pattern
- Design Patterns: Intent, Motiviation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, Related Patterns

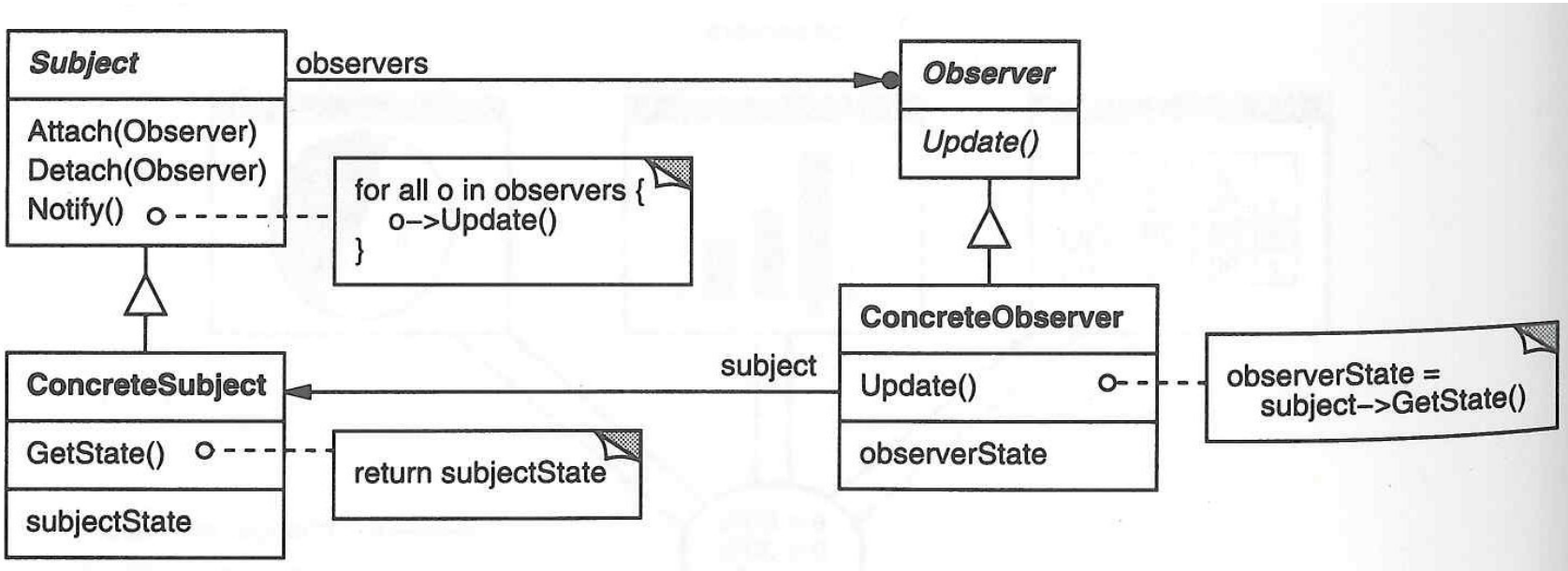
Observer Pattern

Observer Pattern

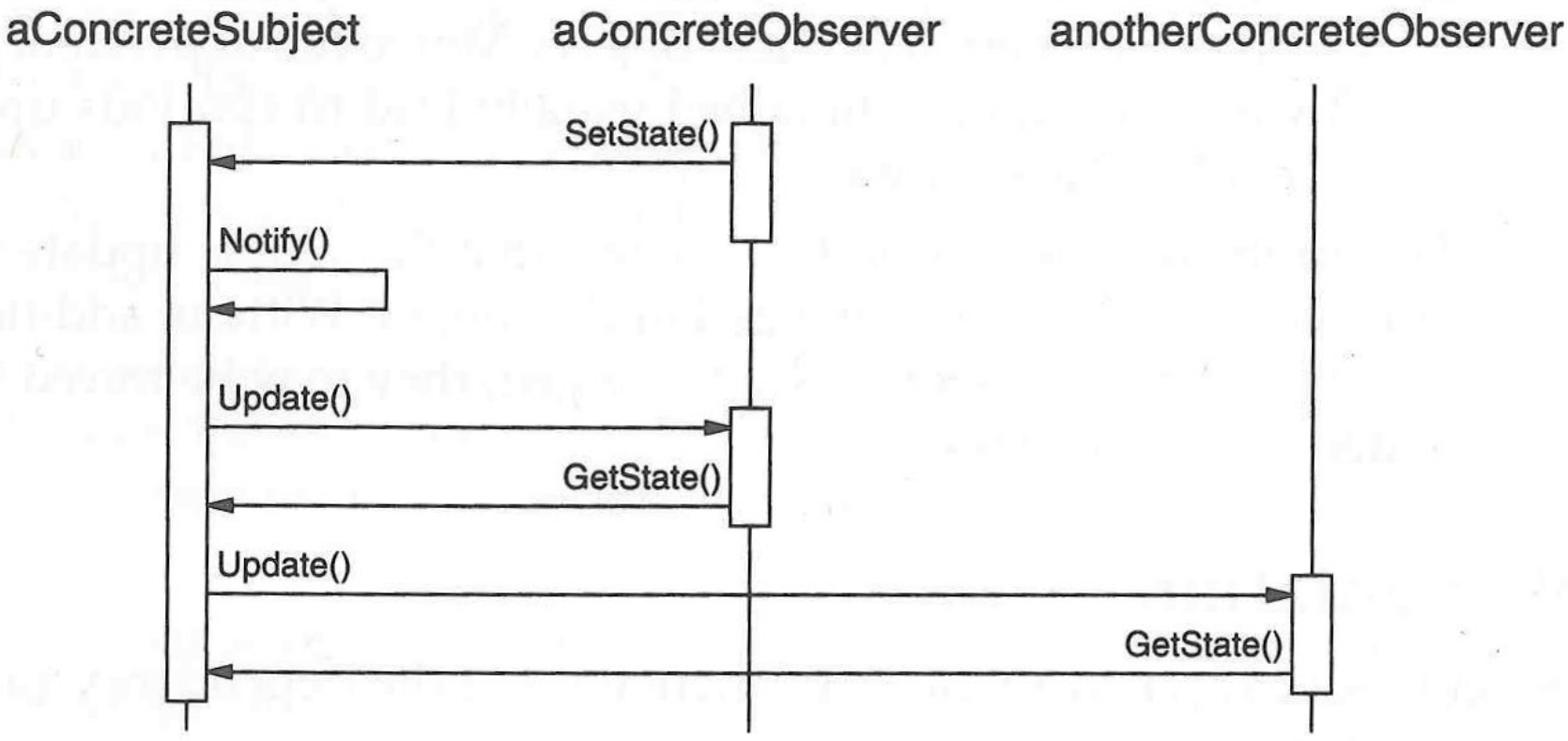
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Observer Pattern



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.

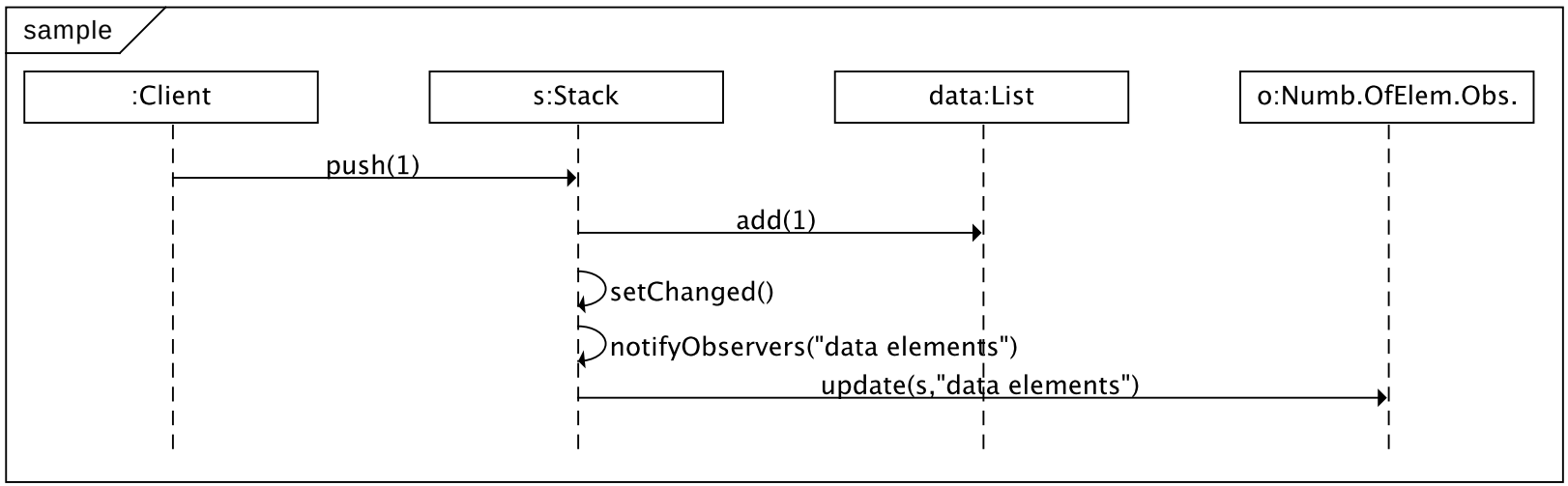
```
class StackObserver() implements Observer {

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

Sequence diagram for the stack



Composite Pattern

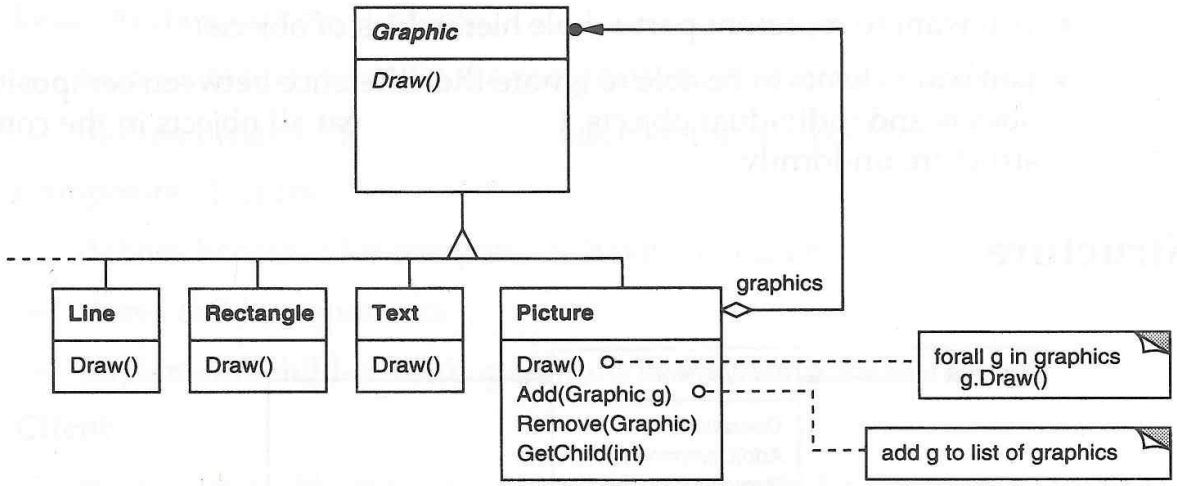
Problem: Graphics Editor

- Line, Rectangle Text
  - can be drawn
- Picture: can contain Line, Rectangle, Text and Picture
  - can be drawn

Composite Pattern

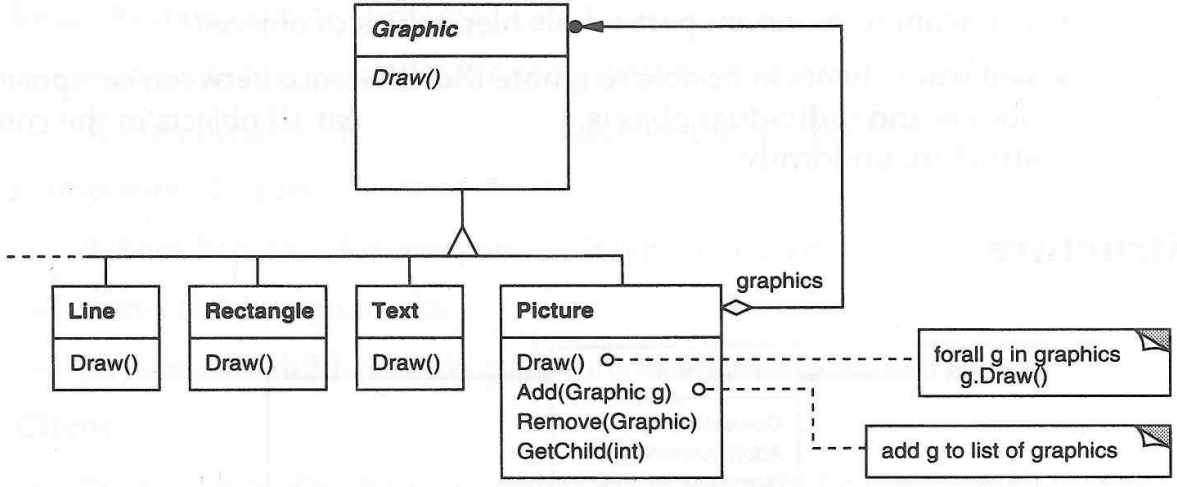
Composite Pattern

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

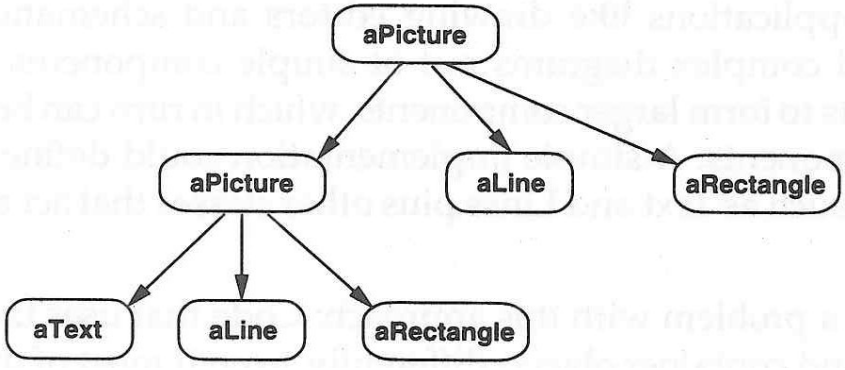


Composite Pattern: Graphics

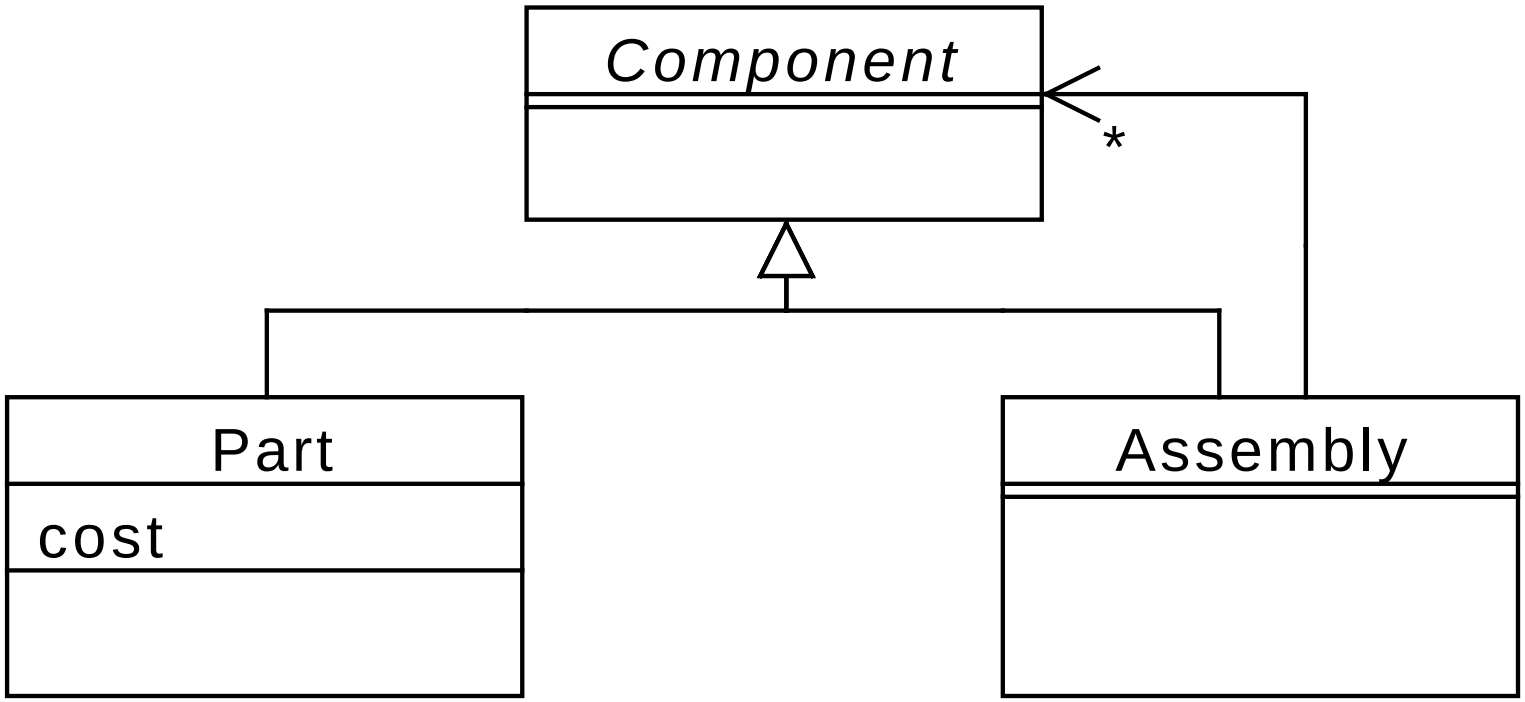
- Class Diagram



- Instance diagram

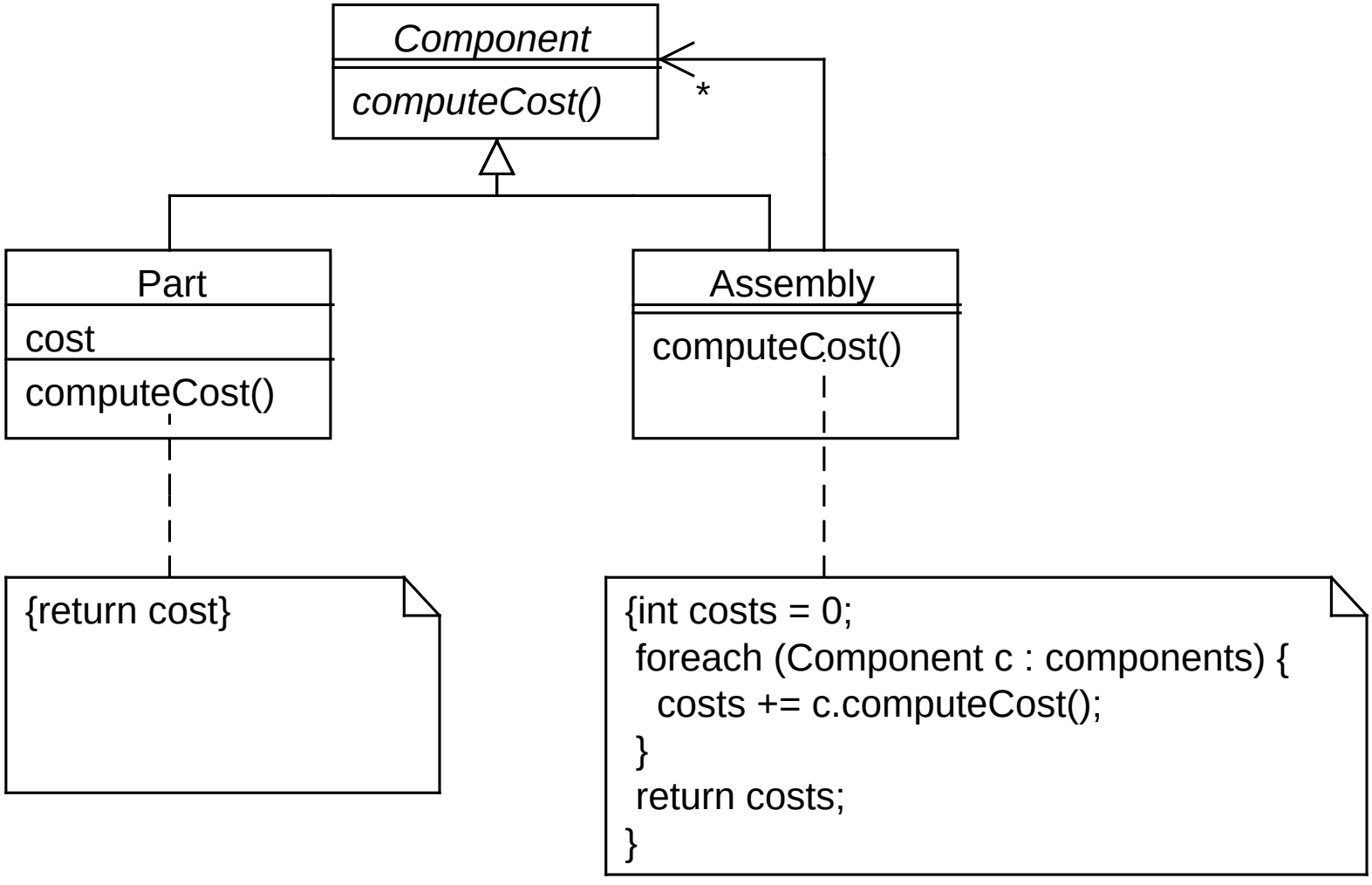


Example: compute costs for components



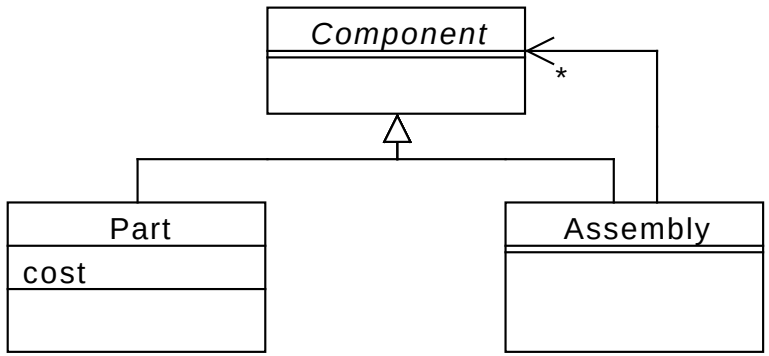
- Bike
  - Frame (1000 kr)
  - Wheel: 28 spokes (1 kr), rim (100 kr), tire (100 kr)
  - Wheel: 28 spokes (1 kr), rim (100 kr), tire (100 kr)
- Task: add a compute cost function computing the overall costs of a bike

Example: compute costs for components

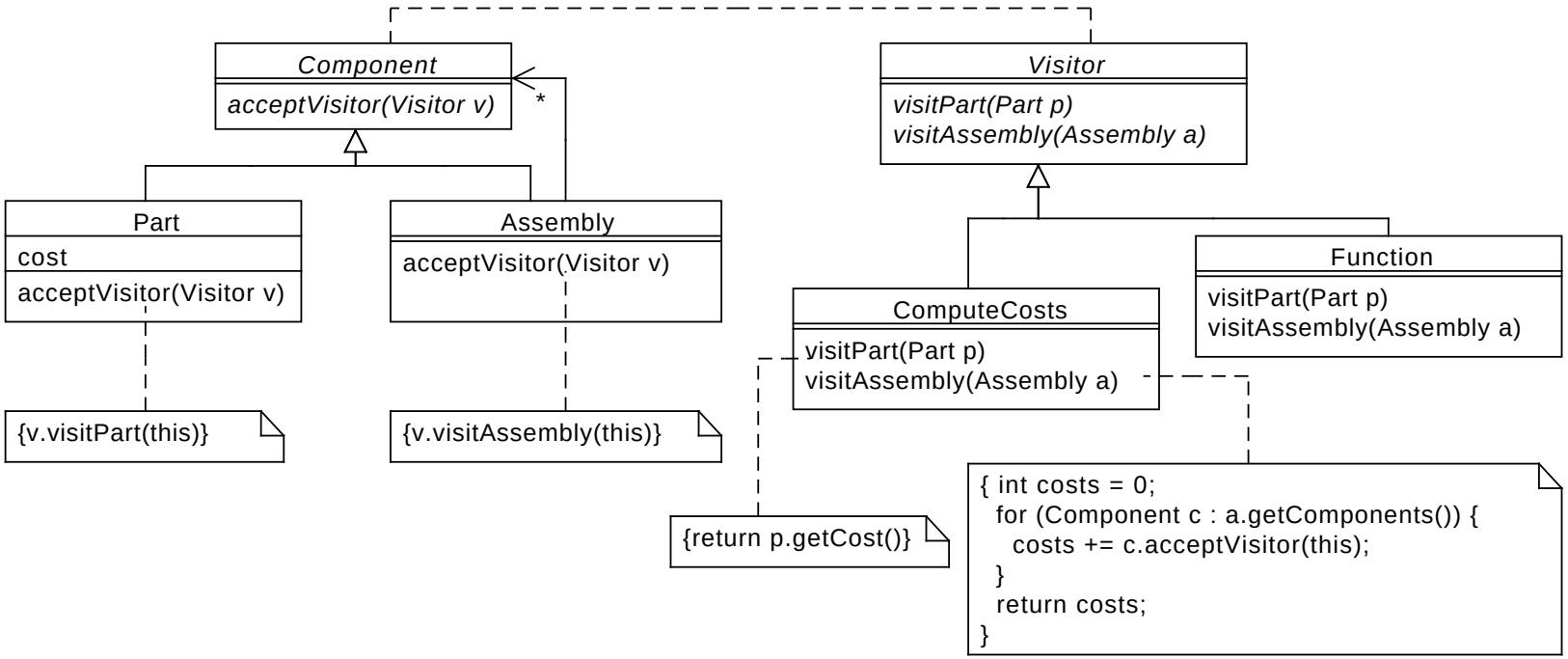


Visitor Pattern: Problem

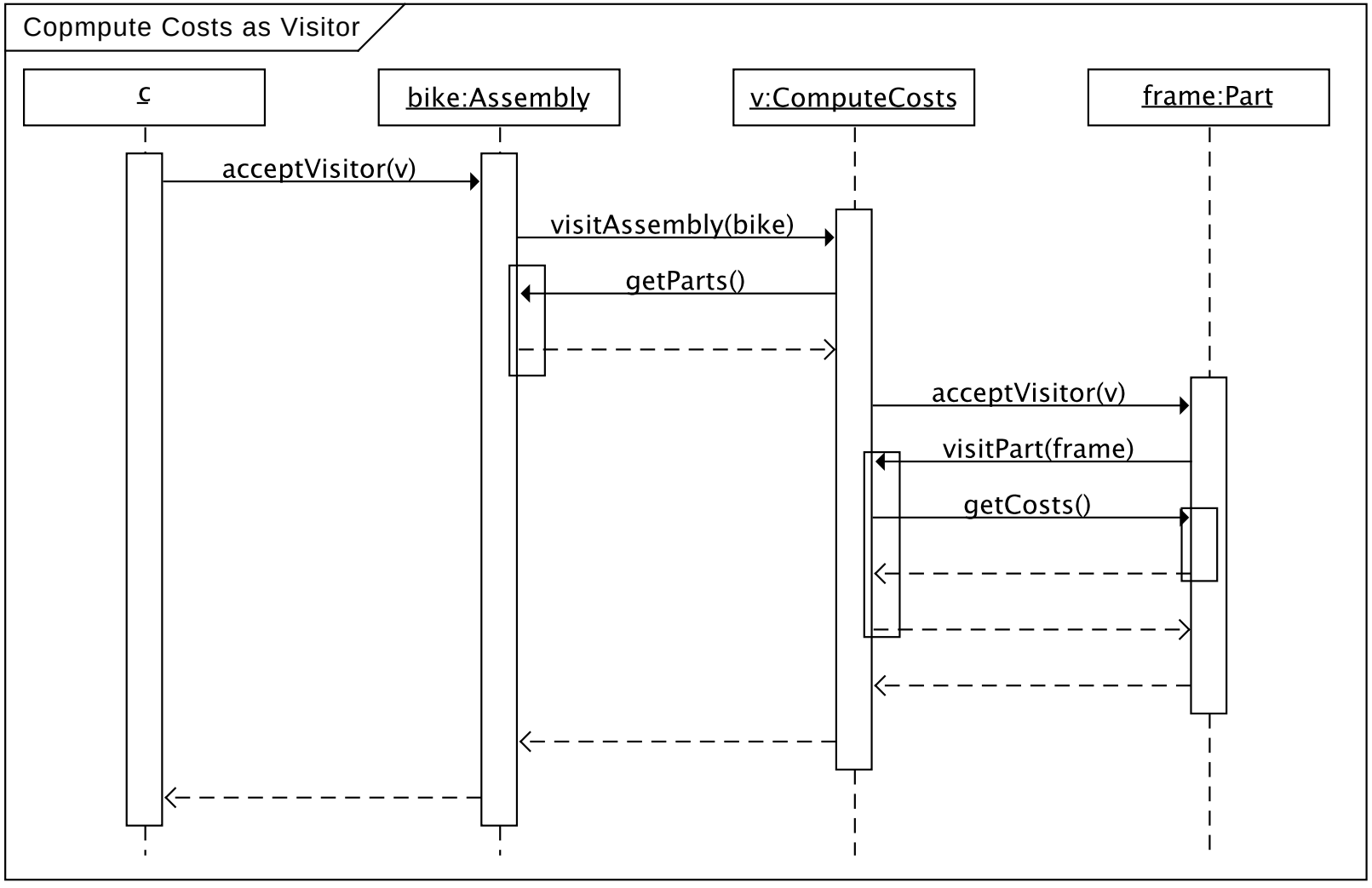
- Define a mechanism to define algorithms on complex datastructures without modifying the class, e.g. when the class is provided in a library
- For example, add a computeCost algorithm without adding the method to the class



Example: compute costs as a visitor



Compute costs as a visitor



Visitor Pattern

Visitor Pattern

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Template Method Problem

Overdue message for Book:

1. compute due date for a book
   a. get the current date
   b. add the max days for loan for the book
2. check if the current date is after the due date

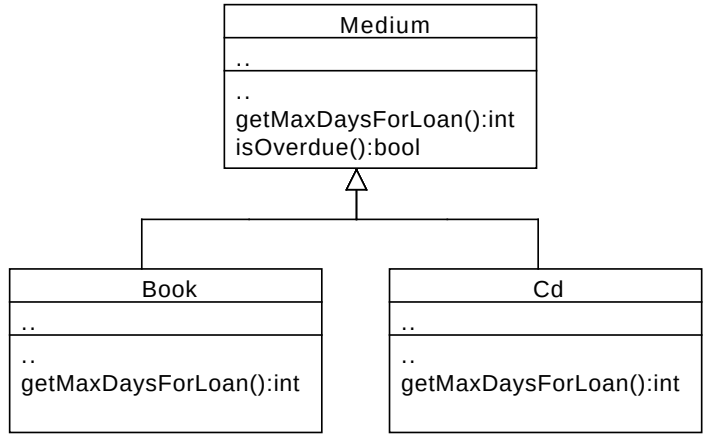
Overdue message for CD:

1. compute due date for a cd
   a. get the current date
   b. add the max days for loan for the cd
2. check if the current date is after the due date

Template method

- Create a template method in class Medium:
  1. compute due date for a medium
     a. get the current date
     b. add the max days for loan for that medium
  2. check if the current date is after the due date
- In book method for getMaxDaysForLoan returning 4 weeks
- In CD getMaxDaysForLoan returns 2 weeks

Template Method

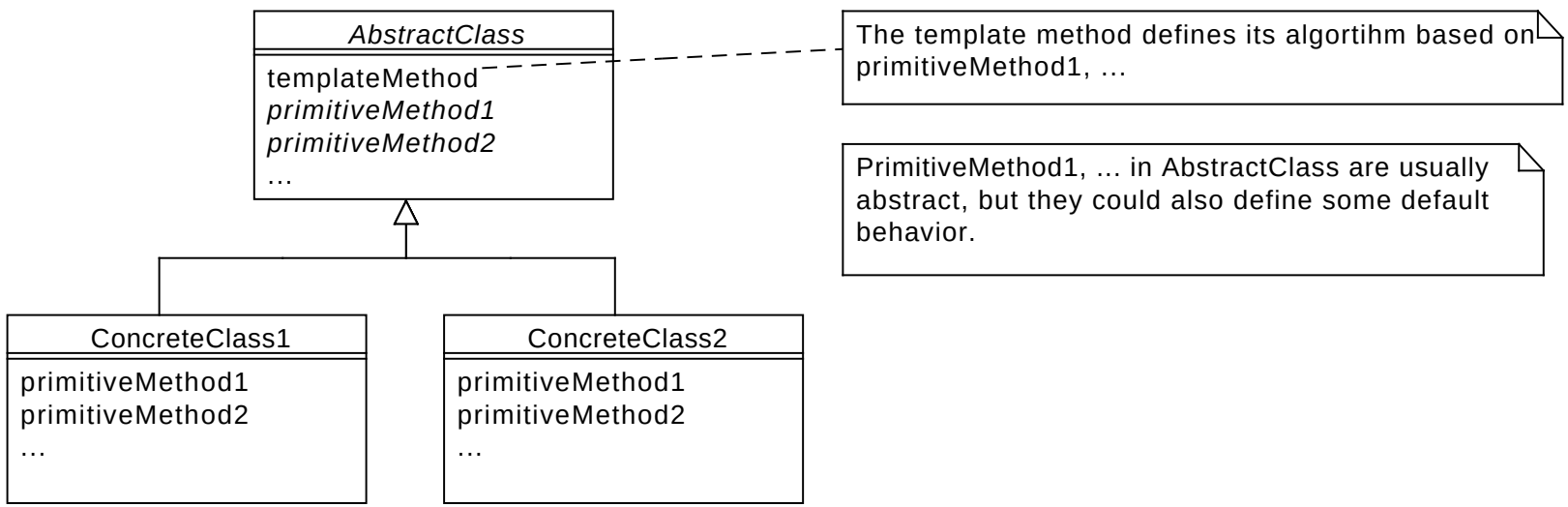



```
public abstract class Medium {

  public boolean isOverdue() {
    if (!isBorrowed()) {
      return false;
    }
    Calendar date = libApp.getDate();
    Calendar dueDate = new GregorianCalendar();
    dueDate.setTime(borrowDate.getTime());
    dueDate.add(Calendar.DAY_OF_YEAR, getMaxDaysForLoan());
    return date.after(dueDate);
  }

  public abstract int getMaxDaysForLoan();
}
```

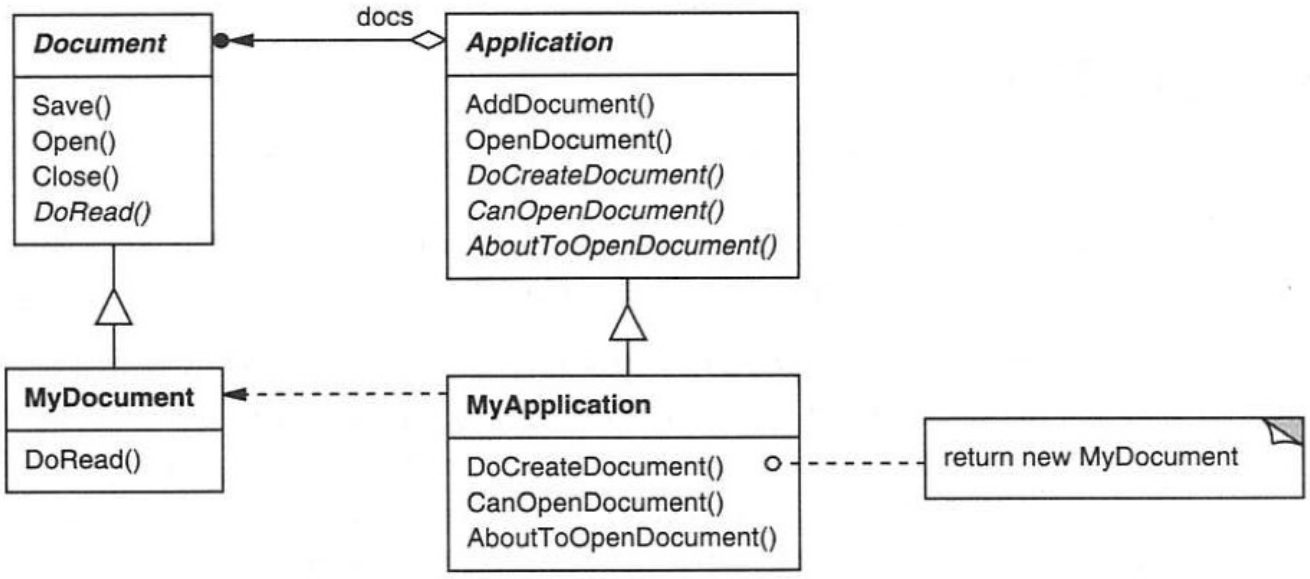
Template Method

Template Method

Define the skeleton of an algortihm in an operation, deferring some steps to subclasses. Template Method lets sublcasses redefine certain steps of an algorithm without changing the algorithm's structure.



Template Method

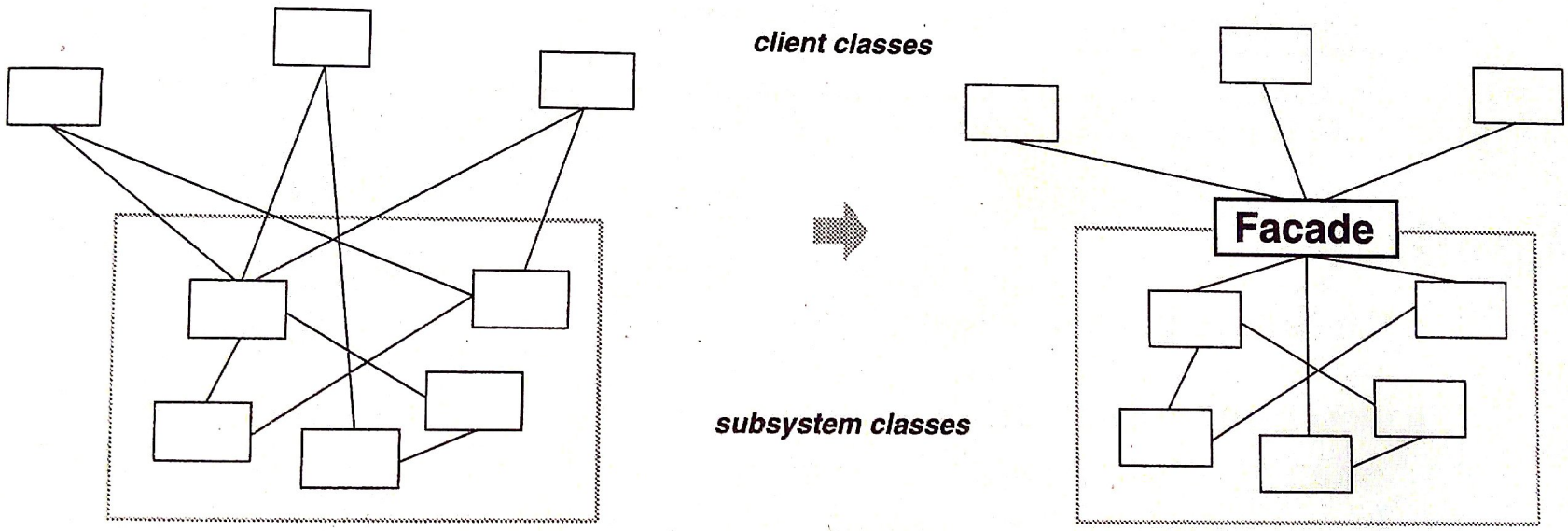



```
public abstract class Application {
  public void openDocument(String name) {
    if (canOpenDocument(name)) {
      Document doc = createDocument(name);
      if (doc != null) {
        docs.add(doc);
        aboutToOpenDocument(doc);
        doc.open();
        doc.read();
      }
    }
  }
}
```

Facade

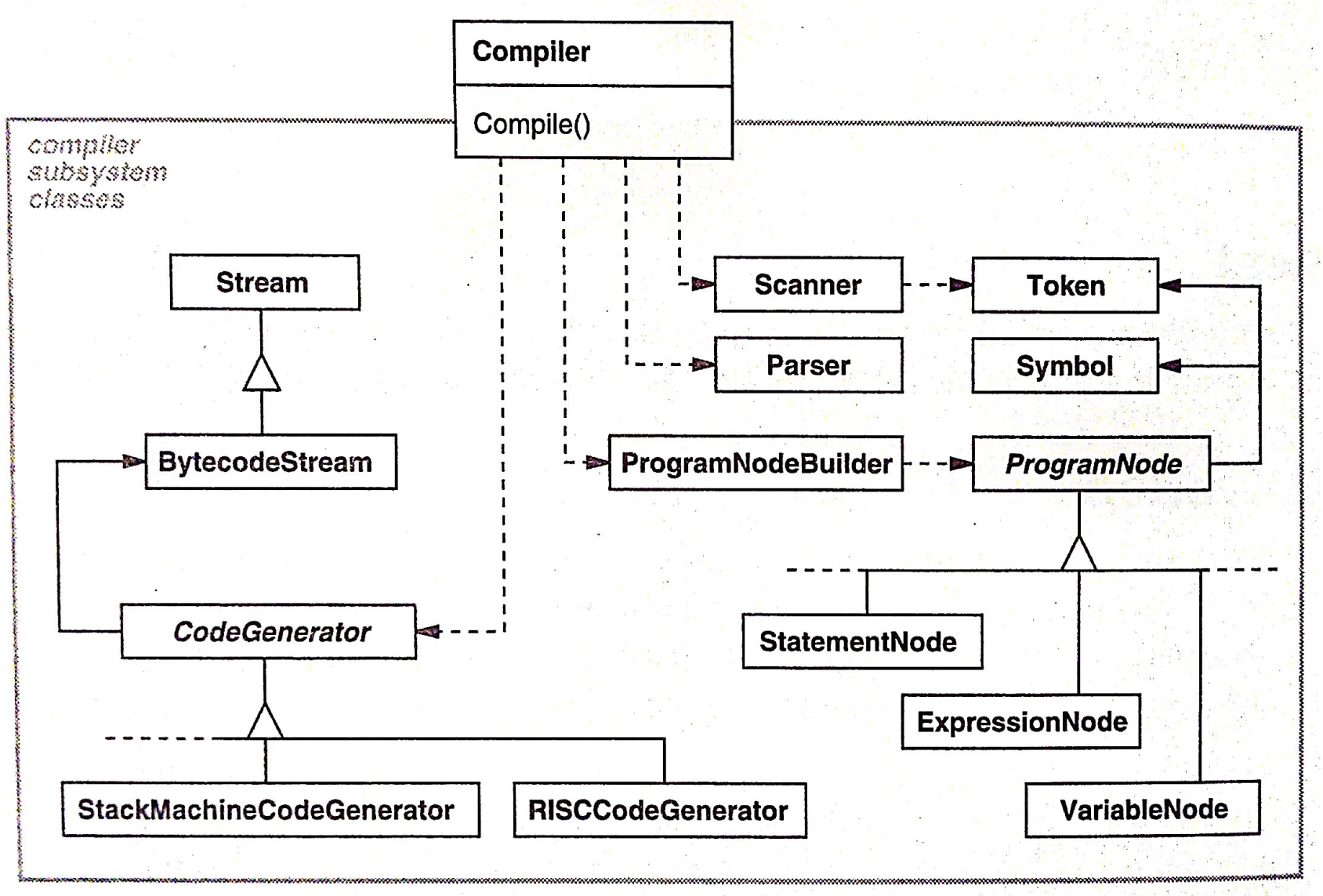
Facade

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.



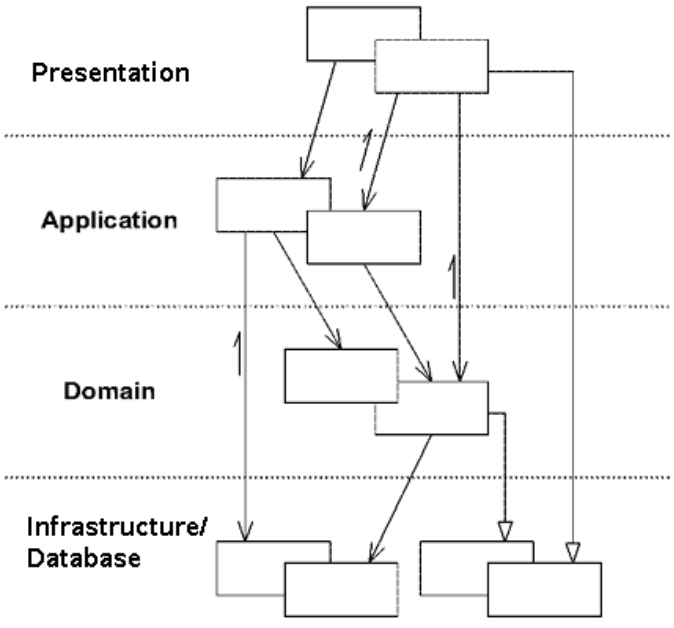
Design Patterns, Addison-Wesley, 1994

Example Compiler



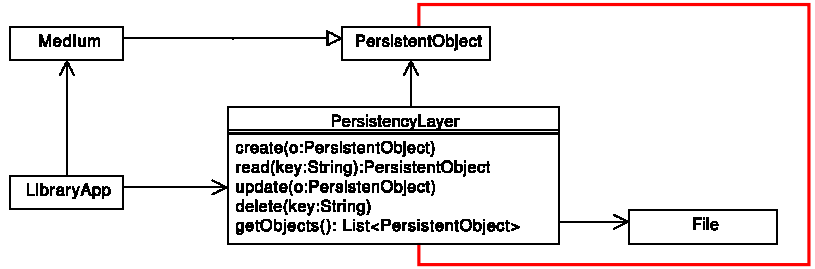
Design Patterns, Addison-Wesley, 1994

Example: Library Application



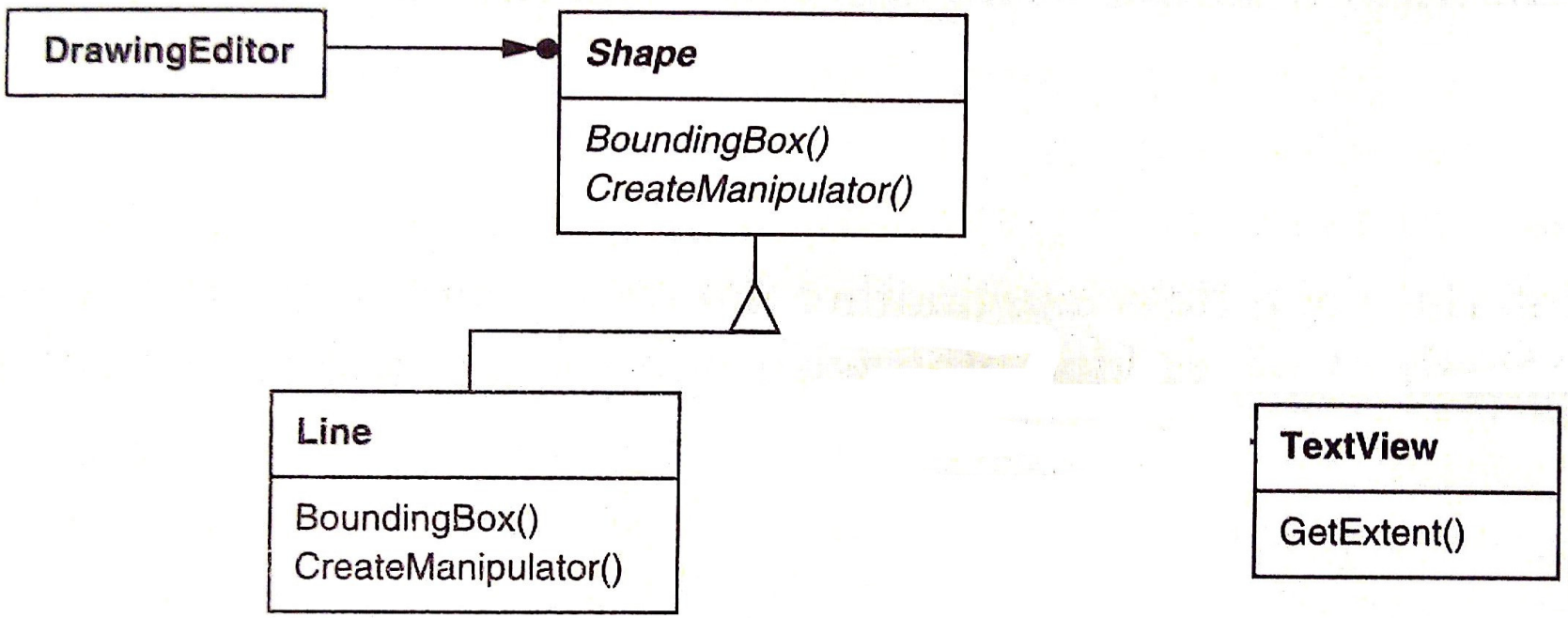
Eric Evans, Domain Driven Design, Addison-Wesley, 2004

- LibApp is the application facade
- Persistency Layer

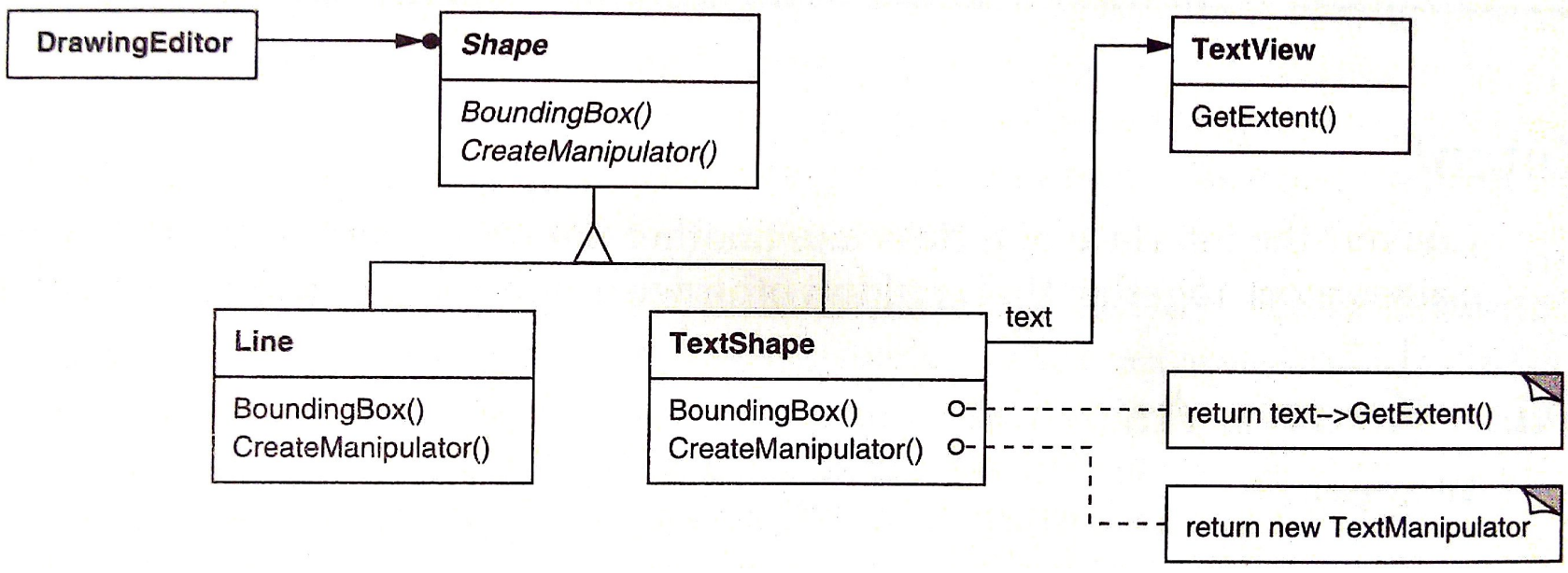


Adapter / Wrapper: Problem

- 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

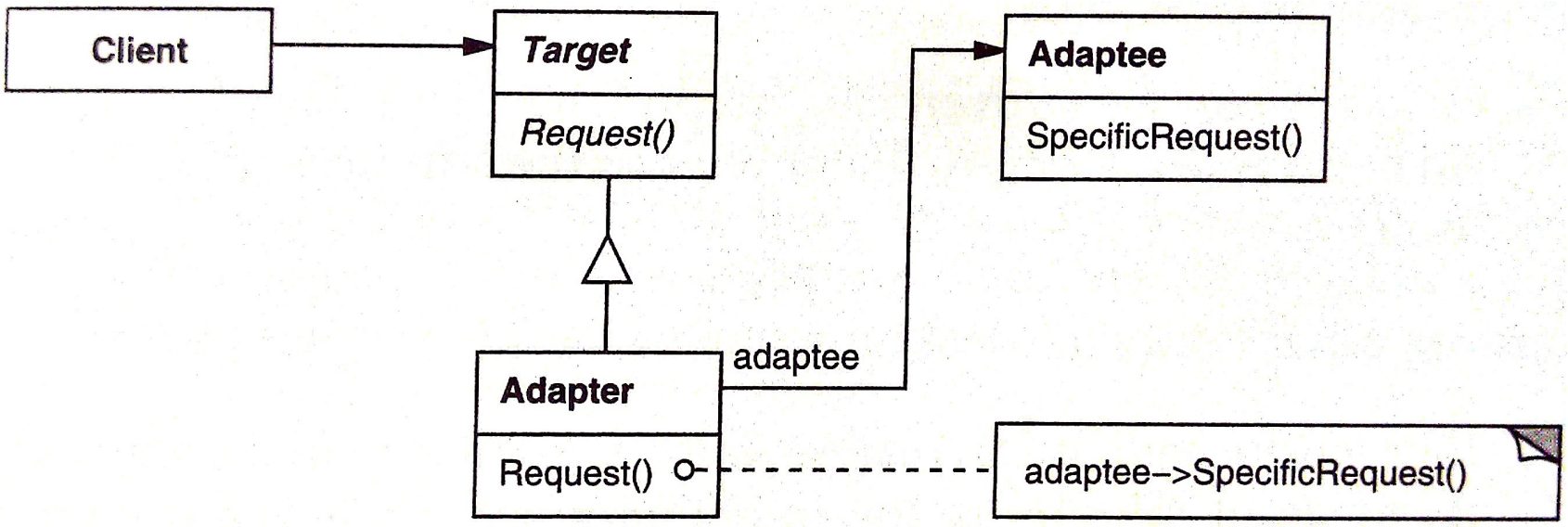


Design Patterns, Addison-Wesley, 1994

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

Next week

- Design by contract
- Activity Diagrams