02161: Software Engineering I

Week 8: Design Patterns

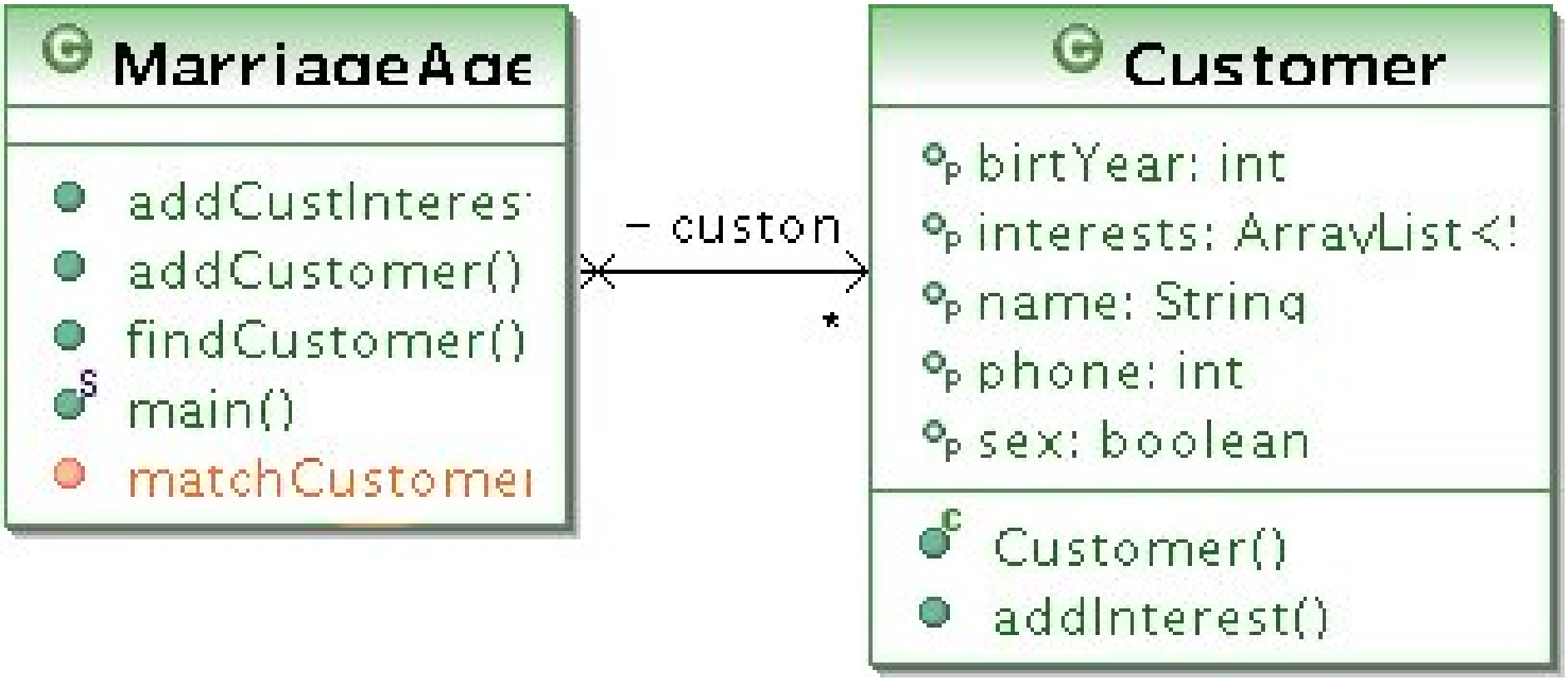
Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2008

Contents

What is good design? MarriageAgency Example

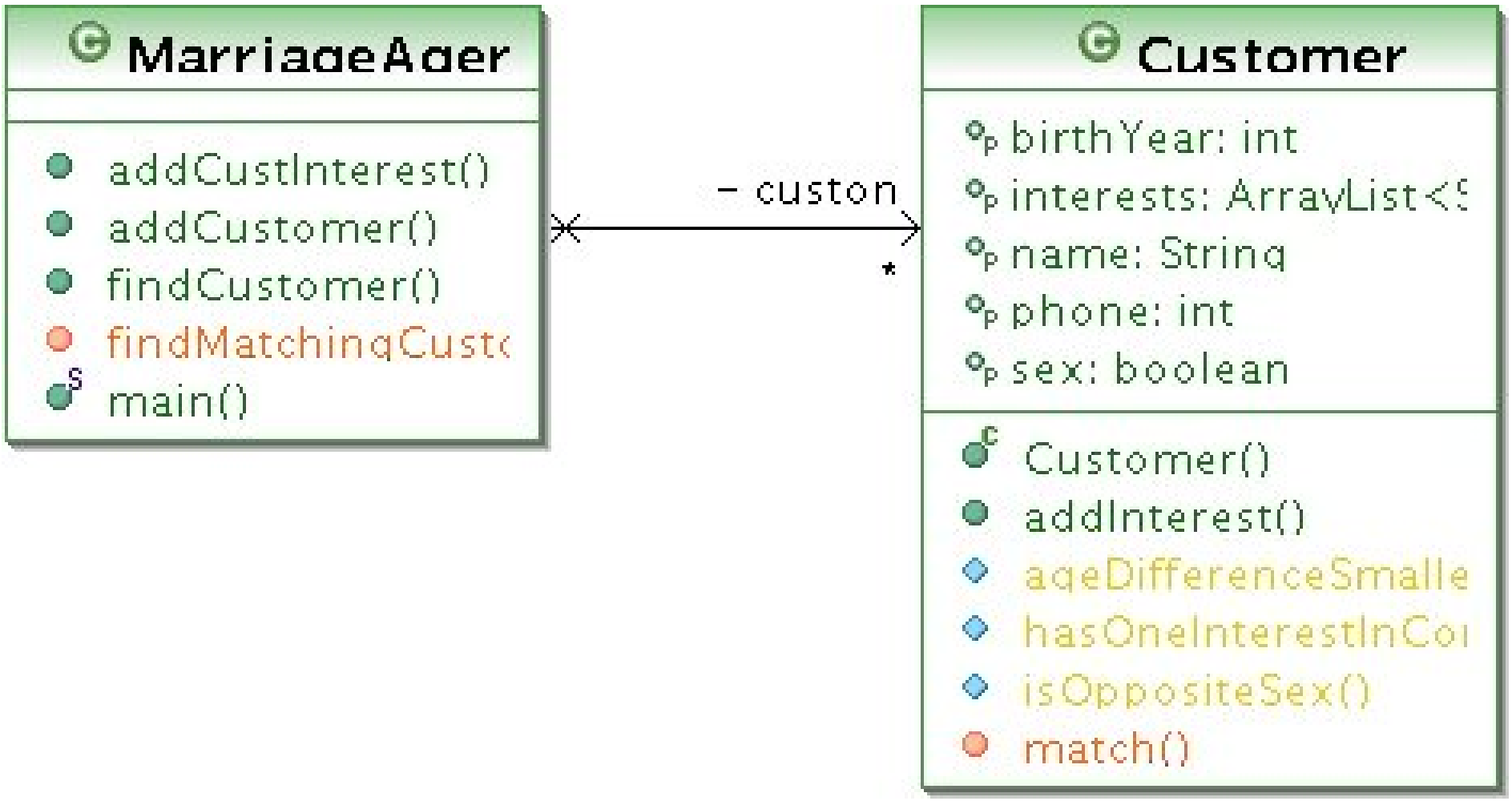


What is good design? MarriageAgency Example 2

Method matchCustomer in class MarriageAgency

```
public ArrayList<Customer> matchCustomer(Customer customer){
  ArrayList<Customer> res = new ArrayList<Customer>();
  for (Customer potential : customers) {
    if (potential.getSex() != customer.getSex()) {
      int yearDiff = Math.abs(potential.getBirtYear()
                     -customer.getBirtYear());
      if (yearDiff  <= 10) {
        for (String interest : potential.getInterests()) {
          if (customer.getInterests().contains(interest)) {
            res.add(potential);
            break;
          } } } } }
  return res;
}
```

What is good design? Improved Design



What is good design? Improved Design 2

Method findMatchingCustomers in class MarriageAgency

```
public ArrayList<Customer>
        findMatchingCustomers(Customer customer) {
  ArrayList<Customer> res = new ArrayList<Customer>();
  for (Customer potential : customers) {
    if (customer.match(potential)) {
      res.add(potential);
    } }
  return res;
}
```

What is good design? Improved Design 3

Methods in class Customer

```
public boolean match(Customer c) {
  return isOppositeSex(c)
       && ageDifferenceSmaller(c,10)
       && hasOneInterestInCommon(c);
}

protected boolean isOppositeSex(Customer c) {
  return sex != c.getSex();
}

protected boolean ageDifferenceSmaller(Customer c, int age) {
  return Math.abs(this.getBirthYear() - c.getBirthYear()) <= age;
}

protected boolean hasOneInterestInCommon(Customer c) {
  for (String interest : getInterests()) {
    if (c.getInterests().contains(interest)) {
    return true;
    } }
  return false;
}
```

Why good design / implementation?

- Enhancing the readability and the quality of the design allows
  - to better understand the structure of the program
  - to make the program more flexible

  → allowing to adapt the program to new requirements

  → to better find bugs
- Implementing larger software is not a linear process (i.e. Requirements analysis, design, implementation)
- Instead it is an evolutionary process
  - Bit of requirements analysis, bits of design, bits of implementation
  - Bit of requirements analysis, bits of design, bits of implementation
  - ...
- Each of these steps provides one with insight to better structure a system
- Don't underestimate how long a program can live and needs to be maintained and adapted

Example of a long living program

- I wrote a program to visualize the structure of TeX-documents
  - written 19 years ago to help me writing my masters thesis
- Today, I am still using and enhancing the program
  - generates interconnected Web pages that from a Web sites
  - can generate read Wiki pages and generate PDF files from them
  - generates the slides for this lecture based on an outline structure
  - creates and reads mind maps

How does one achieve better design?

- Write readable and self documenting programs
  - → Use self documenting names (e.g. class Person or variable person instead of class P or px1
  - → Other tips can be found in the book *Implementation patterns* by Kent Beck and in the book *The Pragmatic Progammer* by Andrew Hunt and David Thomas
- Create a domain language for your problem and then use it in your program code
  - → Use classes and methods to capture the domain language
- Know Design patterns and use them as appropriate

Contents

DRY principle

DRY principle

Don’t repeat yourself

- The idea is to avoid any 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
  - . . .
- 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

DRY principle

- Techniques to avoid duplication
  - Use appropriate abstractions
    - Inheritance
    - Classes with instance variables
    - Methods with parameters
    - ...
  - Use generation techniques
    - generate documention from code
    - generate code from more domain specific languages
    - ...

KISS principle

KISS principle

Keep it simple, stupid

- Try to use the simplest solution first
- Strive for the simplest solution
- Make complex solutions only if needed

Albert Einstein

”Everything should be made as simple as possible, but no 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”.

YAGNI principle

YAGNI principle

You ain't gonna needed it

- Avoid doing something for which there is no need → KISS
  - This happens a lot with design patterns and thinking to much ahead
    - "I am now using observer pattern because I think it provides me with the flexibility later"
    - "I am create a seperate interface which my class implements, because maybe later I need different implementations for that interface"

  → Different kind of flexibility
    - Make your code refactorable: e.g. with tests, using code expressing its intention
  - In practice both are needed: design for change and make your design changable

High cohesion — low coupling

High Cohesion

- A class groups a set of a related methods

→ an object is self contained and represents an entity

Low coupling

- An object / class is connected only to a few other classes
- It fullfils its repsonsibility by delegating responsibility to other objects
  - → c.f. CRC card game rules

- High cohesion & low coupling are a corner stone of good design
  - Low coupling reduces the dependency on other objects
    - It is easier to change / exchange one object when it is only connected to a limited number of other objects
  - High cohesion supports low coupling by grouping related functionality and data

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

- The Law of Demeter is a special case of low coupling

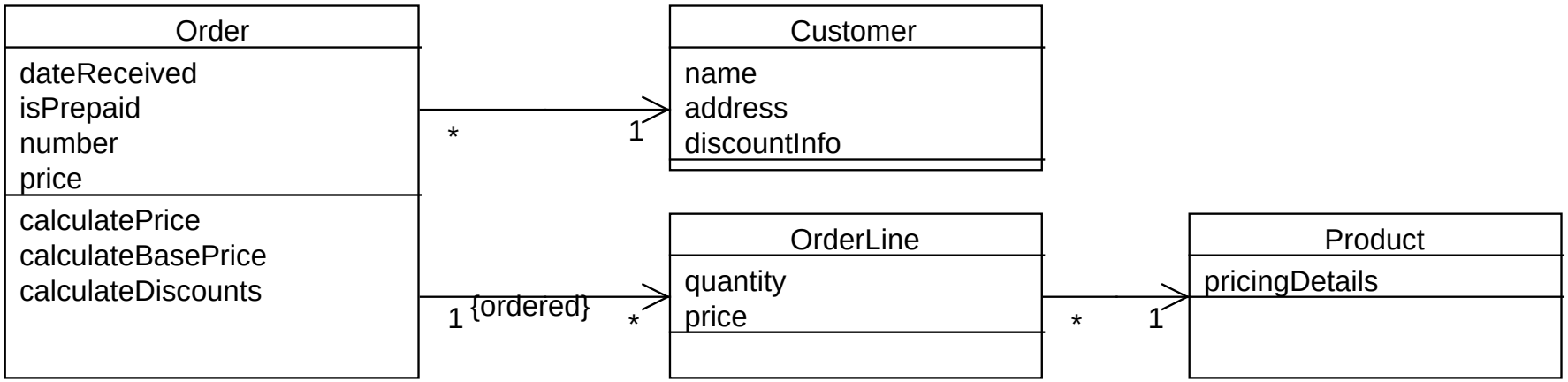
→ To achieve low coupling one needs to delegate functionality, e.g. the computation of the calculation of the price is moved from order to orderline

  → leads to decentralised control

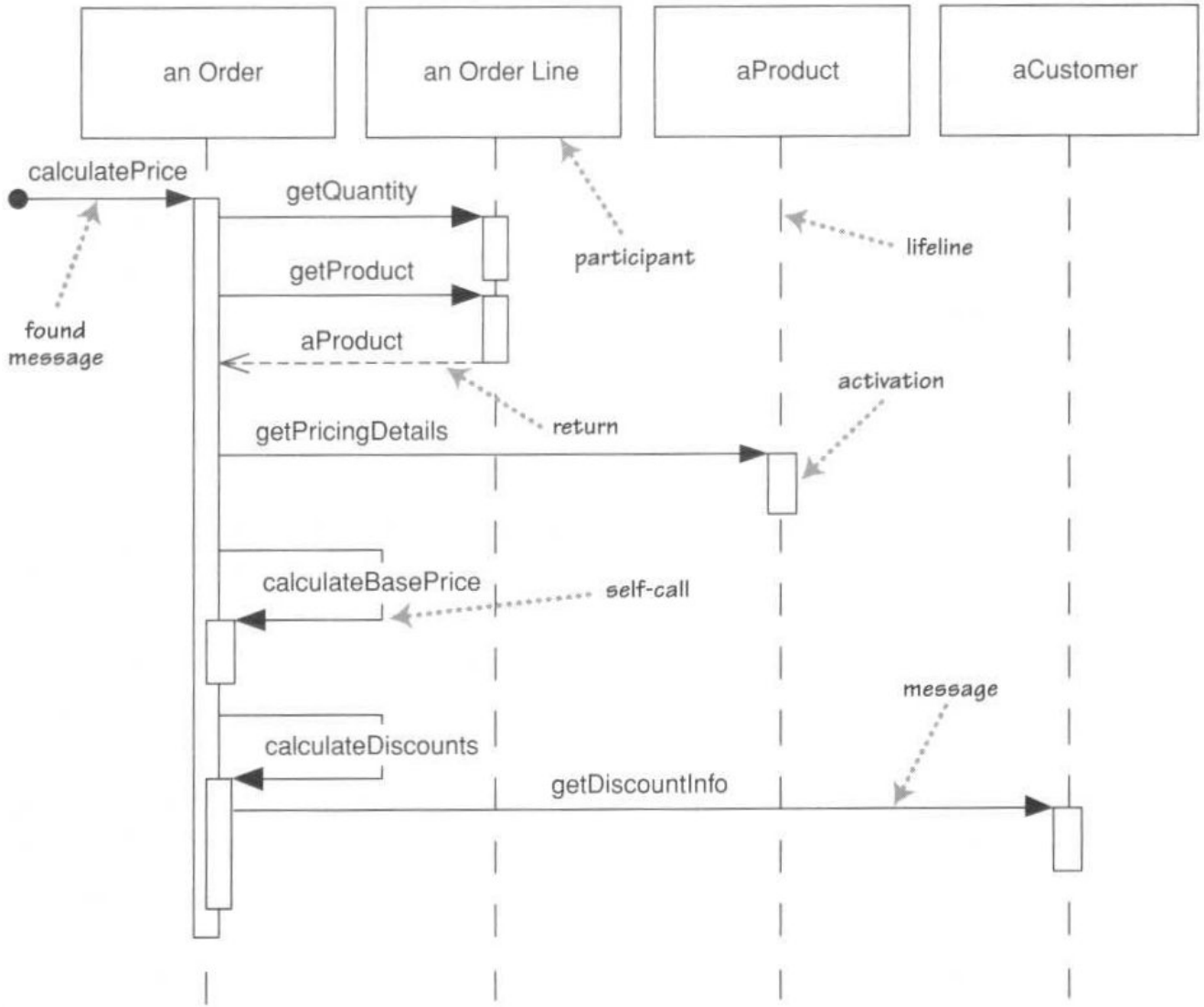
Centralised vs Distributed Control

- Centralised control
  - The method of one object does all the work
  - The remaining objects are merely data objects and usually don't have their own behaviour
  - Typical for a procedural programming style
- Distributed control
  - Objects collaborate to achieve one task
    - "Instead of doing myself the work, I delegate work to other object"
  - Each object in a collaboration has behaviour (= is a "real" object)
  - Typical object-oriented style
    - Each object has its own responsibilities
    - Facilitates polymorphism

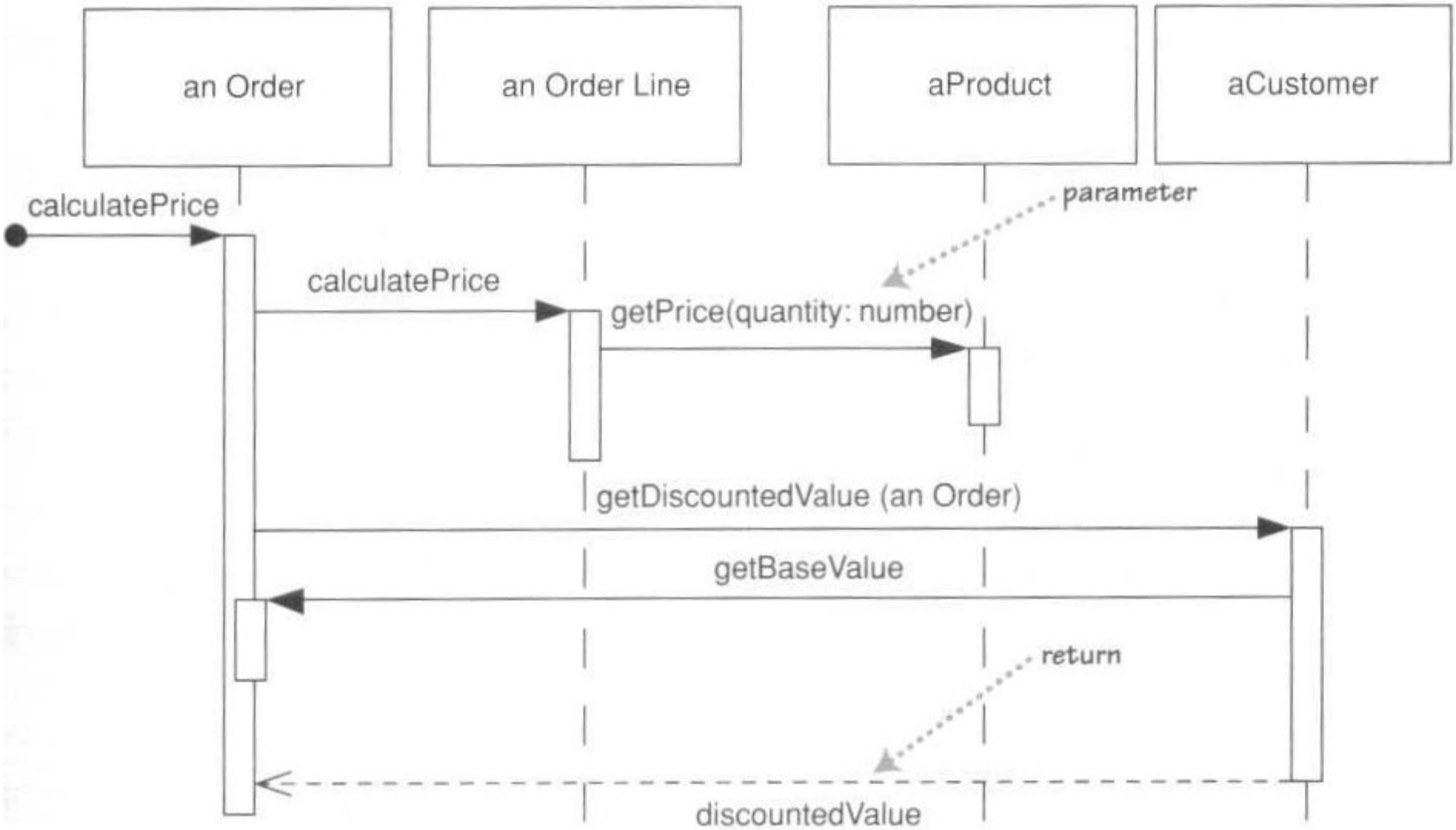
Centralised Control (Class Diagram)



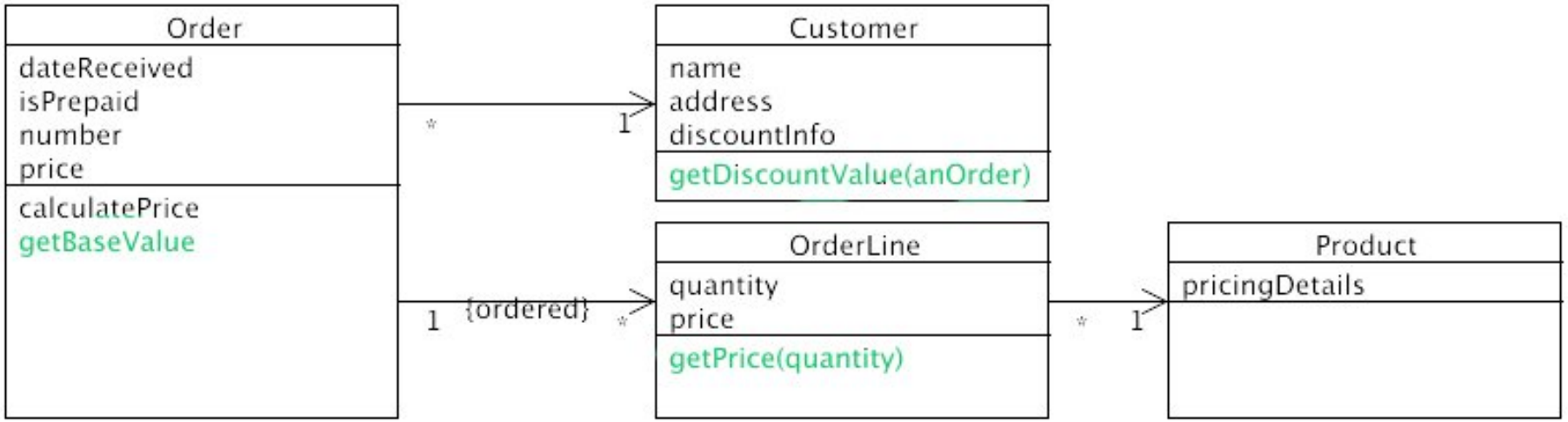
Centralised Control (Sequence Diagram)



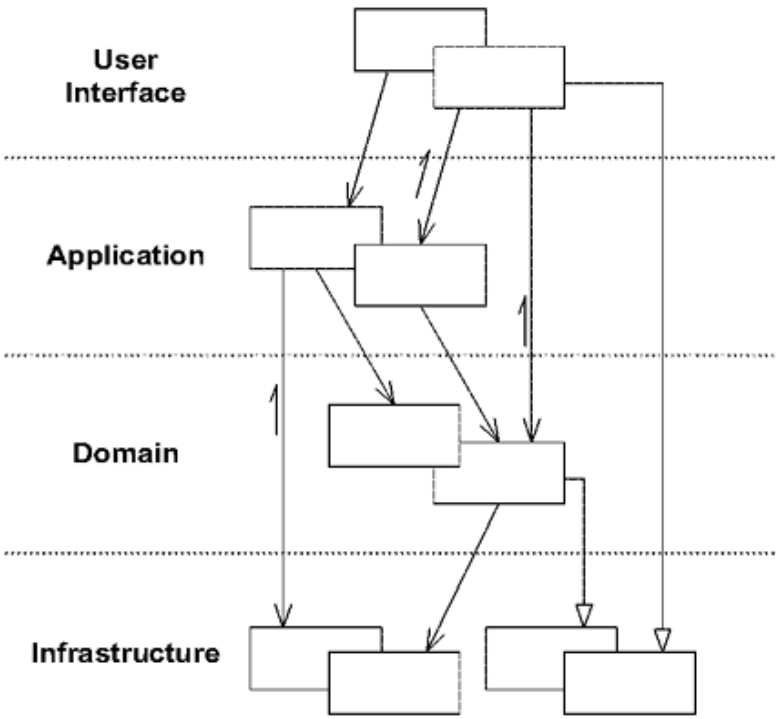
Decentralised Control (Sequence Diagram)



Decentralised Control (Class Diagram)



Layered Architecture



- High cohesion within a layer
  - A layer groups similar functionality, e.g. the User interface layer
- Loose coupling between layers
  - Message flow is directed from higher layers to lower layers but not vice versa
  - Most messages are sent to the adjacent layer

Project

- Exam project: Monday 14.4 — Monday 12.5
- 10 min demonstrations of the software are planned for Tuesday 13.5
- To be delivered
  - a CD with 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

Contents

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)
- Analysis patterns (Martin Fowler)
- Pattern conferences, e.g. PloP (Pattern Languages of Programming) since 1994
- Portland Pattern repository `http://c2.com/cgi/wiki?PeopleProjectsAndPatterns` (since 1995)

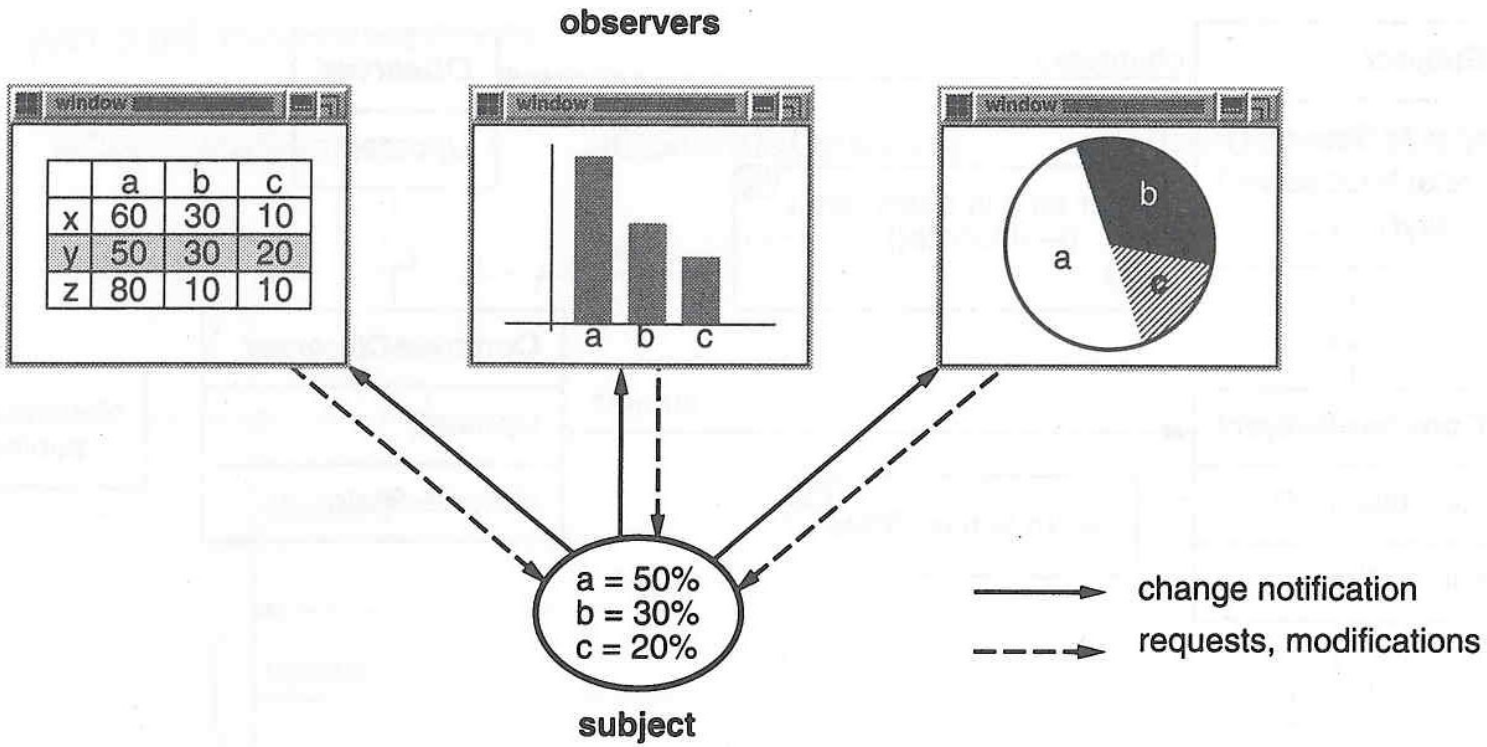
What is a design pattern?

- 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, . . .
- 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`

Observer Pattern

Observer Pattern

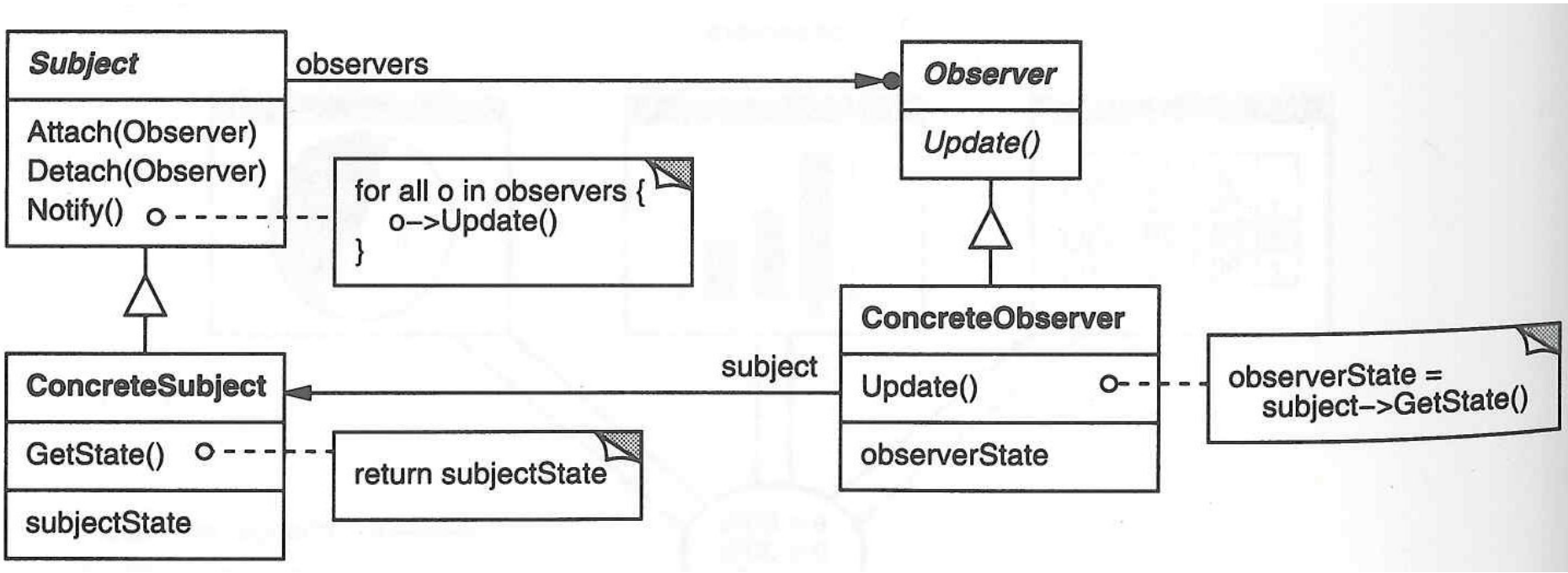
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



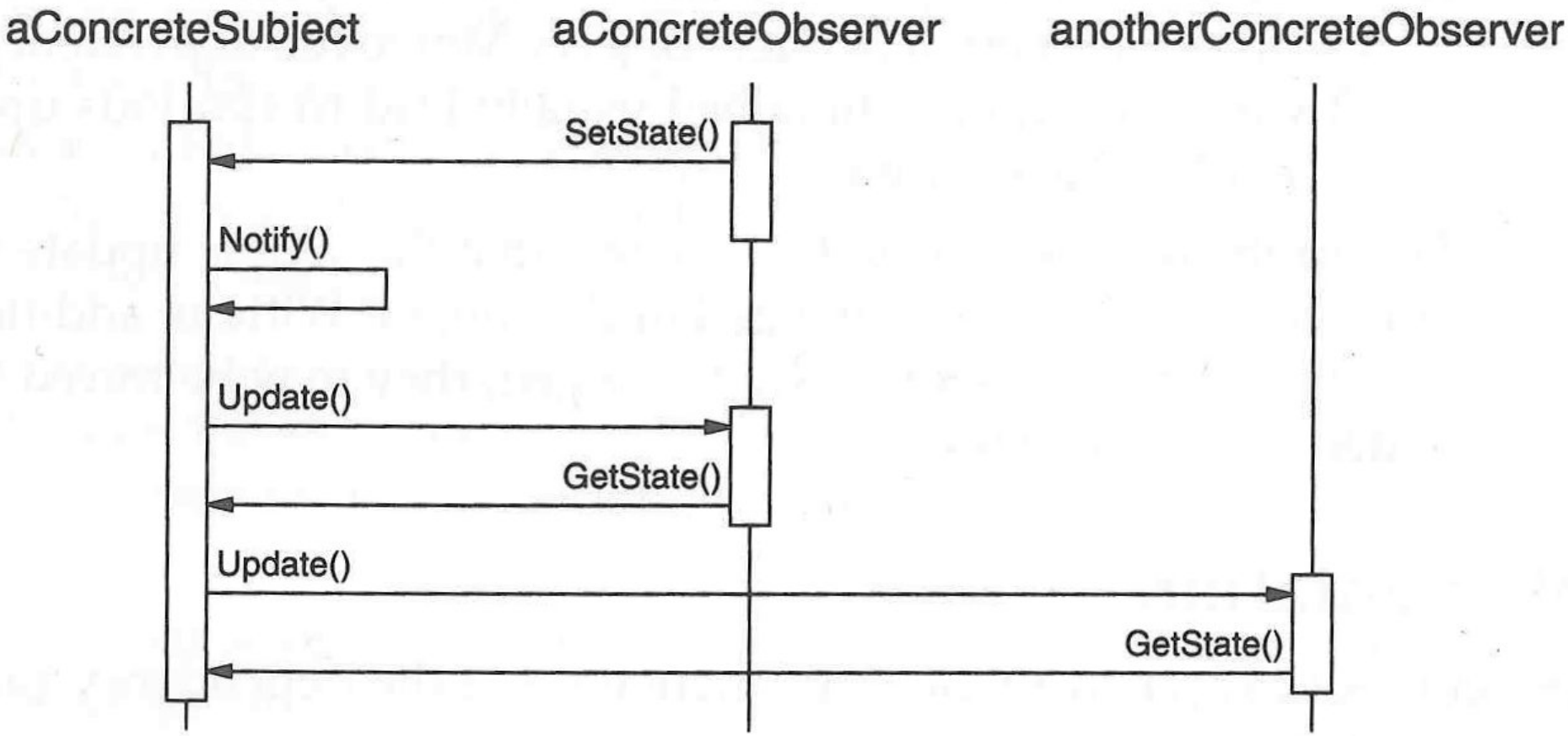
Observer Pattern

- 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 sta
   }
}
```

Example: Stack observer

Adding an observer

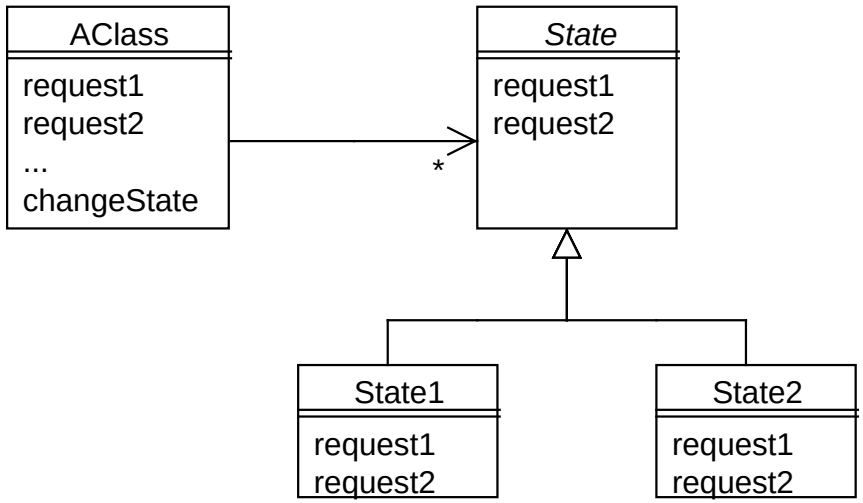
```
....
Stack<Integer> stack = new Stack<Integer>;
NumberOfElementsObserver observer =
            new NumberOfElementsObserver(stack);
stack.addObserver(observer);
stack.push(10);
stack.pop();
...
stack.deleteObserver(observer)
...
```

State Pattern

State Pattern

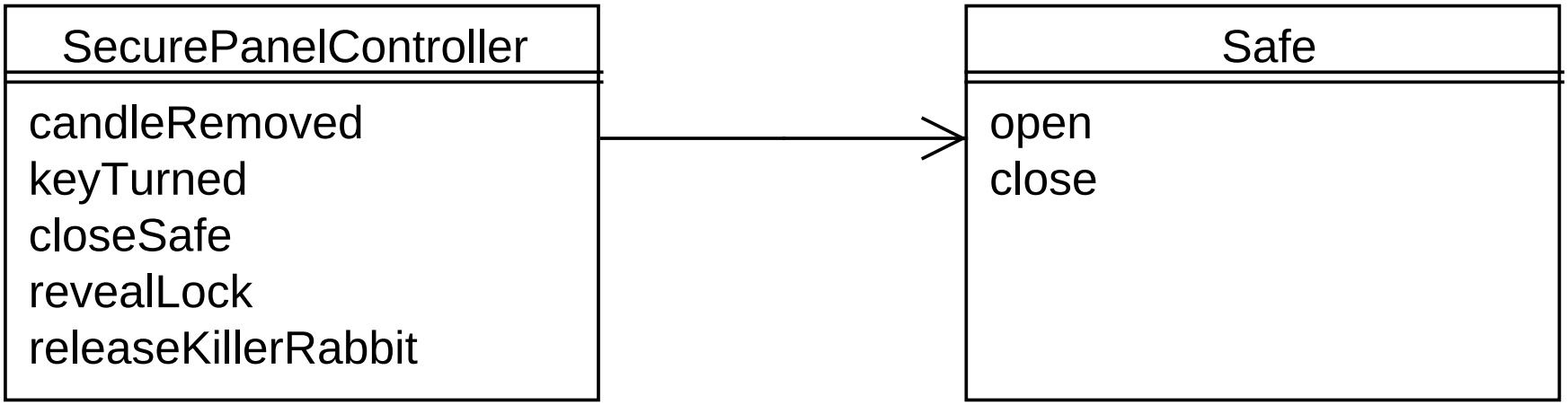
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

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

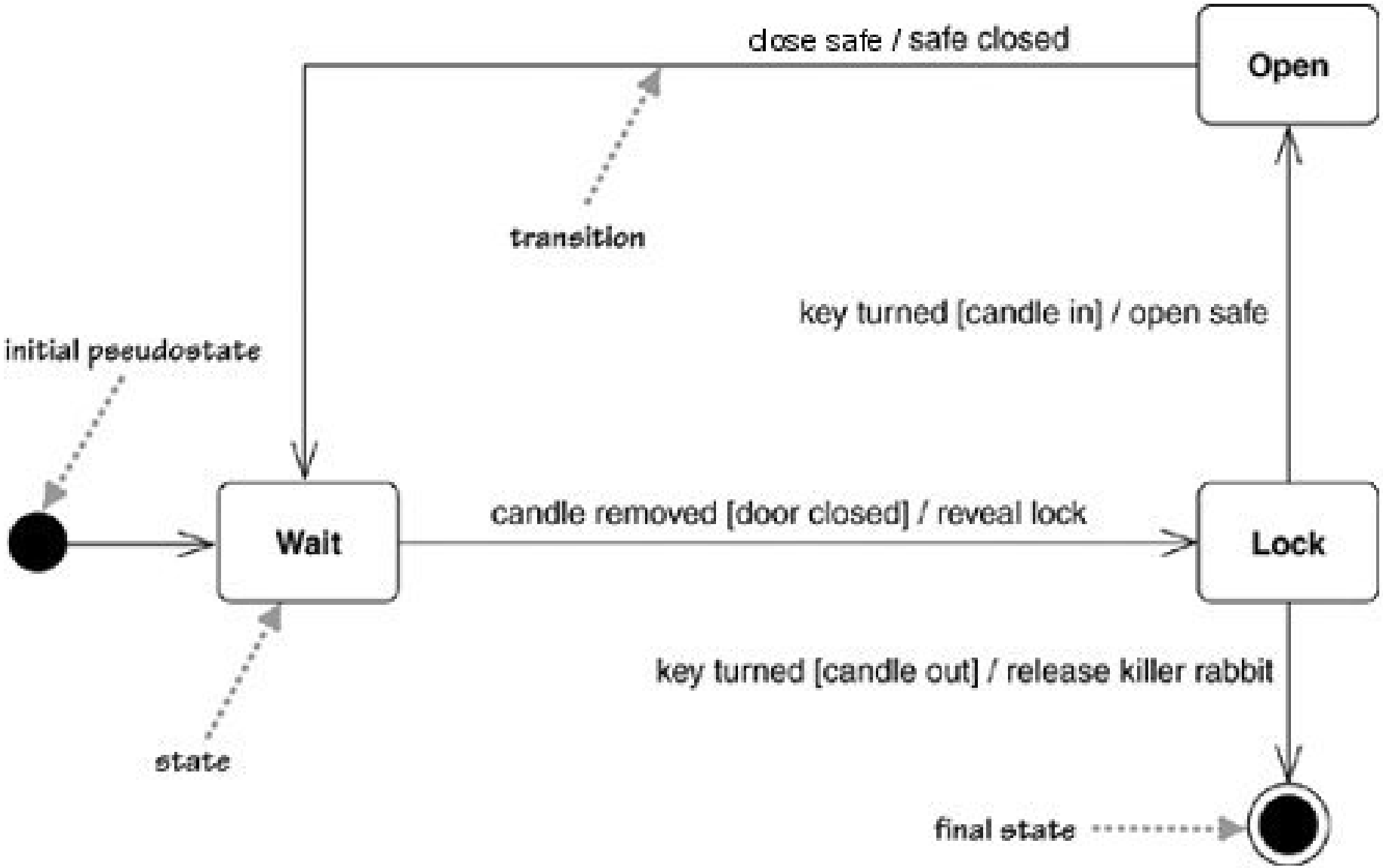


Example

- Task: Implement a control panel for a safe in a dungeon
- The should be visible only when a candle has been removed
- The safe door opens only when the key is turned after the candle has been replaced again
- If the key is turned without replacing the candle, a killer rabbit is released



Example (cont.)



Transitions (UML 2.0)

- General form

  trigger [*guard*]/*effect*

- Triggers (includes events)
  - Call Event
    - messages being sent (e.g. class / interface operation)
    - Can have parameters that can be used in the guard or in the effect
  - ...

  → The event that needs to have happened to fire the transition
- Guard
  - boolean expression

  → Needs to evaluate to true for t he transition to fire
- Effect
  - Sending a message to another object or self
  - Changing the state of an object (e.g. variable assignment)

  → The effect that happens when the transition fires

Alternative Implementation

- The current state is stored in a variable
- Events are method calls

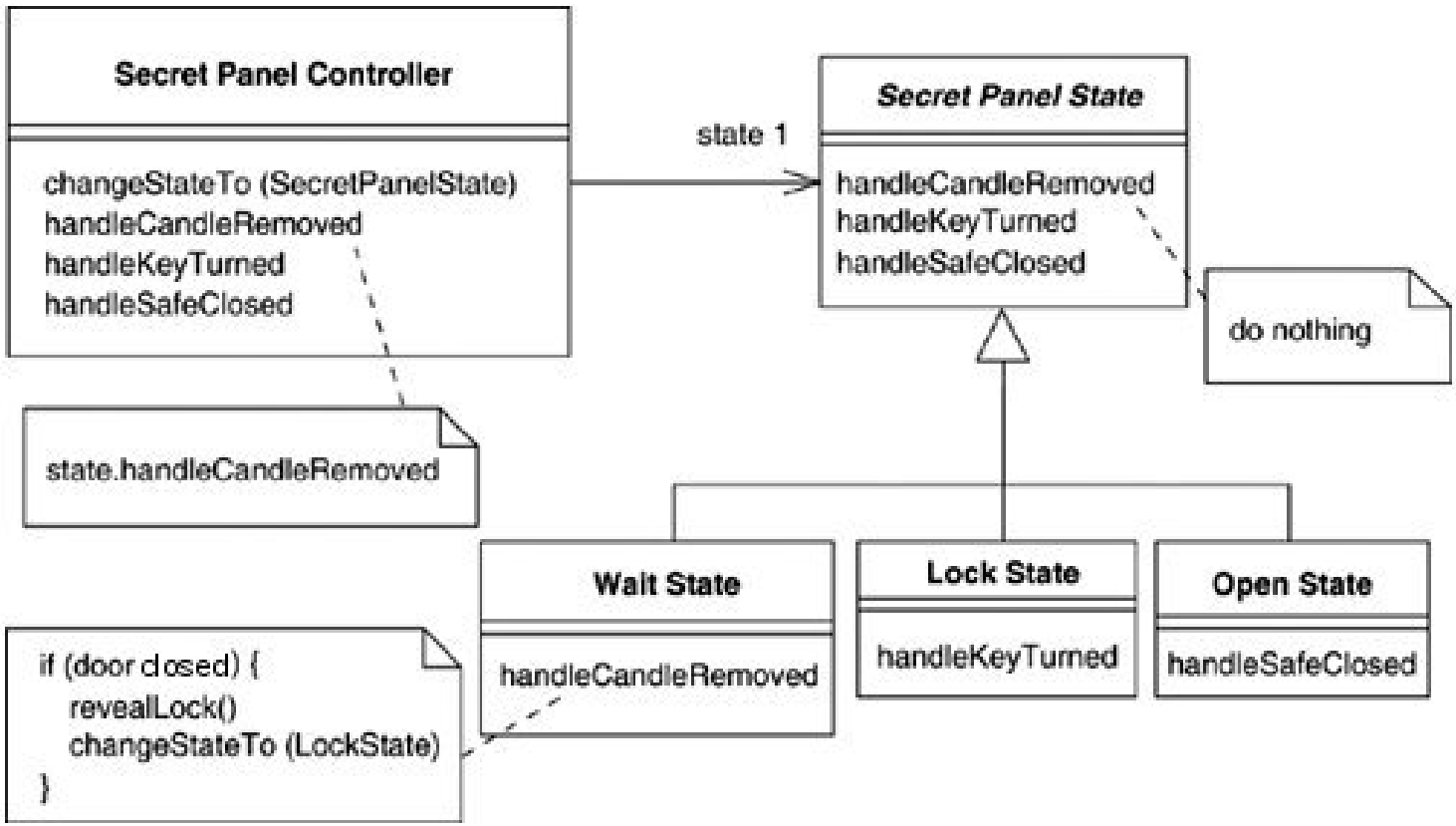
```
public class SecretPanelController {
   enum states { wait, lock, open, finalState };
   states state = states.wait;

   public void candleRemoved() {
      switch (state) {
      case wait:
         if (doorClosed()) {
            state = states.lock;
            break;
         }
      }
   }

   public void keyTurned() {
      switch (state) {
      case lock:
         if (candleOut()) {
            state = states.open;
         } else
         {
            state = states.finalState;
            releaseRabbit();
         }
         break;
      }
   } ... }
```

Implementation using the state pattern

- The current state is an object of a subclass of SecretPanelState
- Events are methods whose implementation is delegated to the state object

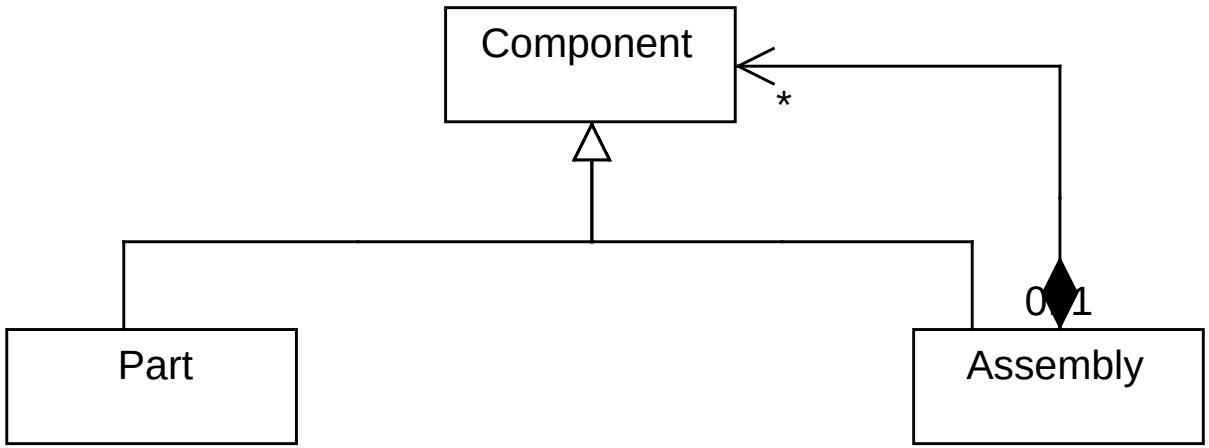


Composite Pattern

Composite Pattern

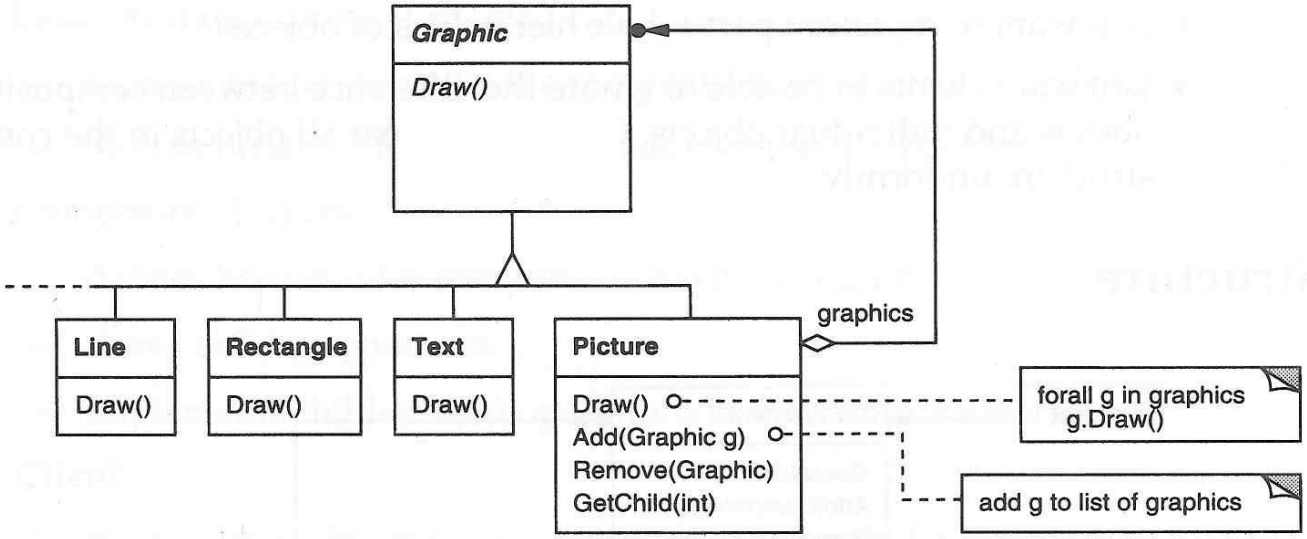
Compose objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.

- Stykkelister example from the first lecture

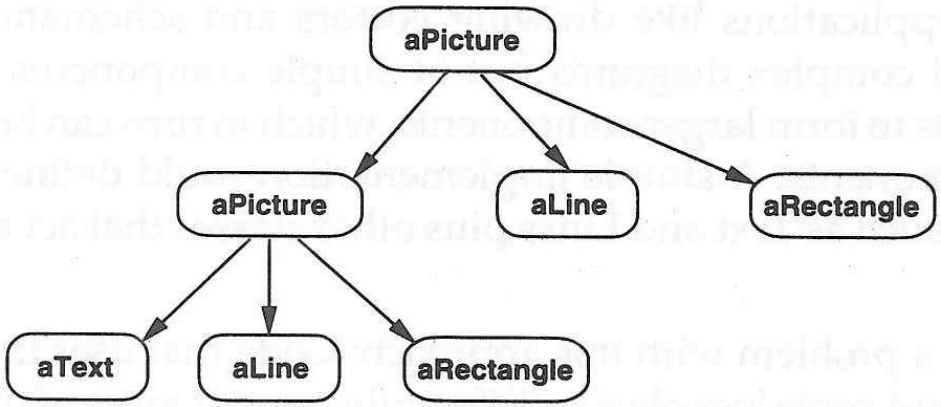


Example: Graphics

- Class Diagram



- Instance diagram



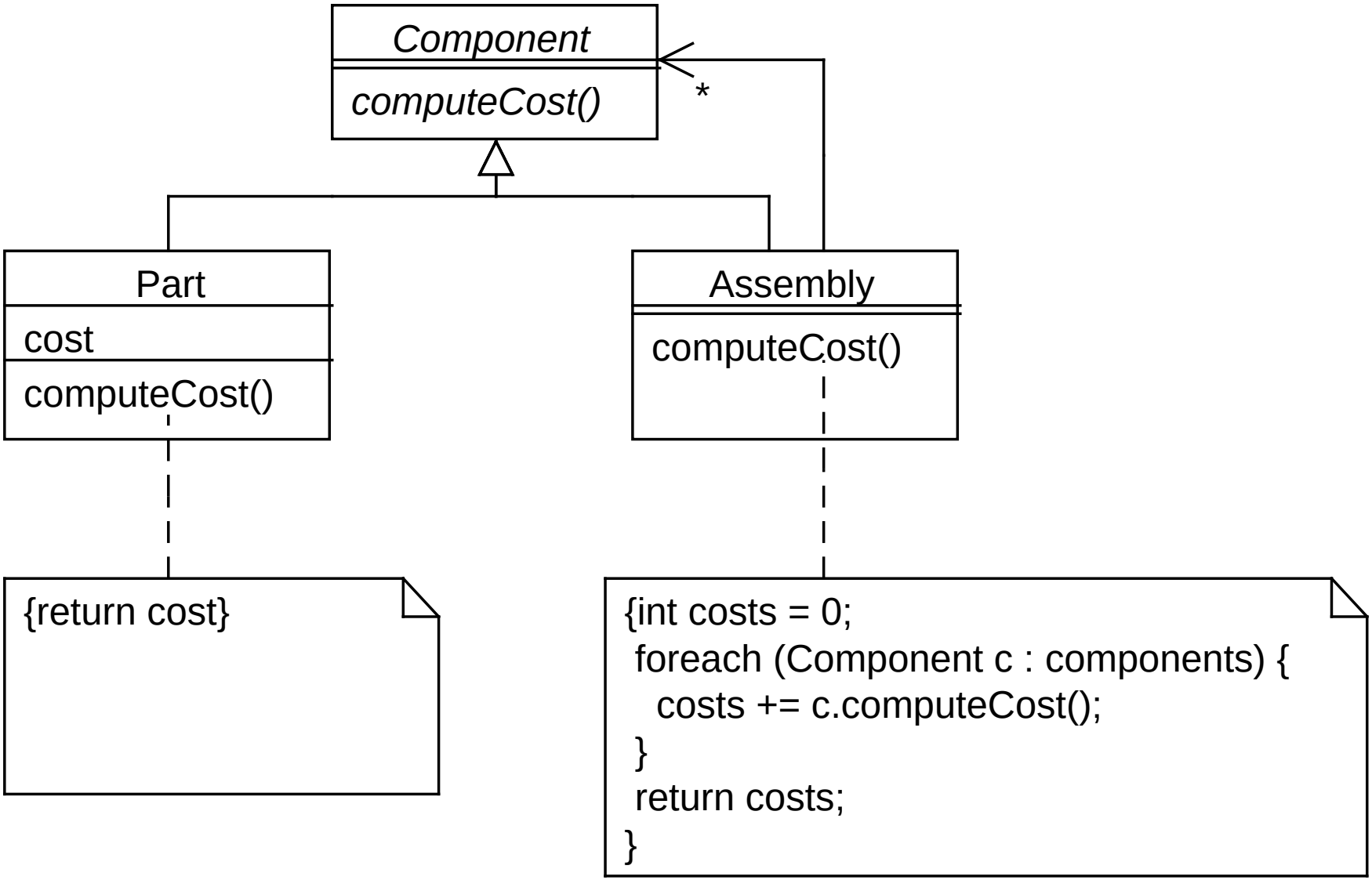
Visitor Pattern

Visitor Pattern

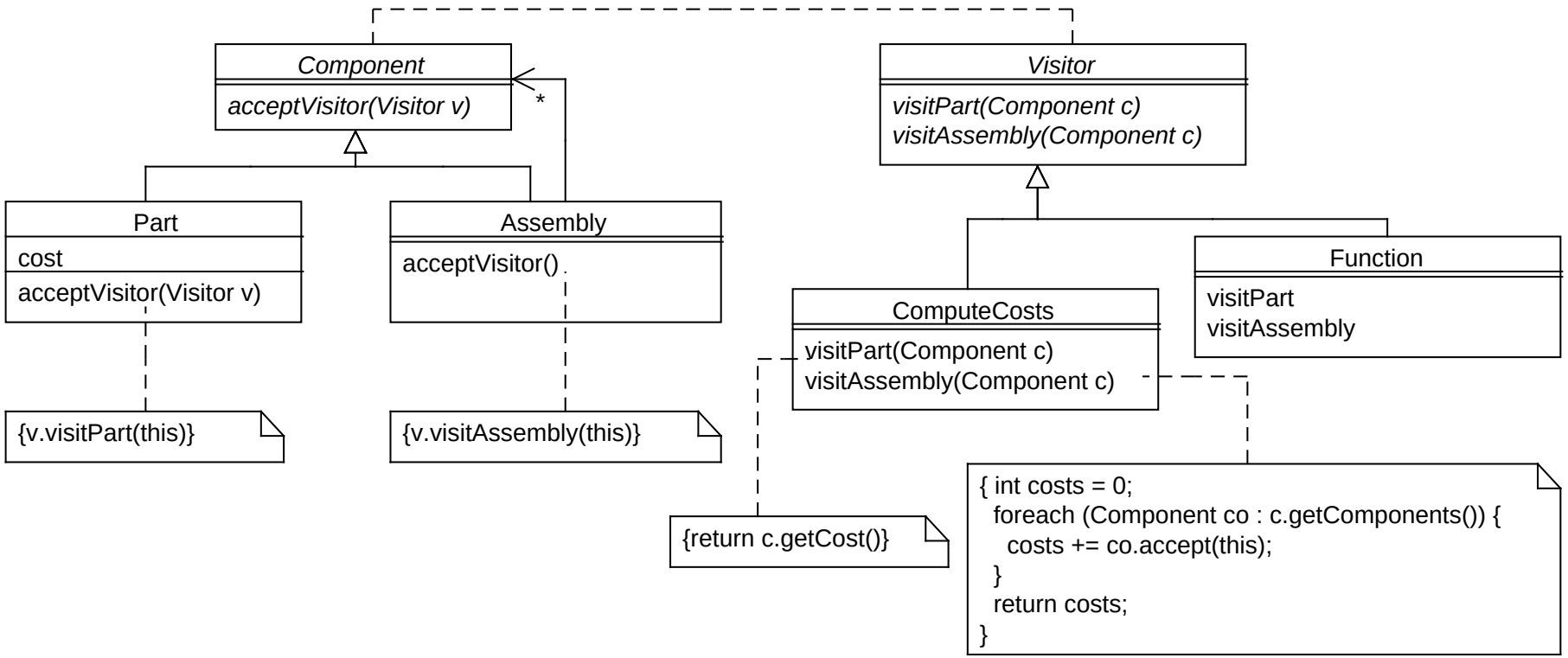
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

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

Example: compute costs for stykkelister



Example: computer 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 are 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 changing 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

Anti-Pattern

Anti Pattern

”In computer science, anti-patterns are specific repeated practices that appear initially to be beneficial, but ultimately result in bad consequences that outweigh the hoped-for advantages.” from Wikipedia (`http://en.wikipedia.org/wiki/Anti-pattern`)

- ”Patterns of failure”
- AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis by William J. Brown, Raphael C. Malveau, and Thomas J. Mowbray
- Example: Analysis Paralysis
  - Stuck with developing the analysis model.
  - The model never is good enough.
  - Each time one revisits the same problem, a new variant comes up
  - Solution: Proceed to design and implementation. This gives new insights into the analysis → iterative / evolutionary approach
- For a list of anti-patterns see `http://en.wikipedia.org/wiki/Anti-pattern#Recognized.2FKnown_Anti-Patterns`)

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

- There is a difference between good and bad design
- Good design is impportant
- Some basic principles:
  - DRY, KISS, YAGNI
  - High cohesion — low coupling
  - Layered architecture
- Patterns capture knowledge
  - Design patterns
  - Analysis patterns
  - Implementation patterns

→ Good design is a life long learning process

- e.g. when or when not to apply certain patterns
- think about what and why you are doing it!