Software Engineering I (02161)

Week 6

Assoc. Prof. Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

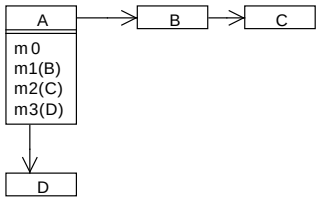
Spring 2011

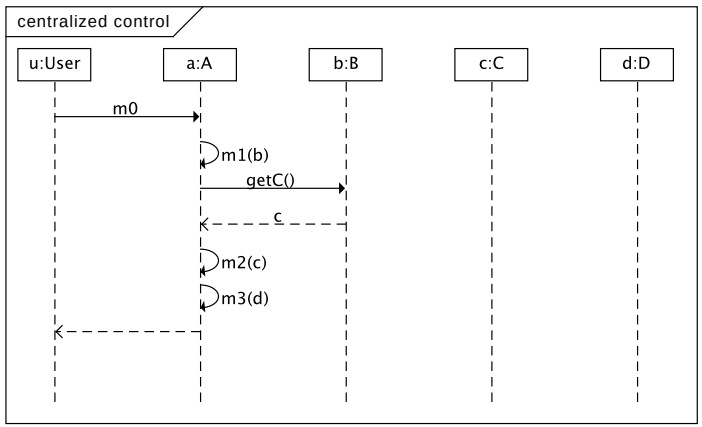
Recap

- Activity Diagrams
  - Represents the flow of activities
  - Can also be used for showing data flow
  - Usage: Business Processes, Workflows, Algorithms
- CRC cards (Class Responsibility Collaboration)
  - simulate OO design (driven by user cases / user stories)
  - Technique to understand problem domain and find +OO Design++
- Sequence Diagrams
  - Shows who sent which message to whom
  - Usually shows one execution trace through the system

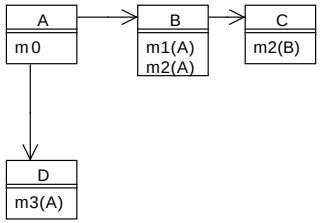
Recap: Distributed Computation

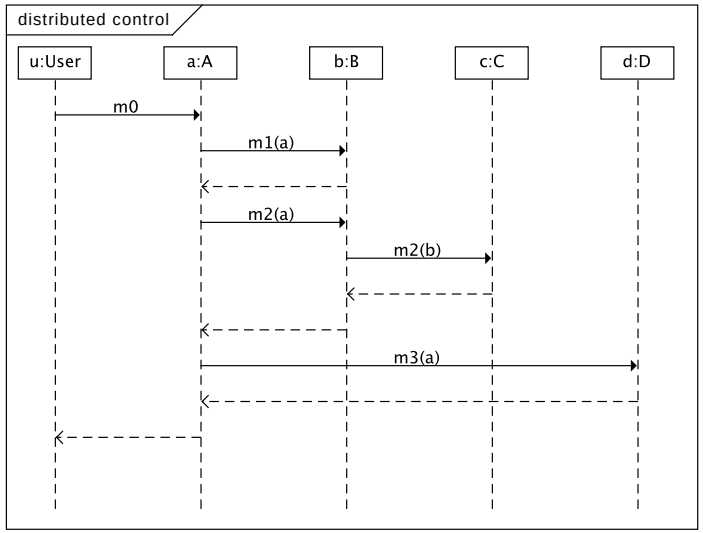
Centralized computation





Distributed Computation





- Object Orientation: Centralized vs Decentralized control

→ Computation can be adapted by using subclasses and overriding of methods

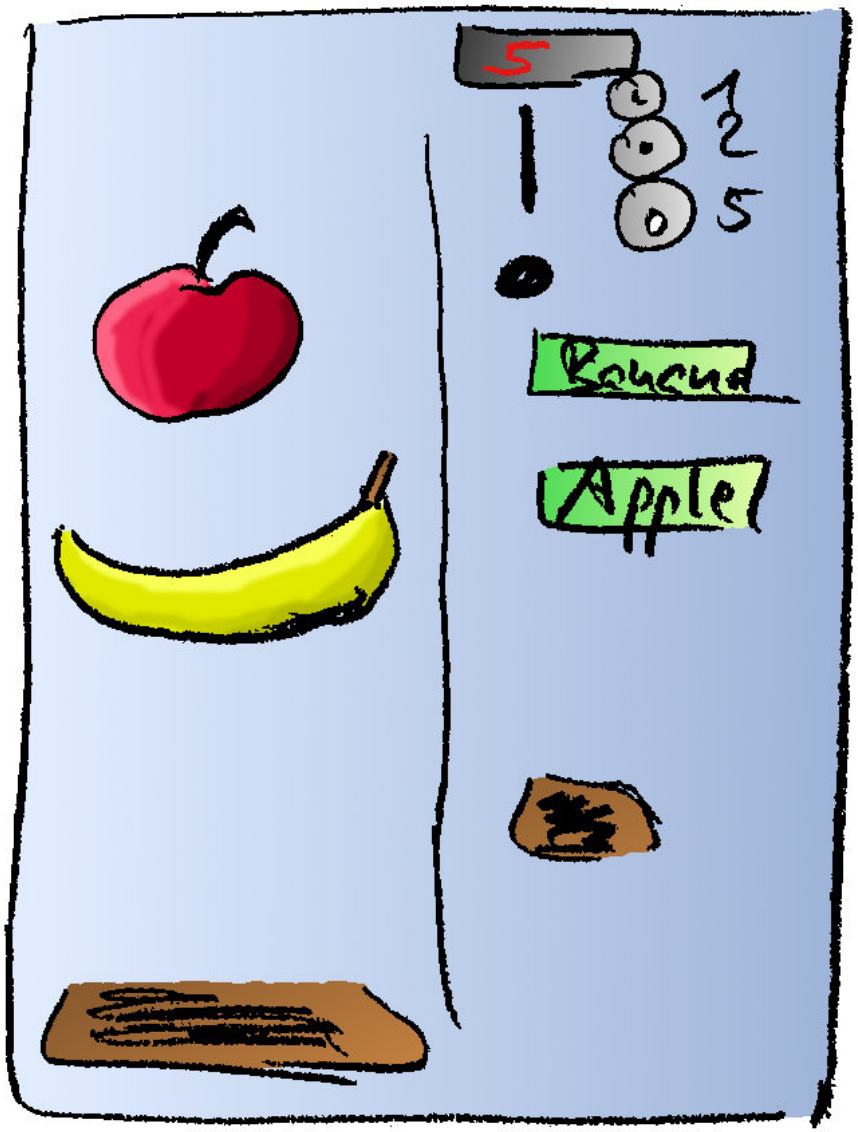
→ Leads to *real* OO design: objects have data and operations

→ A lot of Design Patterns show how to use this principle to solve common design problems

State Machines

- Previously we have seen activity diagrams (focus on activities of the system) and sequence diagrams (focus on how messages are being exchanged
- State machines show the effect of events to an object/system
  - → What does an object do when it receives an event
  - → How does the the object react in the future on receiving that event
    - Being in a state, the occurrence of an event moves the object/system represented by the state machine into a new state
- Very often used to describe system close to hardware, user interfaces, and Web sites
  - e.g. Vending machine
  - UI: You are here and with these actions you can go there

Example Vending Machine

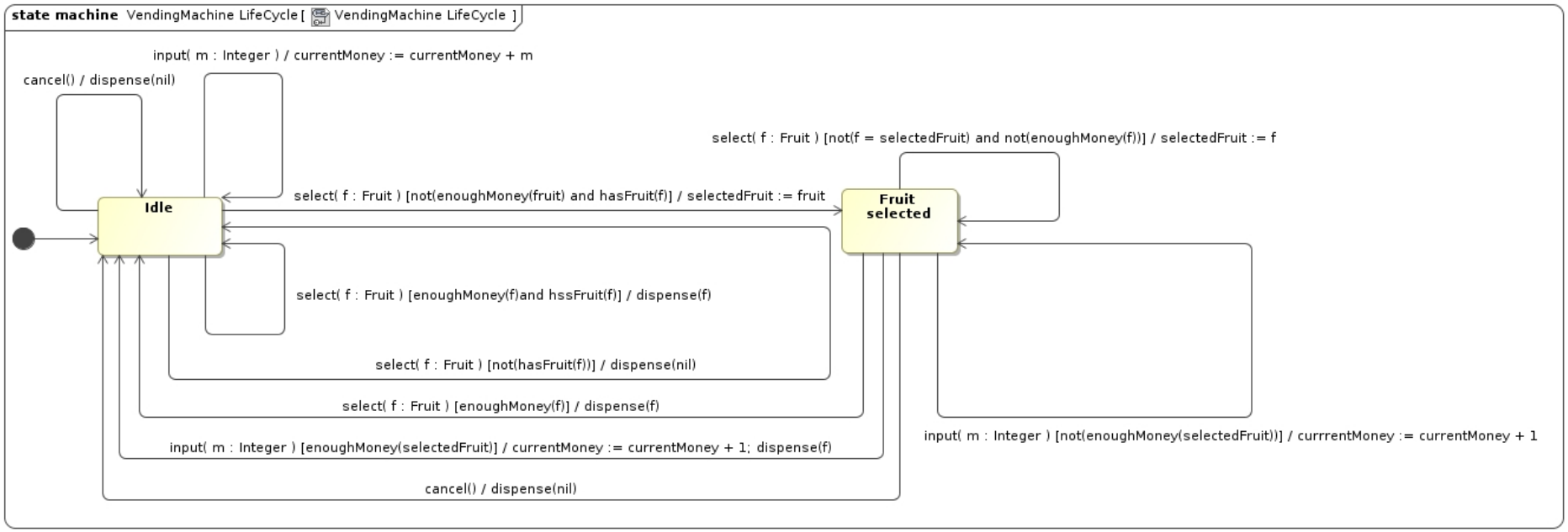


- Actions
  - Input coins
  - Press button for bananas or apples
  - Press cancel
- Displays
  - current amount of money input
- Effects
  - Return money
  - Dispense banana or apple

State transition table

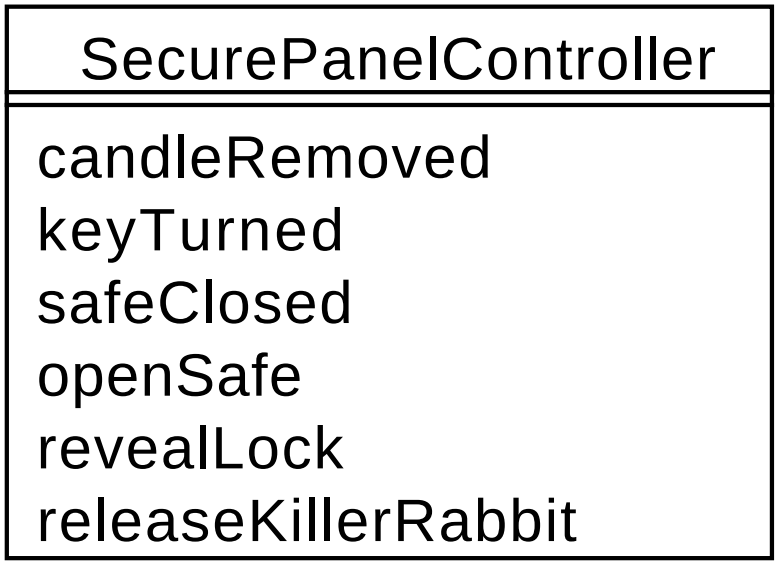
event	guard	state	state	state
		Idle (I)	*Banana selected and not enough money (B)*	*Apple selected and not enough money (A)*
banana	*enough money for banana*	dispense banana and rest money-> I	dispense banana and rest money-> I	dispense banana and rest money-> I
banana	*not enough money for banana*	-> B	-> B	-> B
banana	*no bananas available*	-> I	-> I	-> I
apple	*enough money for apple*	dispense apple and rest money -> I	dispense apple and rest money -> I	dispense apple and rest money -> I
apple	*not enough money for apple*	-> A	-> A	-> A
apple	*no apples available*	-> I	-> I	-> I
money	*enough money for banana*	add money to current money	dispense banana and rest money-> I	add money to current money
money	*enough money for apple*	add money to current money	add money to current money	dispense apple and rest money -> I
money	*not enough money for neither banana nor apple*	add money to current money	add money to current money	add money to current money
cancel		return current money -> I	return current money -> I	return current money -> I

UML State Machines

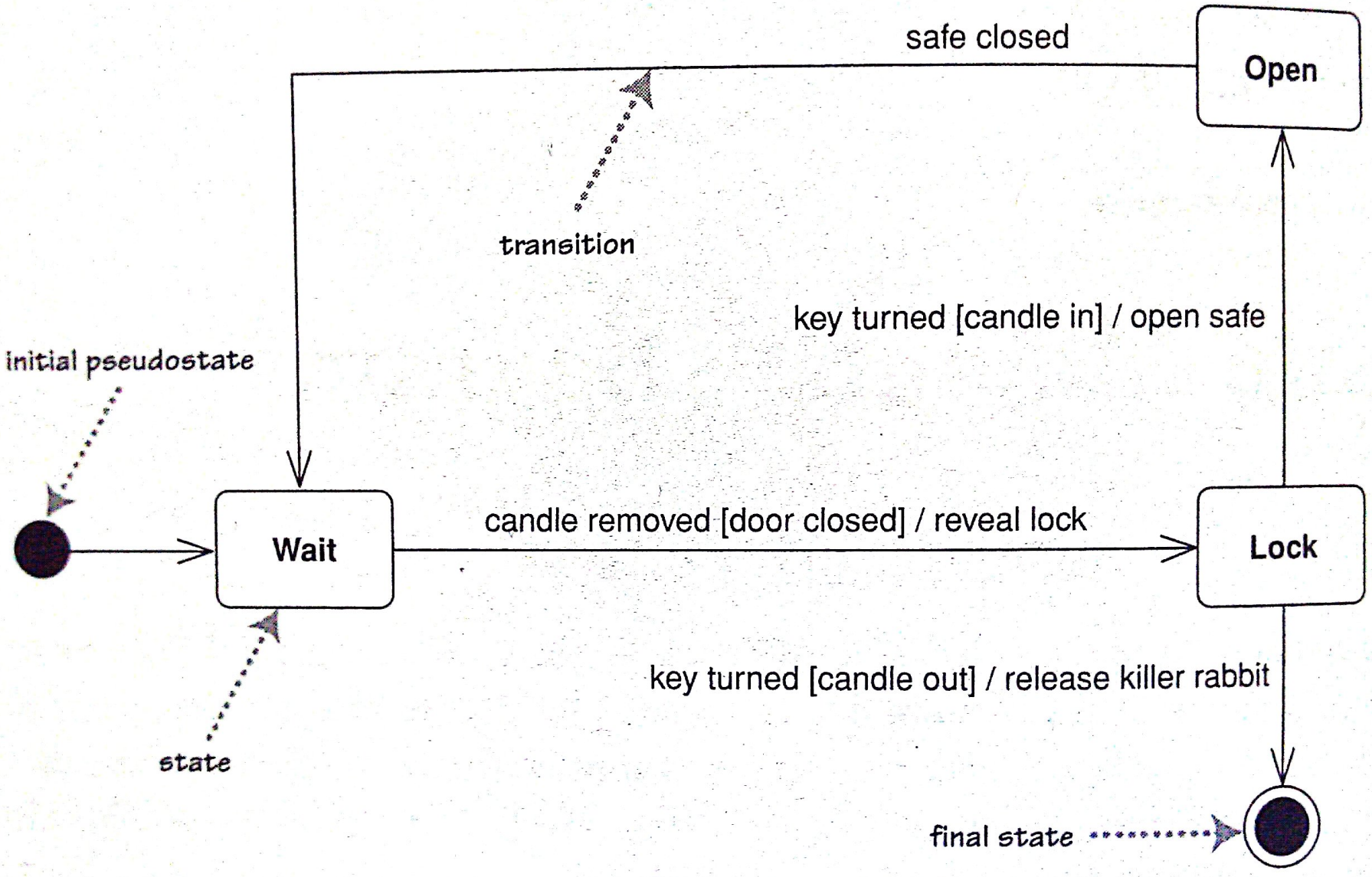


Example

- 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

- 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

Implementation 1

- The current state is stored in a variable
- Events are handled with the HandleEvent method

```
public class SecretPanelController {
 public void HandleEvent (PanelEvent anEvent) {
   switch (CurrentState) {
     case PanelState.Open :
       switch (anEvent) {
         case PanelEvent.SafeClosed :
           CurrentState = PanelState.Wait;
           break;
       }
       break;
     case PanelState.Wait :
       switch (anEvent) {
         case PanelEvent.CandleRemoved :
           if (isDoorOpen) {
             RevealLock();
             CurrentState = PanelState.Lock;
           }
           break;
       }
       break;
     case PanelState.Lock :
       switch (anEvent) {...}
       break;
    }
   }
 }
}
```

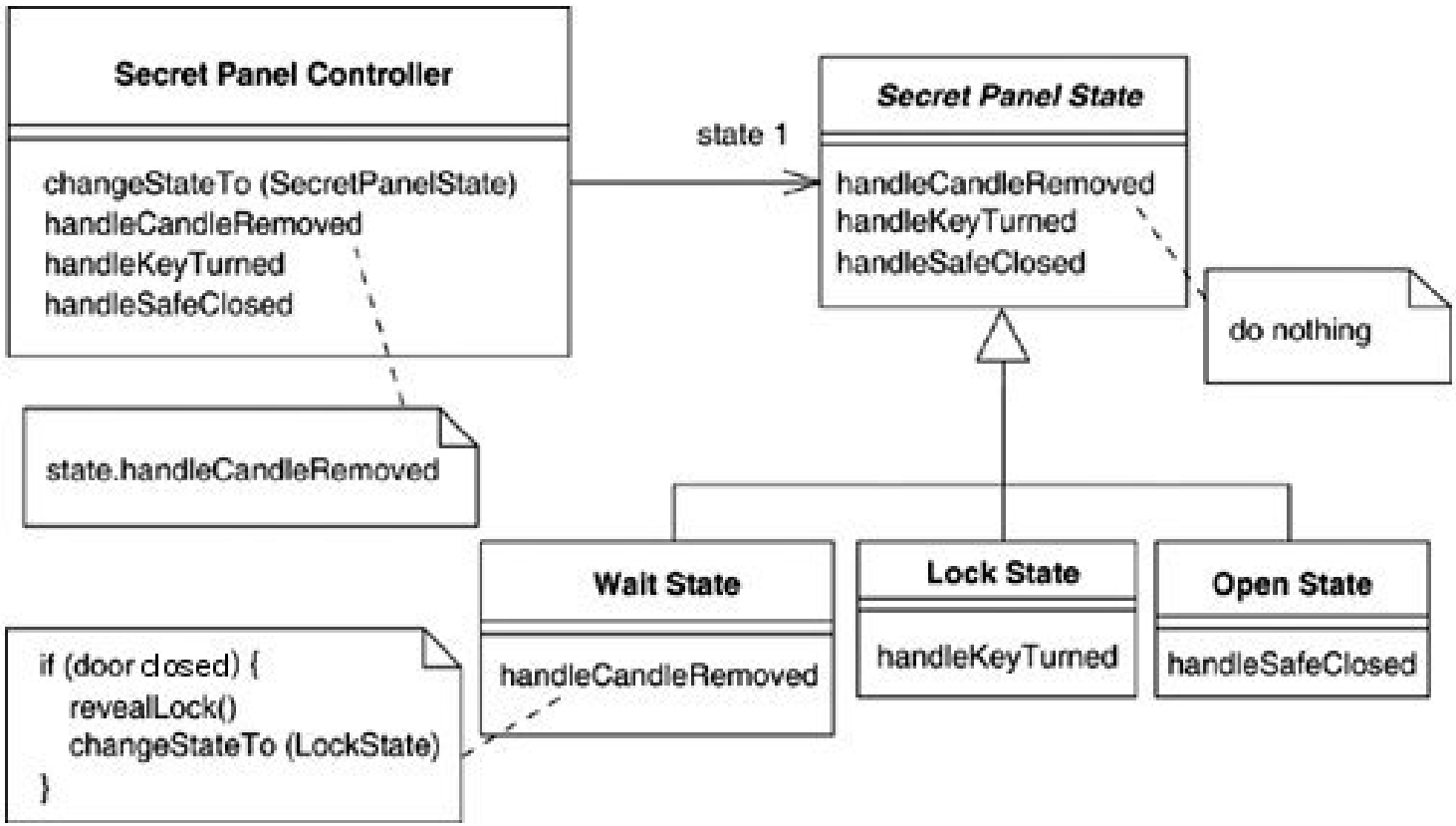
Implementation 2

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

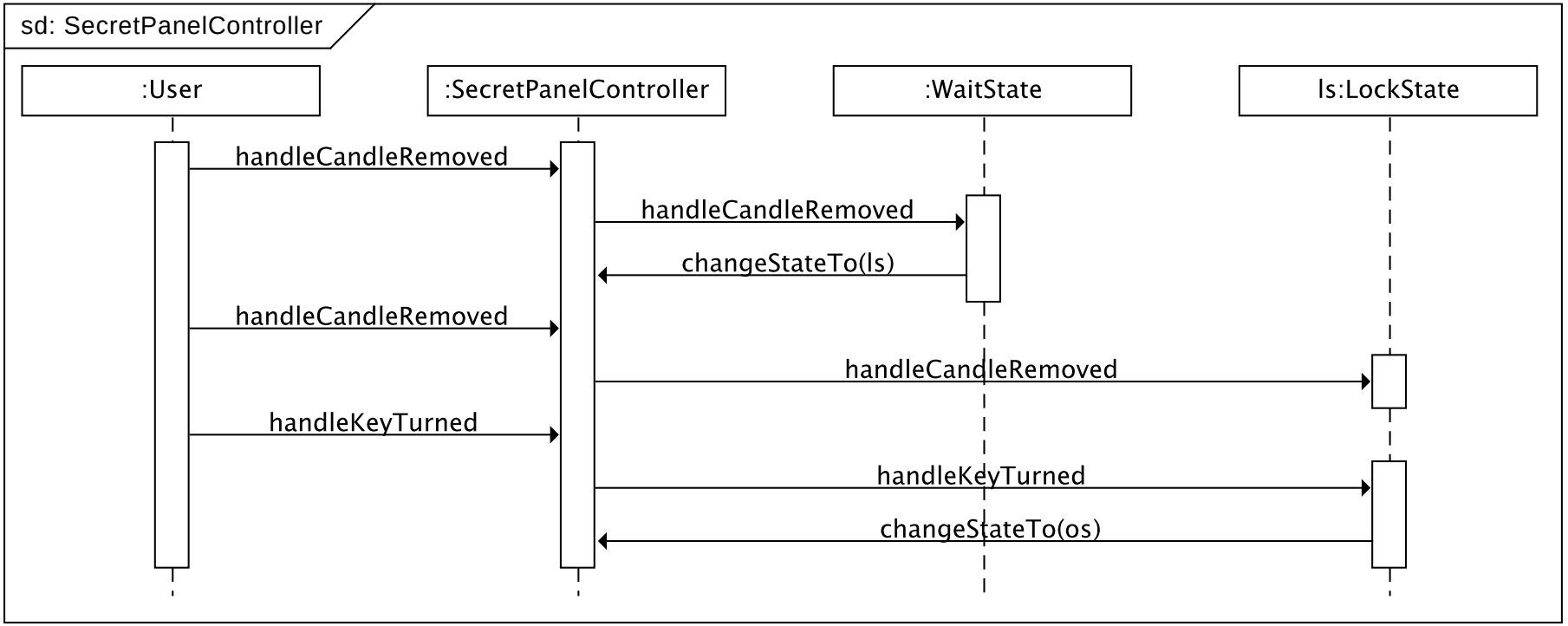
```
public class SecretPanelController {
   enum states { wait, lock, open, finalState };
   states state = states.wait;

   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



SecretPanelController Sequence Diagram

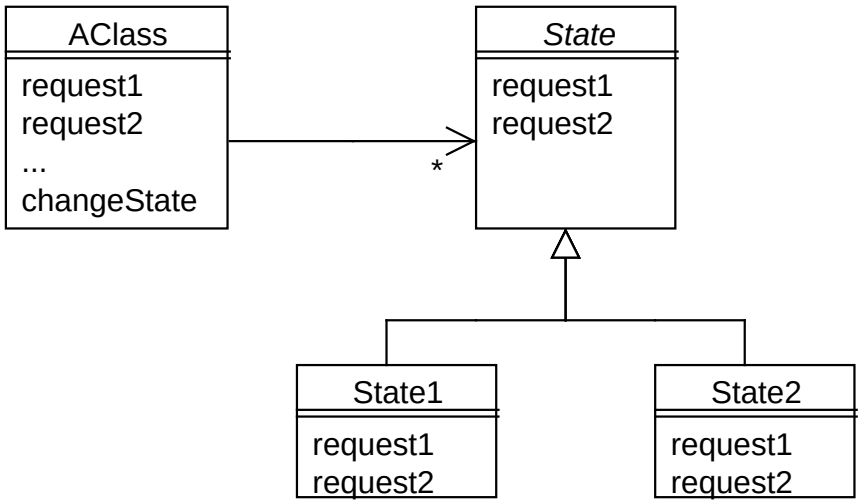


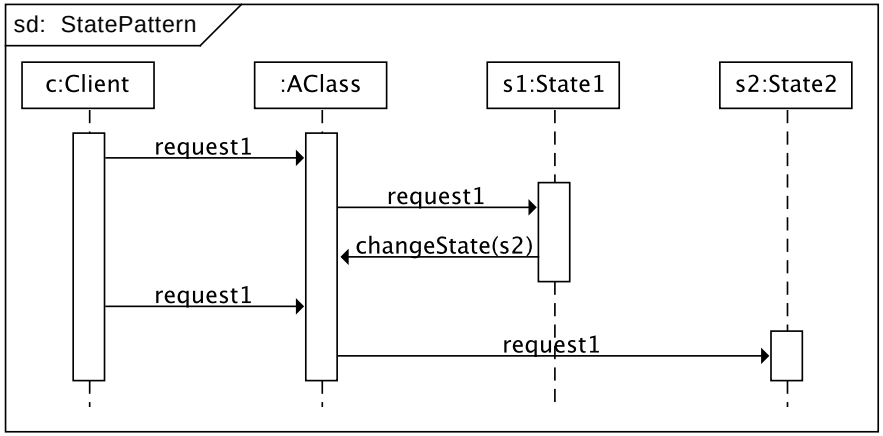
State Pattern

State Pattern

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

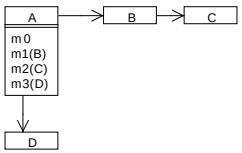
- This pattern delegates the behaviour of one object to another object

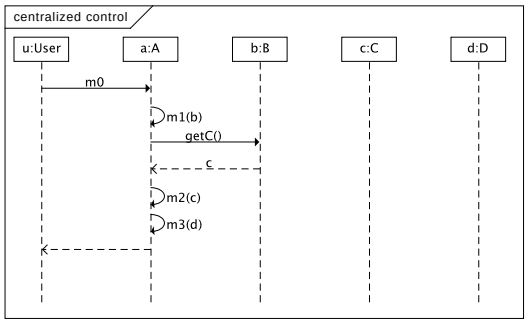


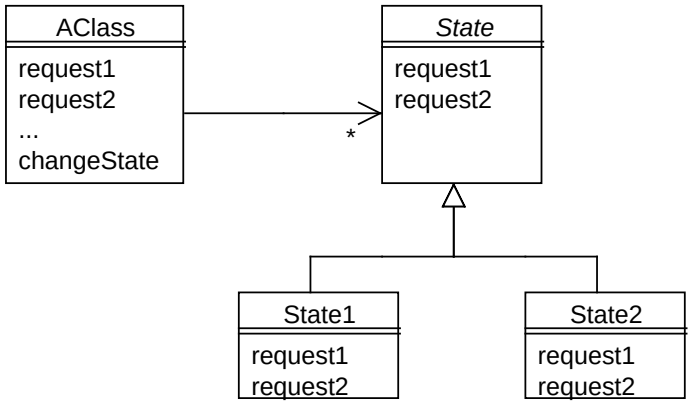


State pattern and distributed control

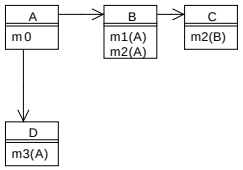
Centralized computation

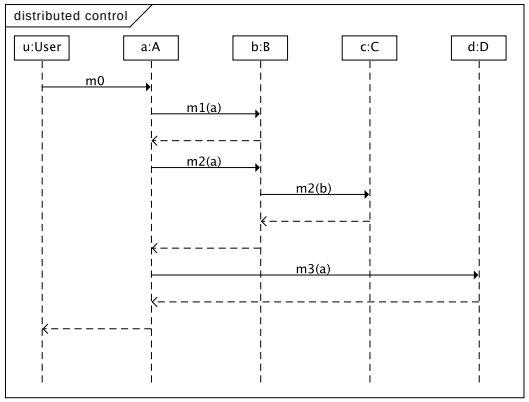


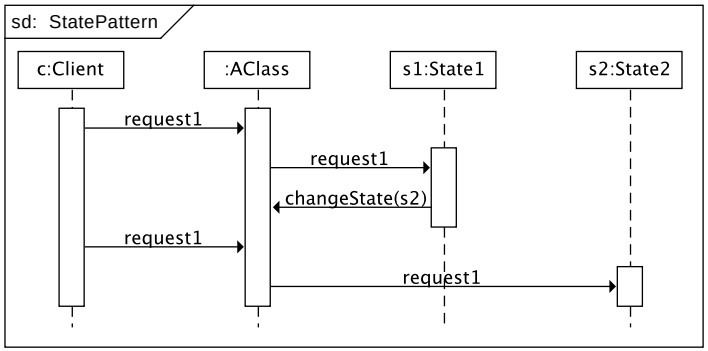




Distributed Computation

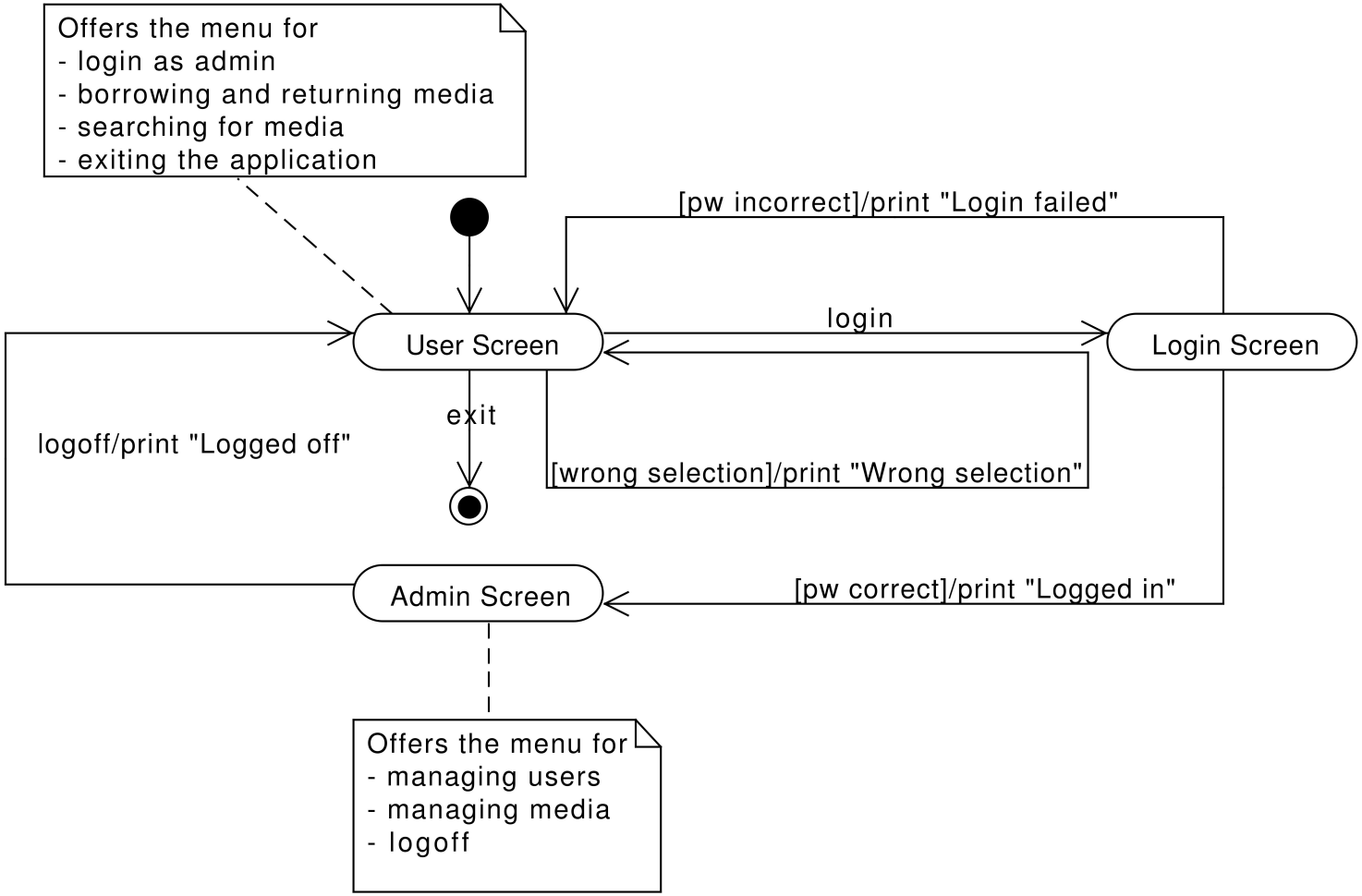






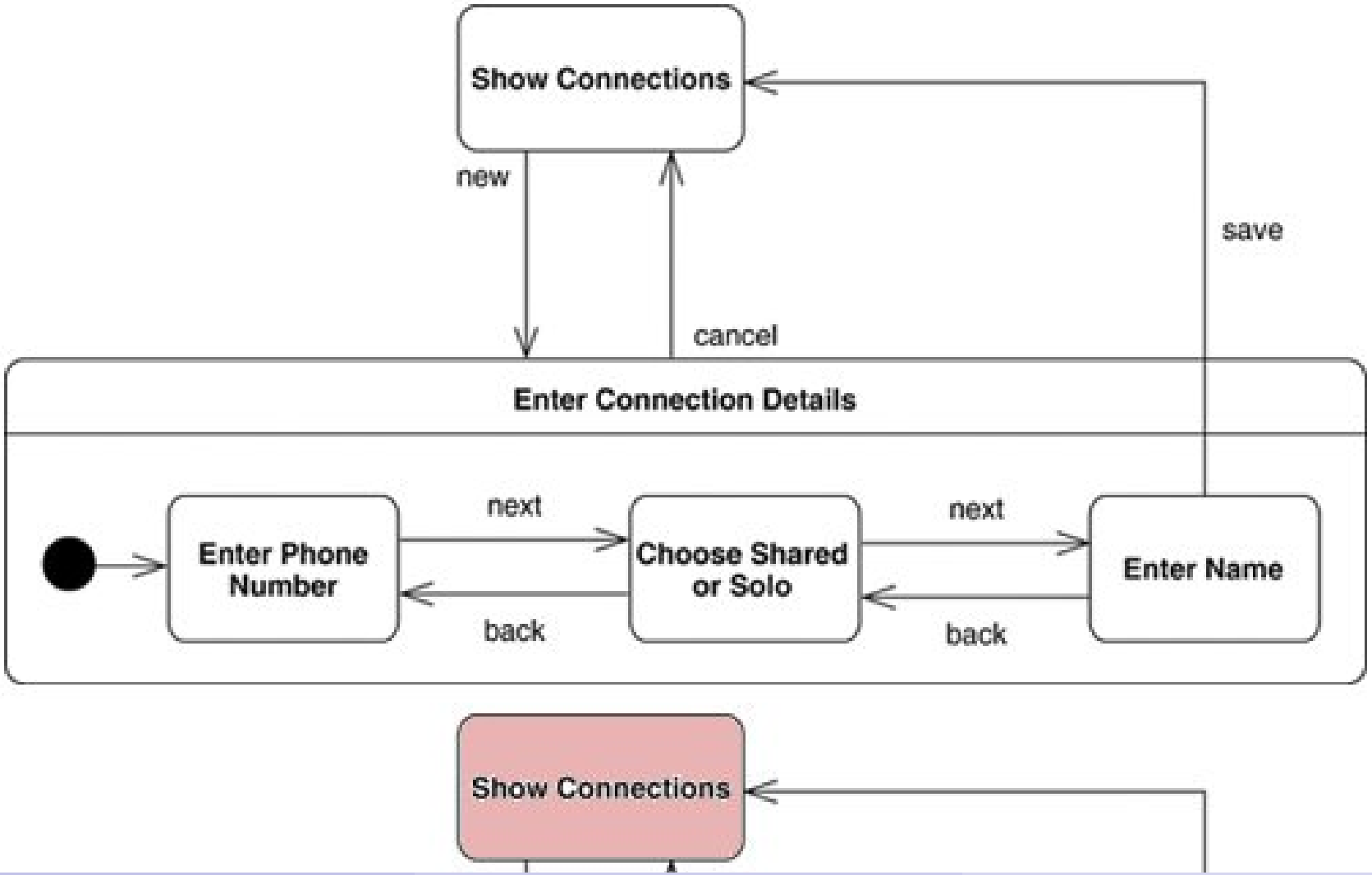
Example Library Application

- State machines can be used to describe user interfaces



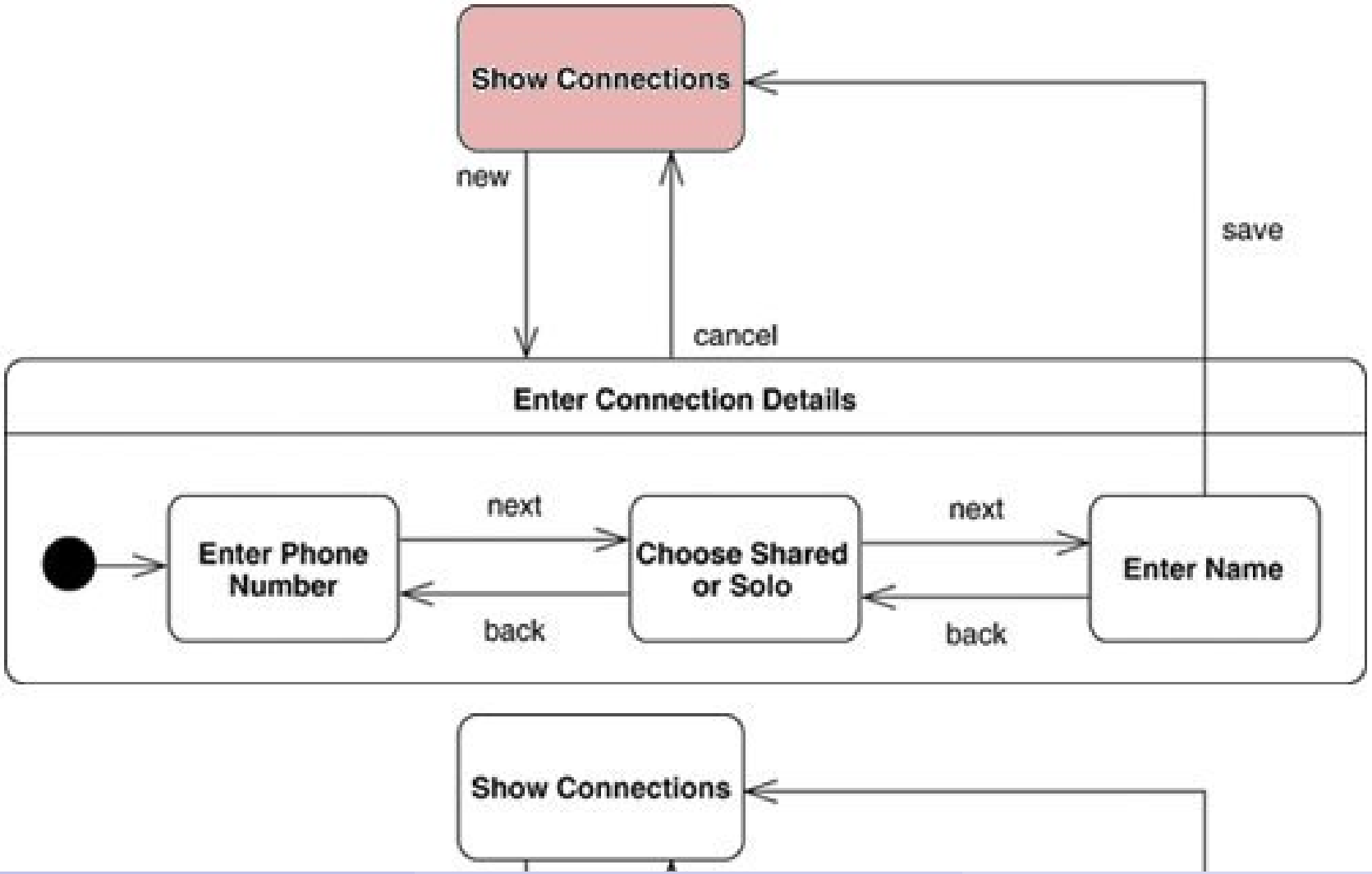
Sub states

- Substates help structure complex state diagrams (similar to subroutines)

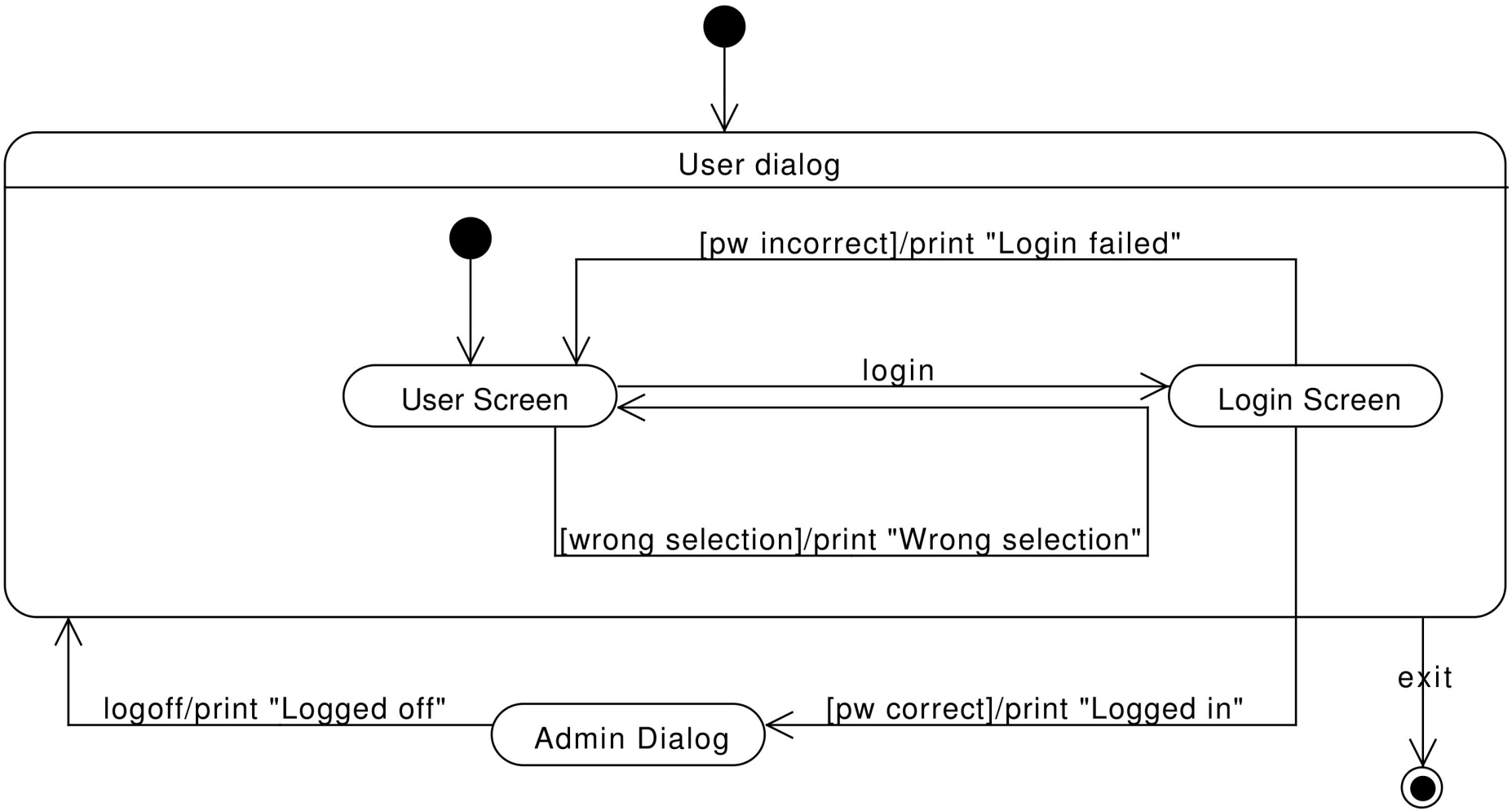


Sub states II: Leaving sub states

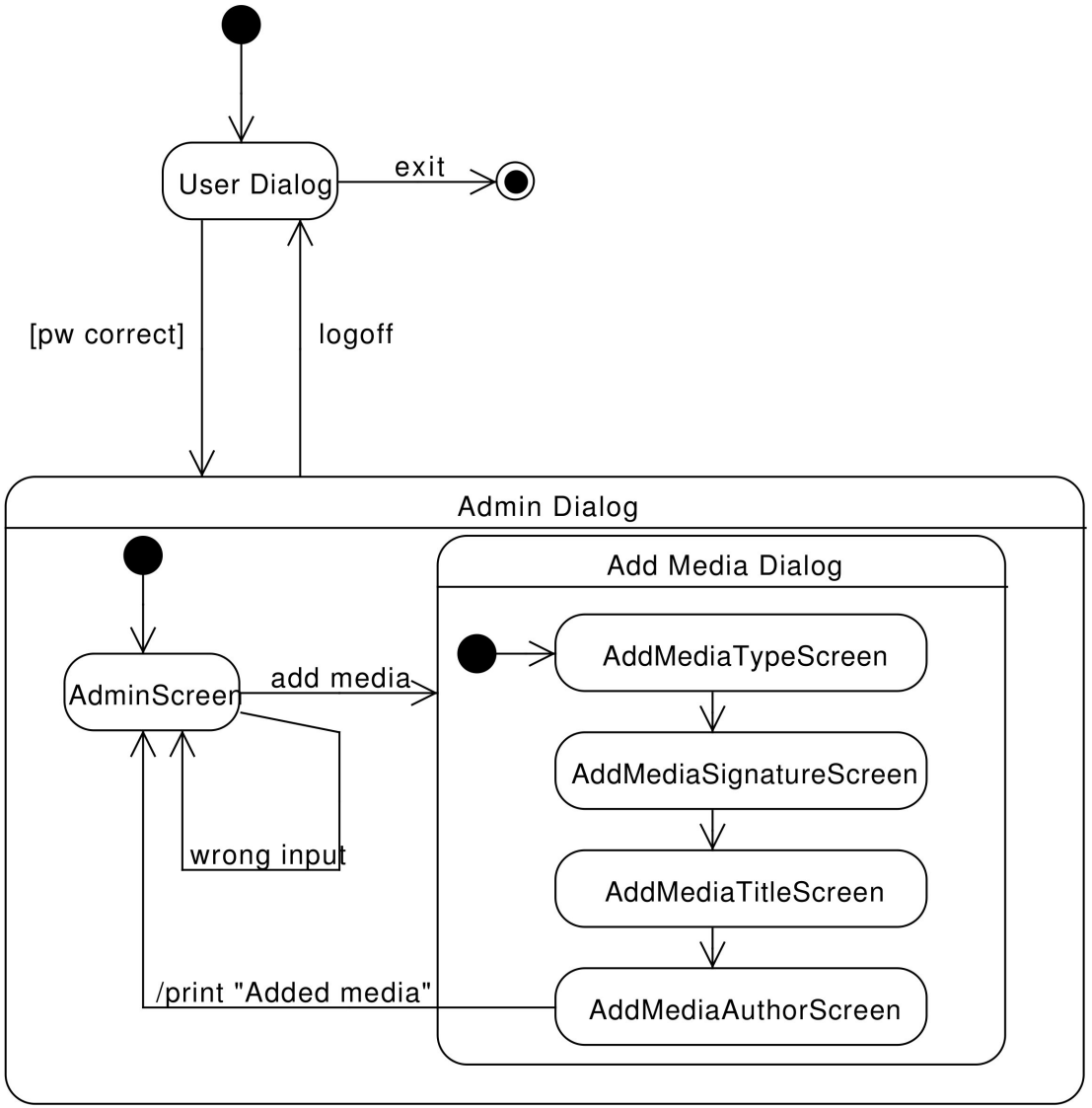
- Substates can be left at any time when an outgoing super state transition fires (e.g. cancel below)



Library App


User dialog
[pw incorrect]/print "Login failed"
User Screen
login
Login Screen
[wrong selection]/print "Wrong selection"
exit
logoff/print "Logged off"
Admin Dialog
[pw correct]/print "Logged in"

Library App: Admin Dialog

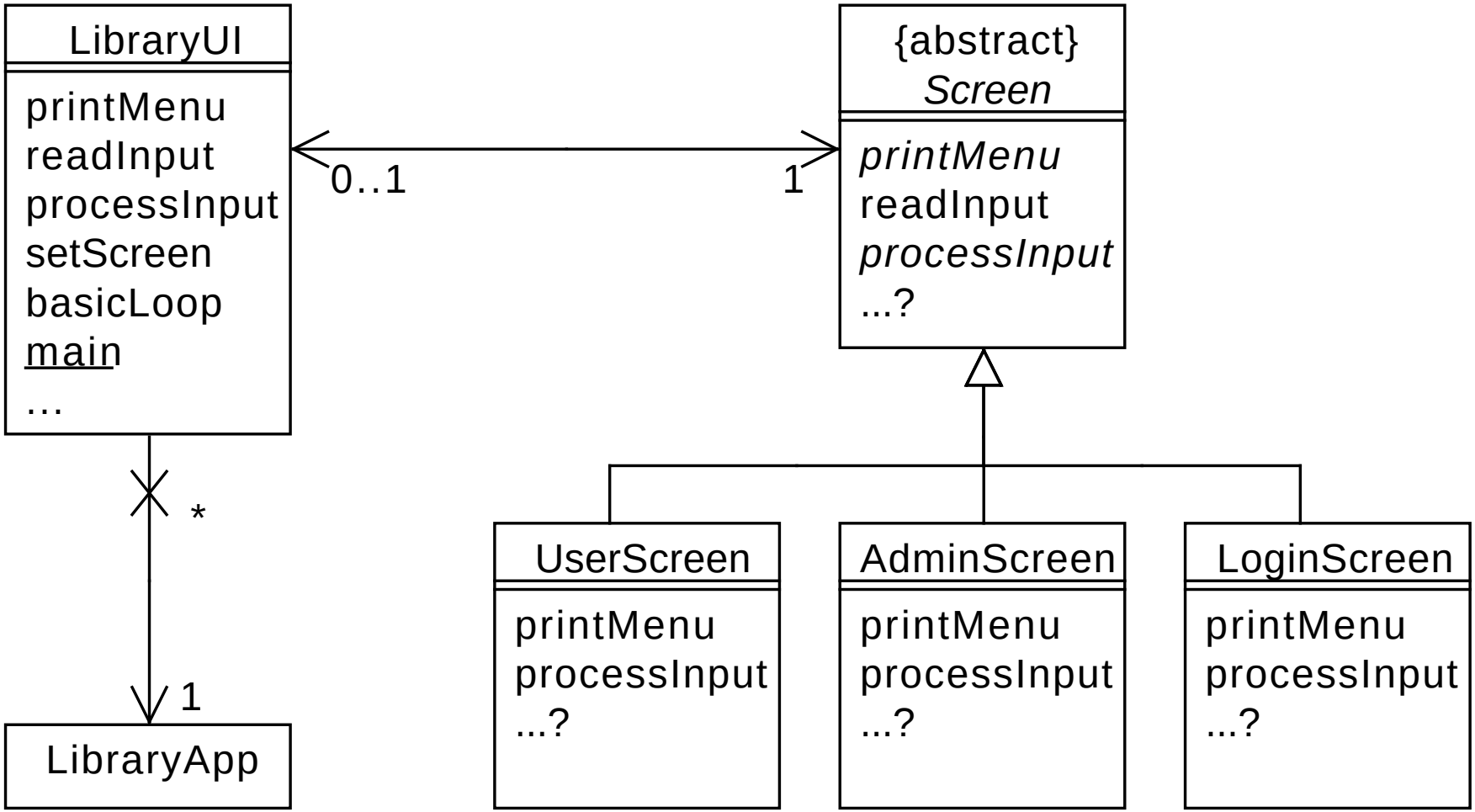


- The contents of composite states can be either hidden or shown
- Composite states can contain composite states

Library App user interface exercise

1) Given tests for the functionality login; implement the tests using the state pattern
2) Design, test, and implement the remaining functionality of the library application

Library App UI: State Pattern



Library App: main application

```
public static void main(String[] args) throws IOException {
   BufferedReader in = new BufferedReader(new InputStreamReader(System.in)
   PrintWriter out = new PrintWriter(System.out, true);
   LibraryUI ui = new LibraryUI();
   ui.basicLoop(in, out);
}

public void basicLoop(BufferedReader in, PrintWriter out)
      throws IOException {
  String selection;
  do {
    printMenu(out);
    selection = readInput(in);
  } while (!processInput(selection, out));
}
```

Library App UI: tests

- In general, user interface code is difficult to test

→ use a layered architecture with a thin presentation layer

- However, sometimes the UI can be made testable

→ Here: the basic methods work on BufferedReader and PrintWriter instead of InputStream and PrintStream

→ Now the methods can be tested by using arbitrary streams instead of just System.in and System.out

Library App UI: tests

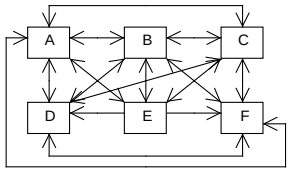
```
@Test
public void testTestExitApplication1() throws IOException {
  LibraryUI libraryUI = new LibraryUI();
  testScreenInteraction(libraryUI, "0) Exit", "0", "Exited.",true);
}


public void testScreenInteraction(LibraryUI libraryUI,
      String expectedMenu,
      String input, String expectedOutput, boolean expectedExitStatus)
      throws IOException {

  StringWriter out = new StringWriter();
  libraryUI.printMenu(new PrintWriter(out));
  assertTrue(out.toString().contains(expectedMenu));

  BufferedReader reader = new BufferedReader(new StringReader(input));
  String line = libraryUI.readInput(reader);
  assertEquals(input,line);

  out = new StringWriter();
  boolean exit = libraryUI.processInput(line, new PrintWriter(out));
  assertEquals(expectedOutput+"\n",out.toString());
  assertEquals(expectedExitStatus,exit);
}
```

Low Coupling

Low coupling

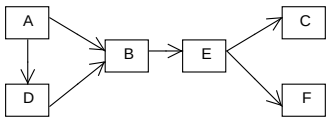
- An object / class is connected only to a few other classes
- It fullfils its repsonsibility by delegating responsibility to other objects

- 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

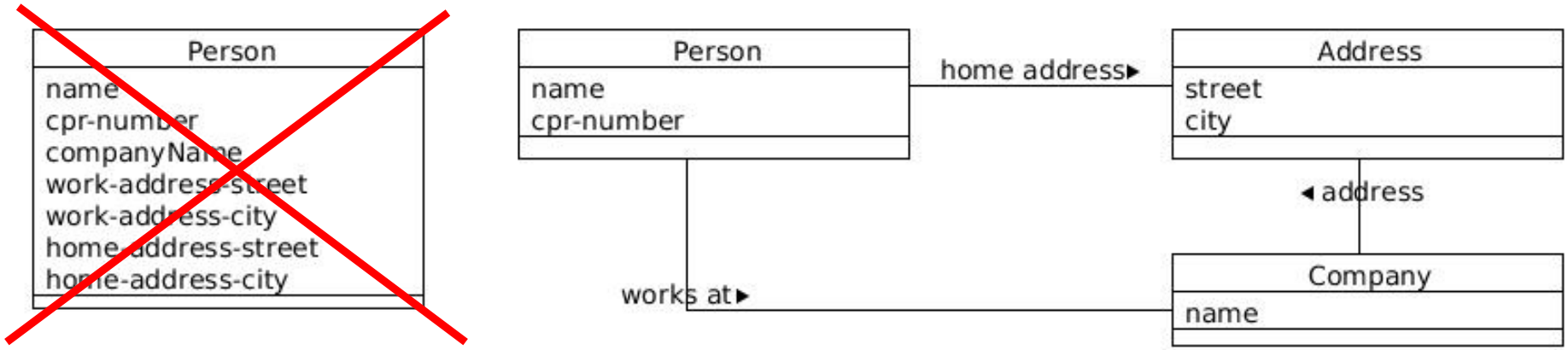
- Groups methods / attributes / classes with a common goal / functionality

→ 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

Example: High Cohesion



- Left side violates high cohesion:
  - Attributes for address details do not belong to the Person concept

Law of Demeter

Law of Demeter

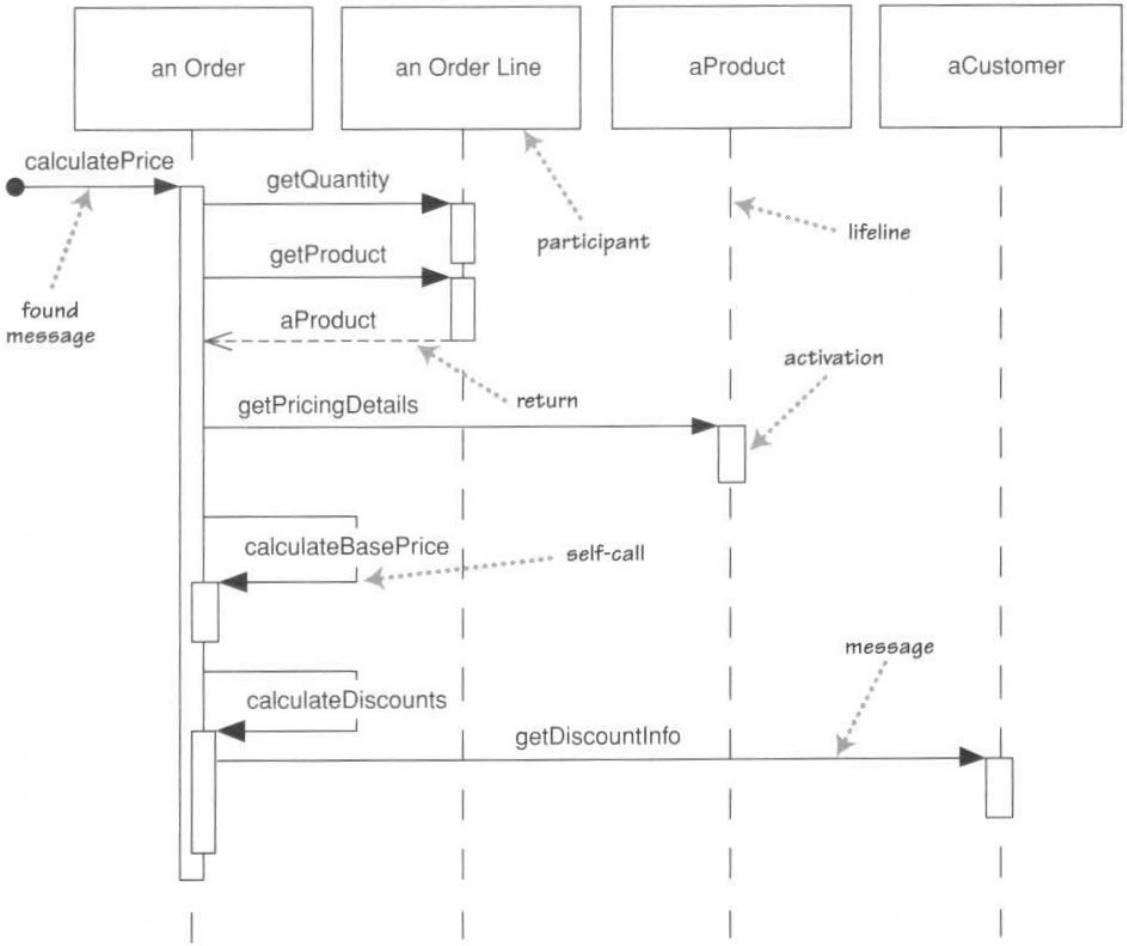
- ”Only talk to your immediate friends”
- Only method calls to the following objects are allowed
  - the object itself
  - its components
  - objects created by that object
  - parameters of methods

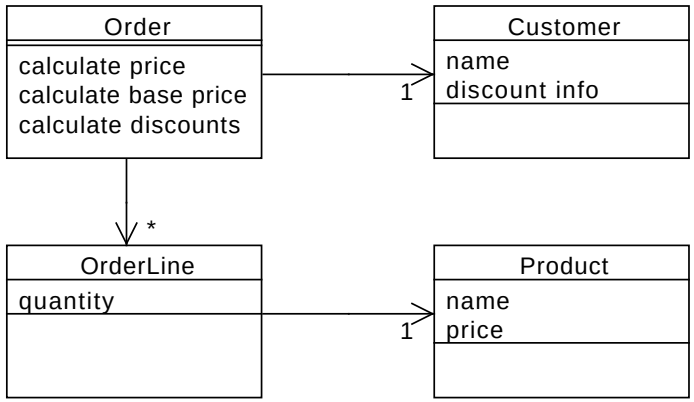
- The Law of Demeter is a special case of low coupling

→ To achieve low coupling one needs to delegate functionality

  → leads to decentralised control

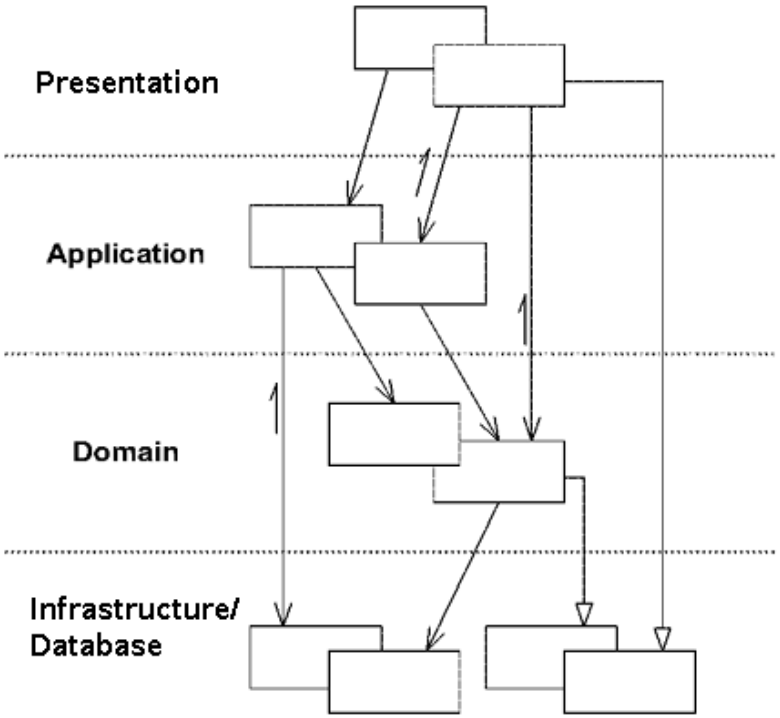
Computing the price of an order





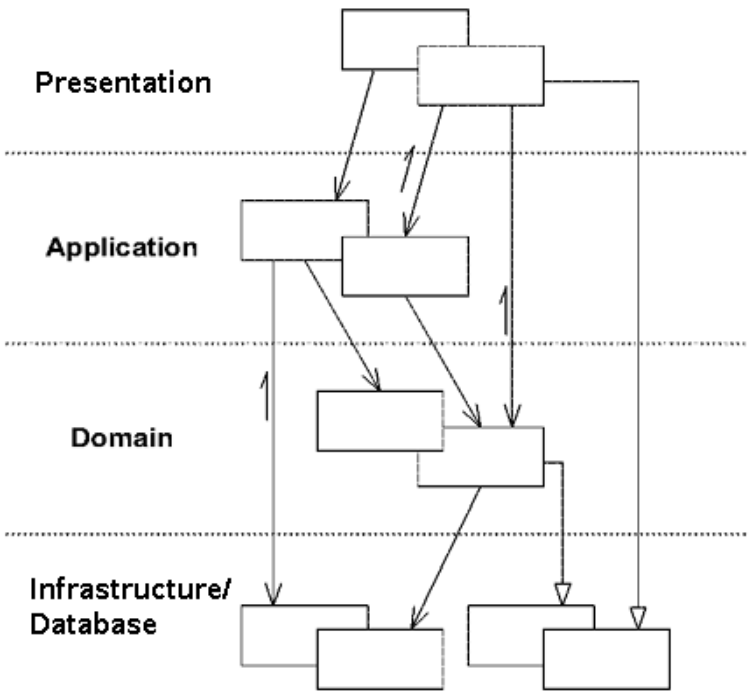
- Law of Demeter is violated because calculate price uses the product, but Order does not have access to the product via a field or as a parameter to a method, only through OrderLine
- Homework: Check if the distributed control example violates the Law of Demeter

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

Layered Architecture



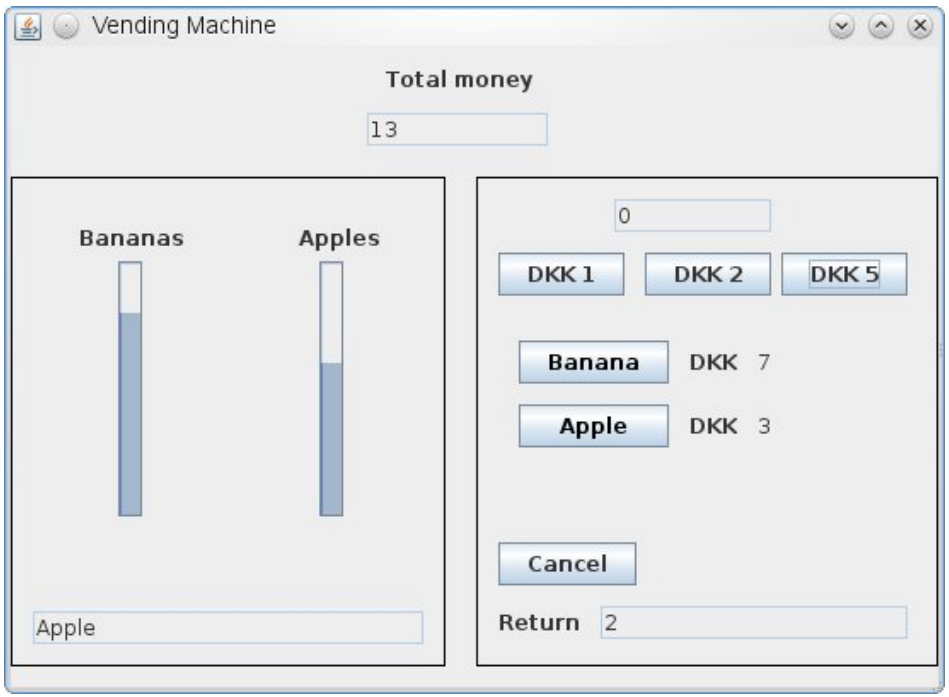
Eric Evans, Domain Driven Design, Addison-Wesley, 2004

- Presentation Layer
  - Translates user gestures to method calls and abstract data to textual/graphical representations; the layer is dumb, it doesn't do any business logic
- Application Layer
  - Contains application specific logic, could be more than one on the same domain model
- Domain Layer
  - Contains the domain specific logic (applicable to all applications)
- Database Layer
  - Contains the storage logic for domain objects

Example Vending Machine

Two different presentation layers

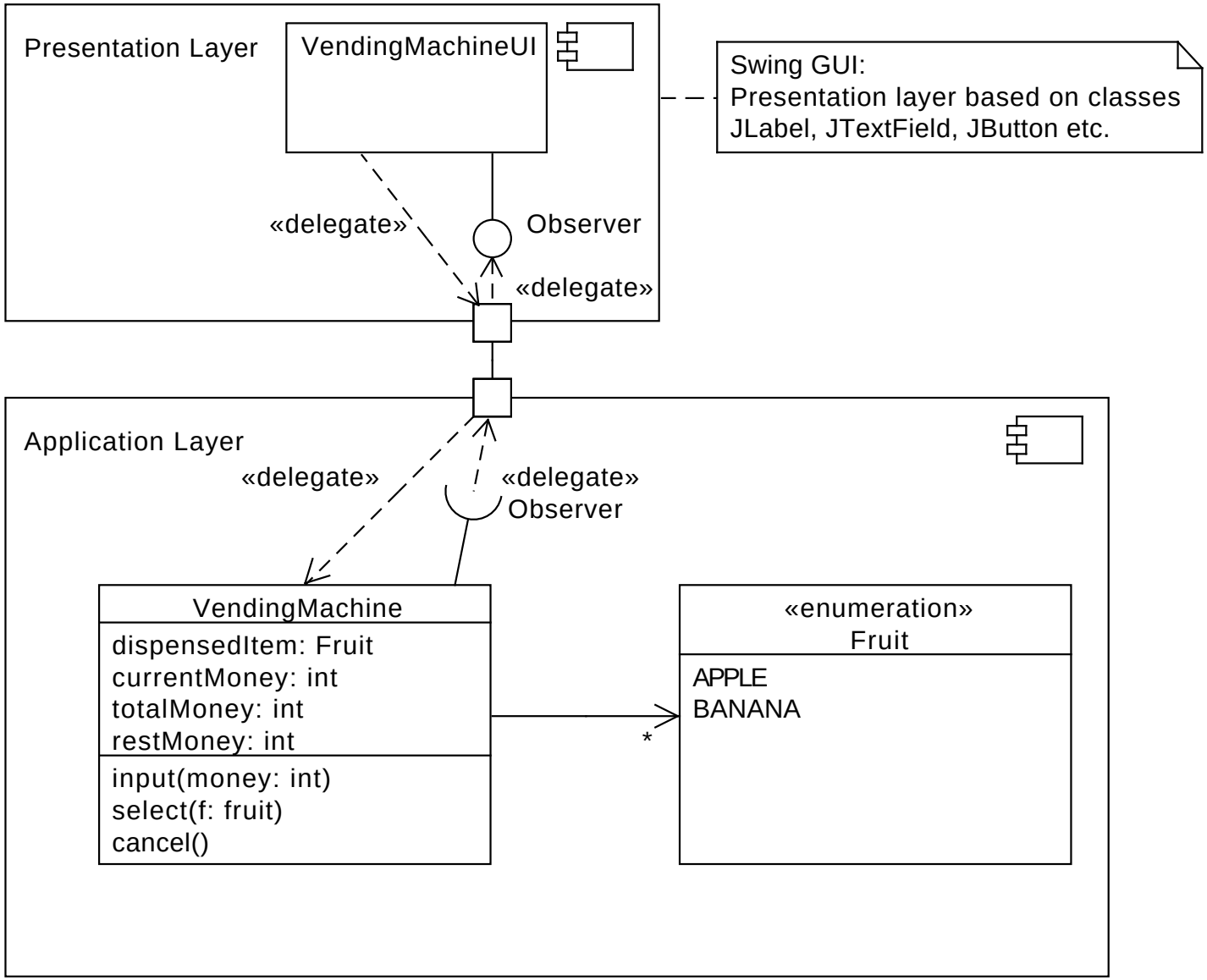
- Swing GUI



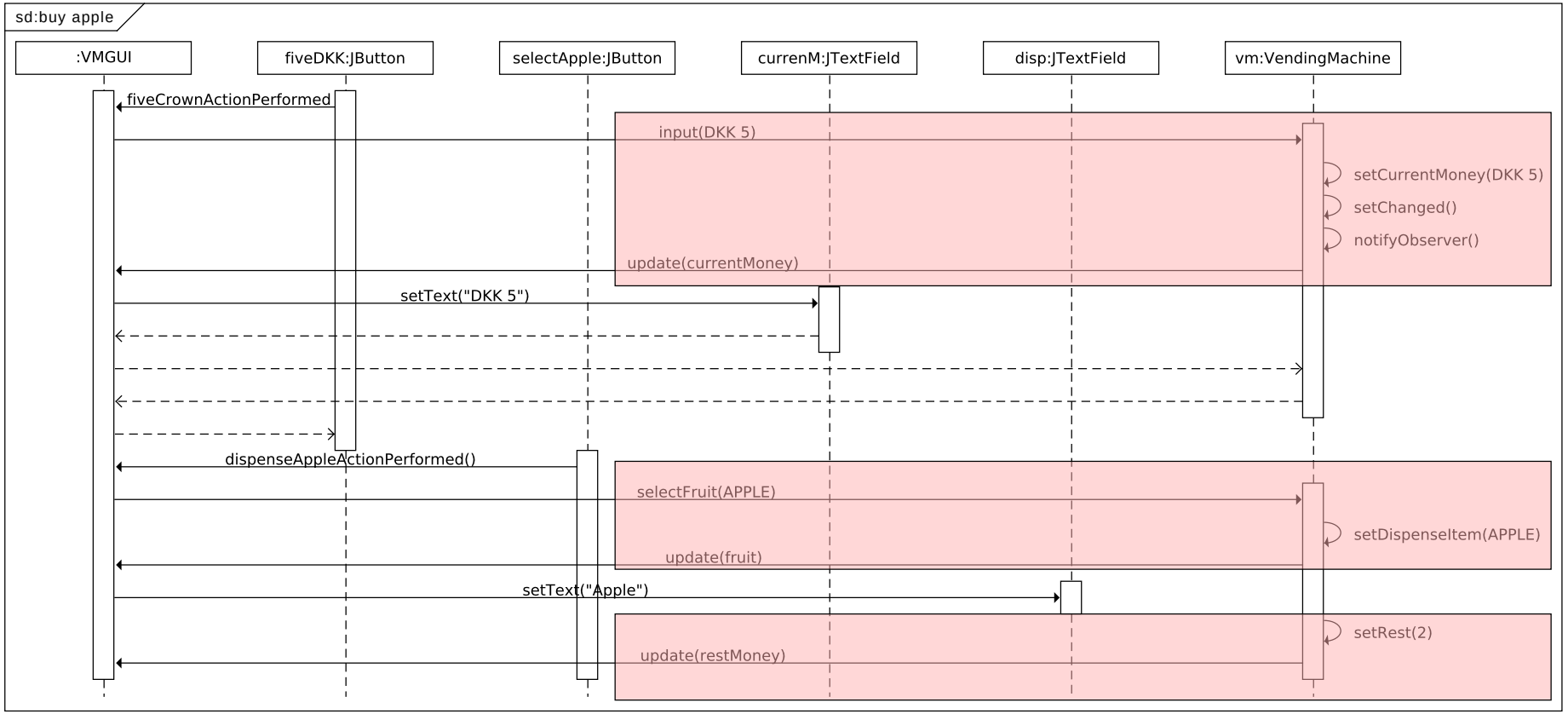
- Command line interface

```
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
```

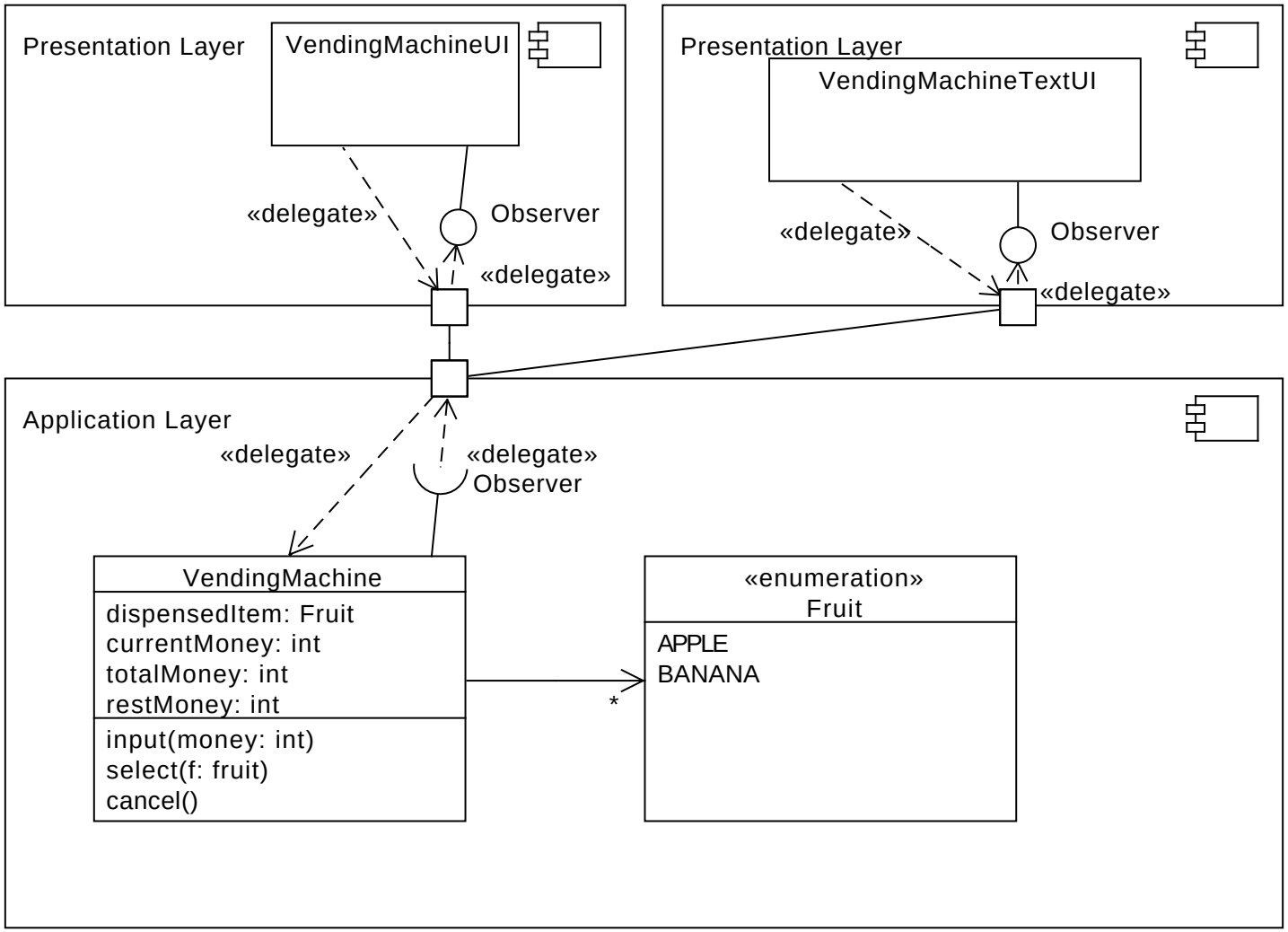
Architecture I



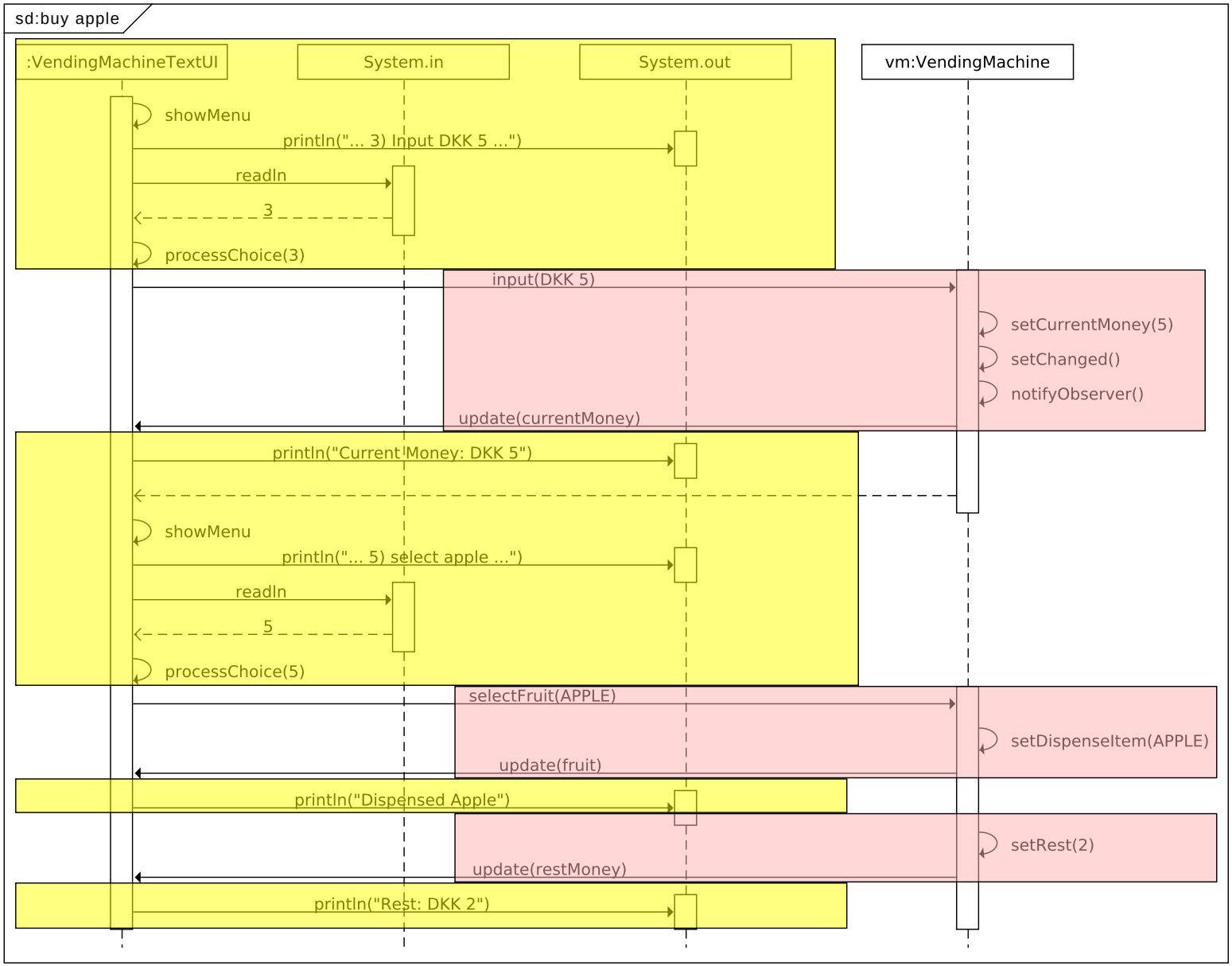
Presentation Layer: Swing GUI



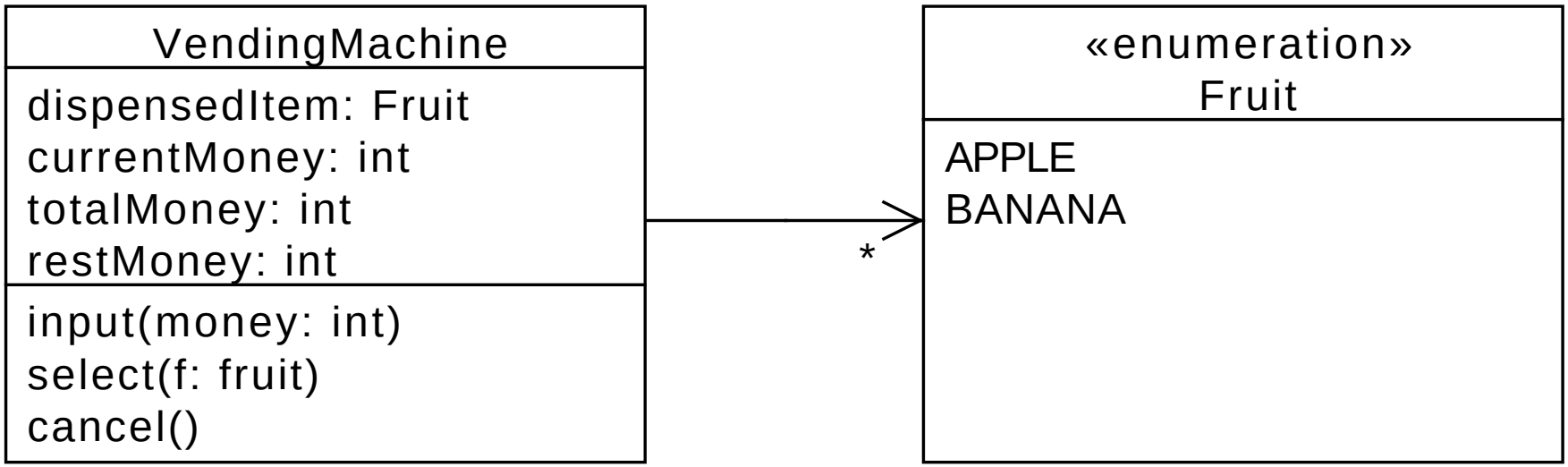
Architecture II



Presentation Layer: Command Line Interface

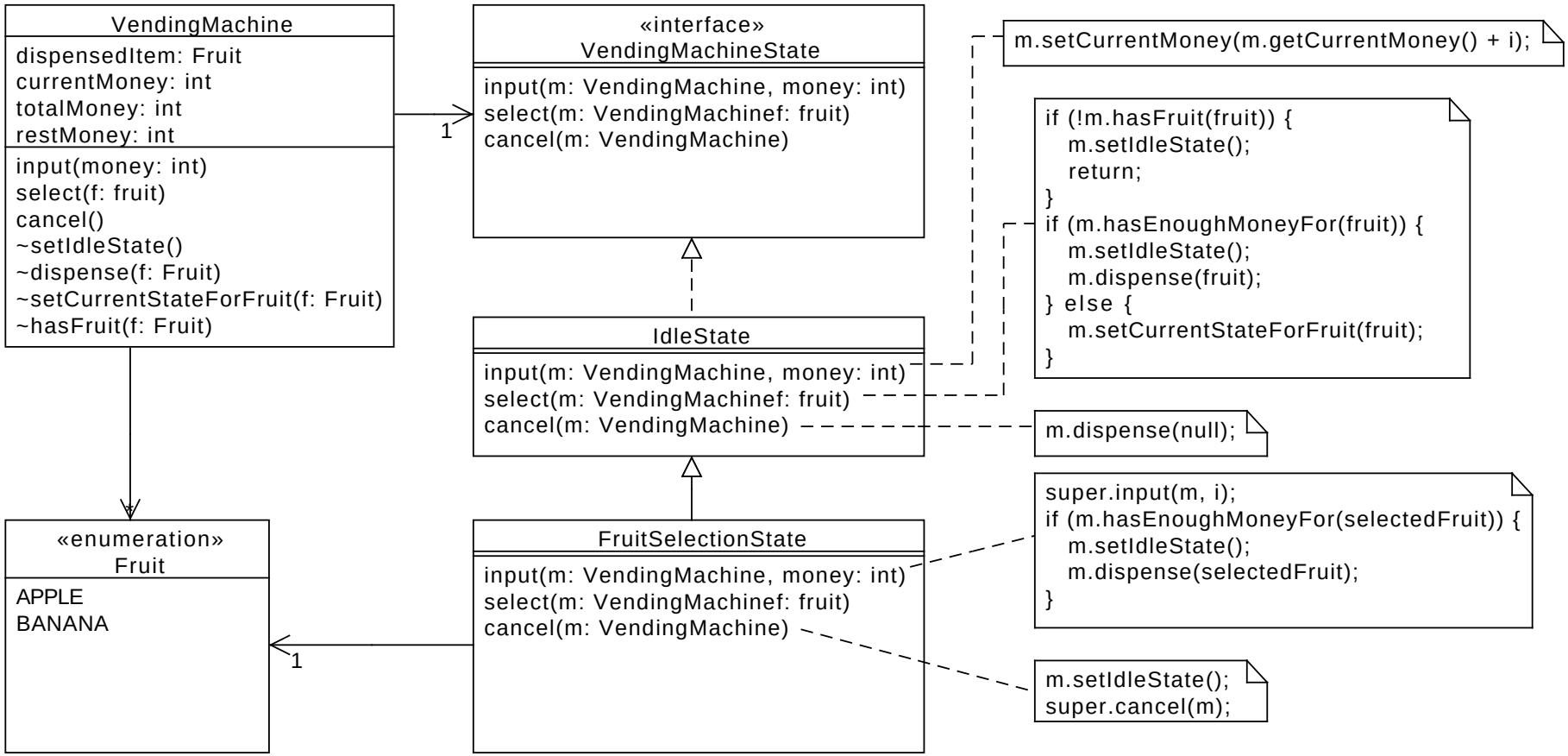


Application Layer



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
    - An ”action performed” method does not do any business relevant 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

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());
}
```

Summary

- 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

Software Engineering I (02161)

Week 6

Assoc. Prof. Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2011