Software Engineering I (02161)

Week 7

Assoc. Prof. Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2011

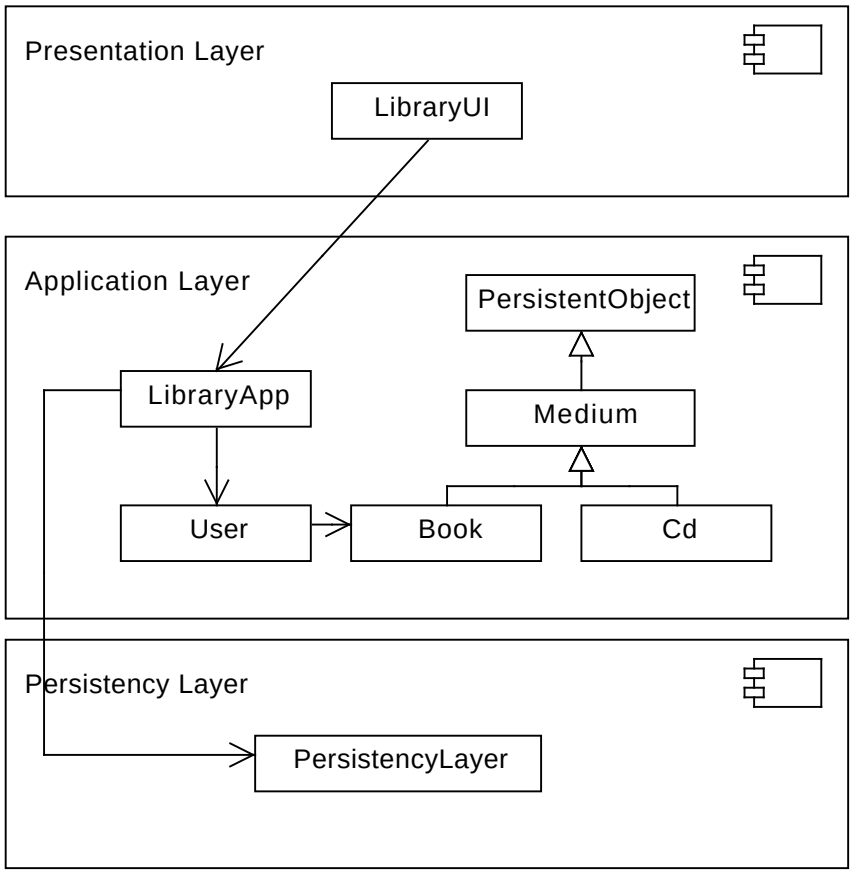
Recap

- State Machines
  - Good for representing that behaviour is changed as a reaction to events:

  → Implementation using the state pattern (a Design Pattern) based on distributed control
- Layered Architecture
  - Layers have their own set of responsibilities: e.g. Presentation–, Application–, Domain–, and Database/Infrastructer layer
  - Interface between layers is small

  → Easy to exchange layer
  - Separation between Presentation– and Application– layer allows to test application logic

Layered Architecture: Persistency Layer for the library application



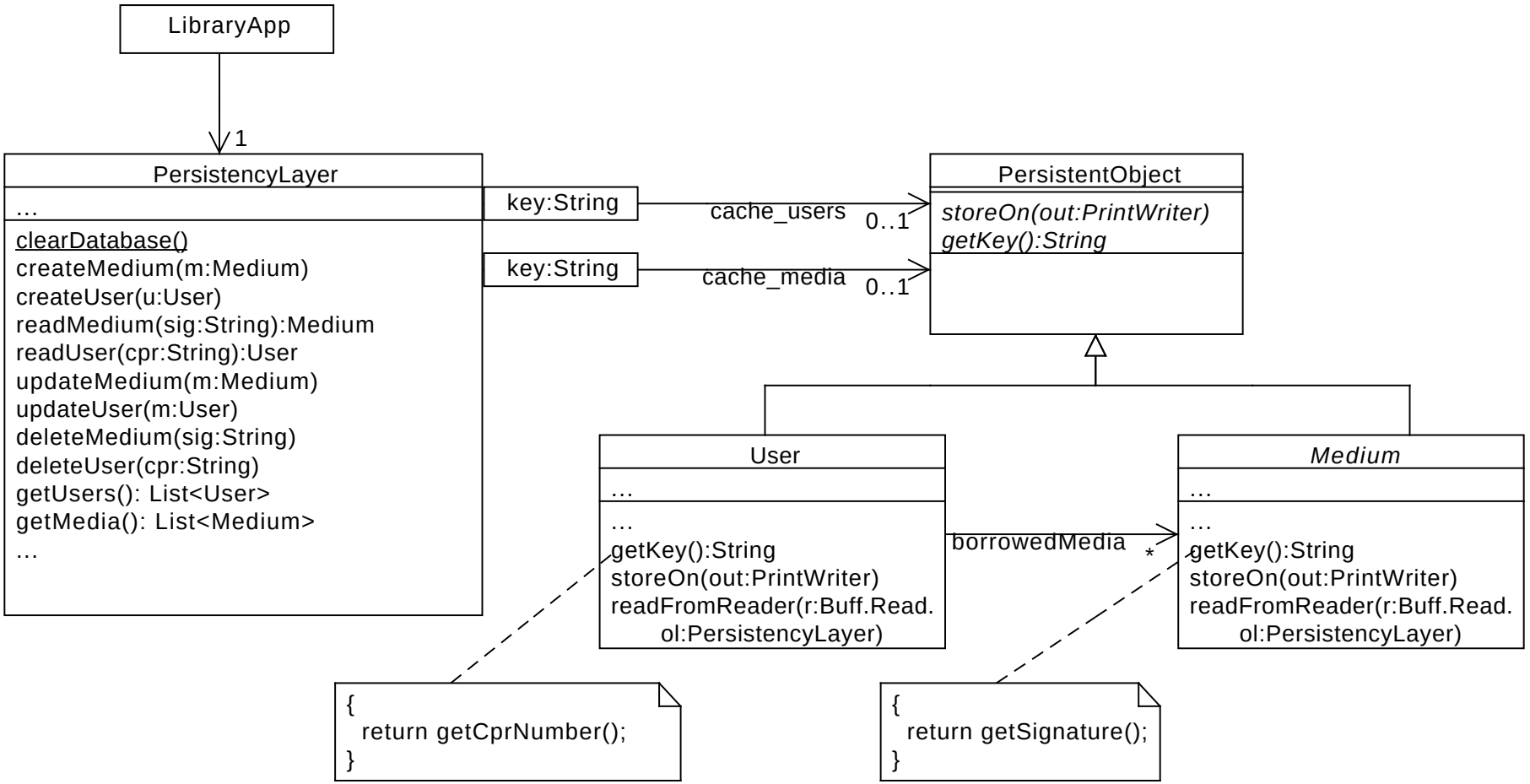
- For simplicity: Data (User and Medium) is stored in two files `users.txt` and `media.txt`; address has no file
- A book

```
dtu.library.app.Book
b01
some book author
some book title
Mar 13, 2011
<empty line>
```

- A user

```
dtu.library.app.User
cpr-number
Some Name
a@b.dk
Kongevejen
2120
København
b01
c01
<empty line>
```

Persistency Layer

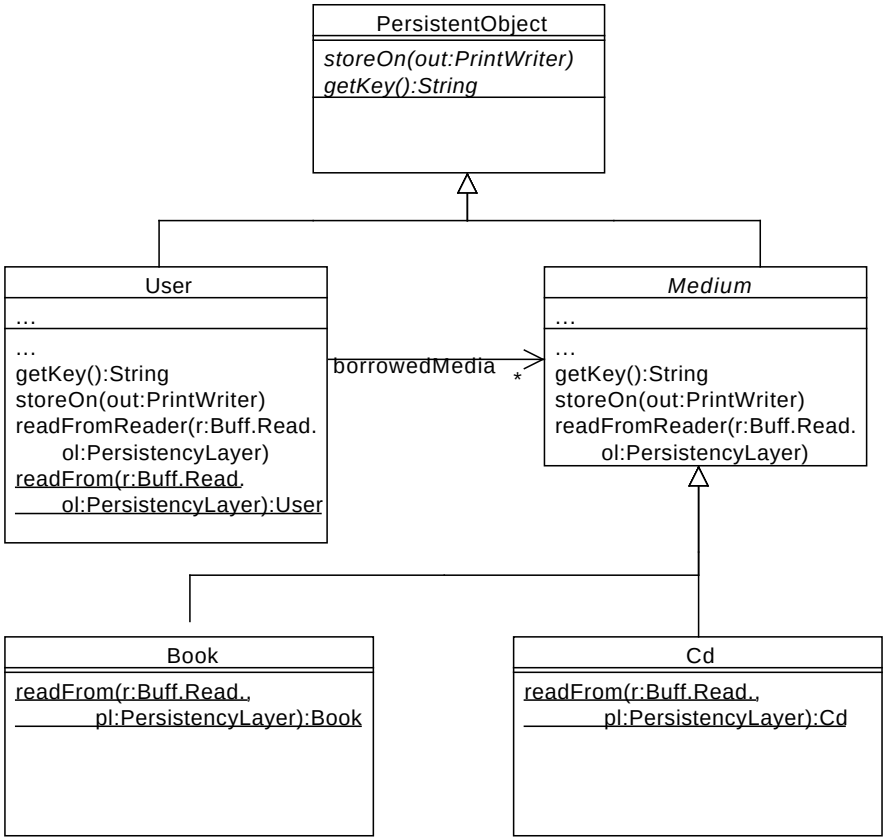


Layered Architecture: Persistency Layer for the library application

PersistencyLayer
cache_users cahce_medium
clearDatabase() createMedium(m:Medium) createUser(u:User) readMedium(sig:String):Medium readUser(cpr:String):User updateMedium(m:Medium) updateUser(m:User) deleteMedium(sig:String) deleteUser(cpr:String) getUsers(): List<User> getMedia(): List<Medium> …

- CRUD: Create, Read, Update, Delete: typical database operations
- clearDatabase: removes the two files users.txt and media.txt
- createMedium/User: appends a new record to the corresponding file
- readMedium/User: go sequentially through the files, reads the object and returns it if the key matches
- updateMedium/User: copy all entries in a new file; replace the old entry with the new entry on copying; rename the new file to the old file
- deleteMedium/User: The same as updateMedium/User, the difference is that the object to delete is not copied
- getUsers/Media

Reading/Writing User and Media objects



- storeOn writes a representation of the object on a writer
- readFrom is a static method that creates a new object from a read; it creates the object and delegates the initialisation to the object itself: i.e.

```
User u = new User();
u.readFromReader(reader,pl);
return u;
```

- readFromReader reads the state of an object from a reader
- Note that the user needs the PersistencyLayer to get from it the borrowed books based on their signatures

Use of Files

- Writing to files: Second argument to FileWriter constructor: if true, this means append to the file if the file exists, false means, replace the file if the file exists

```
FileWriter fw = new FileWriter(filename, true);
PrintWriter out = new PrintWriter(fw);
out.println("Some line");
out.print("Some string without new lline");
```

- Reading from files

```
FileReader fr = new FileReader(filename);
BufferedReader in = new BufferedReader(fr);
String line - in.readLine();
```

- Deleting and renaming files

```
File f = new File(filename);
f.delete();
f.renameTo(new File(new_filename));
```

Issues

- readMedium/User should return the same object if called twice with the same key
  - → use a cache of media/users and return the object in the cache if it exists
    - The cache maps keys to persistent objects, i.e., `Map<String,PersistentObject> cache`
    - Note: the cache needs also to work together with the add and delete operations
- updateMedium/User needs to be called whenever the state of a user/media changes (e.g. through borrowing and returning media)
  - → Don't forget to create tests for that

Tasks

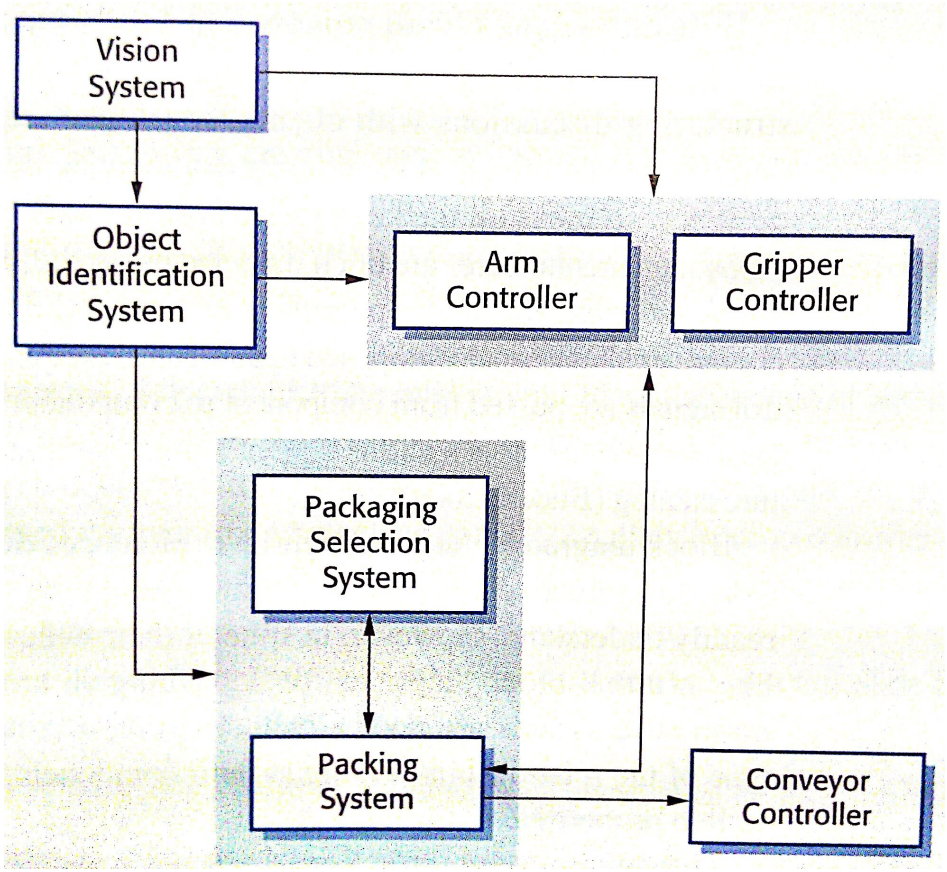
1) Implement the persistency layer as described (based on the provided tests)
   - Note: Don't implement the class diagram directly; instead implement parts of the diagram as necessary to so that the test pass

   → The diagram describes the final state to be achieved using TDD
   - Challenge: the CRUD operations for users are similar to those for media. Actually, they can be implemented for all instances of PersistentObject

   → DRY principle
2) Connect the persistency layer with the library application
   a) Change the library application code to use the persistency layer
      → This requires to add throws clauses to some of the methods of library application
   b) Adapt the tests as necessary
      → Note that most tests require now to have the database cleared before the tests can start (using PersistencyLayer.clearDatabase())
   c) Add additional tests to make sure that the database is updated when, e.g. books are borrowed or returned.

Architectural Design

- Identify the main structural components of the system
- Usually done after the requirements as a first design step
- Outcome is an architectural model or metaphor (XP)
- Example: Packing robot control system



Overlap Requirements Engineering and Architectural Design

- Ideally: System specification should not include design information
- Reality: For complex system the system specification will come with a structure
  - Also: Requirements may state architectural requirements:
  - e.g. application should run on the Web, mobile phone, desktop, ...

→ Requirements engineering may already yield a coarse system architecture

Software architecture

- Importance: It influences performance, robustness, distributability, and maintainability
  - Functional requirements are implemented by the components
  - Non-functional requirements depend on the architecture
- Advantage of an explicit system architecture
  1) Manage the complexity of the system and development task
  2) Communication with stakeholders
  3) Allows to analyse the system according to emergent properties (like performance, security, safety, etc.)
  4) Possibility for large-scale reuse

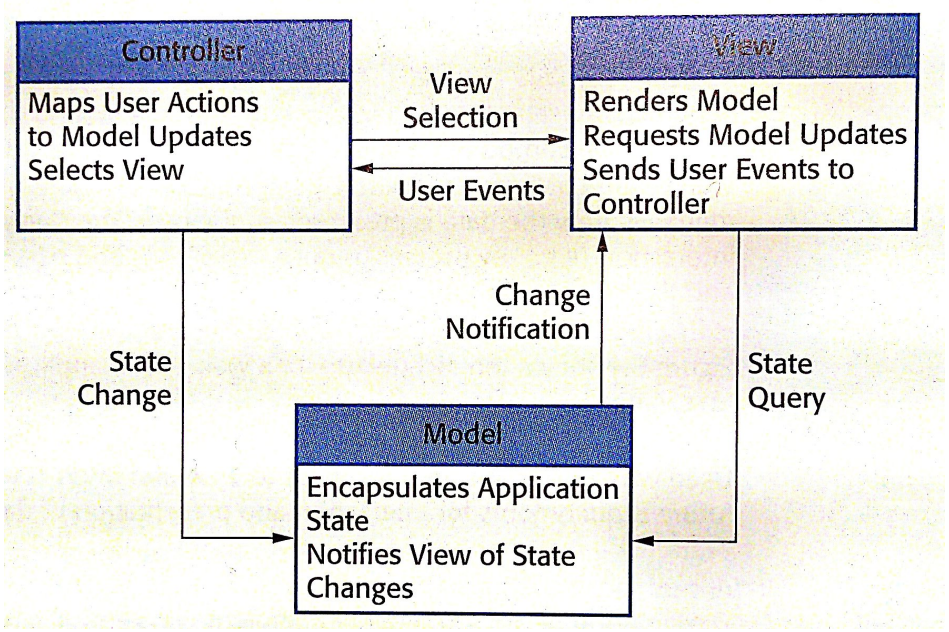
Notation for architectural design

- Commonly: Informal, boxes denoting components and sub-components and lines denoting data– or control-flow
- Also: UML component diagram with precise defined interfaces for the components
- Two purposes
  a) A way of facilitating discussion abut the system design
     - e.g metaphor in XP
  b) A way of documenting an architecture that has been designed
     - e.g. base of model-driven architecture where the implementation is derived from the architecture through model-transformations

Architectural Patterns/styles

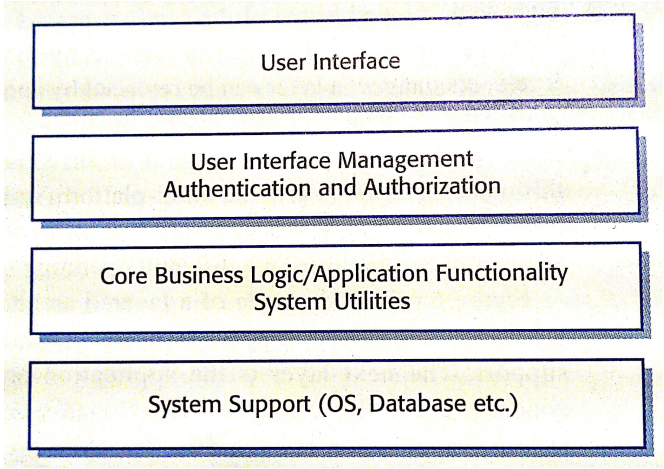
- There are a set of common architectural patterns:
  - Model View Controller
  - Layered Architecture
  - Repository Architecture
  - Client-Server Architecture
  - Pipe and filter Architecture
  - ...
- Quite often, the patterns are mixed, e.g. the client in a client-server architecture could be build using a layered architecture where the presentation layer uses the model-view-controller pattern

Model View Controller



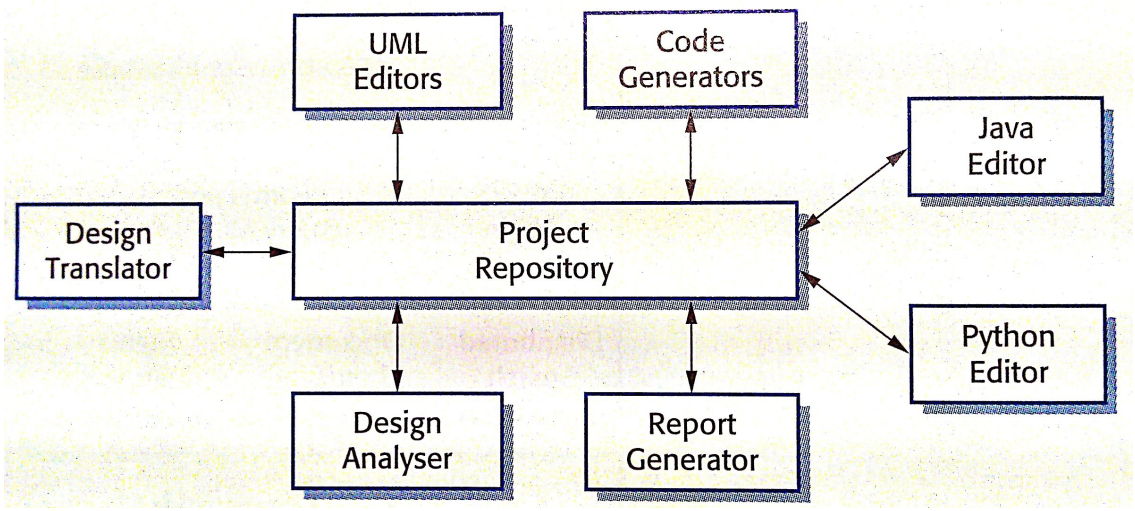
- Used when there are multiple ways to view and interact with data.
- Advantage: Easy to change and add data representations and interactions; supports different presentations of the same data where changes in one presentation are reflected in all others (→ Vending Machine UI's from last week)
- Disadvantage: Can involve additional code with simple data models and interactions.

Layered Architecture



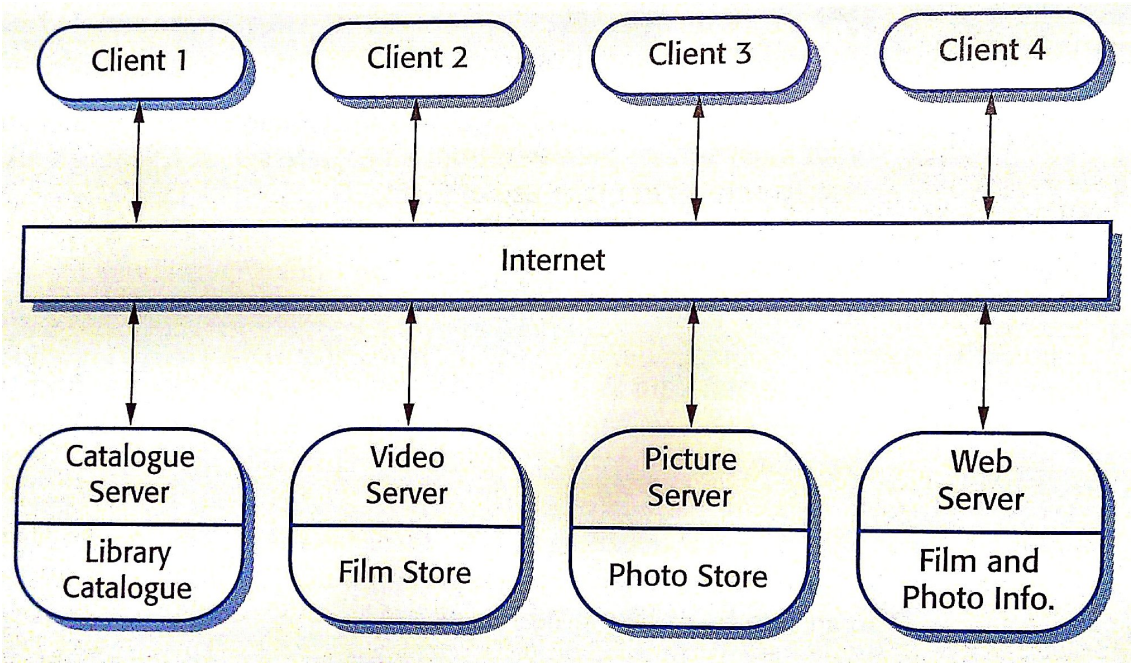
- Used when building new facilities on top of existing systems or separation of concerns is important (e.g. easy exchange of presentation– or data layer)
- Advantage: Easy replacement of entire layers; redundant facilities (like authentification) can be provided in each layer to increase the dependability of the system
- Disadvantage: Sometimes higher layers have to talk directly to lower layers; possible performance issues by propagating messages through layers

Repository Architecture



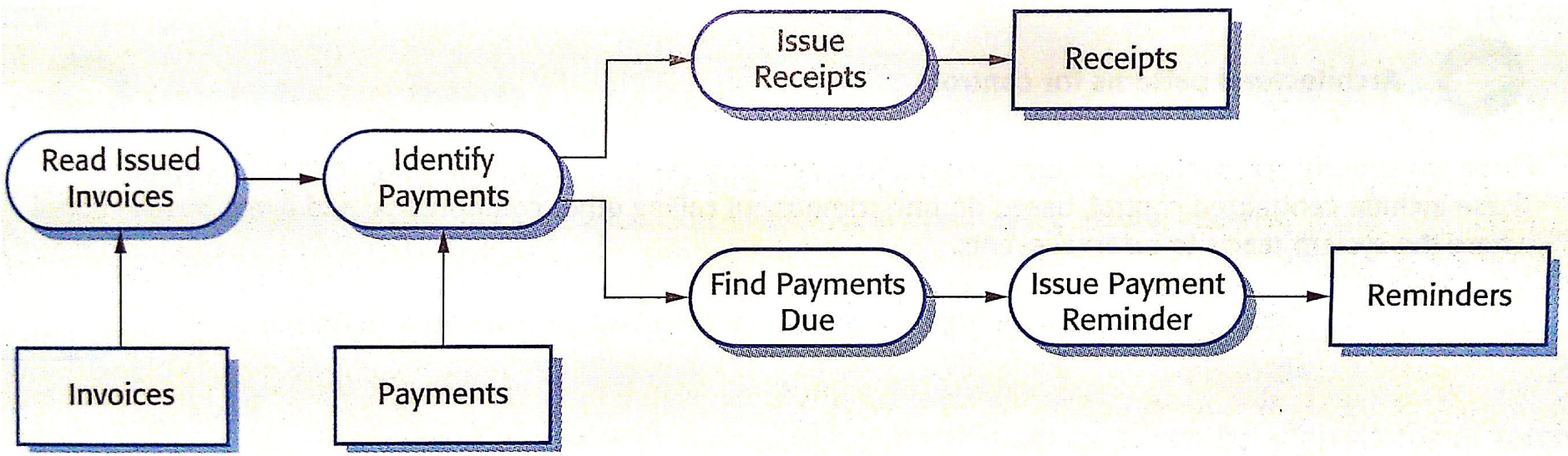
- Used when the system has large volumes of data
- Advantage: Independent of components (they don't have to know each other); components can easily be added or exchanged
- Disadvantage: Repository is a single point of failure; possible source for inefficiencies

Client-Server Architecture



- Used when data in a shared database has to be accessed from a range of locations
- Advantage: access to remote servers/data
- Disadvantage: single point of failure (the server); denial-of-service attacks

Pipes-and-Filter Architecture



- Used in data processing applications
- Advantage: Easy to understand; reuse of transformations
- Disadvantage: agreement on data structures; each transformation needs to parse and unparse the data

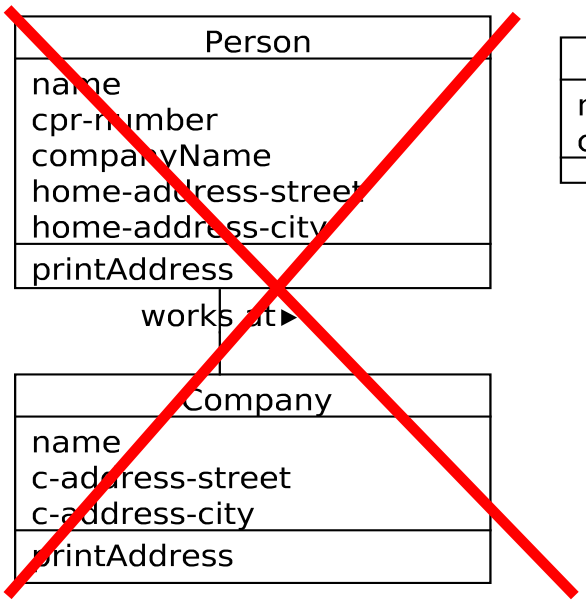
DRY principle

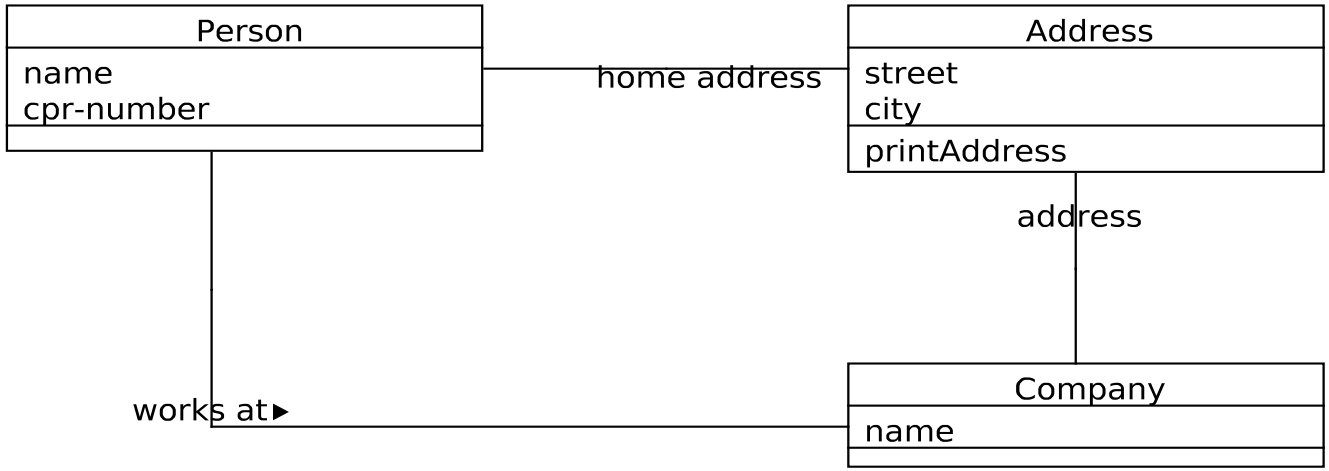
DRY principle

Don't repeat yourself: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

- Problem with duplication
  - Consistency: Changes need to be applied to each of the duplicates
  - Changes won't be executed because changes needed to be done in too many places
- Kind of duplication
  - Code duplications
  - Concept duplications
  - Code / Comments / Documentation
    - → Self documenting code
    - → Only document ideas, concepts, . . . that are not expressible (expressed) clearly in the code: e.g. What is the idea behind a design, what were the design decisions
      - Example: eUML: Class diagrams == Code
  - . . .

Example: Code Duplication





DRY principle

- Techniques to avoid duplication
  - Use appropriate abstractions
    - Inheritance
    - Classes with instance variables
    - Methods with parameters
    - refactor your software to remove duplications
    - ...

to refactor software

Change the structure of the software without changing its functionality

- Use generation techniques
  - generate documention from code
    - e.g. Javadoc generates HTML documentation from Java source files
    - e.g. `http://java.sun.com/javase/6/docs/api/`
  - generate code from UML models
    - most modern tools support this for class diagrams in both directions (i.e. code → diagram and diagram → code
  - ...

KISS principle

KISS principle

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

- Try to use the simplest solution first
  - Make complex solutions only if needed
- Strive for simplicity
  - Takes time!!
  - refactor your software to make it simpler

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”.

Course 02161 Exam Project

- Exam project: Monday 21.3 — Monday 9.5
- 10 min demonstrations of the software are planned for Monday 9.5
- To be delivered
  - the running software and source code
  - a report describing the software (Use cases, class diagrams, sequence diagrams, . . .)
- Group size: 2 – 4
- Group forming: next week
  - Either you are personally present or someone can speak for you
  - If not, then there is no guarantee for participation in the exam project

What is a pattern and a pattern language?

Pattern

A pattern is a solution to a problem in context

A pattern usually contains a

- discussion on the problem,
- the forces involved in the problem,
- a solution that addresses the problem,
- and references to other patterns

Pattern language

A pattern language is a collection of related patterns

History of patterns

- Christopher Alexander (architect)
  - Patterns and pattern language for constructing buildings / cities
  - → Timeless Way of Building and A Pattern Language: Towns, Buildings, Construction (1977/79)
- Investigated for use of patterns with Software by Kent Beck and Ward Cunningham in 1987
- Design patterns book (1994)
- Pattern conferences, e.g. PloP (Pattern Languages of Programming) since 1994
- Portland Pattern repository `http://c2.com/cgi/wiki?PeopleProjectsAndPatterns` (since 1995)

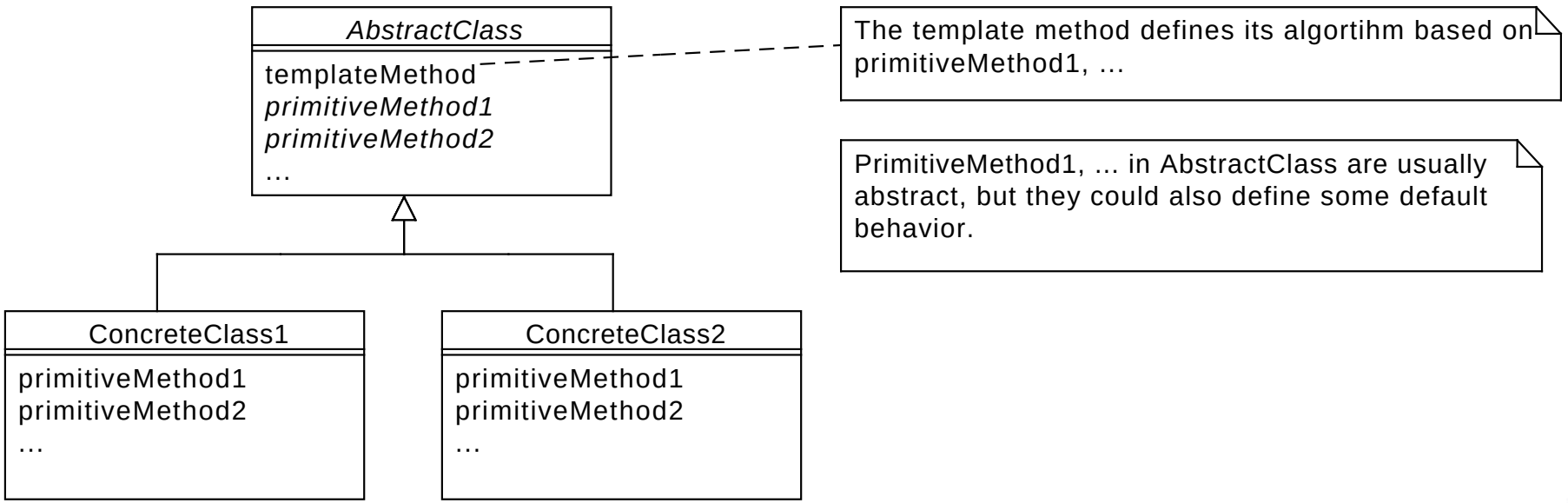
What is a design pattern?

- Design patterns book by ”Gang of Four” (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides)
- A set of best practices for designing software
  - E.g. Observer pattern, Factory pattern, Composite pattern, . . .
  - Many of the design patterns describe how to use decentralised control (i.e. object-oriented techniques) to solve common design problems
- Places to find patterns:
  - Wikipedia `http://en.wikipedia.org/wiki/Design_pattern_(computer_science)`
  - Portland Pattern repository `http://c2.com/cgi/wiki?PeopleProjectsAndPatterns` (since 1995)
  - Wikipedia `http://en.wikipedia.org/wiki/Category:Software_design_patterns`

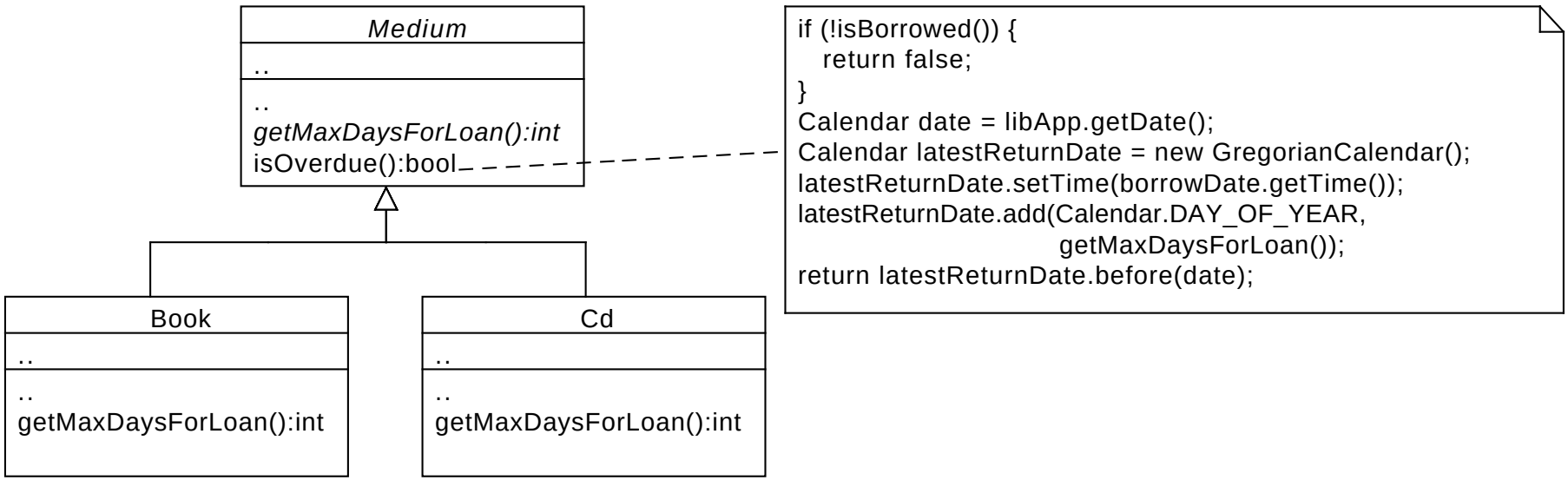
Template Method

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: Library Application

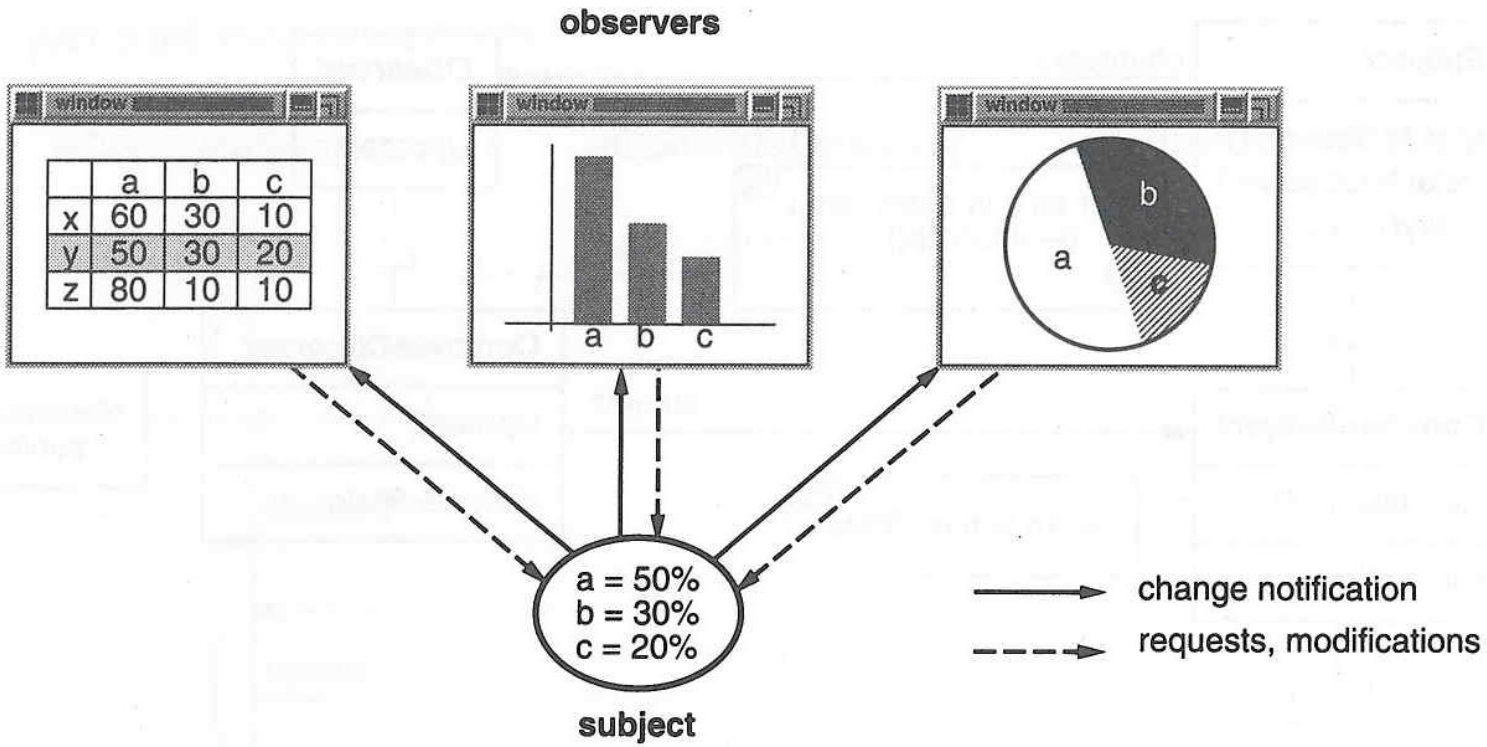


The computation of isOverdue depends on getMaxDaysForLoan which is different for Book and Cd.

Observer Pattern

Observer Pattern

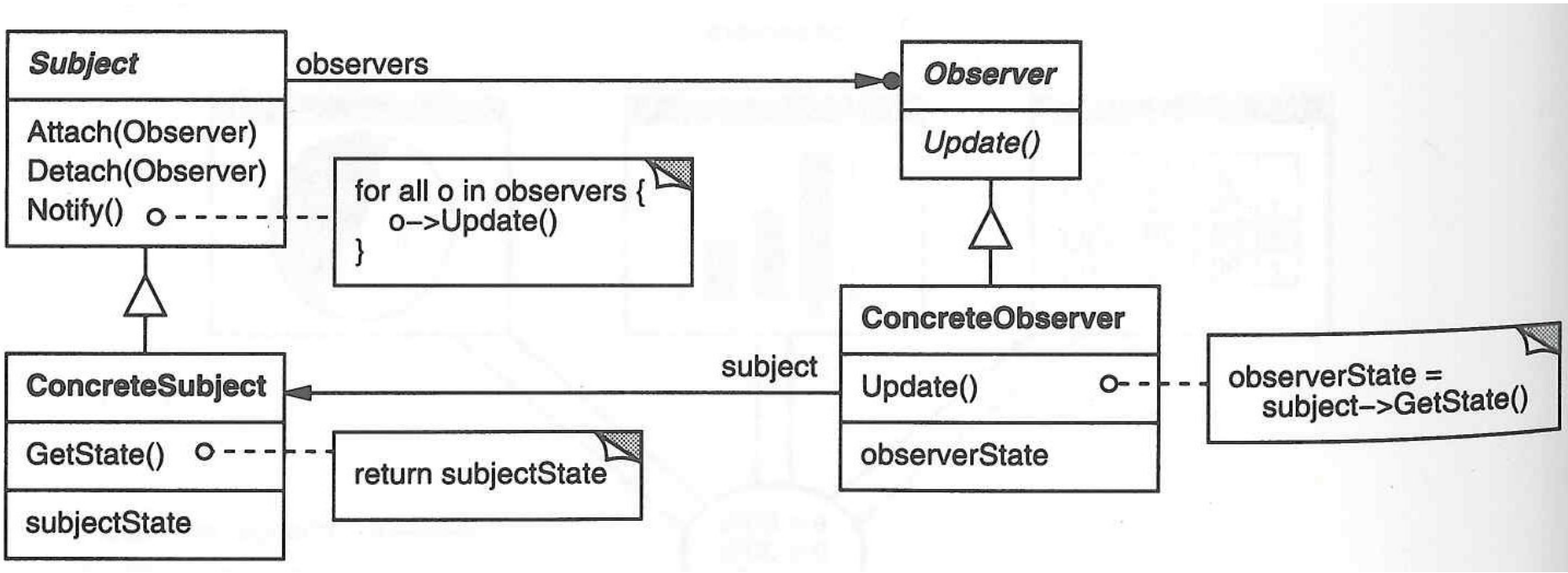
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



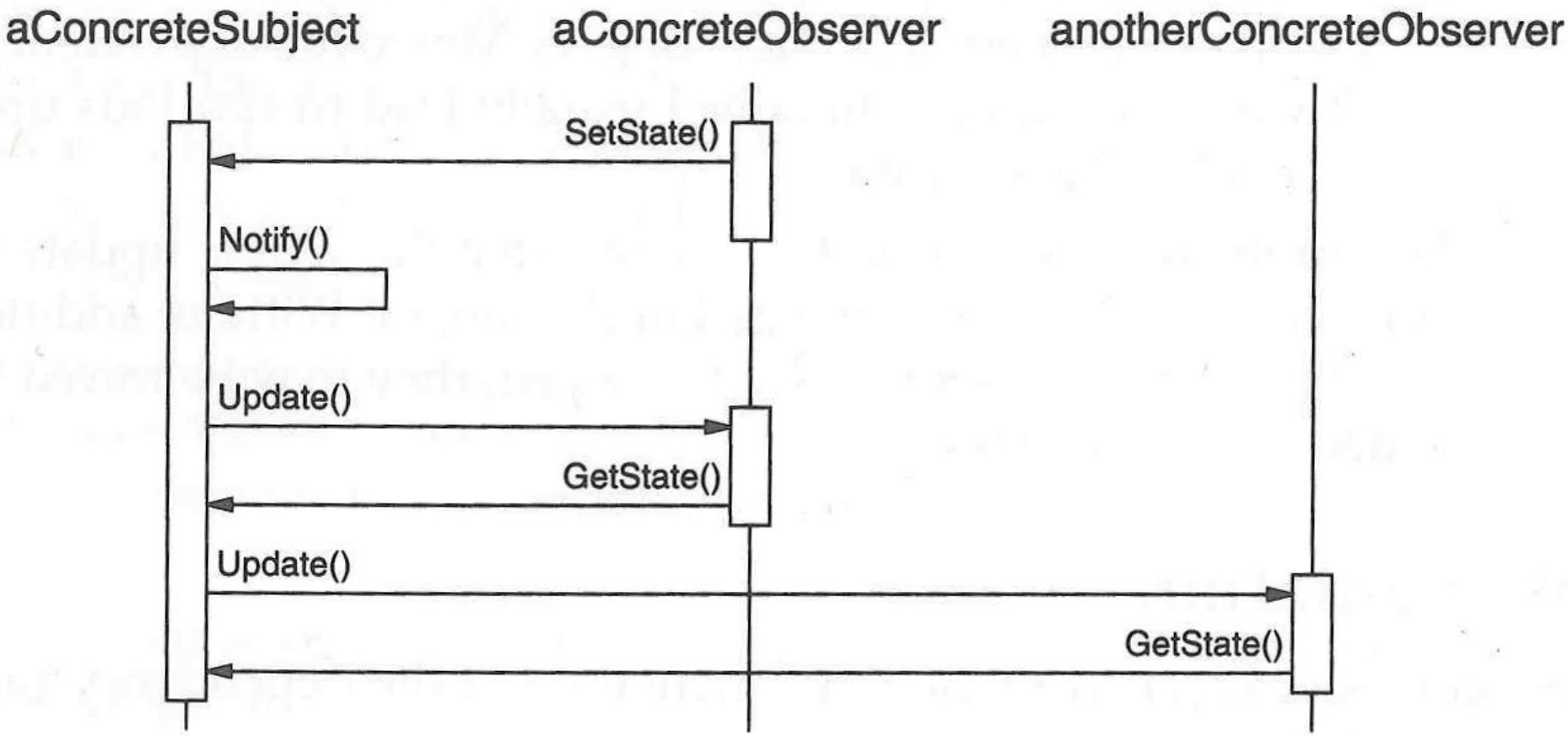
Observer Pattern

- The basic idea is that the object being observed does not know that there are observers
  - → observers can be added independently on the observable (also called subject)
  - → new types of observers can be created without changing the subject
- The observer pattern is used often in GUI programming to connect the presentation of a model with the model itself

Observer Pattern



Observer Pattern



Implementation in Java

- Support from the class library: One abstract class and interface:
- Interface java.util.Observer
  - Implement update(Observable o, Object aspect)
- Class java.util.Observable
  - Provides connection to the observers
  - Provides methods addObserver(Observer o), deleteObserver(Observer o)
    - To add and delete observers
  - setChanged()
    - Marks the observable / subject as dirty
  - notifyObservers(), notifyObservers(Object aspects)
    - Notify the observers that the state of the observable has changed
    - The aspect can be used to say what has changed in the observable

Example: Stack with observers

```
public class Stack<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();
   }
   ...
}
```

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.

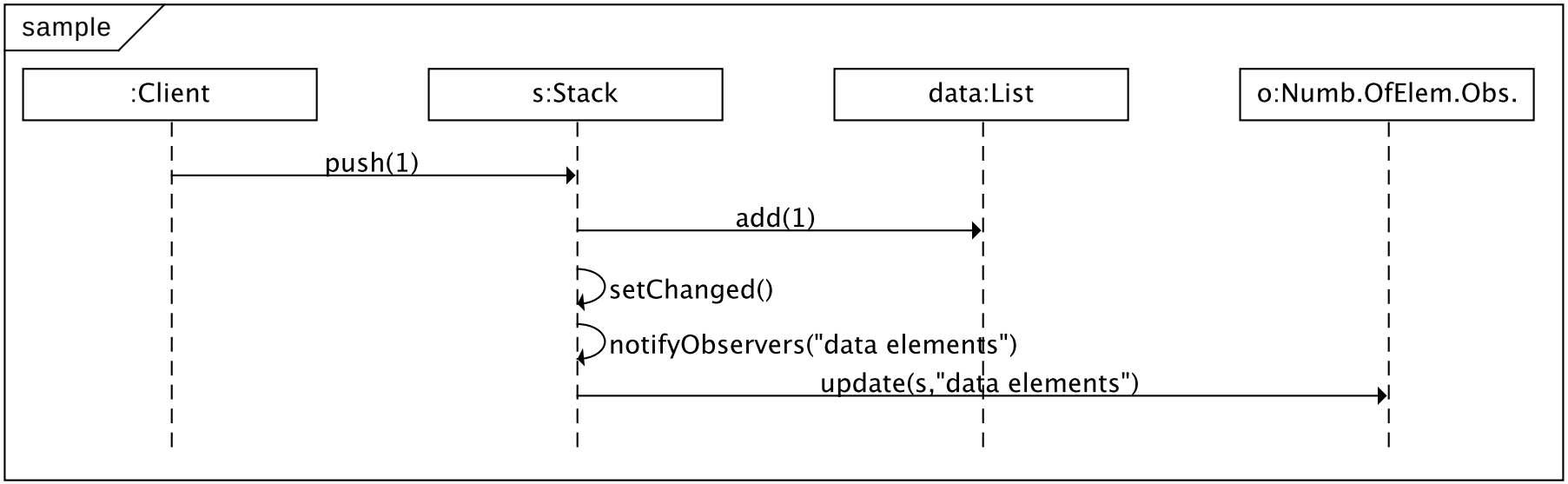
```
class NumberOfElementsObserver() implements Observer {
   Stack<E> stack;

   NumberOfElementsObserver(Stack<E> st) {
      stack = st;
   }

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

Example: Stack observer

Adding an observer

```
....
Stack<Integer> stack = new Stack<Integer>;
NumberOfElementsObserver observer =
            new NumberOfElementsObserver(stack);
stack.addObserver(observer);
stack.push(10);
stack.pop();
...
stack.deleteObserver(observer)
...
```

Sequence diagram for the stack

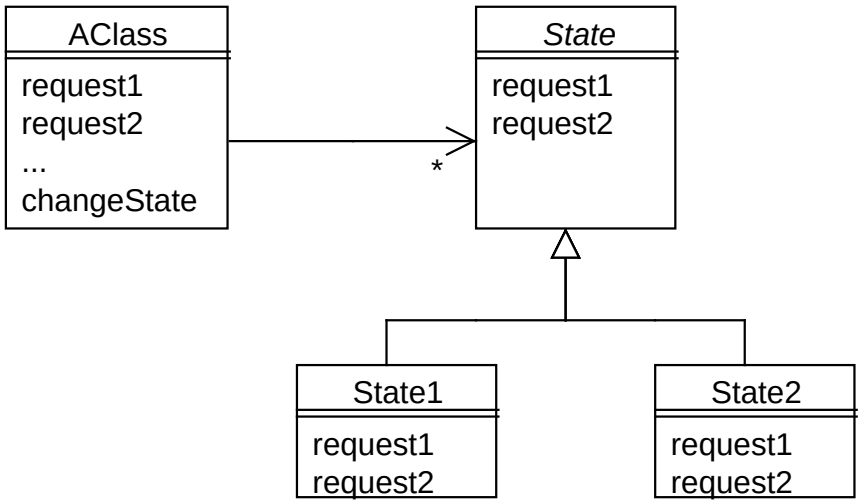


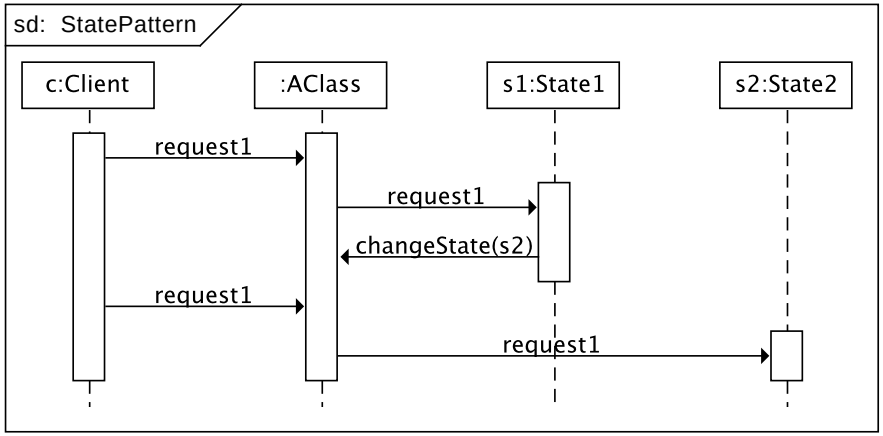
State Pattern

State Pattern

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

- This pattern delegates the behaviour of one object to another object

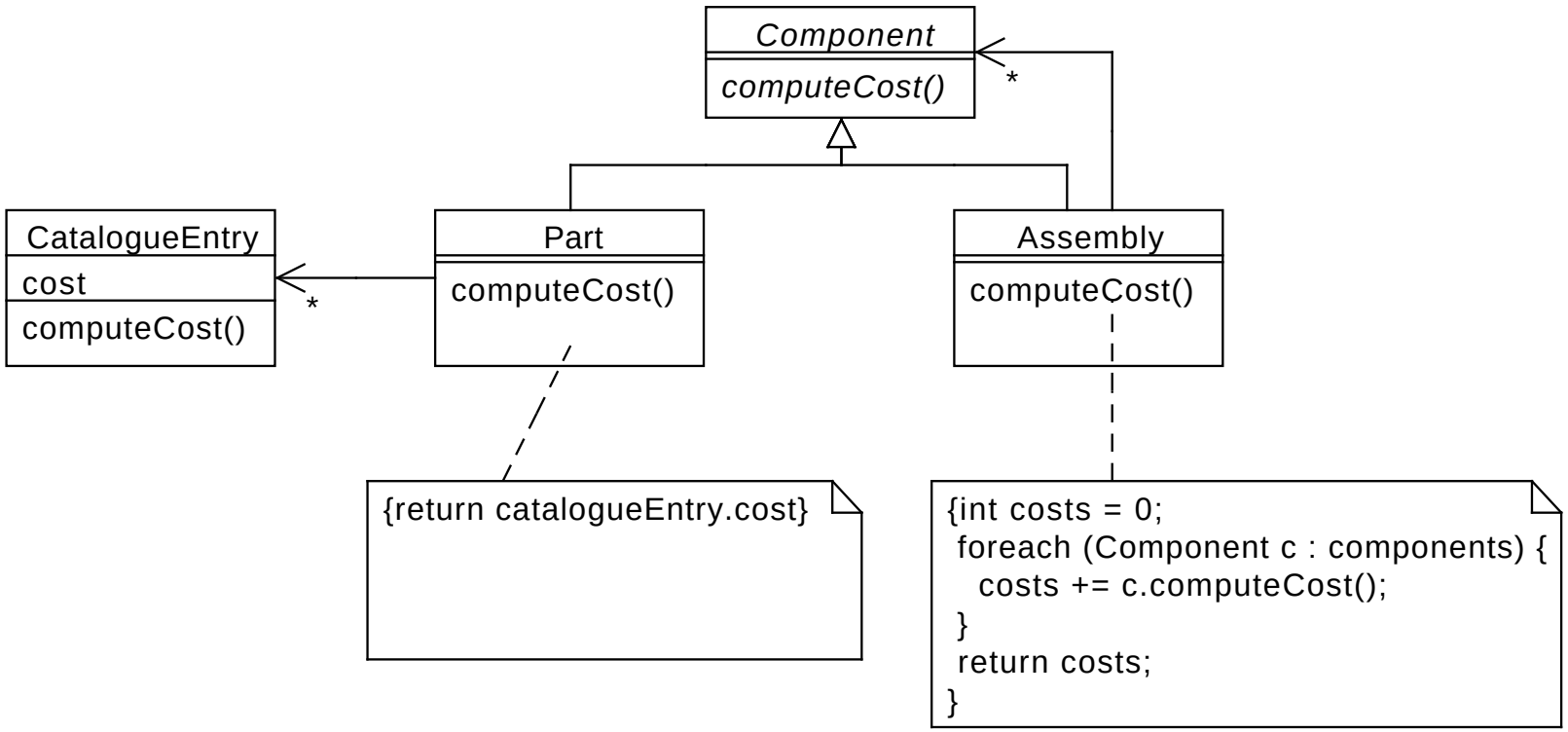




Composite Pattern

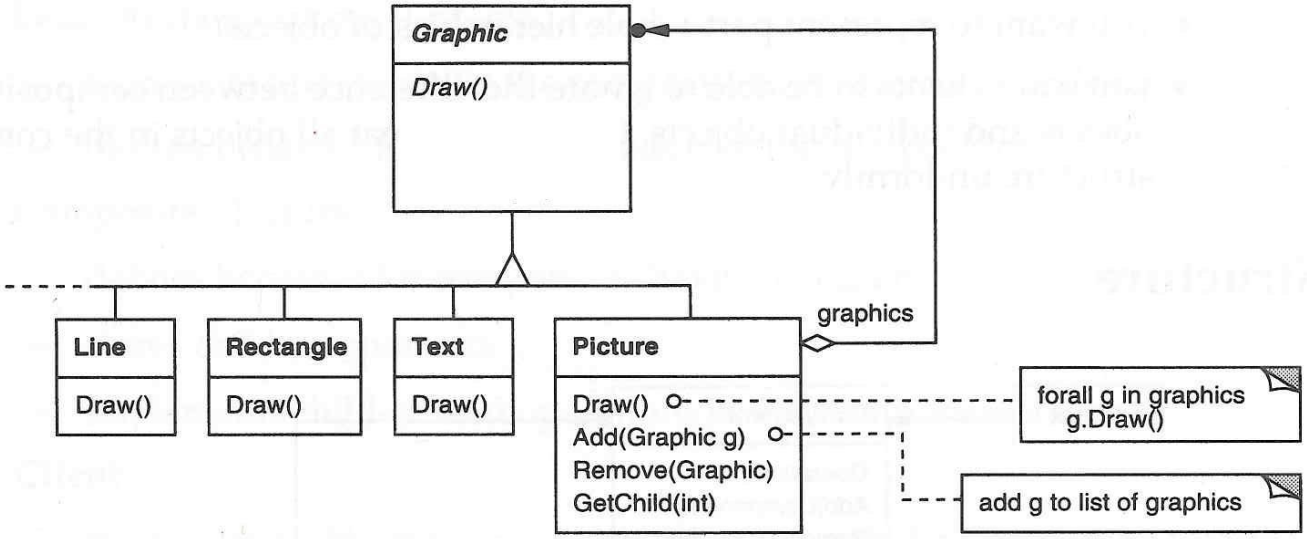
Composite Pattern

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

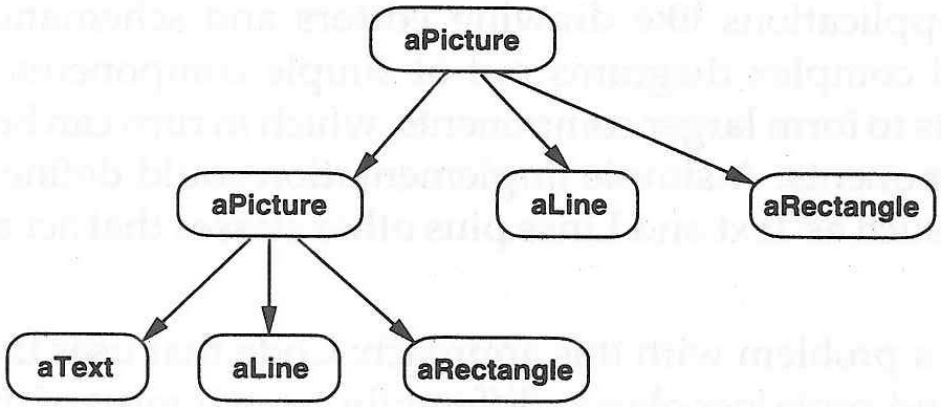


Example: Graphics

- Class Diagram



- Instance diagram



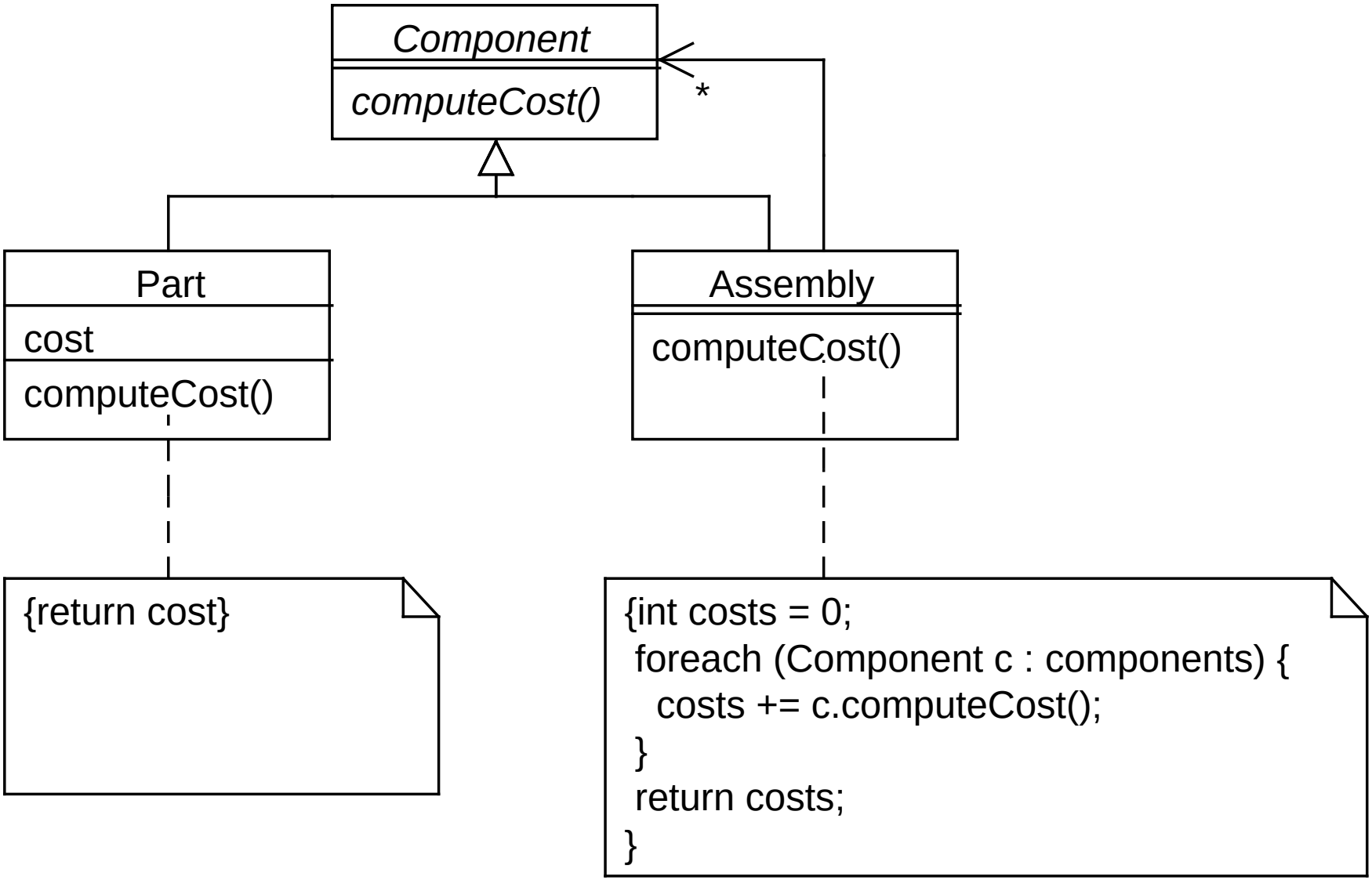
Visitor Pattern

Visitor Pattern

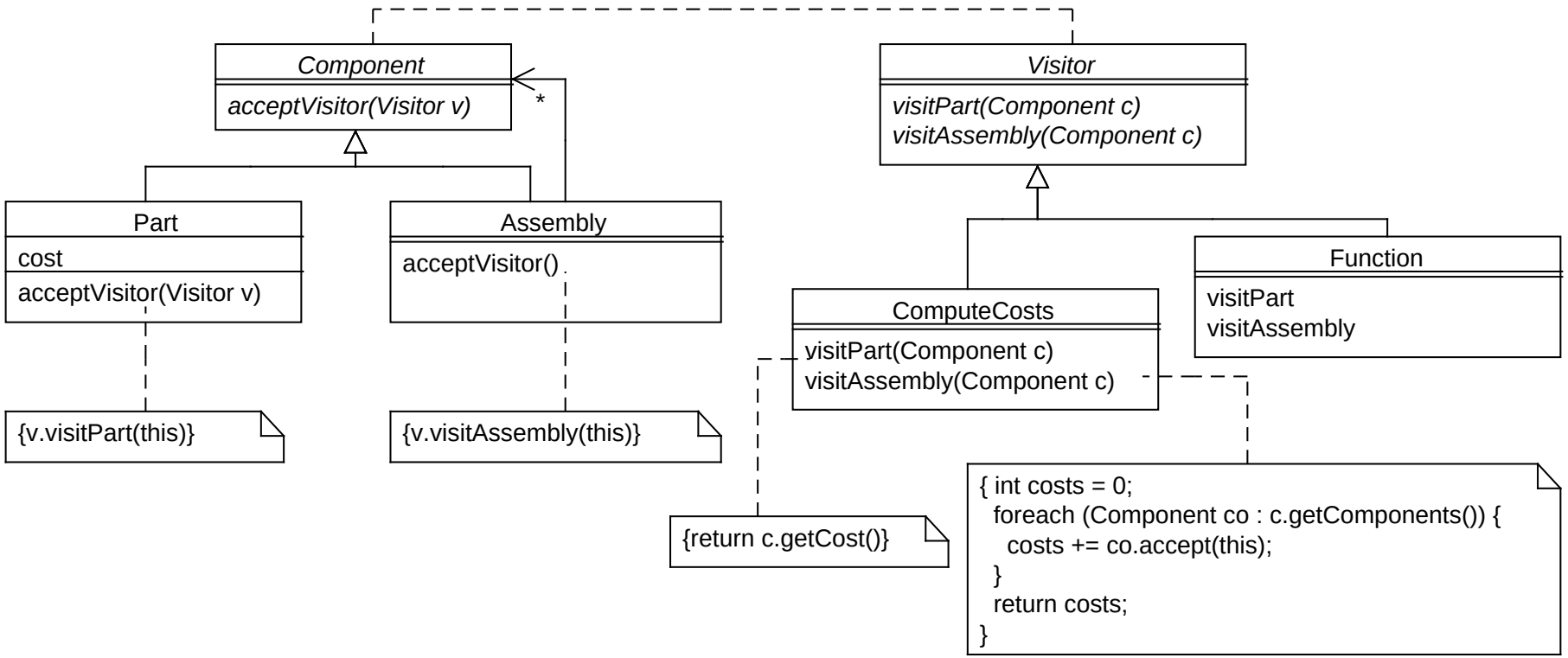
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

- The object structure (e.g. based on a composite pattern) provides access to itself through a set of methods

Example: compute costs for components



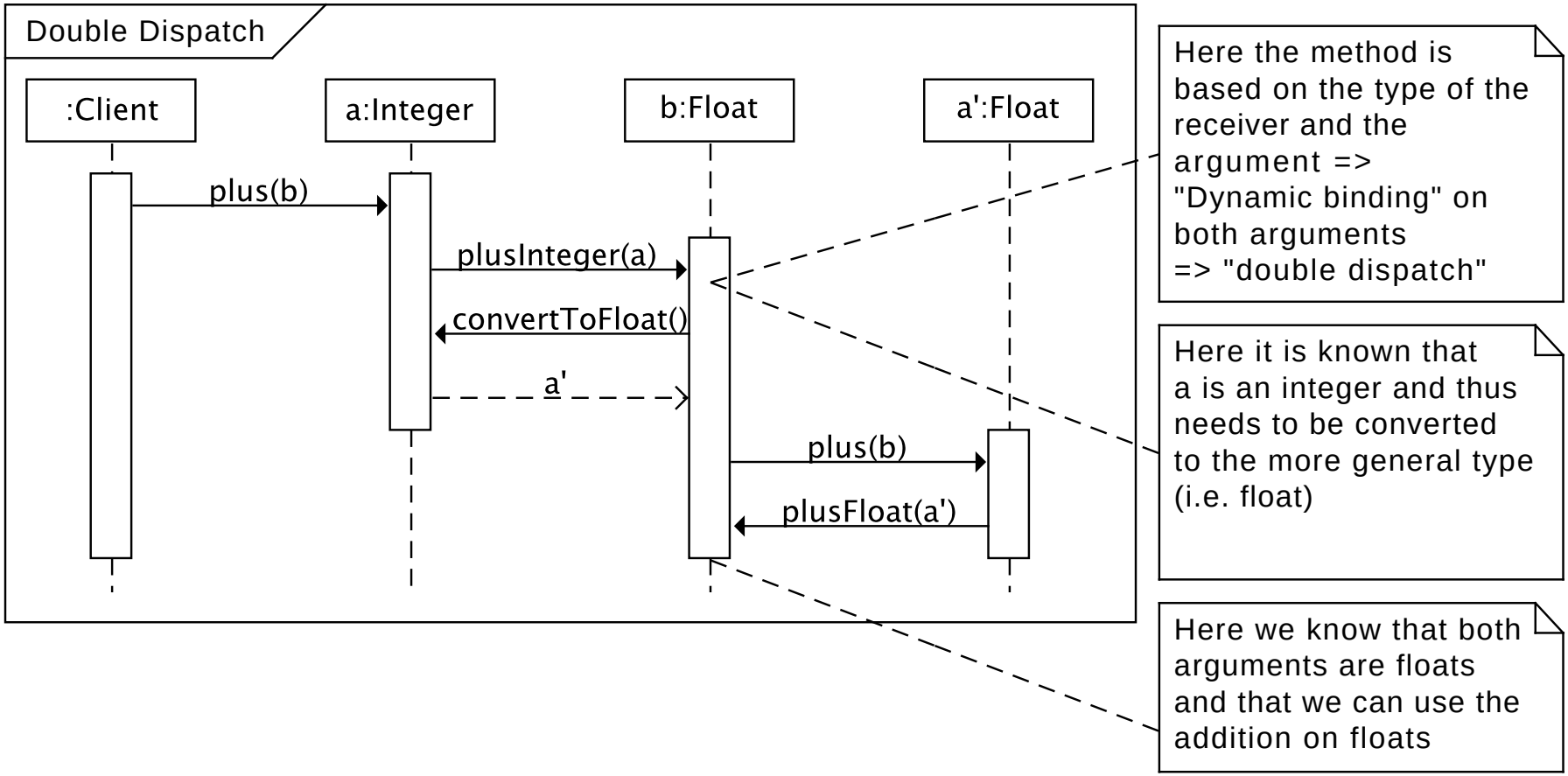
Example: compute costs as a visitor



Visitor pattern

- The trick of the visitor is to use double dispatch
  - add type information to the method name
    - acceptVisitor → visitPart, visitAssembly
- Use the visitor pattern if
  - The functions don't belong to the concept of the object structure:e.g. generator functions
  - One should be able to do traverse an object structure without wanting to add operations to the object structure
  - One has several functions almost the same. Then one can use the visitor pattern and inheritance between the visitors do define slight variants of the functions (e.g. only overriding acceptPart)
- Do not use it
  - if the complexity of the visitor pattern is not justified
  - if the functions belongs conceptually to the object structure
  - If the flexibility of the visitor is not needed, e.g.. if one only wants to add one function

Double Dispatch



e.g. found in Smalltalk or Ruby where numbers are first class objects

Summary Design Patterns

- Original Gang of Four book:
- 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 ...

Summary

- Layered Architecture: Persistent Layer
- Architecture
- Good Design
- Design Patterns (I):
  - (Object-oriented) solutions to common design problems
  - Template Method, Observer Pattern, State Pattern, Composite Pattern, Visitor Pattern