Software Engineering I (02161)

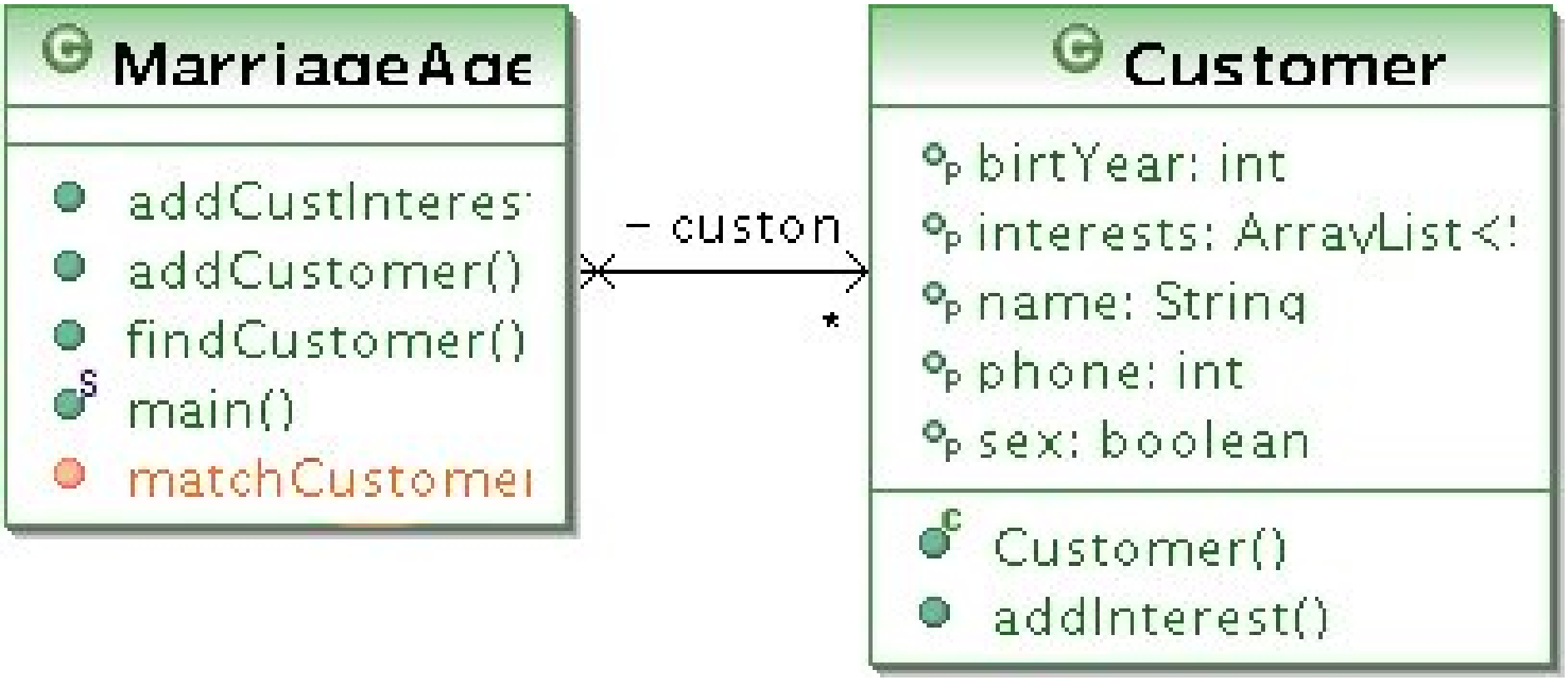
Week 9: Principles of good design, Patterns, Layered Architecture

Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2010

What is good design? MarriageAgency Example



What is good design? MarriageAgency Example 2

Method matchCustomer in class MarriageAgency

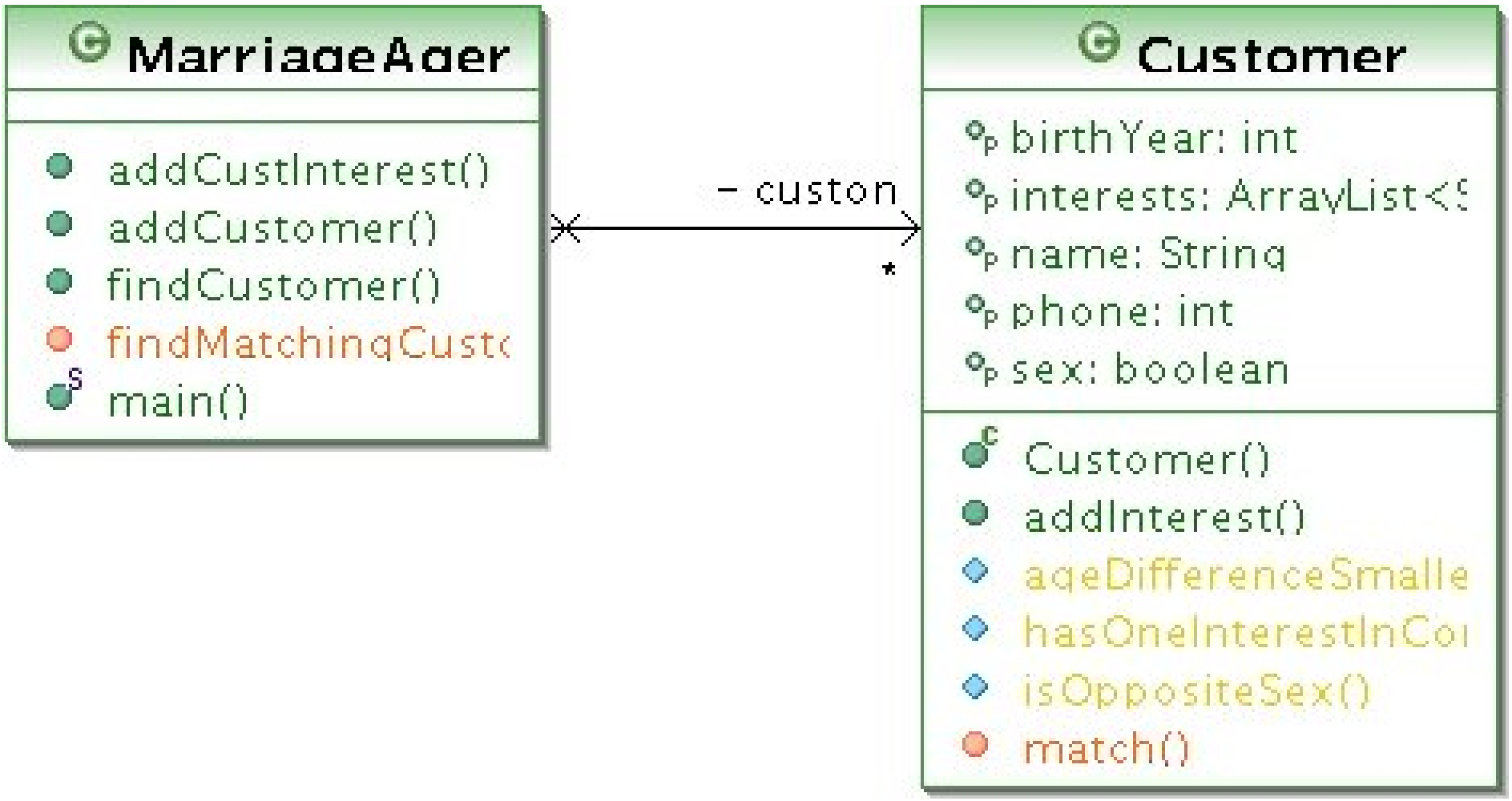
```
public ArrayList<Customer> matchCustomer(Customer customer){
  ArrayList<Customer> res = new ArrayList<Customer>();
  for (Customer potential : customers) {
    if (potential.getSex() != customer.getSex()) {
      int yearDiff = Math.abs(potential.getBirtYear()
                    -customer.getBirtYear());
      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

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 2



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

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
- Modularize your Program
  - → High cohesion — low coupling

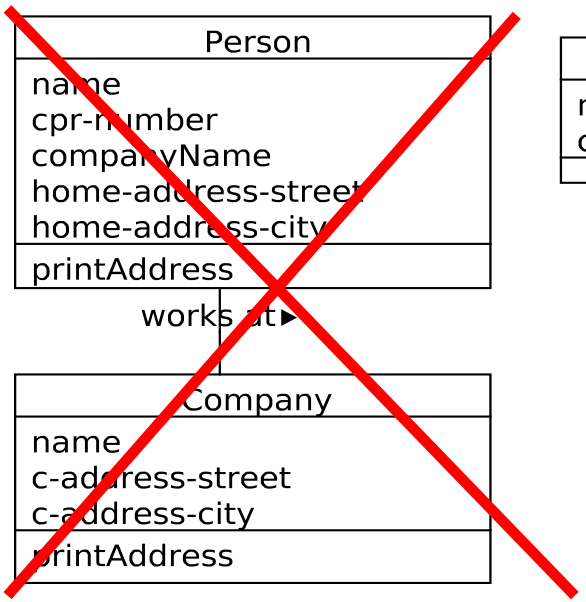
DRY principle

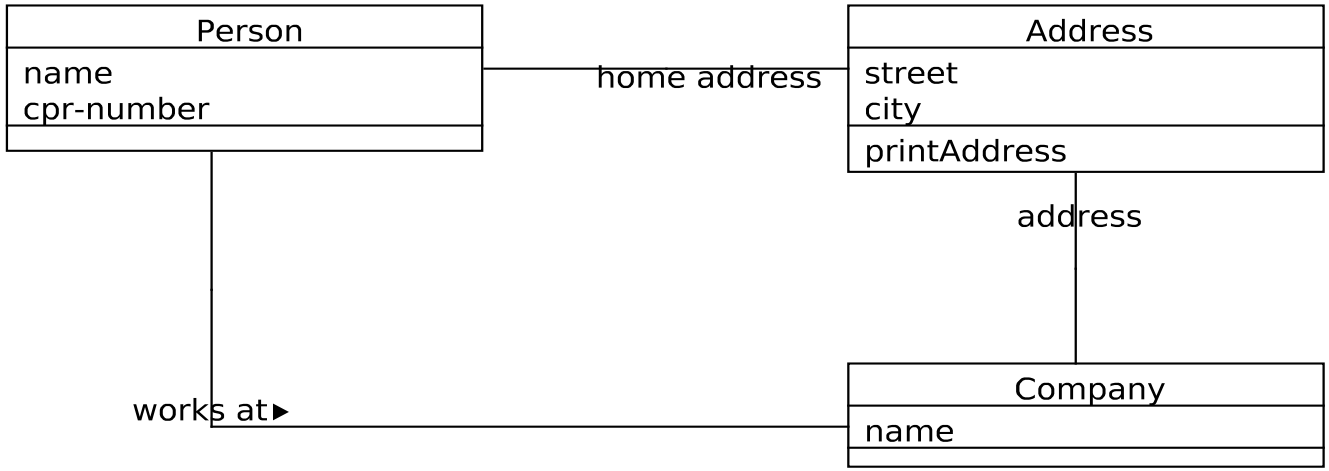
DRY principle

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”.

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)

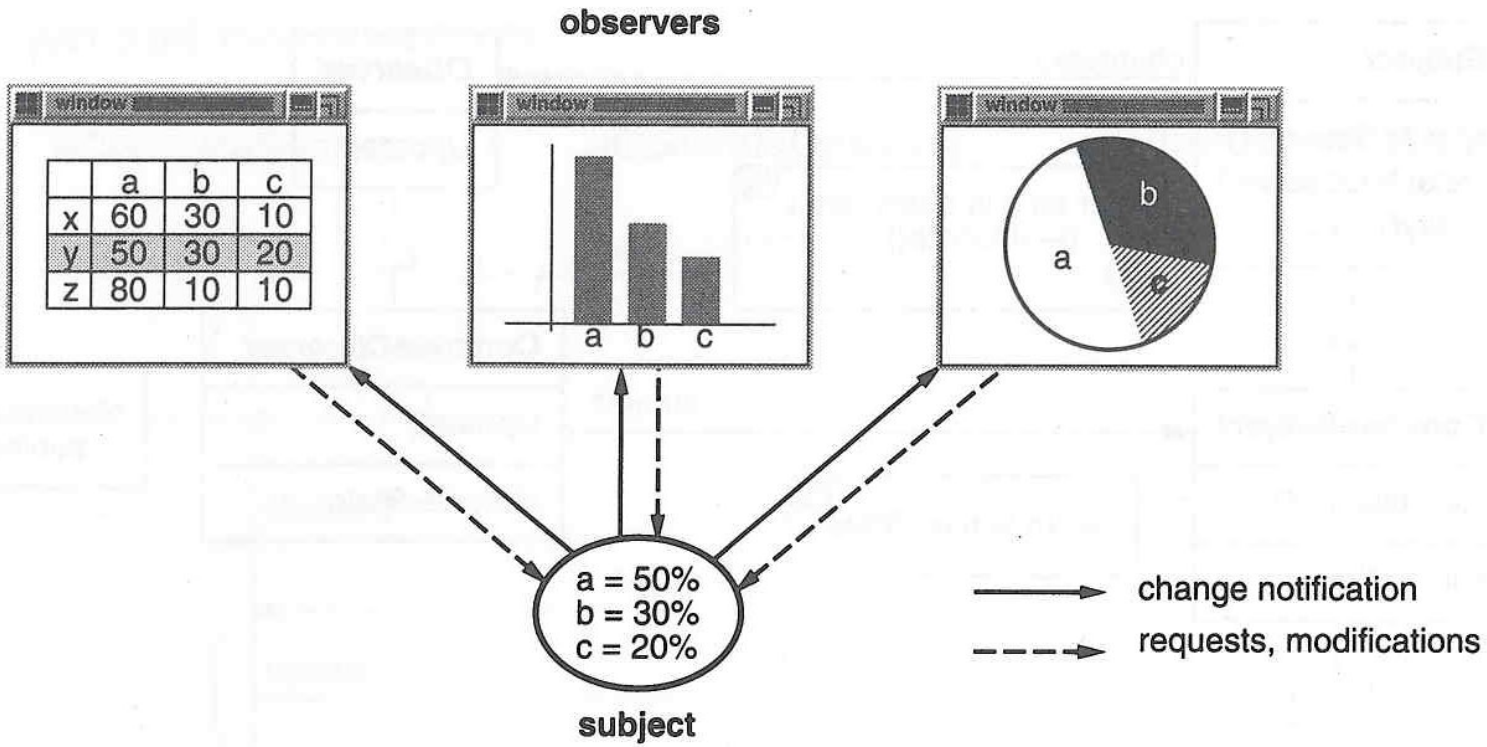
What is a design pattern?

- 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, . . .
- 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

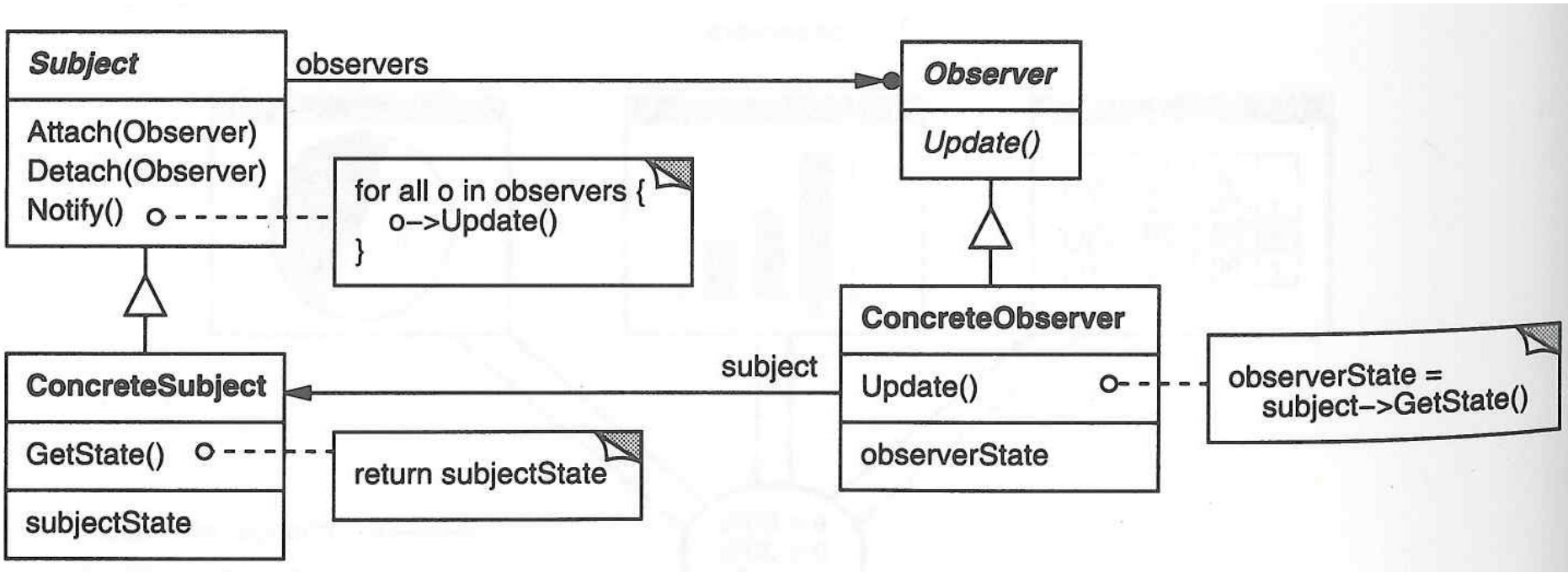
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



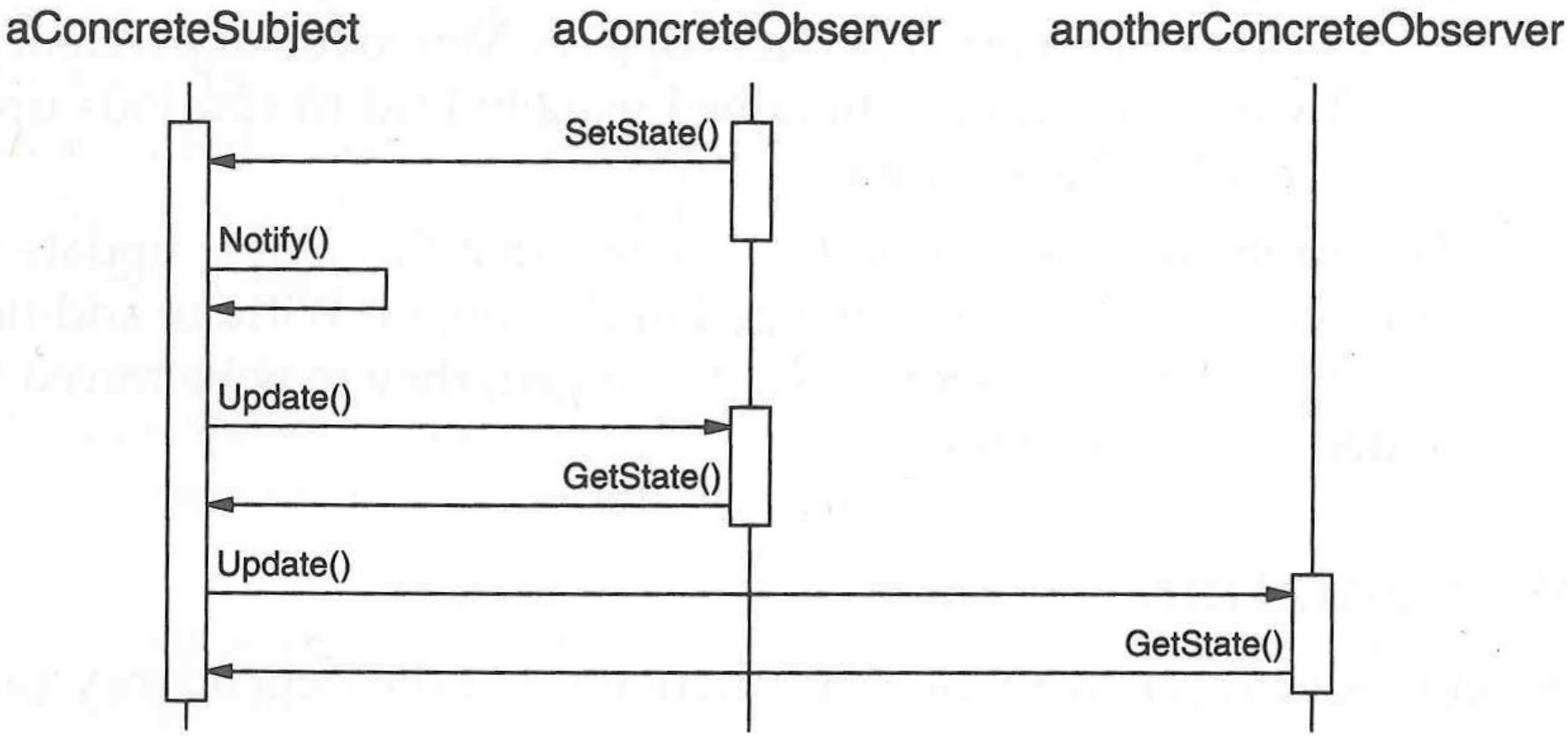
Observer Pattern

- 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 stac
   }
}
```

Example: Stack observer

Adding an observer

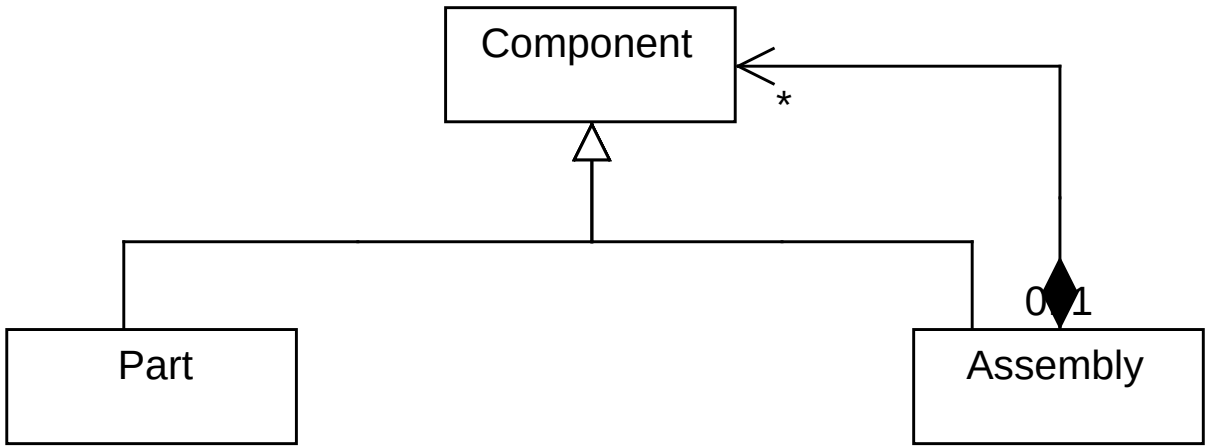
```
....
Stack<Integer> stack = new Stack<Integer>;
NumberOfElementsObserver observer =
            new NumberOfElementsObserver(stack);
stack.addObserver(observer);
stack.push(10);
stack.pop();
...
stack.deleteObserver(observer)
...
```

Composite Pattern

Composite Pattern

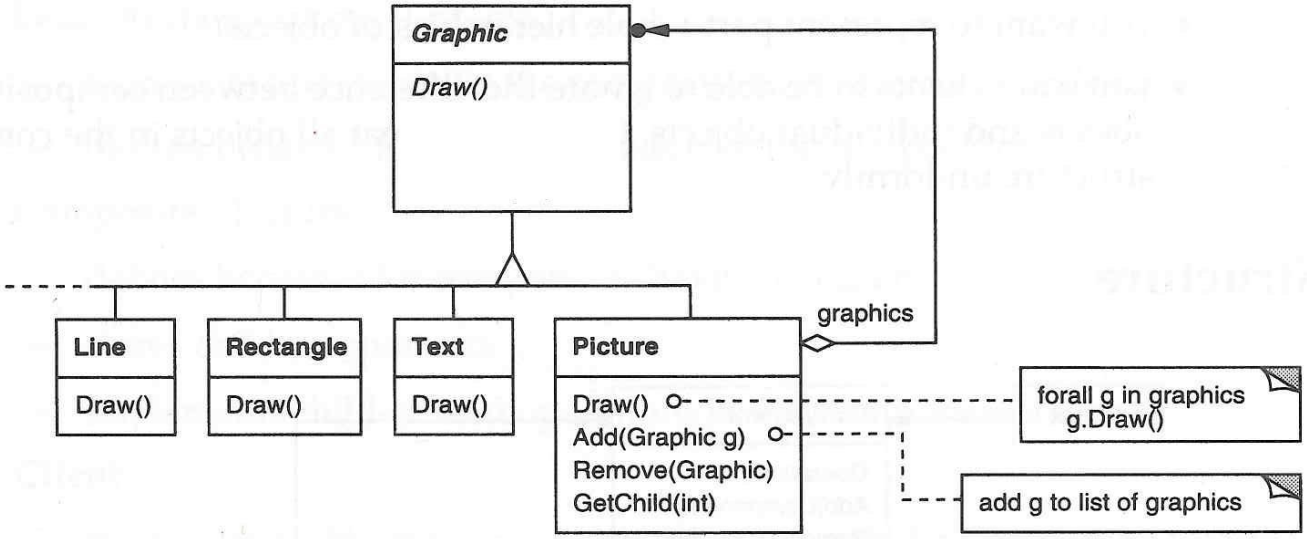
Compose objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.

- Stykkelister example from the first lecture

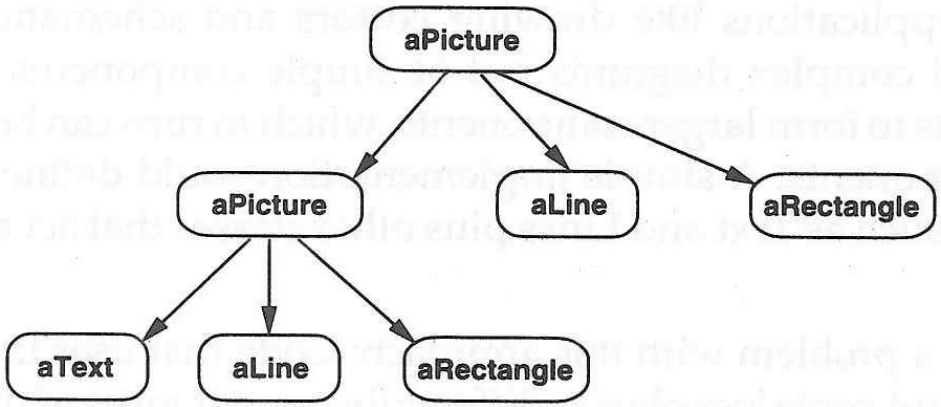


Example: Graphics

- Class Diagram



- Instance diagram

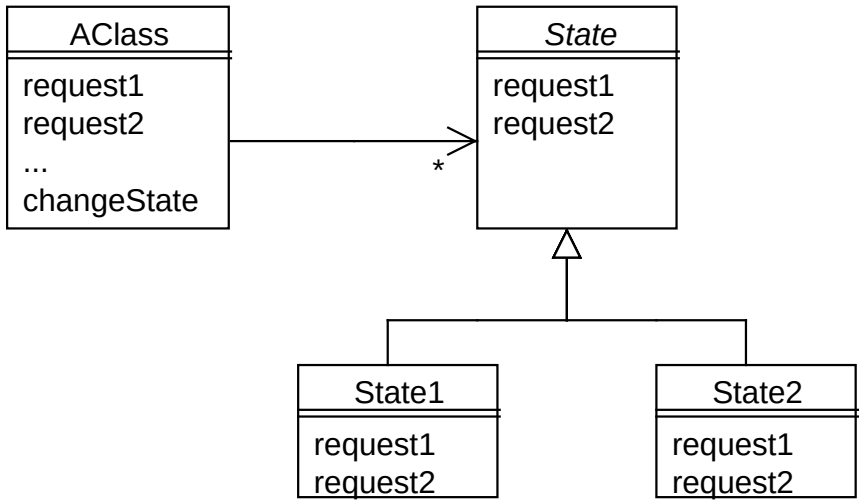


State Pattern

State Pattern

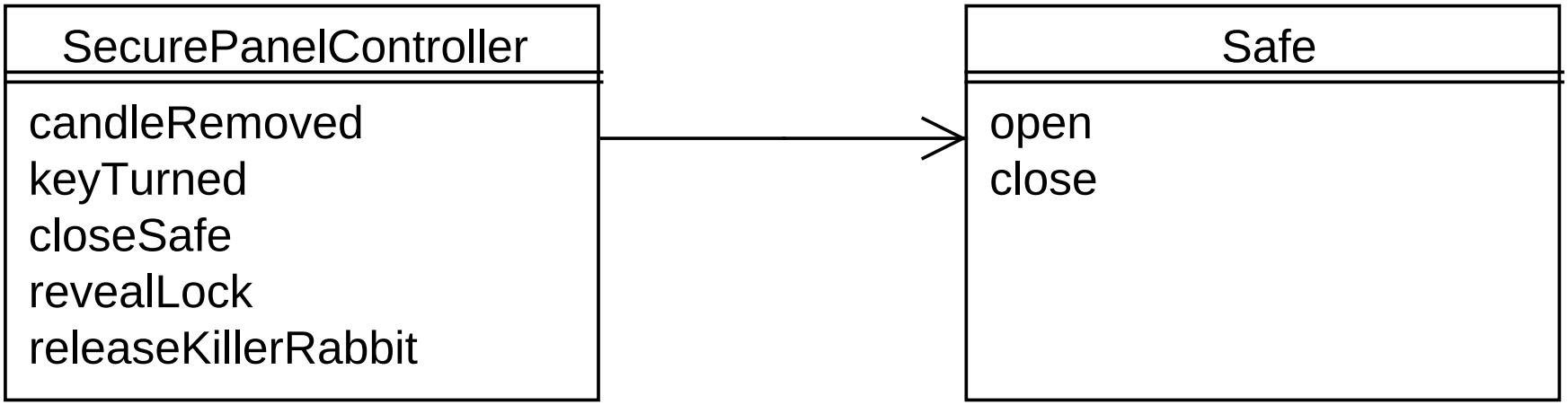
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

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

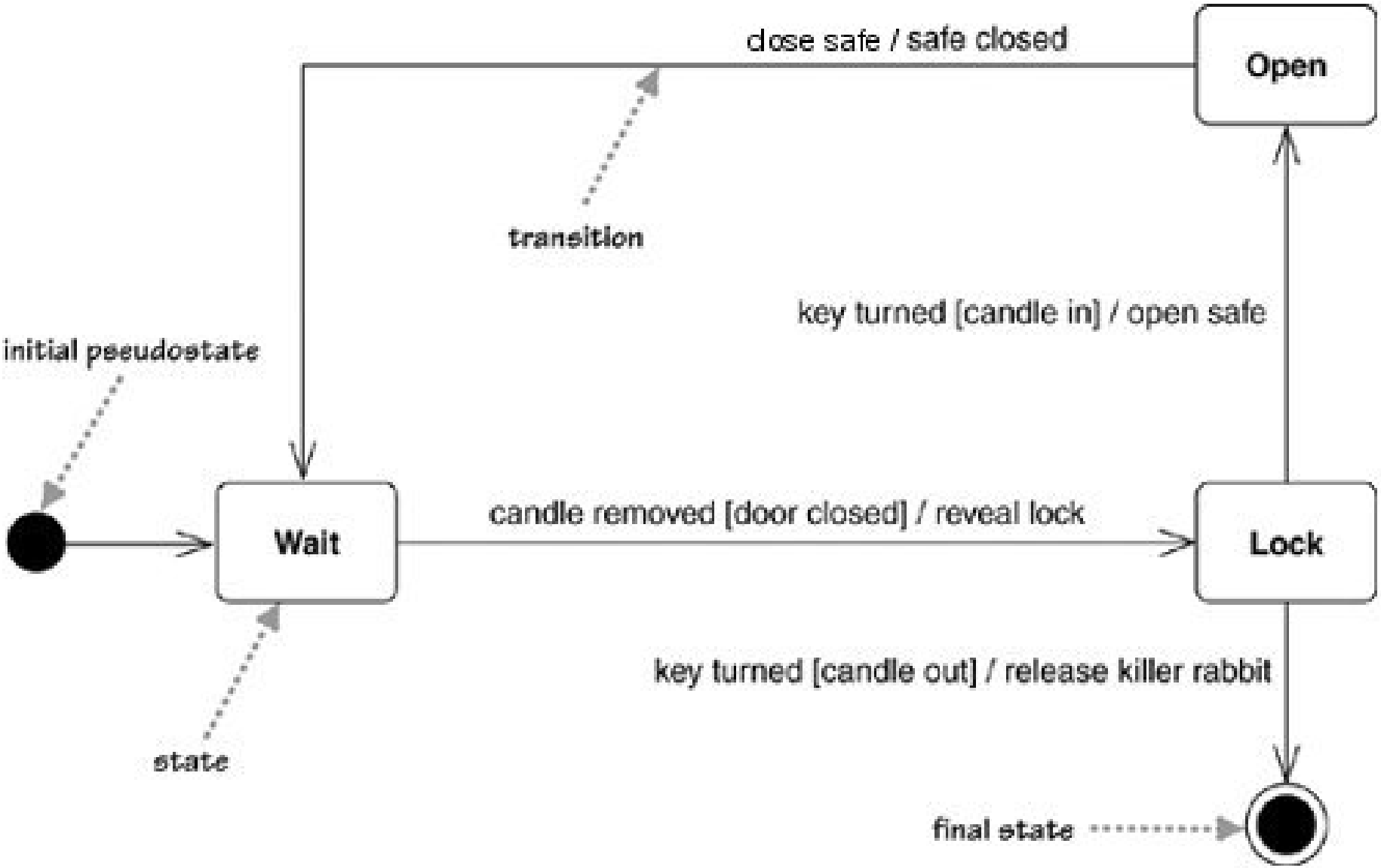


Example

- 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 the 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

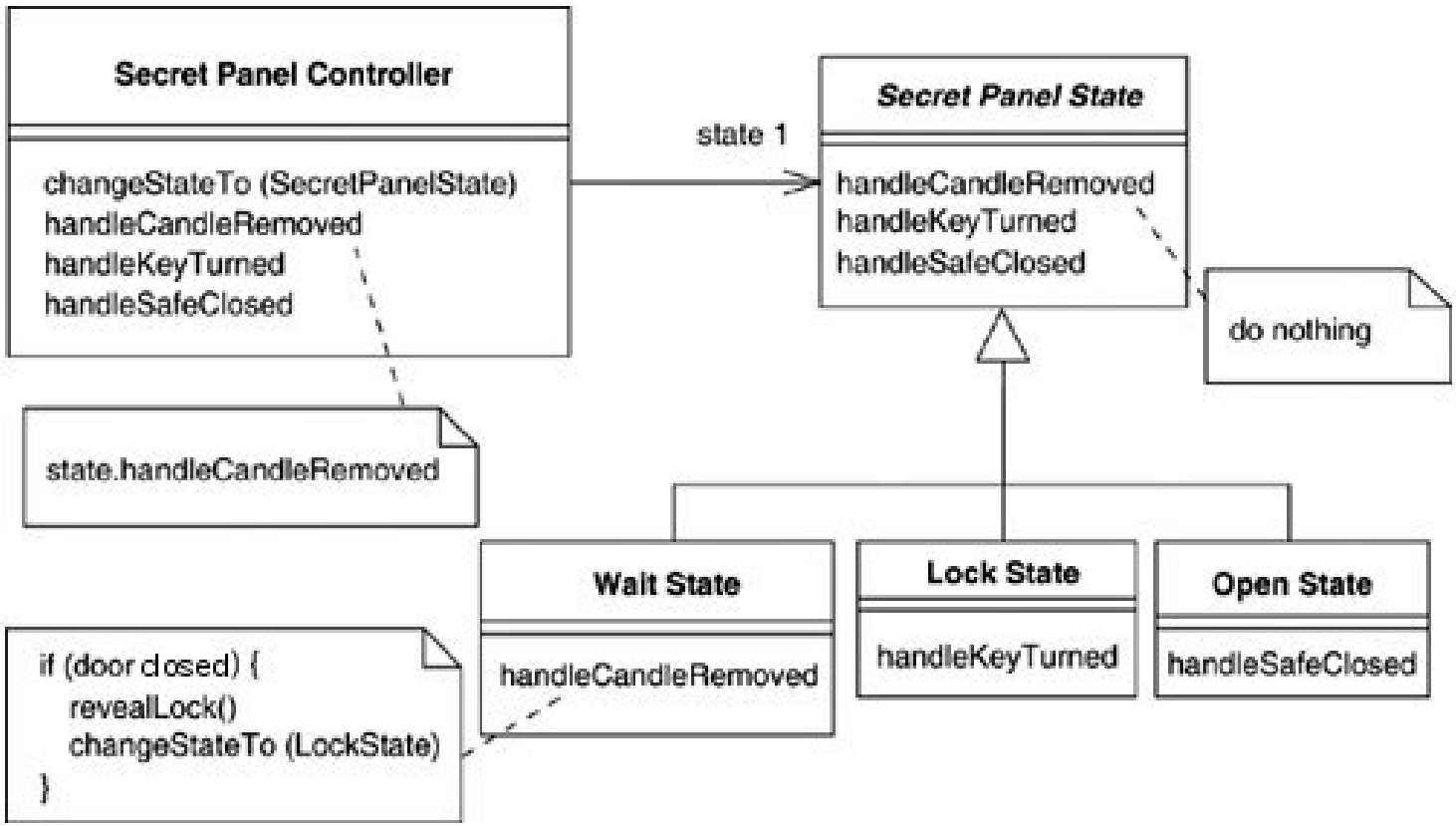
Alternative Implementation

- 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



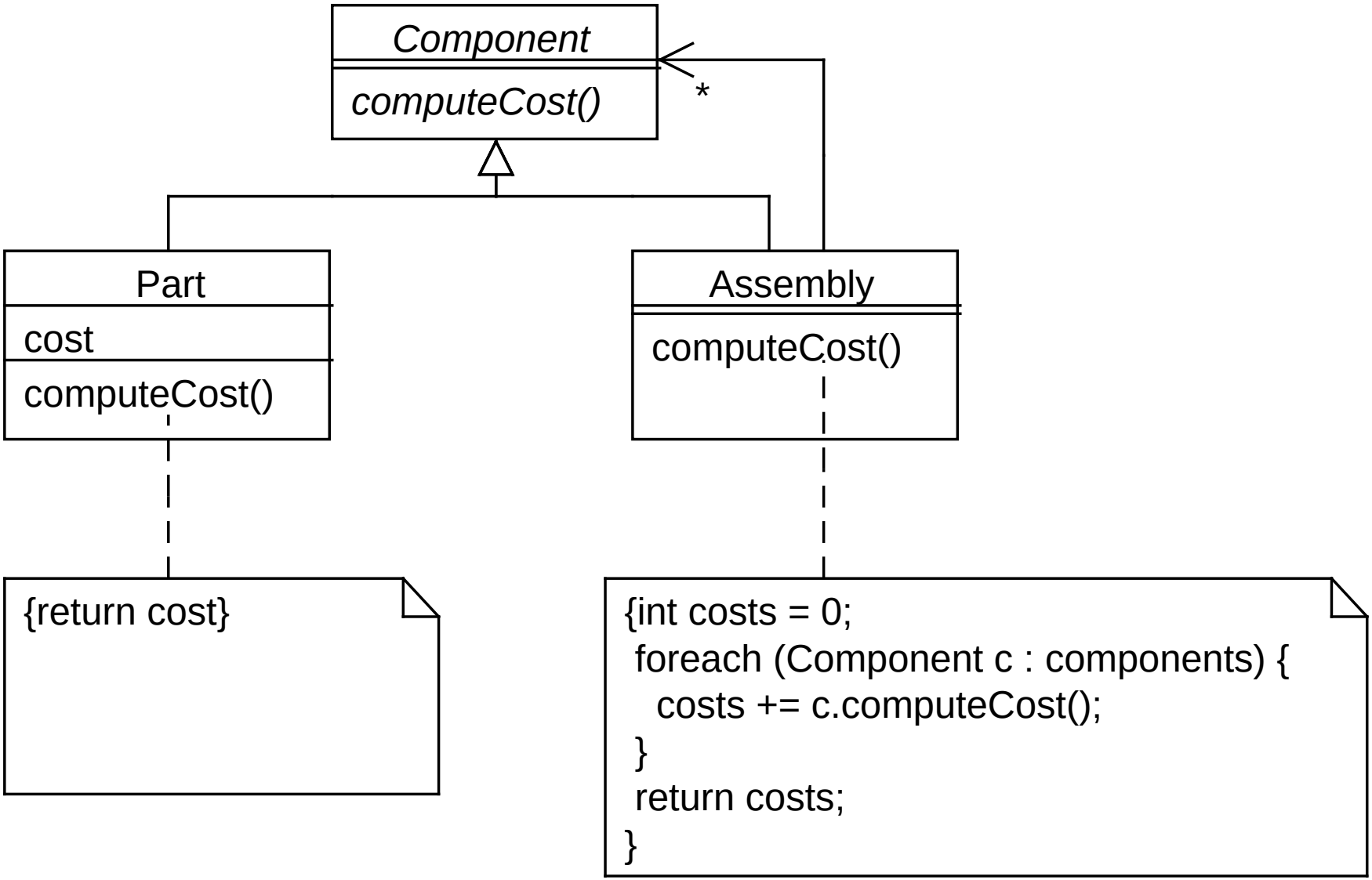
Visitor Pattern

Visitor Pattern

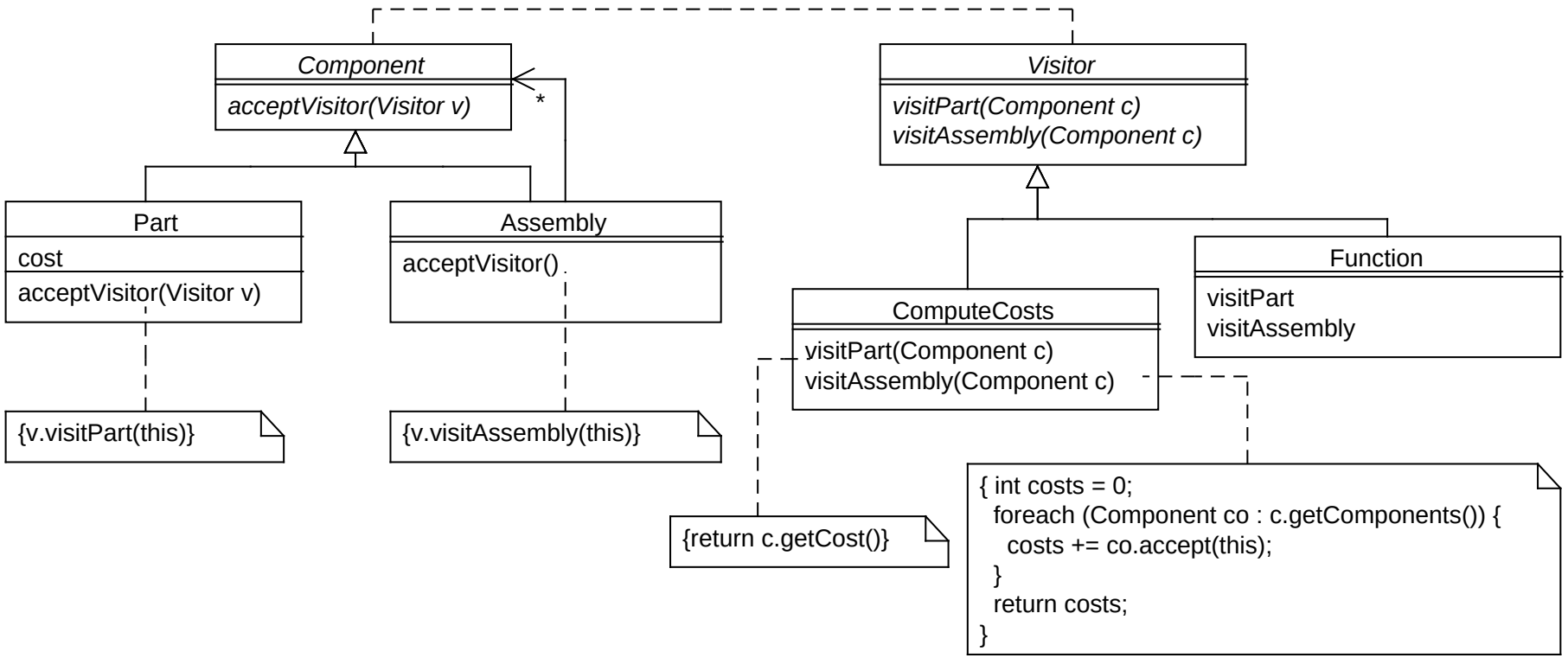
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

- 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: 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

Summary Design Patterns

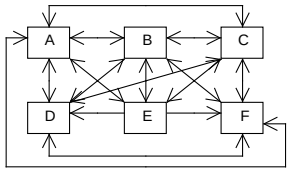
- Original Gang of Four book:
- Creational Patterns
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton
- Structural Patterns
  - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
- Behavioral Patterns
  - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor
- There are more: Implementation Patterns, Architectural Patterns, Analysis Patterns, Domain Patterns ...

Low Coupling

Low coupling

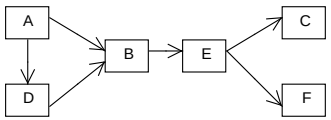
- An object / class is connected only to a few other classes
- It fullfils its repsonsibility by delegating responsibility to other objects

- High coupling: Every class is connected with every class



→ Difficult to change / exchange classes: dependency to all other classes need to be considered

- Low coupling: Classes are only connected to few other classes



High Cohesion

High Cohesion

- Groups methods / attributes / classes with a common goal / functionality

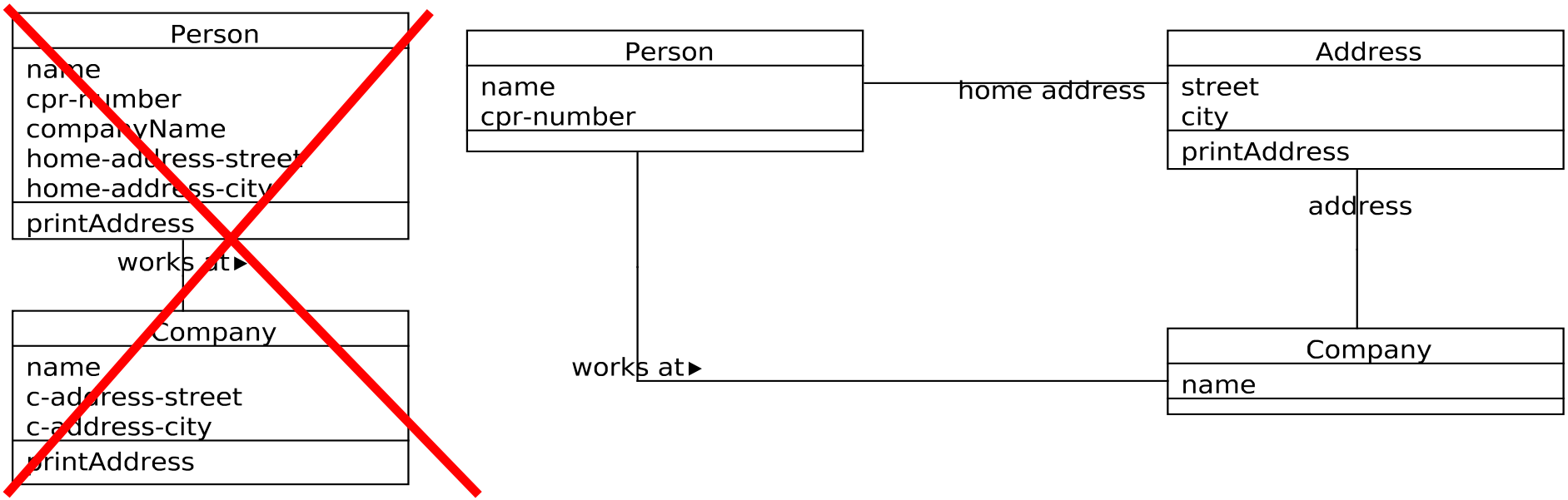
→ E.g. A class groups a set of a related methods and attributes

→ an object is self contained and represents an entity

- 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

High cohesion — low coupling

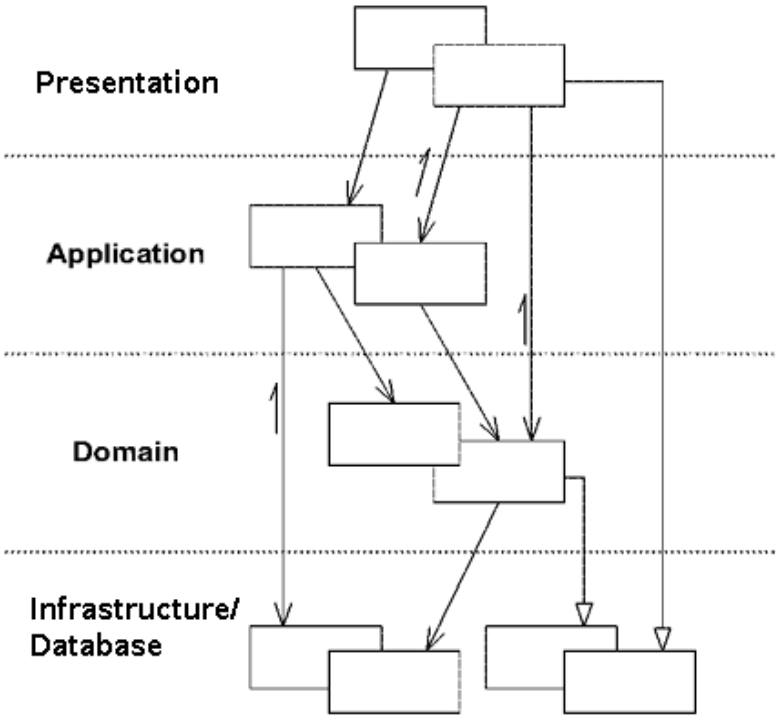
Example: High Cohesion



- Left side violates high cohesion:
  - Attributes for address and method for printing the address do not belong to the Person or the Company concept

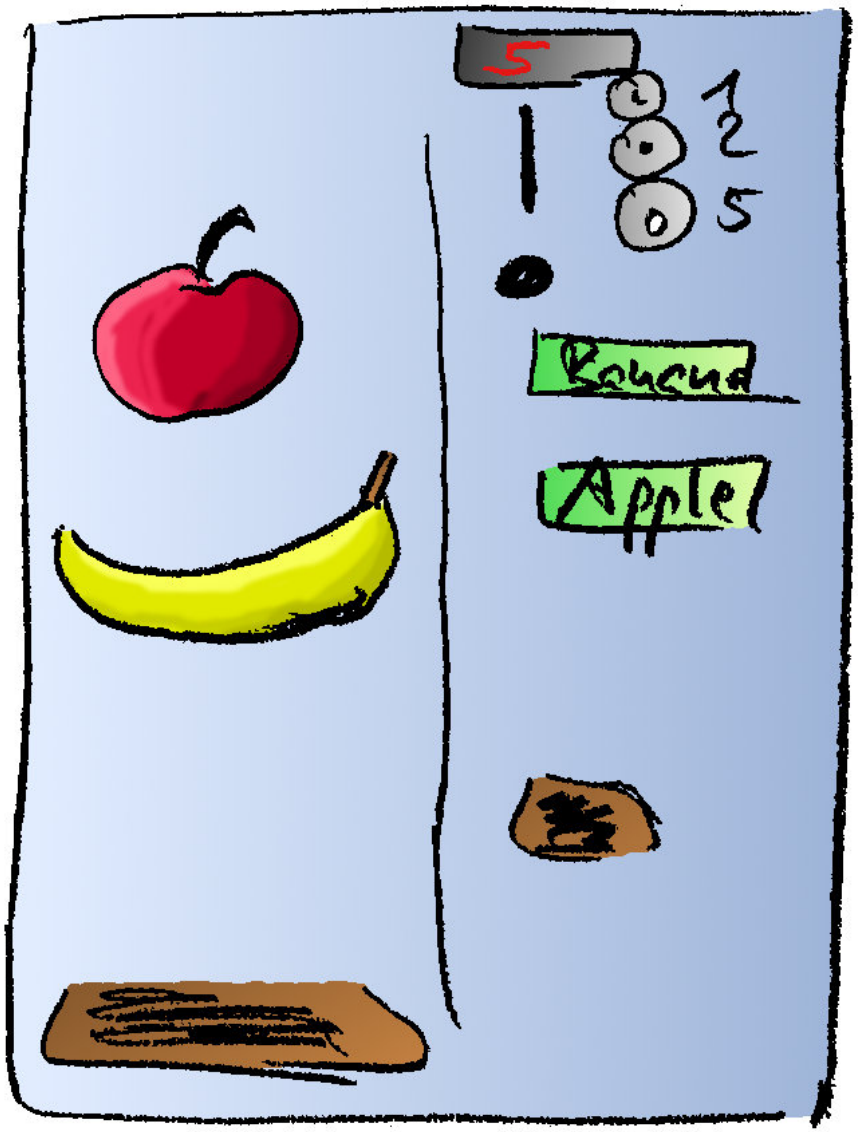
High cohesion — low coupling

Layered Architecture



- Low 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
- High cohesion within a layer
  - A layer groups similar functionality, e.g. the User interface / Presentation layer

Example Vending Machine



- Actions
  - Input coins
  - Press button for bananas or apples
  - Press cancel
- Displays
  - current amount of money input
- Effects
  - Return money
  - Dispense banana or apple

Use Case: Buy Fruit

name: Buy fruit

description: Entering coins and buying a fruit

actor: user

main scenario:

1. Input coins until the price for the fruit to be selected is reached
2. Select a fruit
3. Vending machine dispenses fruit

alternative scnearios:

a1. User inputs more coins than necessary
a2. select a fruit
a3. Vending machine dispenses fruit
a4. Vending machine returns excessive coins

Use Case: Buy Fruit (cont.)

alternative scenarios (cont.)

- b1 User inputs less coins than necessary
- b2 user selects a fruit
- b3 No fruit is dispensed
- b4 User adds the missing coins
- b5 Fruit is dispensed
- c1 User selects fruit
- c2 User adds sufficient or more coins
- c3 vending machine dispneses fruit and rest money
- d1 user enters coins
- d2 user selects cancel
- c3 money gets returned

Use Case: Buy Fruit (cont.)

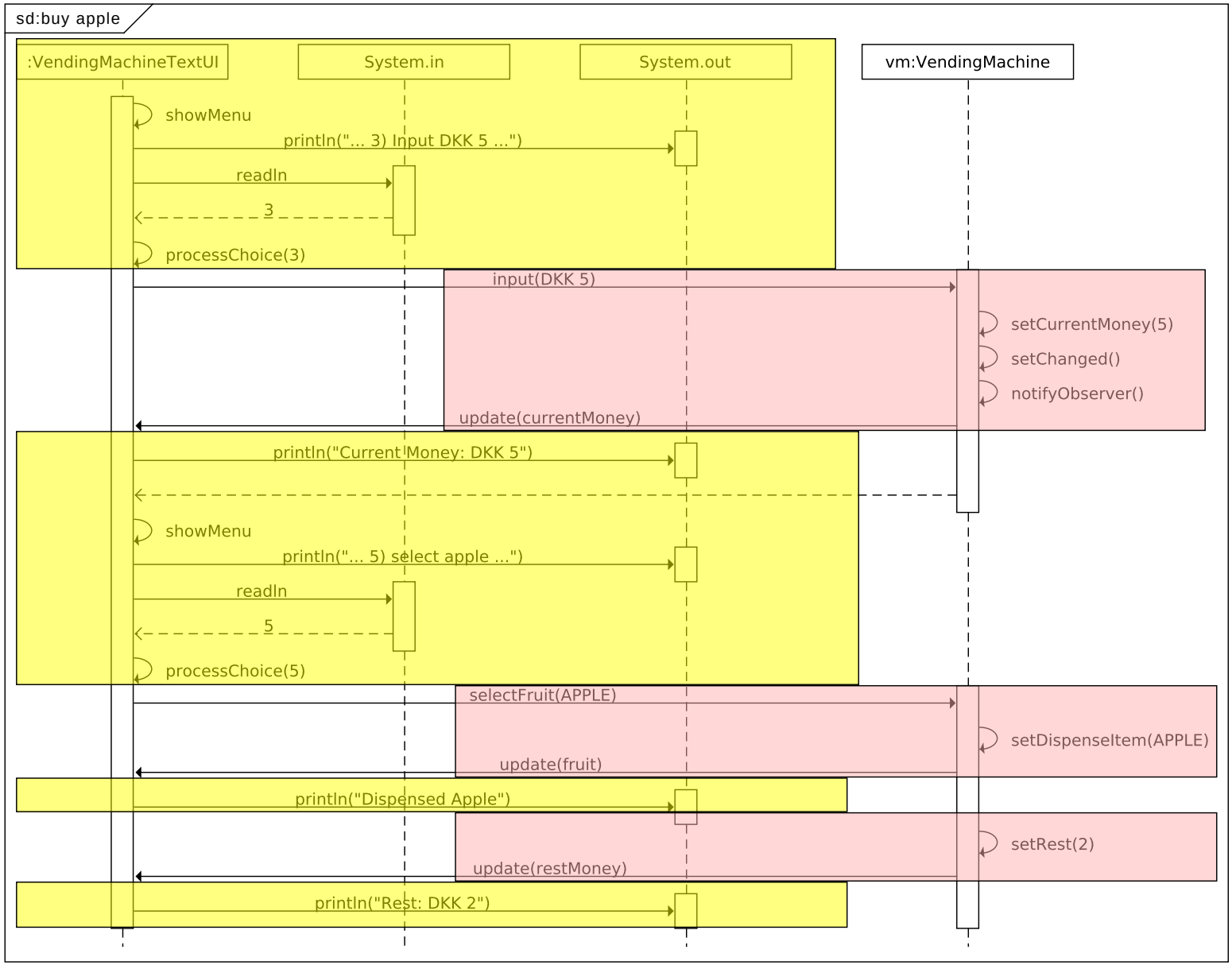
alternative scenarios (cont.)

- e1 user enters correct coins
- e2 user selects fruit but vending machine does not have the fruit anymore
- e3 nothing happens
- e4 user selects cancel
- e5 the money gets returned
- f1 user enters correct coins
- f2 user selects a fruit but vending machine does not have the fruit anymore
- f3 user selects another fruit
- f4 if money is correct fruit with rest money is dispensed; if money is not sufficient, the user can add more coins

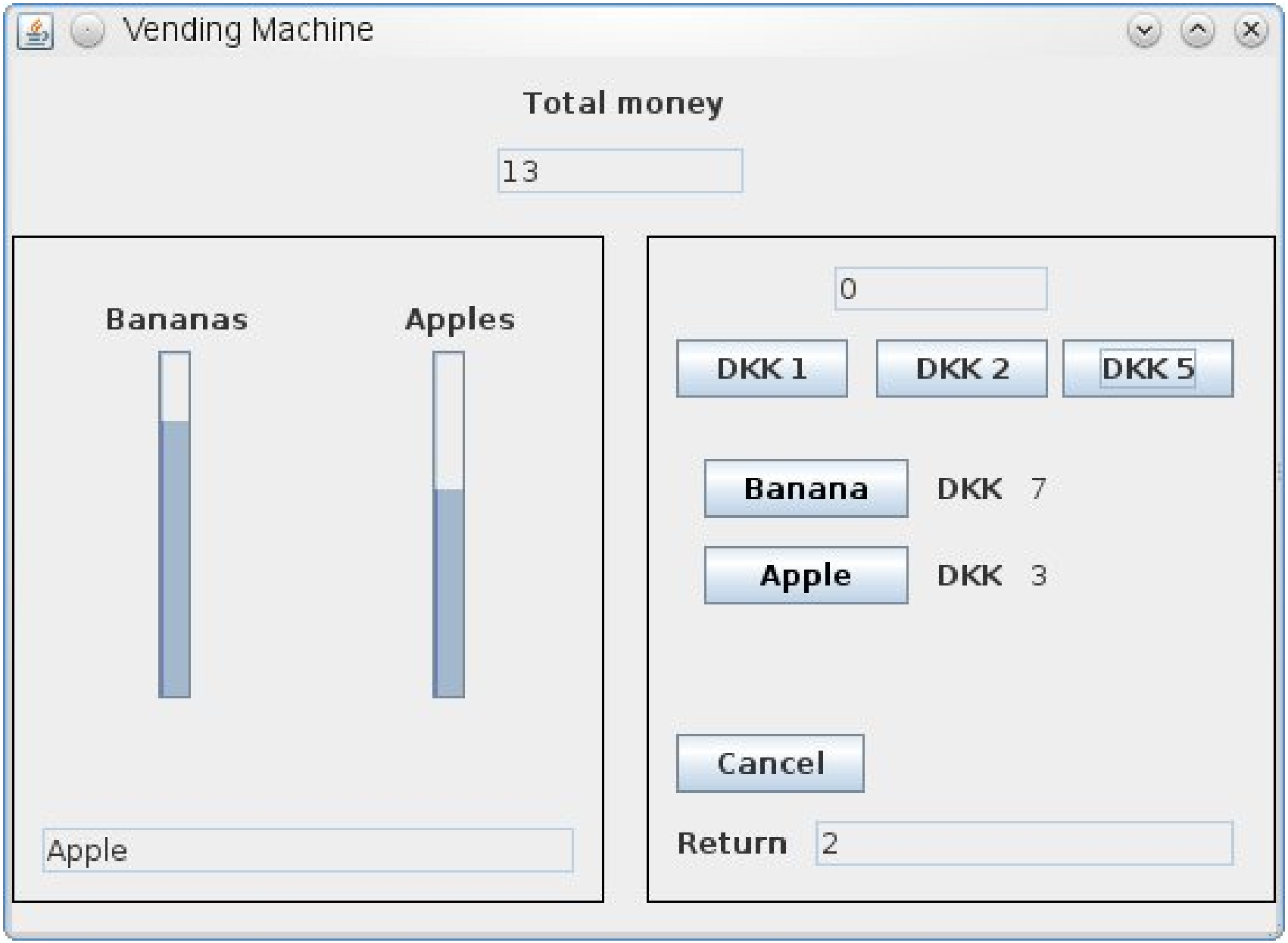
Presentation Layer: Command Line Interface

```
Current Money: DKK 5
    0) Exit
    1) Input 1 DKK
    2) Input 2 DKK
    3) Input 5 DKK
    4) Select banana
    5) Select apple
    6) Cancel
Select a number (0-6): 5
Rest: DKK 2
Current Money: DKK 0
Dispensing: Apple
```

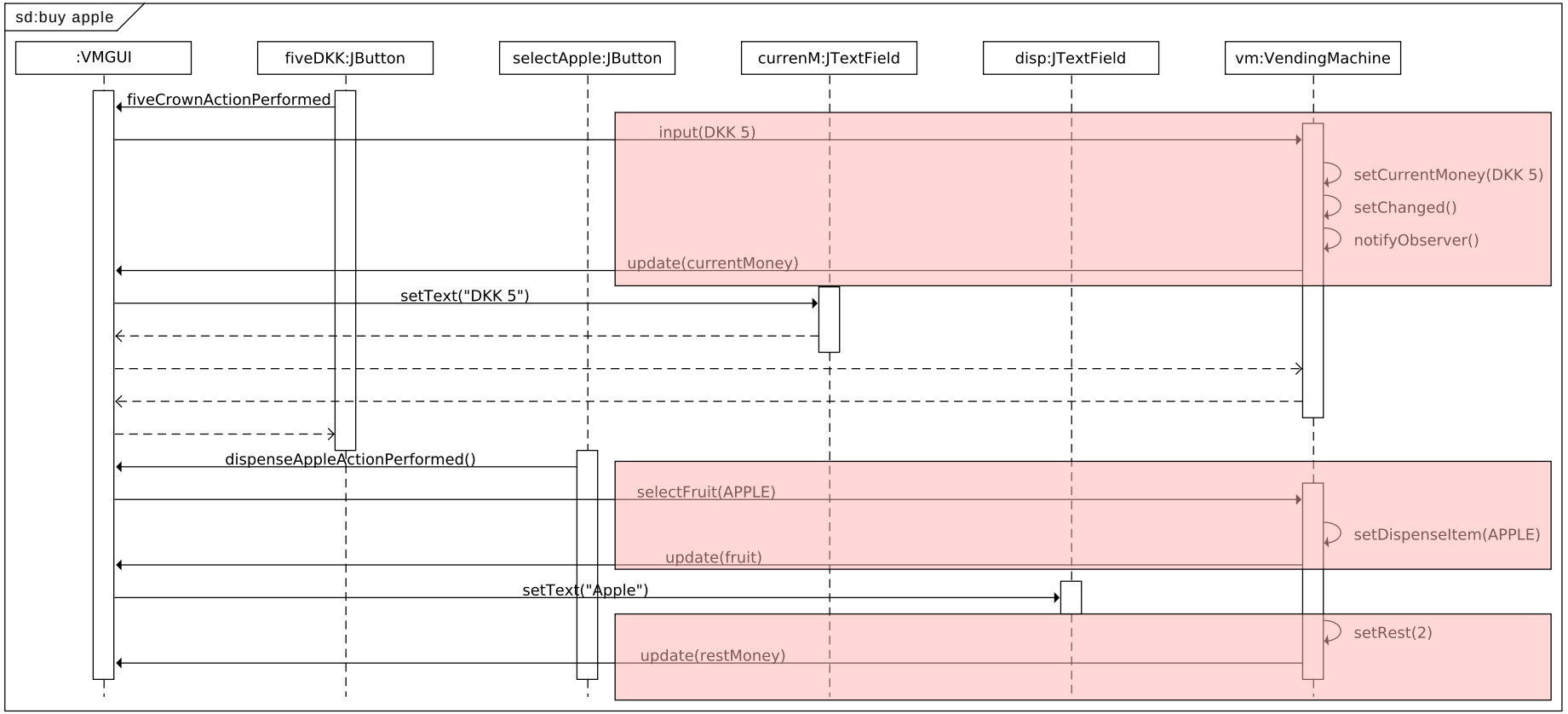
Presentation Layer: Command Line Interface



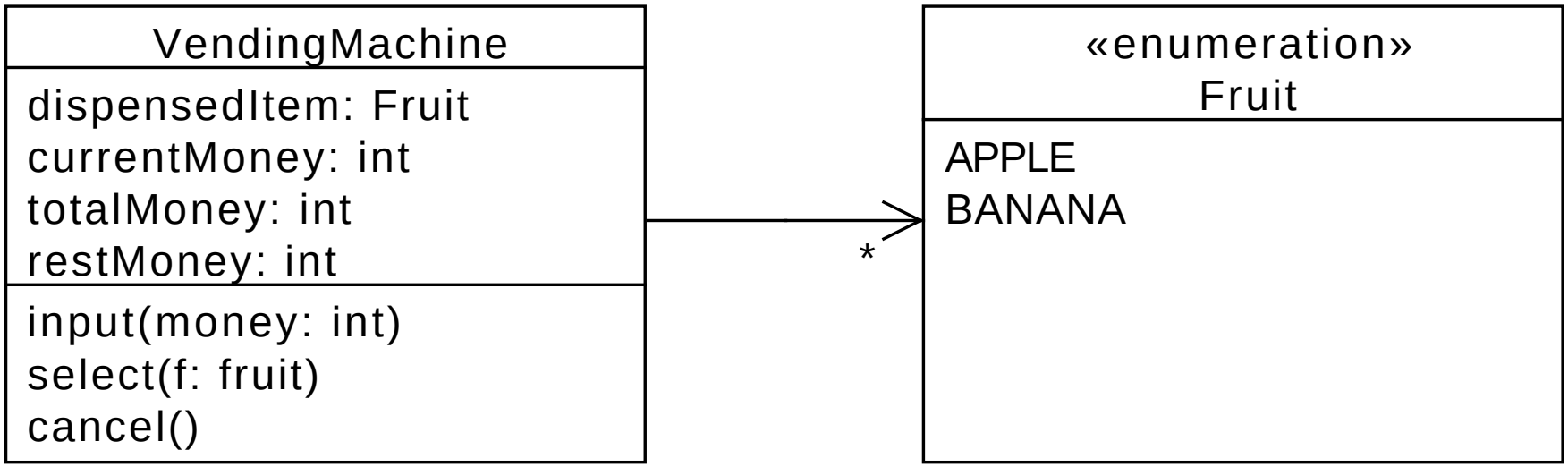
Presentation Layer: Swing GUI



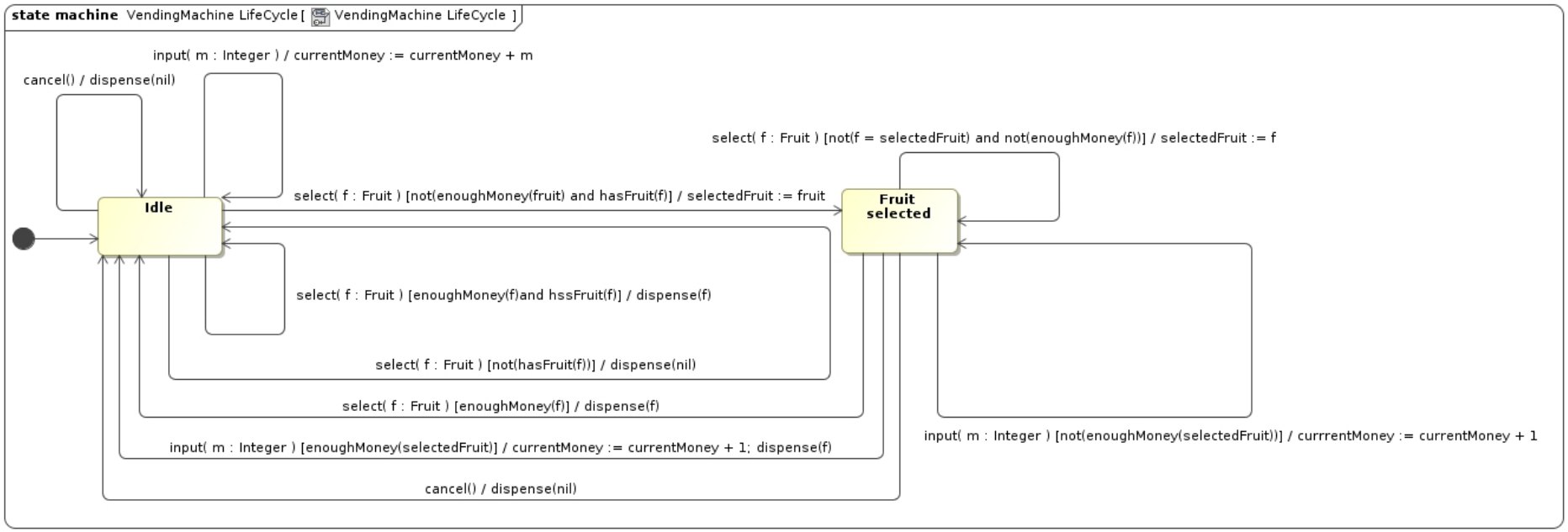
Presentation Layer: Swing GUI



Application Layer

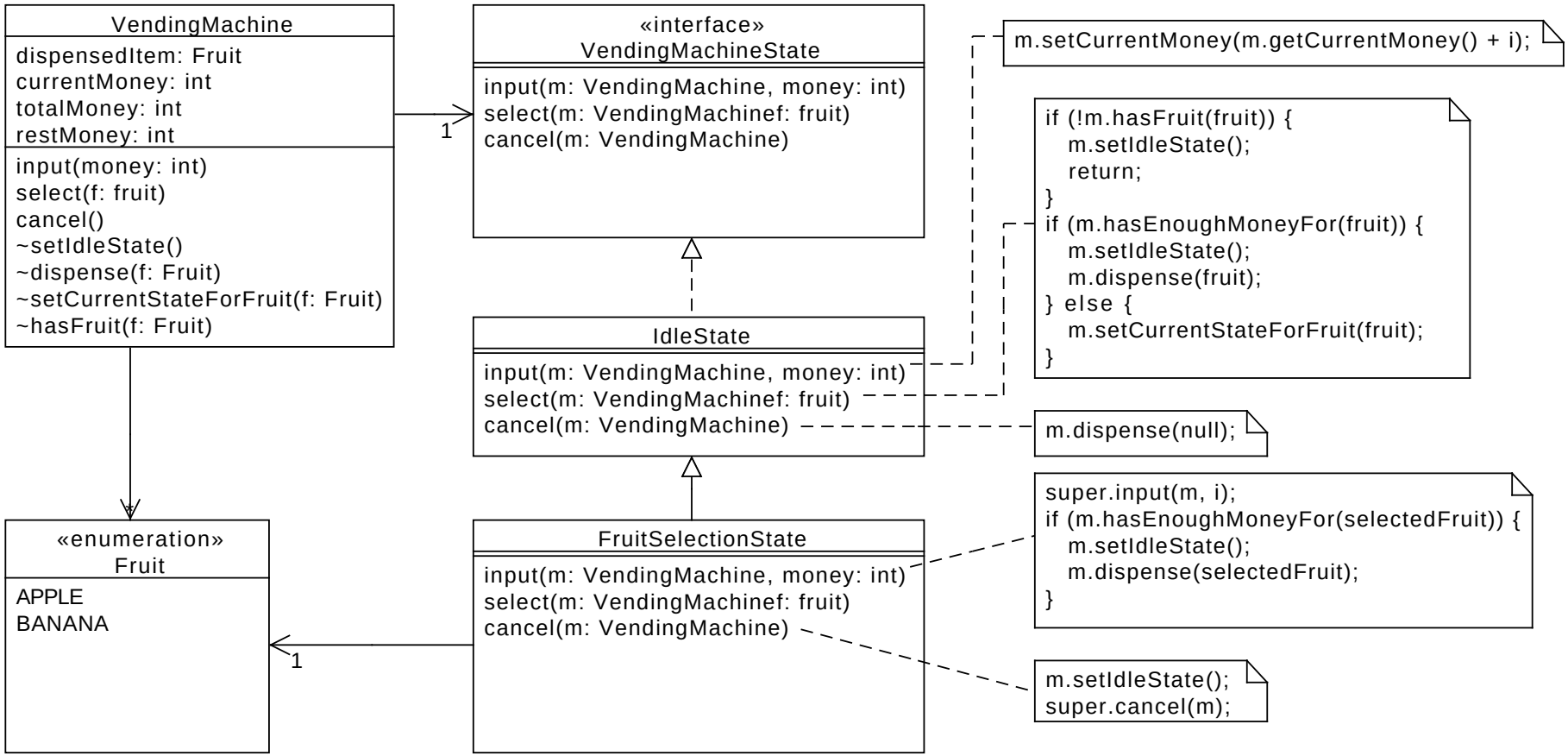


Application Logic



Application Logic Implementation

Uses the state pattern discussed before



Separation Presentation Layer from Application Layer

- Presentation layer translates
  - keyboard events, mouse movements . . . to messages in the application layer
  - application specific information as text or graphics
  - Reacts on events coming from the application layer
    - e.g. via the observer pattern (e.g. update messages)
    - also possible: use of event listeners
  - The presentation does not contain any business logic
    - Am action performed method does not do any business relevent computations
- Application layer
  - offers an abstract interface of messages to the presentation layer
    - e.g. input(int amount); select(Fruit fruit), getDispensedItem(), . . .
  - implements the business logic
  - Application logic does not provide any presentation logic
    - No calling of dialogs, no returning of images etc.

Advantages of the separation

1. Presentation layer can be exchanged/changed easily without compromising the business logic
2. It is easy to add different presentation layers on top of the same business logic at the same time
   - collaborative work: a person working on the application from a Web interface, the other from a stand-alone application
3. Automatic tests of business logic is easily possible because the application layer can be tested as any Java program

Use Case: Buy Fruit

name: Buy fruit

description: Entering coins and buying a fruit

actor: user

main scenario:

1. Input coins until the price for the fruit to be selected is reached
2. Select a fruit
3. Vending machine dispenses fruit

alternative scnearios:

a1. User inputs more coins than necessary
a2. select a fruit
a3. Vending machine dispenses fruit
a4. Vending machine returns excessive coins

...

Functional Test: for Buy Fruit Use Case: Input Data Sets

Input data set	Input property
A	Exact coins; enough fruits; first coins, then fruit selection
B	Exact coins; enough fruits; first fruit selection, then coins
C	Exact coins; not enough fruits; first coins, then fruit selection, then cancel
D	Exact coins; not enough fruits; first fruit selection, then coins, then cancel
E	More coins; enough fruits; first coins, then fruit selection
F	More coins; enough fruits; first fruit selection, then coins
G	More coins; not enough fruits; first coins, then fruit selection, then cancel
H	More coins; not enough fruits; first fruit selection, then coins, then cancel
I	Less coins; enough fruits; first coins, then fruit selection
J	Less coins; enough fruits; first fruit selection, then coins
K	Less coins; not enough fruits; first coins, then fruit selection, then cancel
L	Less coins; not enough fruits; first fruit selection, then coins, then cancel

Functional Test for Buy Fruit Use Case: Test Cases

Input data set	Contents	Expected Output
A	1,2; apple	apple dispensed
B	Apple; 1,2	apple dispensed
C	1,2; apple; cancel	no fruit dispensed; returned DKK 3
D	Apple; 1,2; cancel	no fruit dispensed; returned DKK 3
E	5, apple	apple dispensed; returned DKK 2
F	Apple; 5	apple dispensed; returned DKK 2
G	5, apple; cancel	no fruit dispensed; returned DKK 5
H	Apple; 5; cancel	no fruit dispensed; returned DKK 5
I	5; banana	no fruit dispensed; current money shows 5
J	Banana; 5,1	no fruit dispensed; current money shows 6
K	5,1; banana; cancel	no fruit dispensed; returned DKK 6
L	Banana; 5,1;cancel	no fruit dispensed; returned DKK 6

Manual vs Automated Tests

- The test cases can be tested manually
  - Open the application
  - Input the data according to the input data set description
  - Check that the output is the expected one
- But also automatically
  - There are tools that execute the user interface automatically
    - But they have a lot of problems: e.g. don't tolorate additional GUI elements or changed GUI elements
  - Easier: test the application layer automatically
    - application layer works with plain Java methods
    - → they can be exercised using, e.g., JUnit
    - Presentation independent; focuses on the business logic

Functional Test for Buy Fruit Use Case: JUnit Tests

```
public void testInputDataSetA() {
    VendingMachine m = new VendingMachine(10, 10);
    m.input(1);
    m.input(2);
    assertEquals(3, m.getCurrentMoney());
    m.selectFruit(Fruit.APPLE);
    assertEquals(Fruit.APPLE, m.getDispensedItem());
}

public void testInputDataSetB() {
    VendingMachine m = new VendingMachine(10, 10);
    m.selectFruit(Fruit.APPLE);
    m.input(1);
    m.input(2);
    assertEquals(0, m.getCurrentMoney());
    assertEquals(Fruit.APPLE, m.getDispensedItem());
}
```

Functional Test: JUnit Tests (cont.)

```
public void testInputDataSetC() {
    VendingMachine m = new VendingMachine(0, 0);
    m.input(1);
    m.input(2);
    assertEquals(3, m.getCurrentMoney());
    m.selectFruit(Fruit.APPLE);
    assertEquals(null, m.getDispensedItem());
    m.cancel();
    assertEquals(null, m.getDispensedItem());
    assertEquals(3, m.getRest());
}

public void testInputDataSetD() {
    VendingMachine m = new VendingMachine(0, 0);
    m.selectFruit(Fruit.APPLE);
    m.input(1);
    m.input(2);
    assertEquals(3, m.getCurrentMoney());
    m.cancel();
    assertEquals(null, m.getDispensedItem());
    assertEquals(3, m.getRest());
}

...
```

Summary

- There is a difference between good and bad design
  - Good design is important
- Some basic principles:
  - DRY, KISS
- Patterns capture knowledge
  - → Design patterns capture common object oriented design principles
- Low Coupling and High Cohesion
  - → Leads to modularized / object oriented design
- Layered Architecture
  - → application of the low coupling and high cohesion principle
  - supports independence of application from UI
  - supports automated tests

→ 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!