Software Engineering I (02161)

Week 6

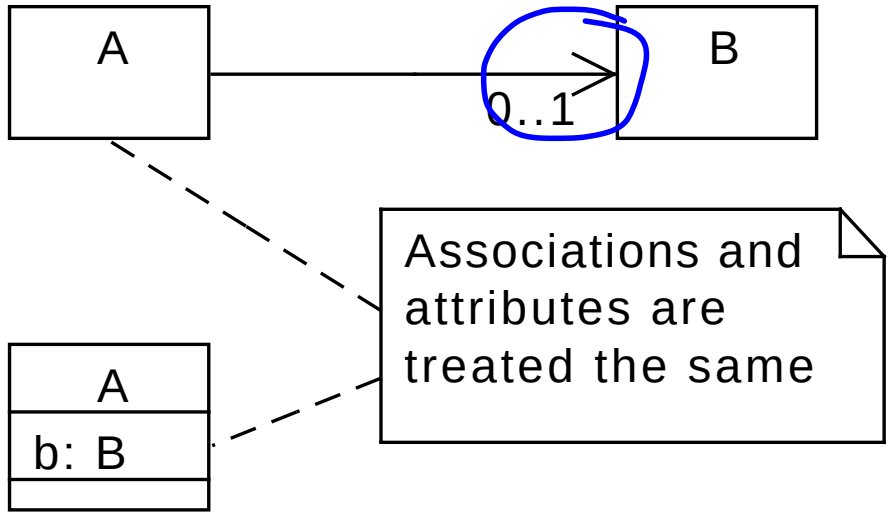
Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2013

Contents

Implementing Associations: Cardinality 0..1

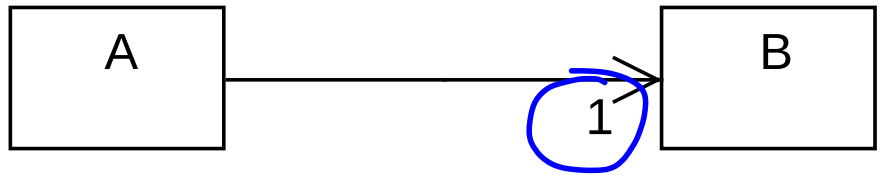


- Field can be null

```
public class A {

   private B b;

   public B getB() {
      return b;
   }

   public void setB(B b) { this.b = b; }
}
```

role name

Implementing Associations: Cardinality 1

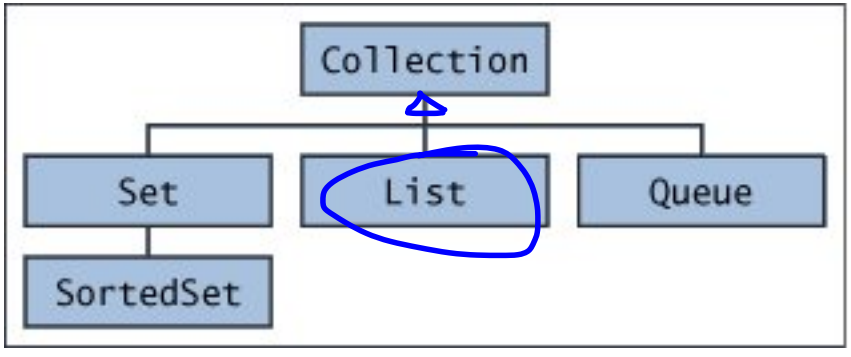


- Field may not be null

```
public class A {

   private B b = new B();

   public A(B b) { this.b = b;}

   public B getB() {
     if (b == null) {b = computeB();}
     return b;
   }

   public void setB(B b) { if (b != null) {this.b = b;} }
}
```

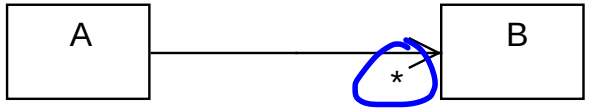
Interface *Collection<E>*

Operation	Description
`boolean add(E e)`	returns **false** if e is in the collection
`boolean remove(E e)`	returns **true** if e is in the collection
`boolean contains(E e)`	returns **true** if e is in the collection
`Iterator<E> iterator()`	allows to iterate over the collection
`int size()`	number of elements




```
for (Book book: books) { .. }
```

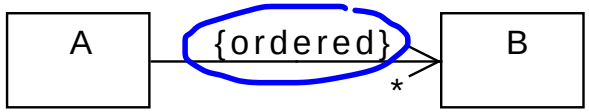
Implementing Associations: Cardinality *



Default: Unordered, no duplicates

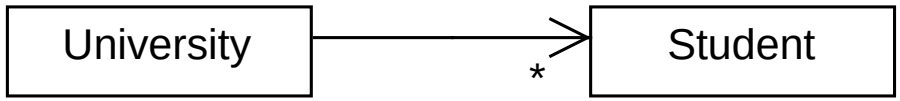
```
public class A {
   private Set<B> bs = new HashSet<B>();
   ...
}
```

Interface



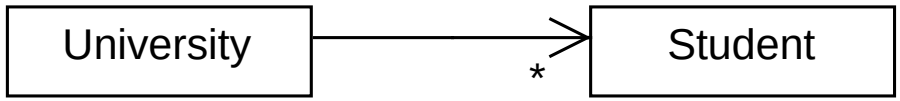

```
public class A {
   private List<B> bs = new ArrayList<B>();
   ...
}
```

Encapsulation problem: getStudents




```
University dtu = new University("DTU");
..
Student hans = new Student("Hans");
Set<Student> students = dtu.getStudents();
```

Encapsulation problem: getStudents

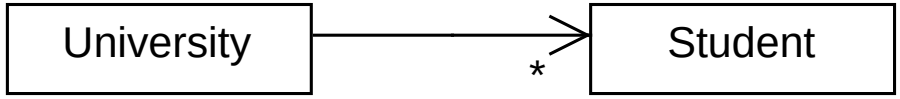



```
University dtu = new University("DTU");
..
Student hans = new Student("Hans");
Set<Student> students = dtu.getStudents();


Student hans = new Student("Hans");
students.add(hans);
students.remove(ole);
...
```

Solution: getStudents returns an unmodifiable set

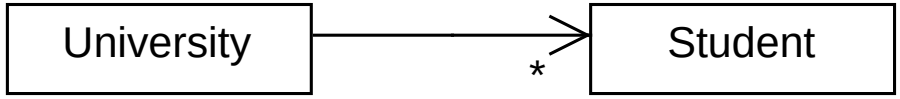
```
public void Set<Student> getStudents() {
   students = Collections.unmodifiableSet();
}
```

Encapsulation problem: setStudents




```
University dtu = new University("DTU");
..
Set<Student> students = new HashSet<Student>();
dtu.setStudents(students);
```

Encapsulation problem: setStudents

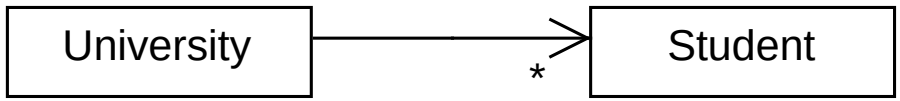



```
University dtu = new University("DTU");
..
Set<Student> students = new HashSet<Student>();
dtu.setStudents(students);


Student hans = new Student("Hans");
students.add(hans);
...
```

Solution: setStudents copies the set

```
   public void setStudents(Set<Student> stds) {
      students = new HashSet<Student>(stds);
}
```

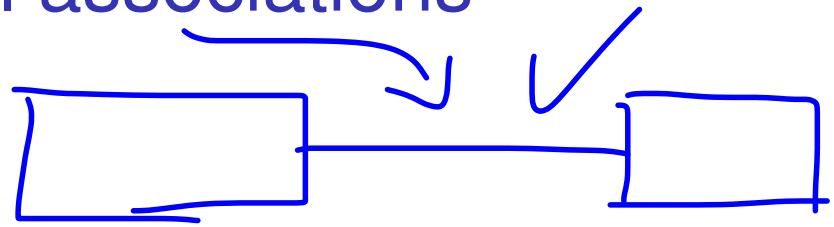
Solution: How to change the association?

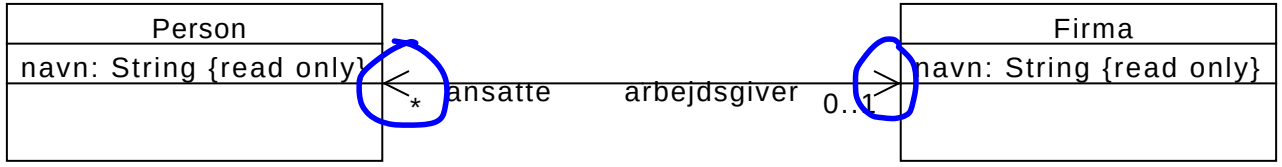



```
public class University {

   private Set<Student> bs = new HashSet<Student>();

   public void addStudent(Student s) {students.add(student);}
   public void containsStudent(Student s) {return students.contains(s)
   public void removeStudent(Student s) {students.remove(s);}
}
```

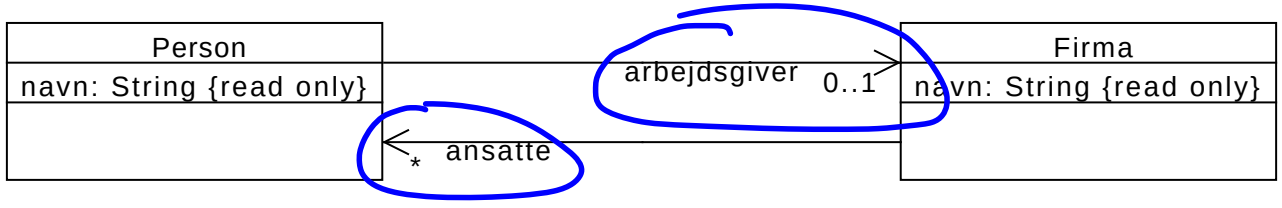
Even better: domain specific methods like *registerStudent*

Bi-directional associations





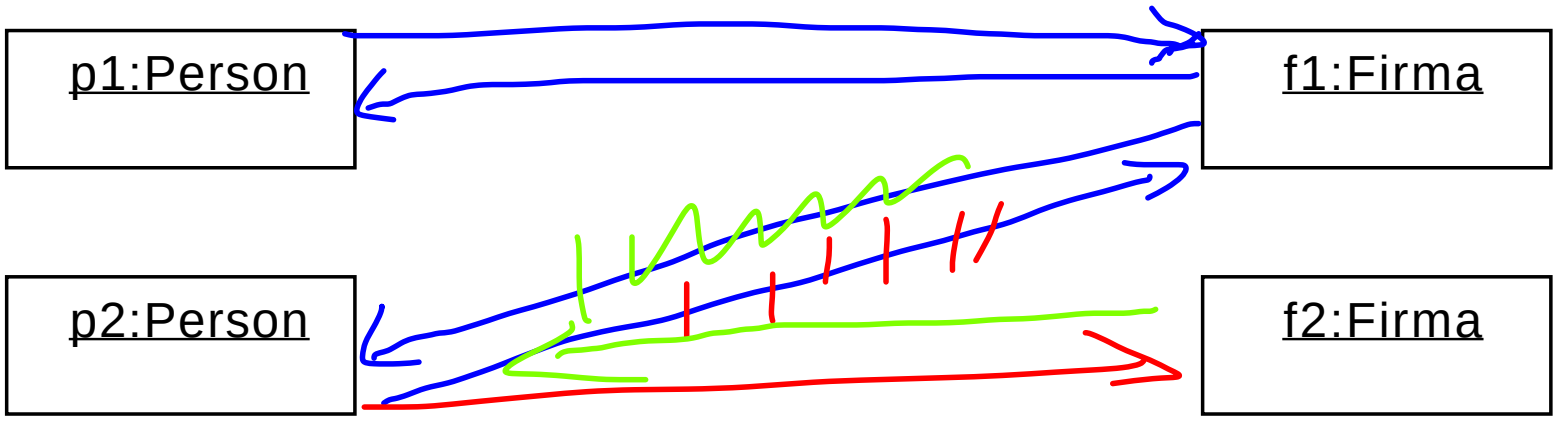
Implemented as two *uni-directional* associations



2 uni directional

→ Problem of referential integrity

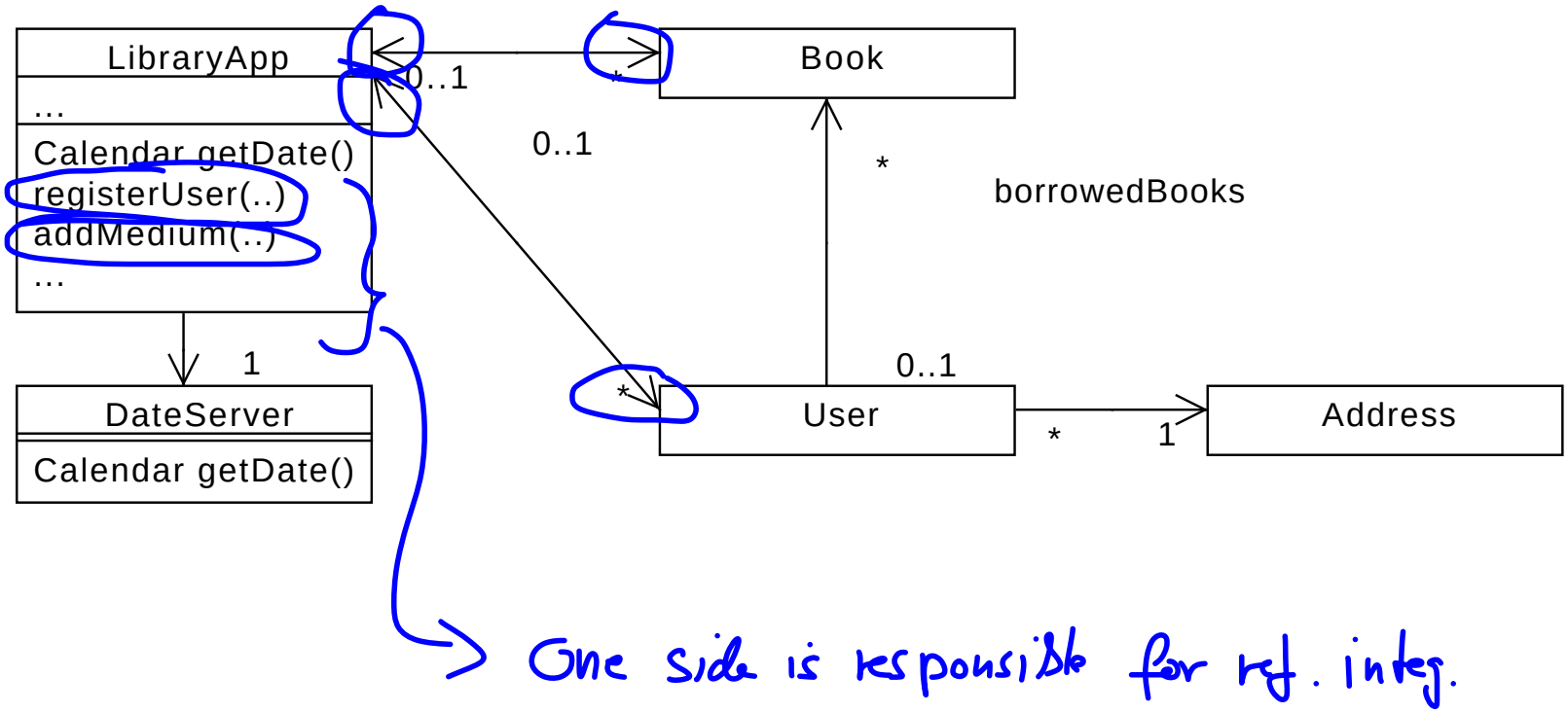
Referential Integrity



Referential Integrity: setArbejdsgiver

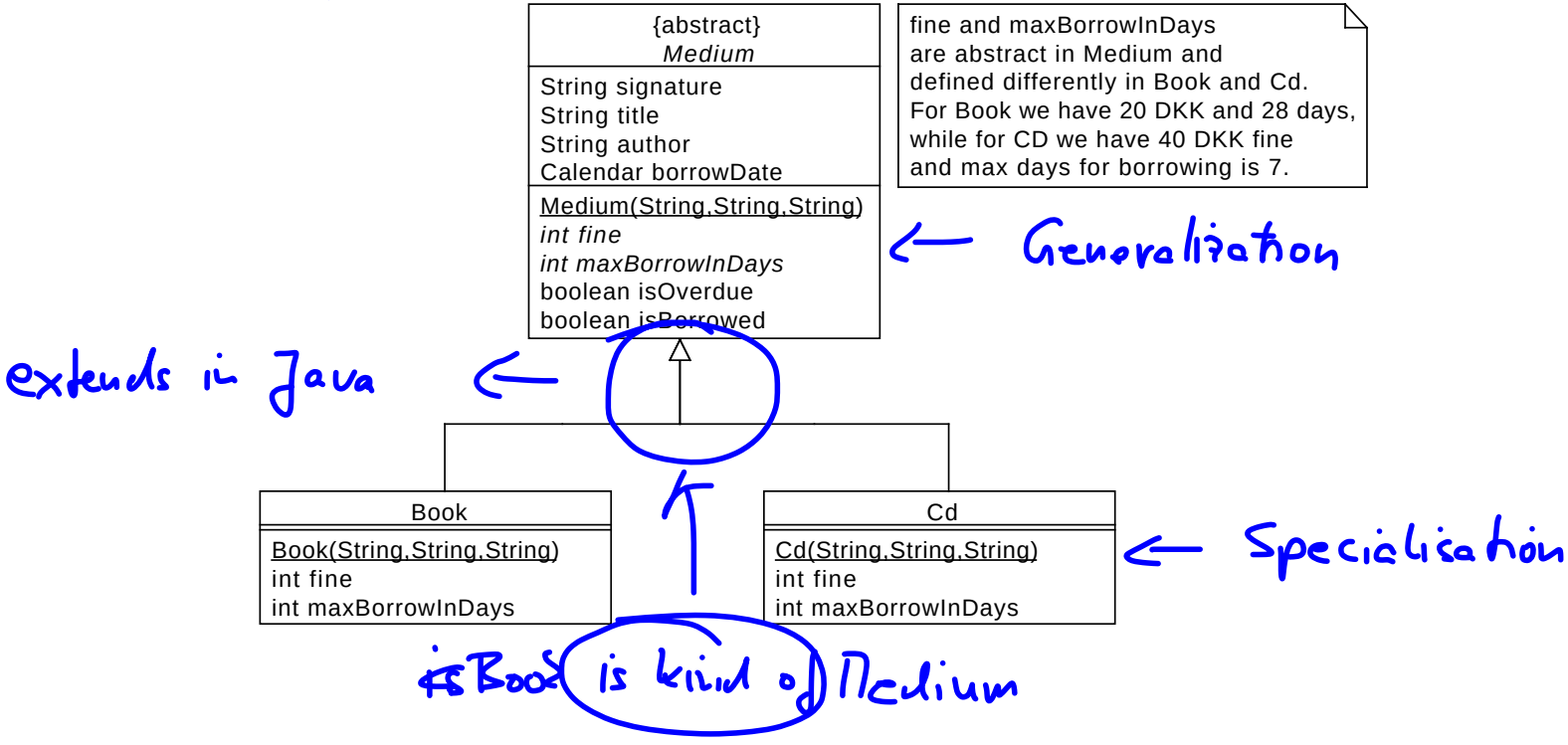
Referential Integrity: addAnsatte

Library application



Generalisation Example

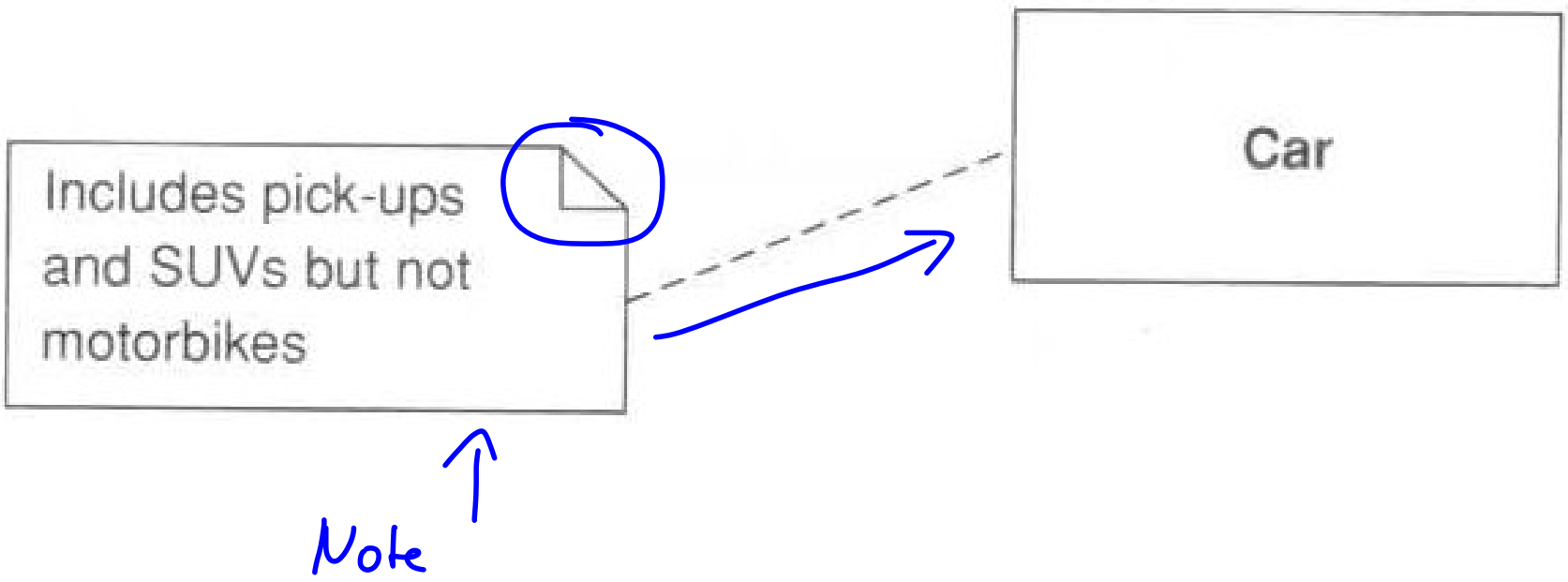
aka Inheritance



Liskov-Wing Substitution Principle

> *"If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness)."*

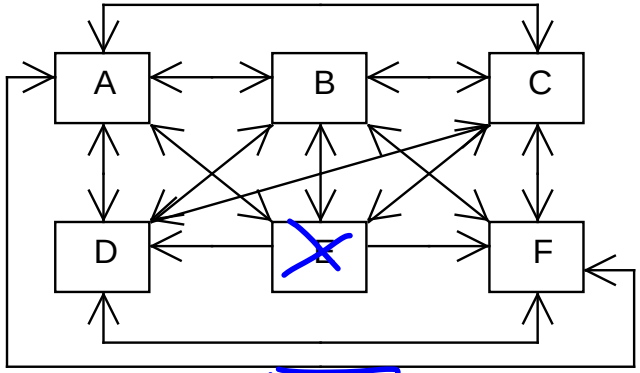
Notes and Comments



Contents

Low Coupling

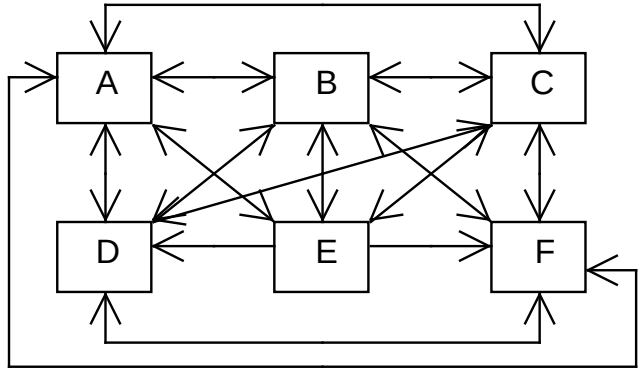
High coupling



n^2 connections

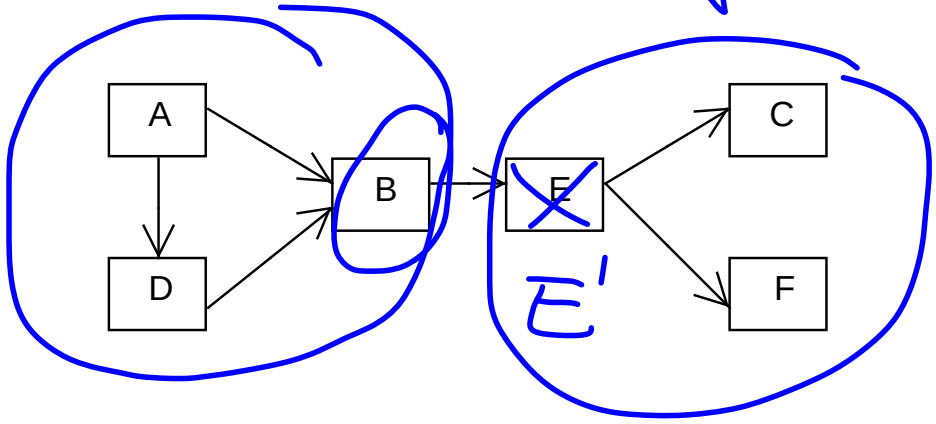
Low Coupling

High coupling



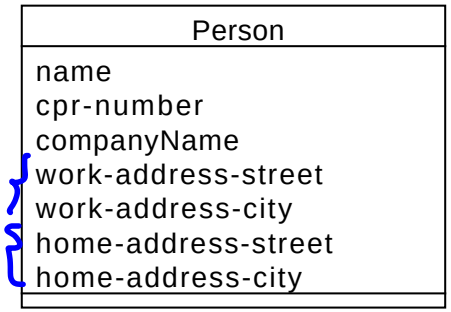
Low coupling

cluster of classes



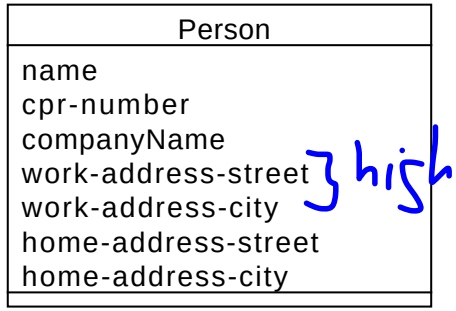
High Cohesion

Low Cohesion

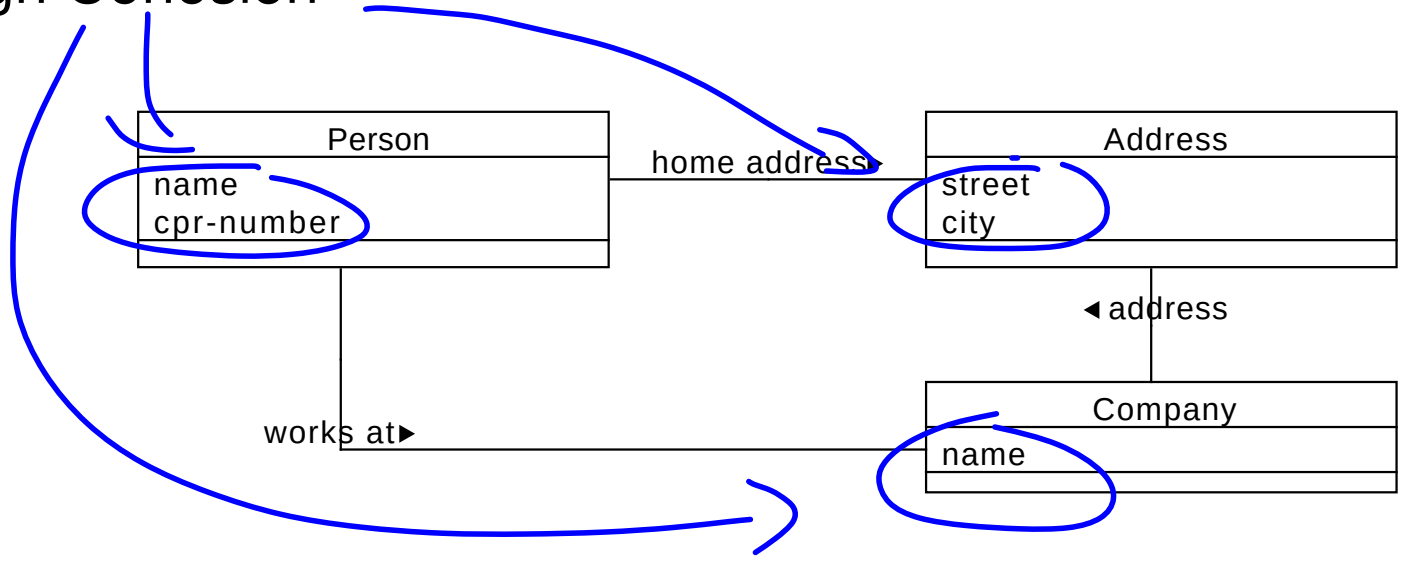


High Cohesion

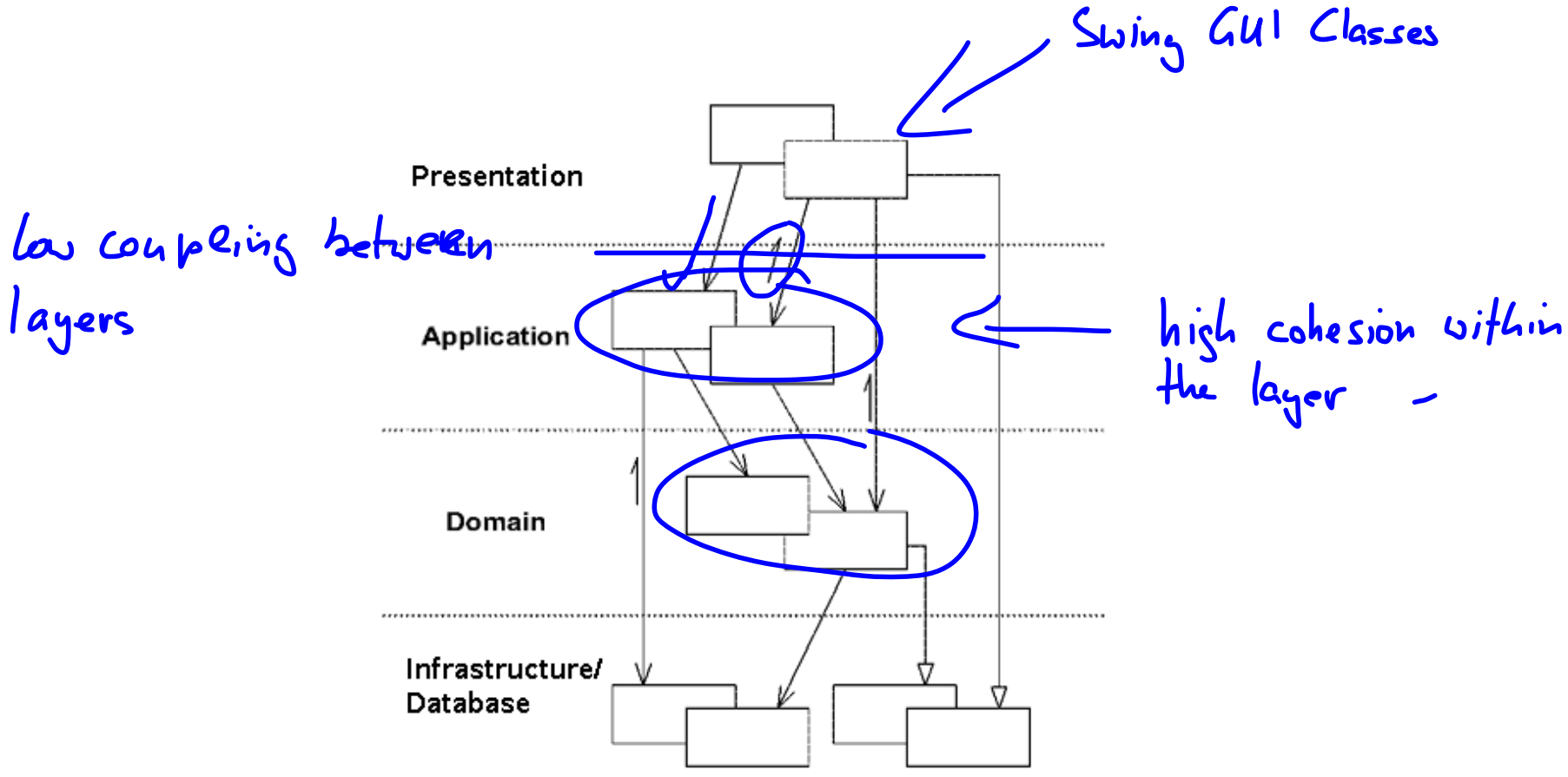
Low Cohesion



High Cohesion



Layered Architecture

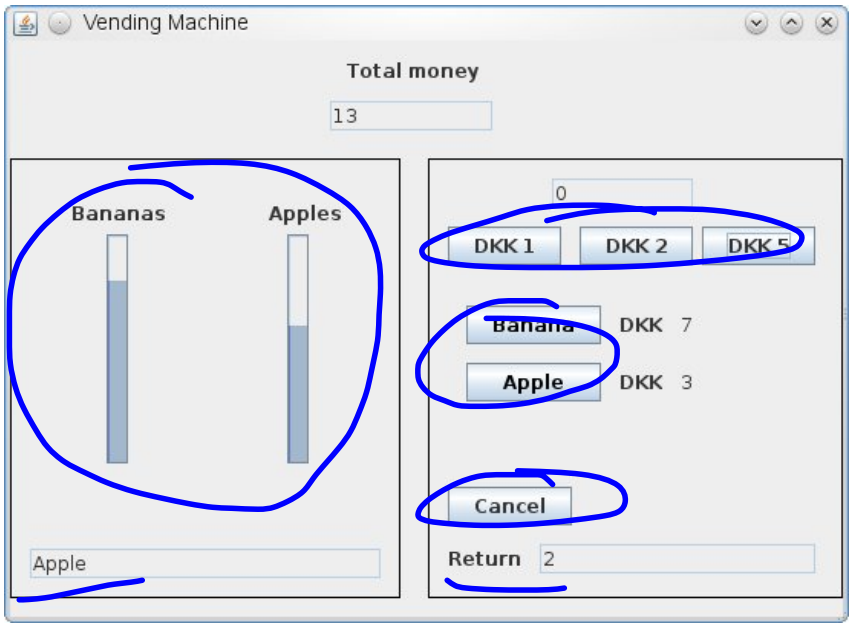


Eric Evans, Domain Driven Design, Addison-Wesley, 2004

Example Vending Machine

Two different presentation layers; same application layer

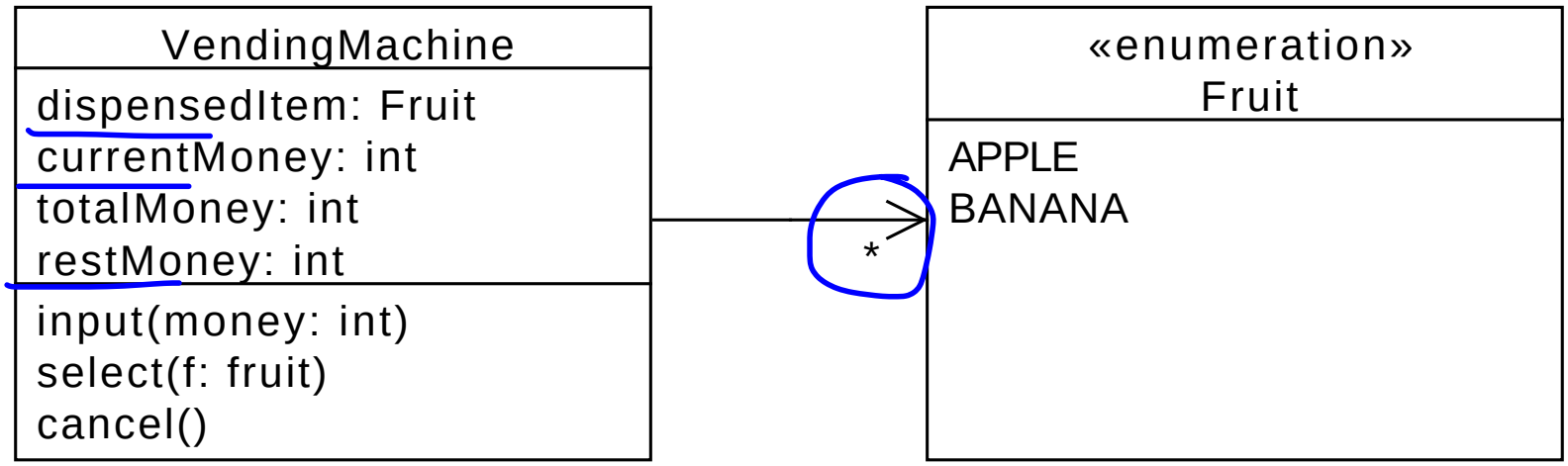
- Swing GUI



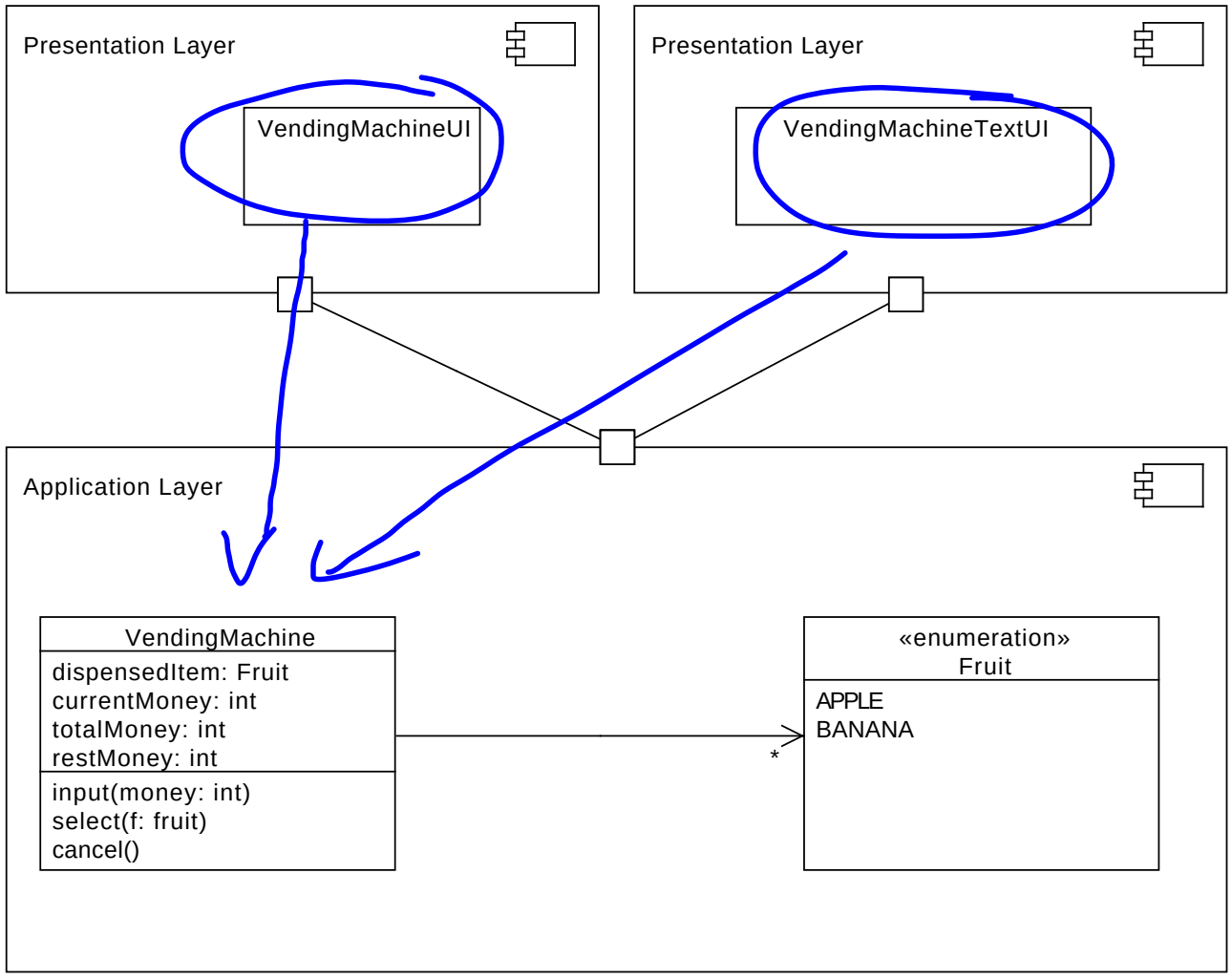
- Command line interface

```
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):
Rest: DKK 2
Current Money: DKK 0
Dispensing: Apple
```

Application Layer



Architecture



Presentation Layer: Swing GUI

```
public class VendingMachineUI extends javax.swing.JFrame
      implements java.util.Observer {
   private VendingMachine vendingMachine = new VendingMachine(10, 10);
   ...
   private javax.swing.JButton fiveCrowns = new javax.swing.JButton();
   ...

   private void initComponents() {
      fiveCrowns.setText("DKK 5");
      fiveCrowns.addActionListener(new java.awt.event.ActionListener()
            public void actionPerformed(java.awt.event.ActionEvent evt
               vendingMachine.input(5);
            }
         }
      )
      ...
   };
   ...
   public void update(Observable o, Object arg) {
      displayState();
   }
}
```

→ Observer Pattern

Presentation Layer: TextUI

```
public class VendingMachineTextUI implements Observer {
   VendingMachine vendingMachine;
   public static void main(String[] args) throws Exception {
      new VendingMachineTextUI(5,5).mainLoop(System.in, System.out);
   }
   public void mainLoop(InputStream in, PrintStream out) throws IOExce
      BufferedReader rs = new BufferedReader(new InputStreamReader(in)
      do {
         showMenu(out);
         int number = Integer.valueOf(rs.readLine());
         processChoice(number, out);
      } while (number != 0);
   }
   private void processChoice(int number, PrintStream out) {
      switch (number) {
        case 3: vendingMachine.input(5); break;
         ...
      }
   }
   public void update(Observable o, Object arg) {
     NotificationType type = (NotificationType) arg;
     if (type == NotificationType.CURRENT_MONEY) {
         System.out.println("Current Money: DKK " +
            vendingMachine.getCurrentMoney());
         return;
      }
   }
}
```

→ application logic

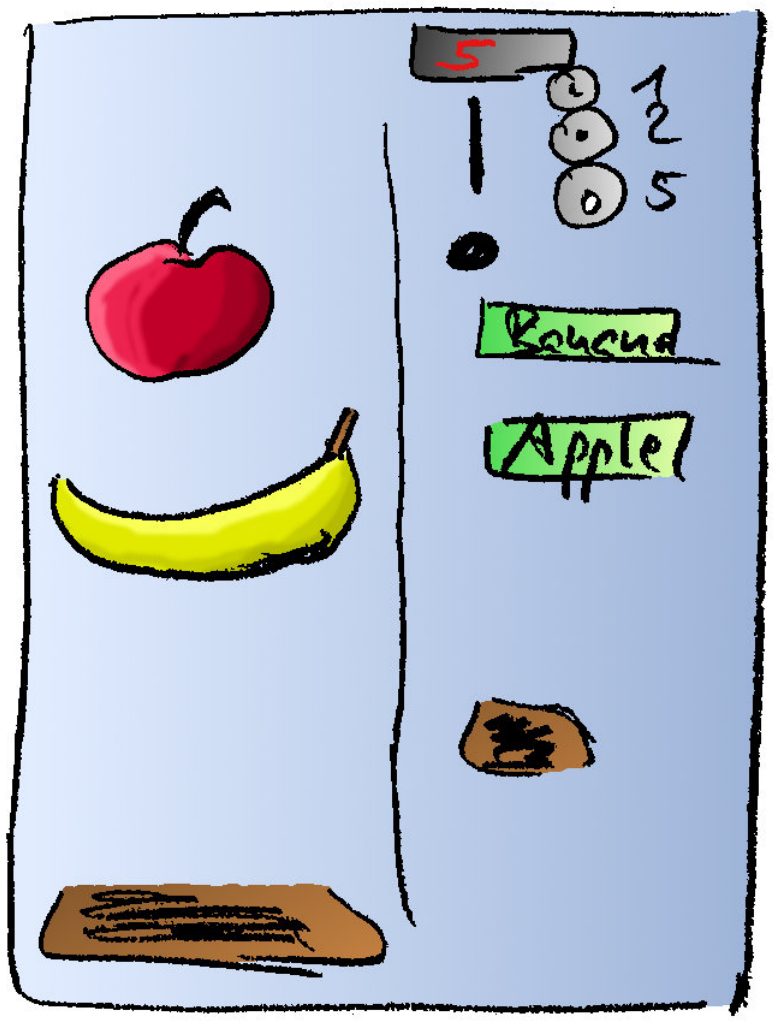
Advantages of the separation

1. Presentation layer easily changed
2. Additional presentation layers
3. Automatic 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());
}
```

Contents

Example Vending Machine



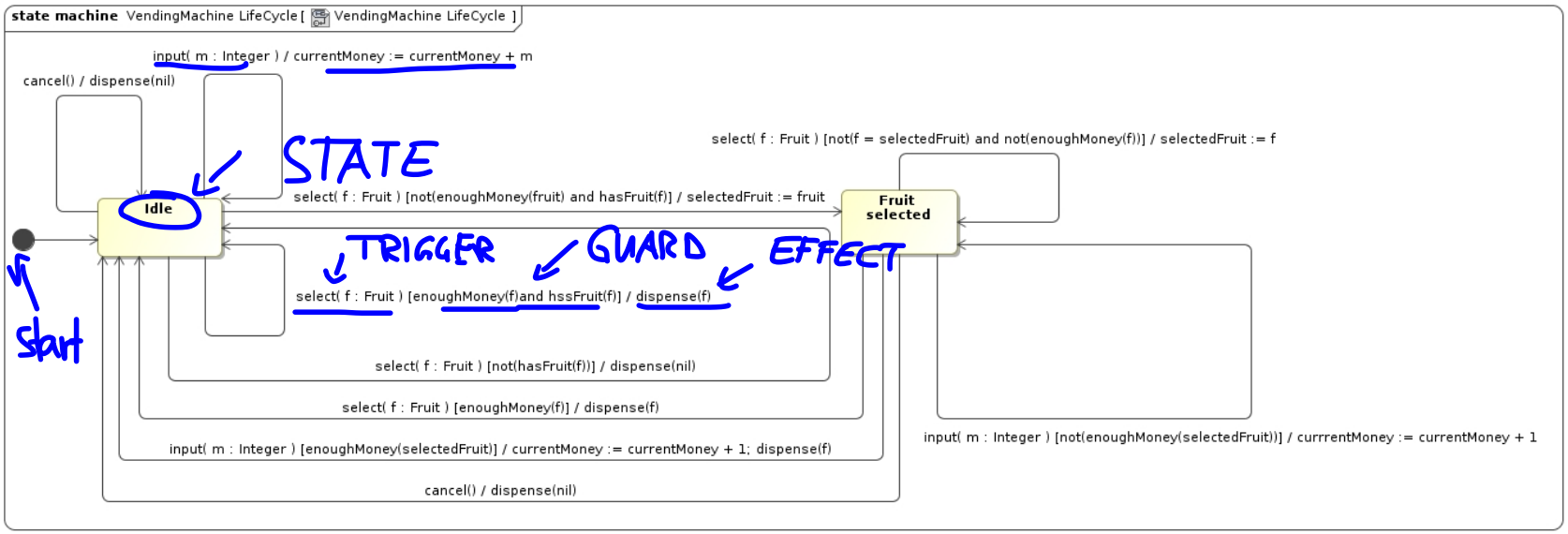
- Actions
  - Input coins
  - Press button for bananas or apples
  - Press cancel
- Displays
  - current amount of money input
- Effects
  - Return money
  - Dispense banana or apple

Vending Machine: state machine

State transition table

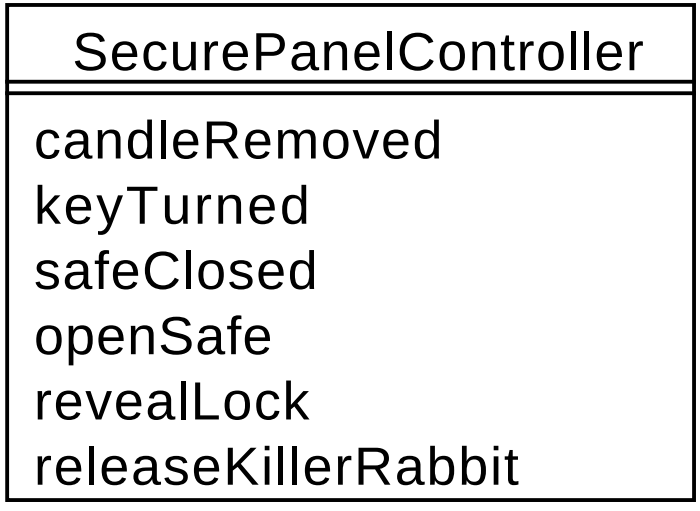
event	guard	state	state	state
		Idle (I)	*Banana selected and not enough money (B)*	*Apple selected and not enough money (A)*
banana	*enough money for banana*	dispense banana and rest money	dispense banana and rest money-> I	dispense banana and rest money-> I
banana	*not enough money for banana*	-> B		-> B
banana	*no bananas available*		-> 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	
apple	*no apples available*		-> 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	return current money -> I	return current money -> I

UML State Machines

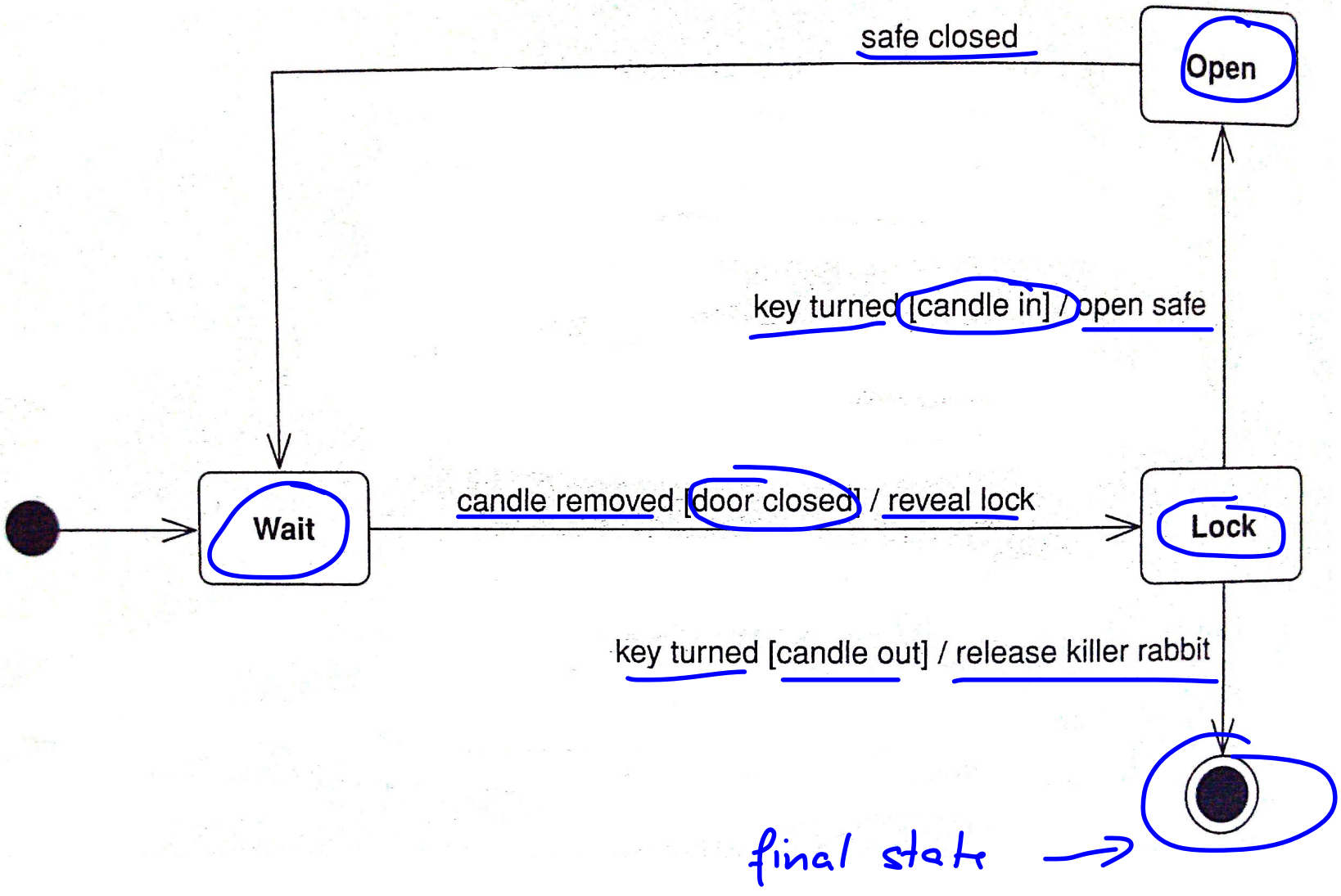


Example: Safe

- Task: Implement a control panel for a safe in a dungeon
- The lock 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

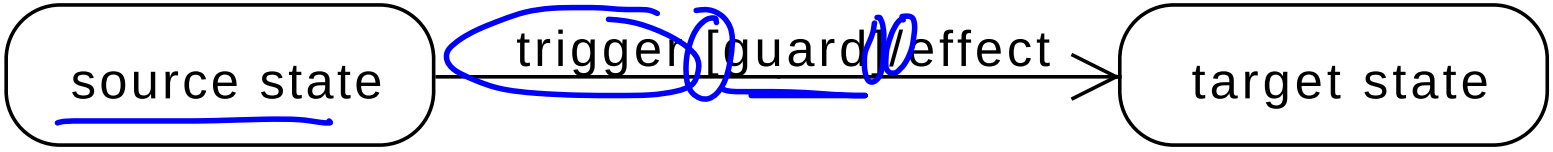


Example: Safe



Transitions

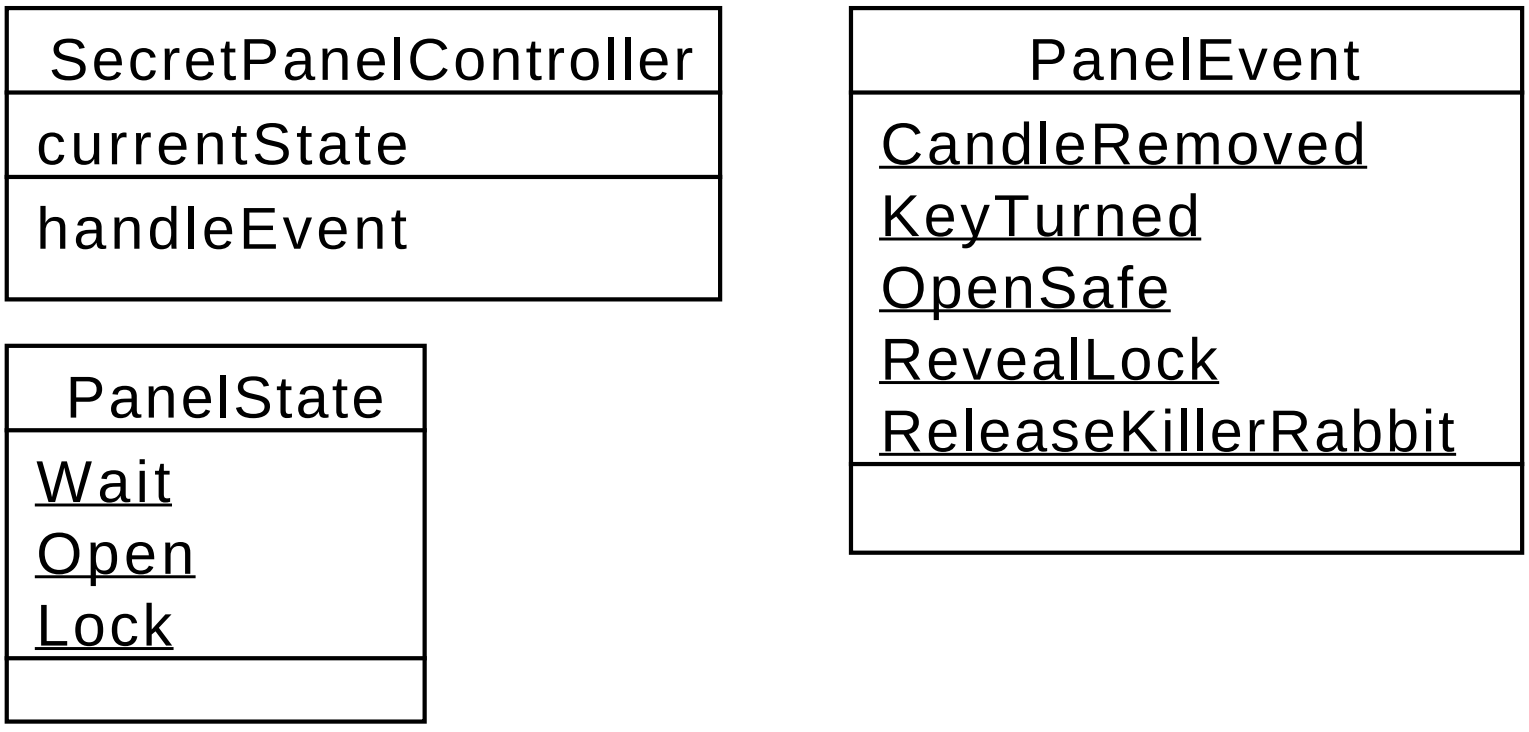
- General form



- Triggers (events, method calls)
- Guard: boolean expression
- Effect: a statement
- Fireing a transition
  - trigger + guard is true then the effect is executed

System is in source state

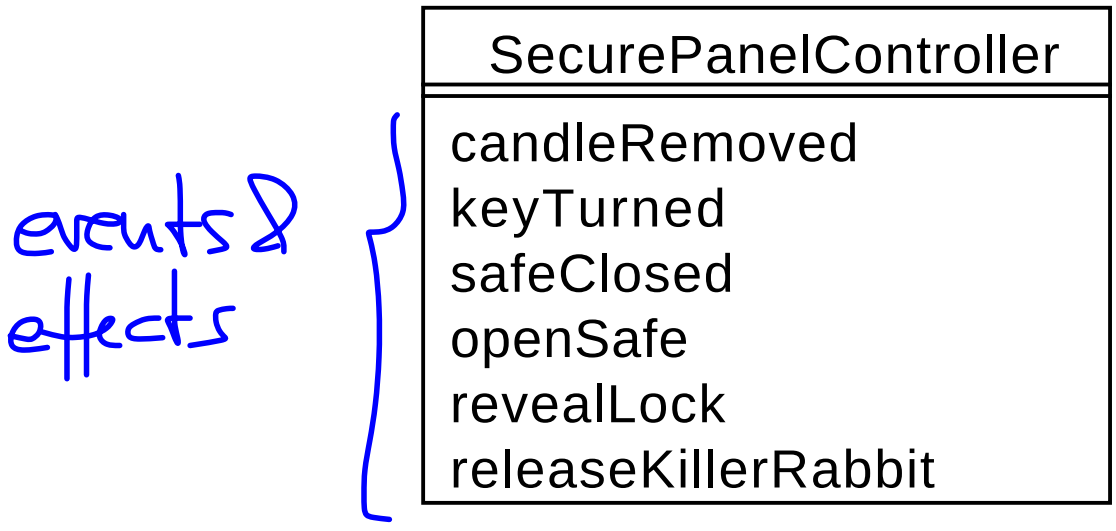
Implementation 1: Class diagram



Implementation 1

```
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: Class diagram

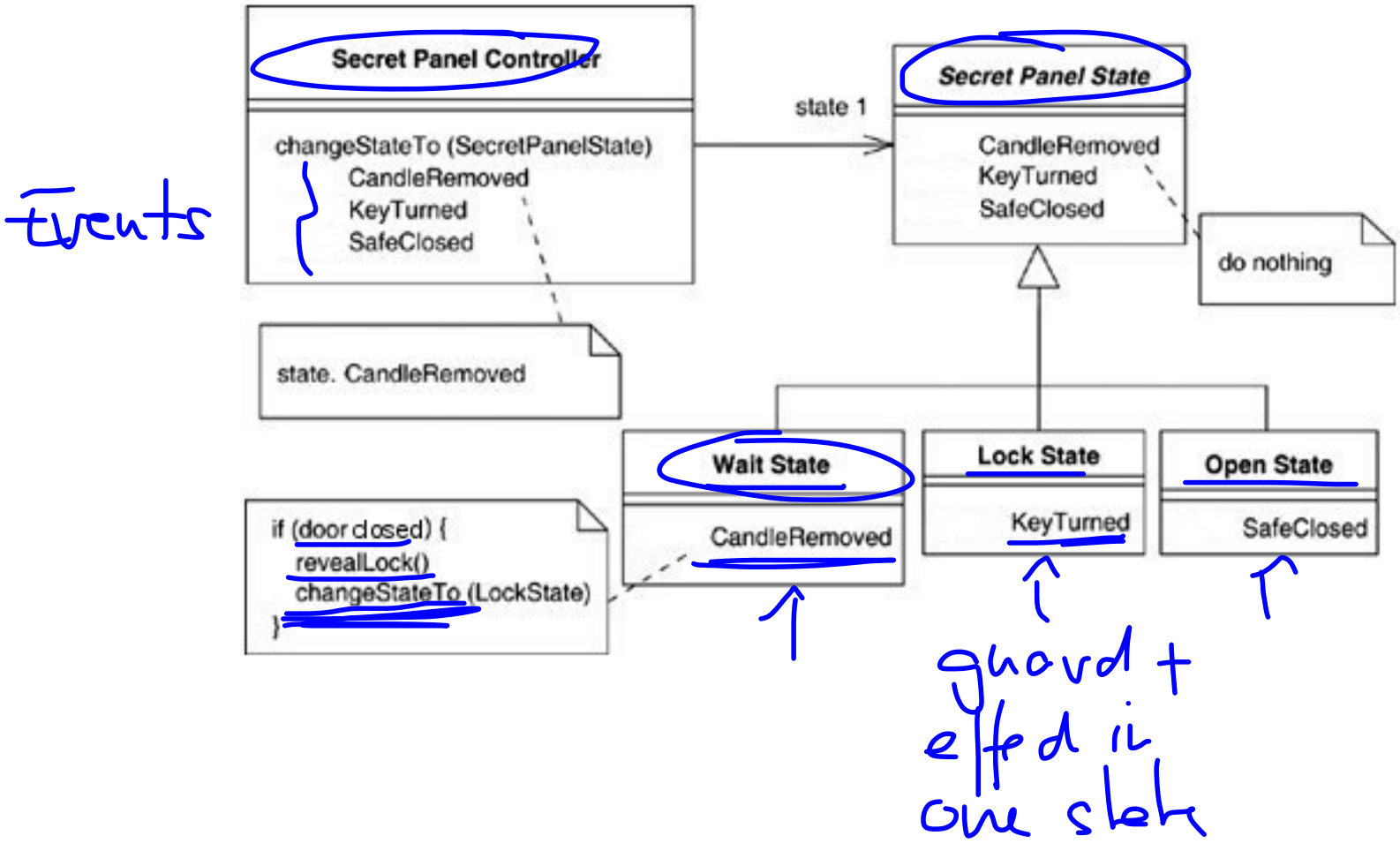


Implementation 2

```
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 3: Using the state pattern

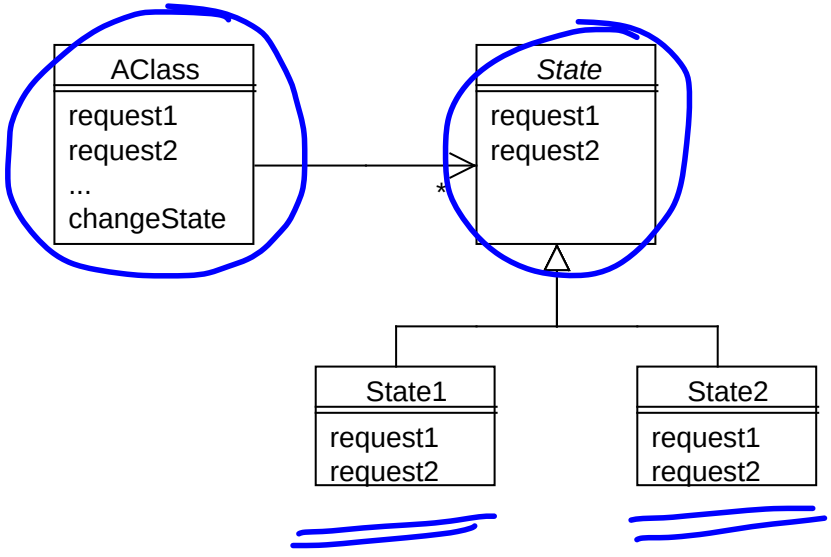
↳ Design Pattern

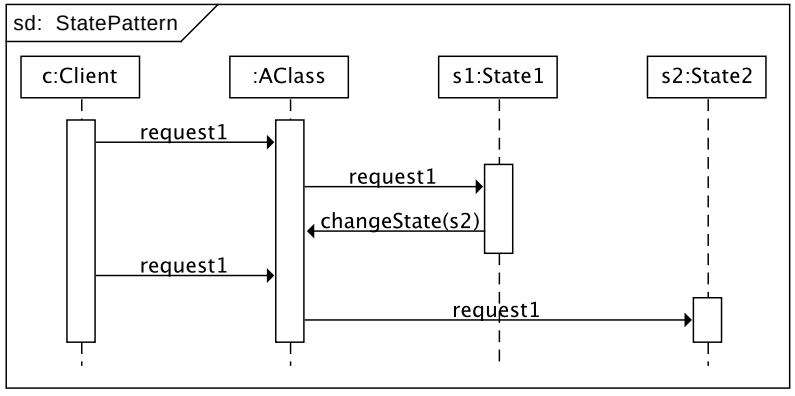


State Pattern

State Pattern

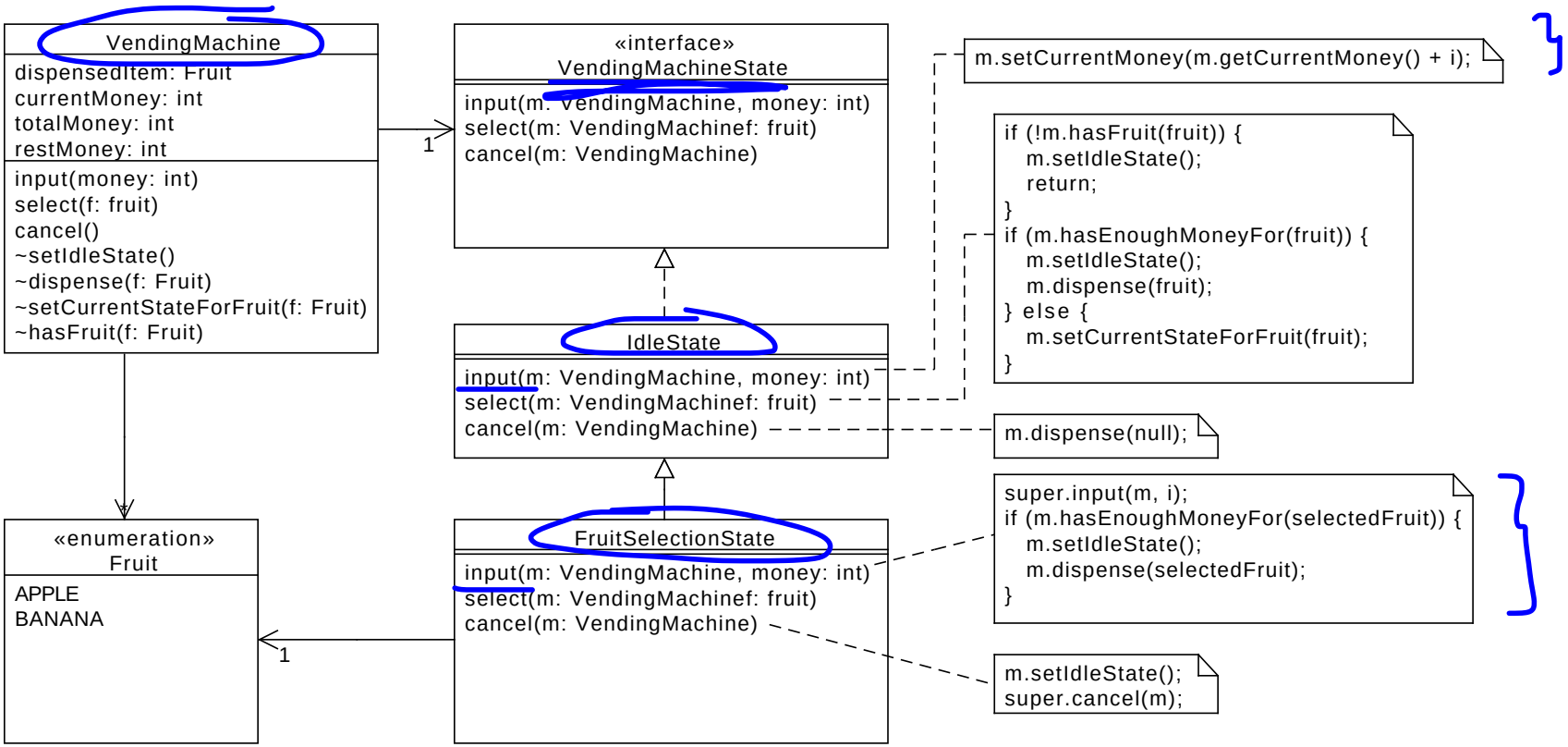
”Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.” Design Pattern book





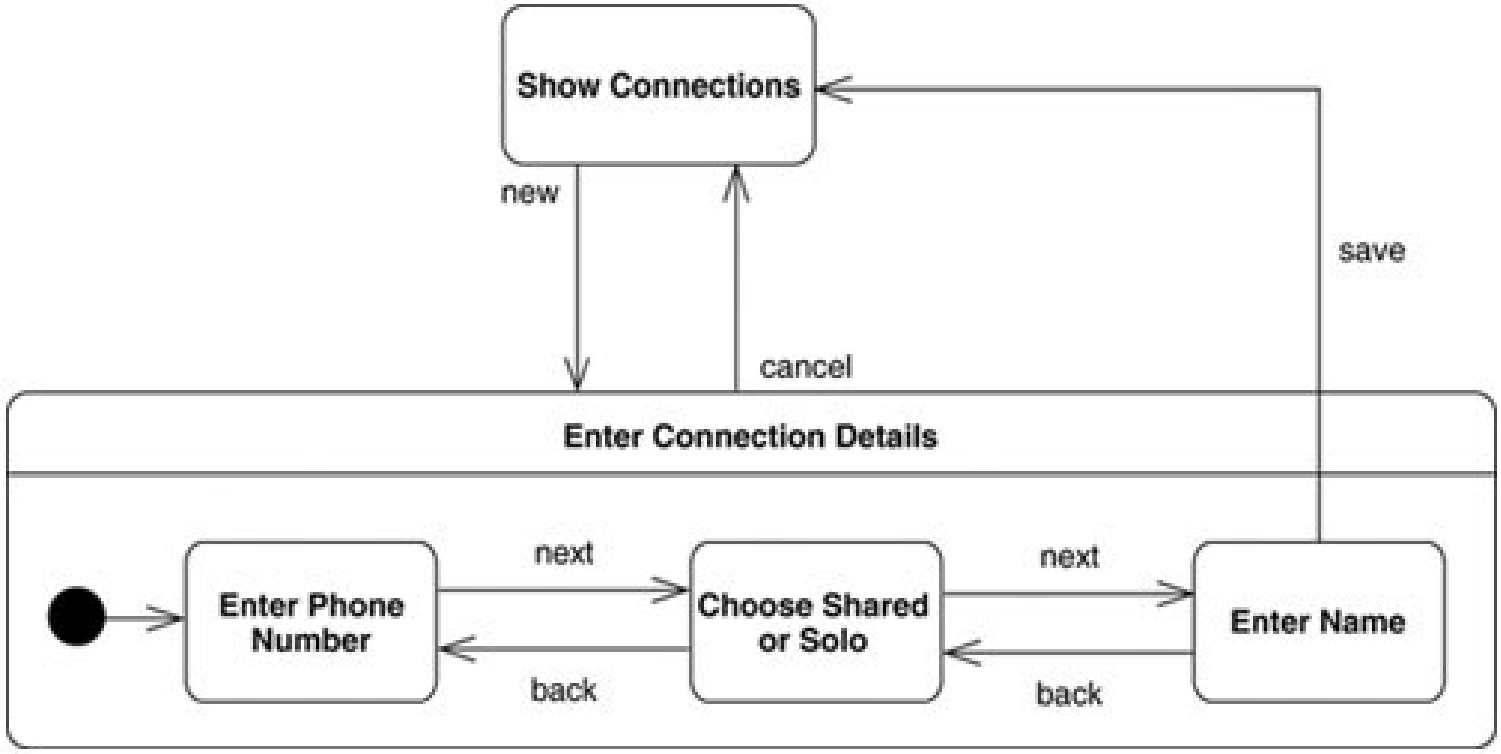
Vending machine Implementation

Uses the state pattern



Sub states

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



Contents

Library Application: Text based UI

User Screen

0) Exit
1) Login as administrator

1

Login Screen

password

adminadmin

Logged in.

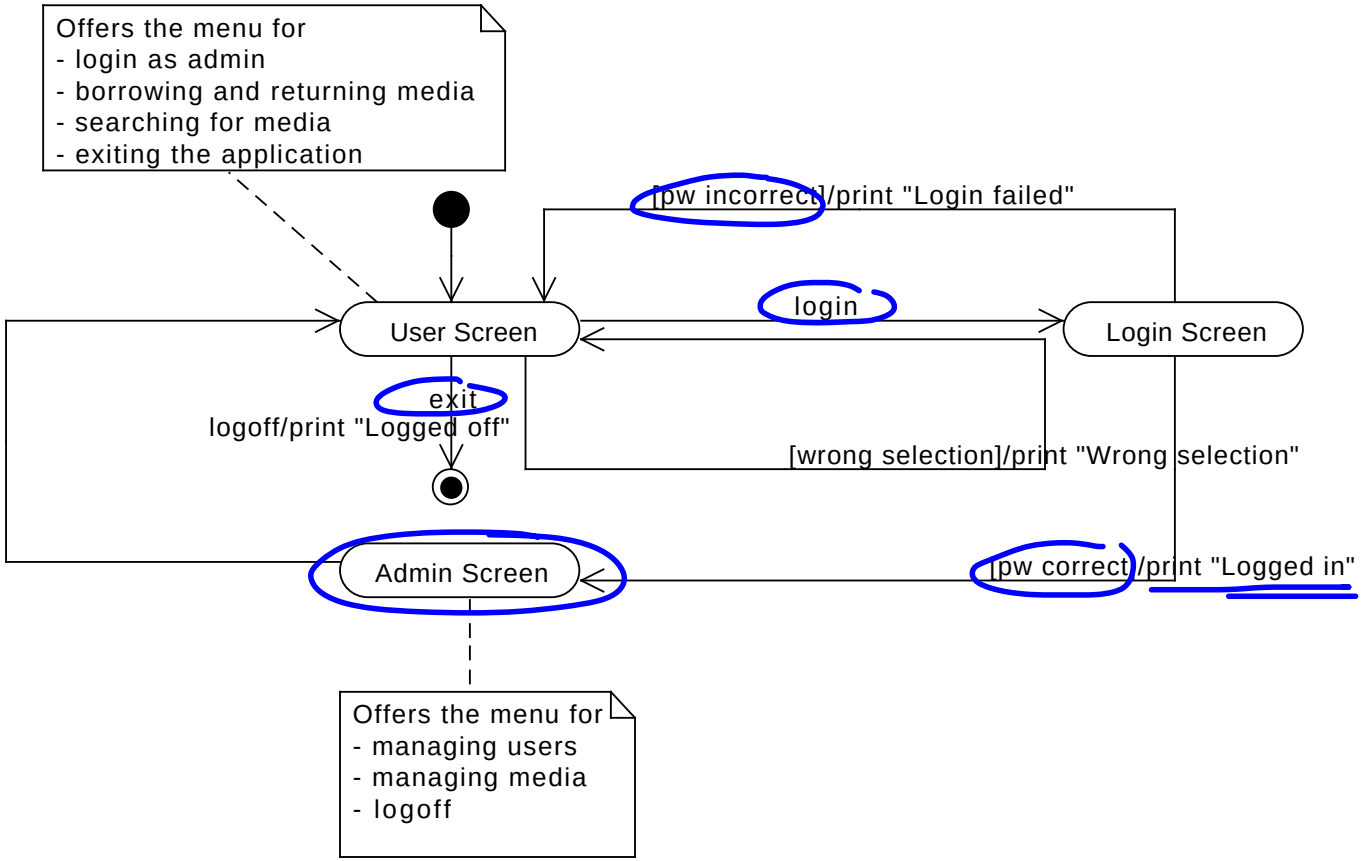
Admin Screen

0) Logoff

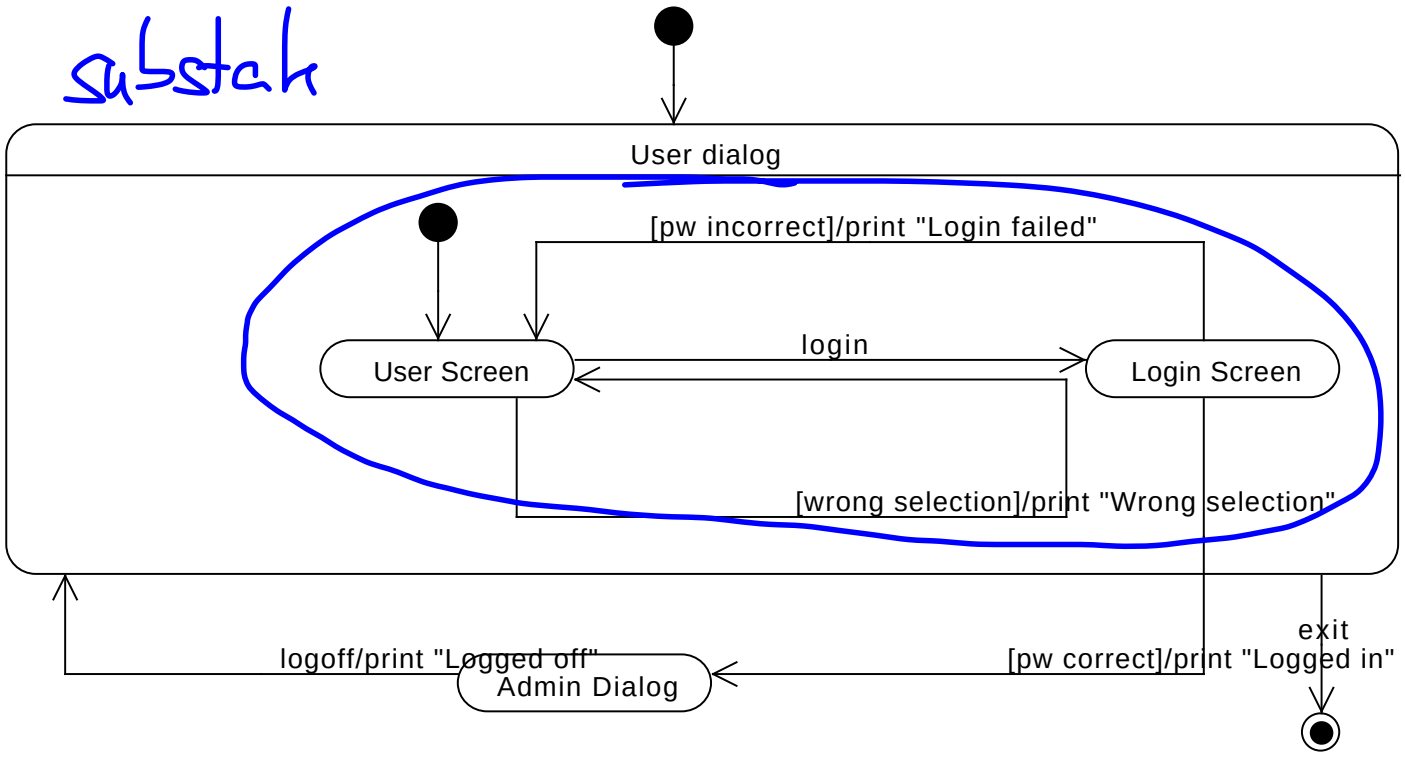
0

Logged off.

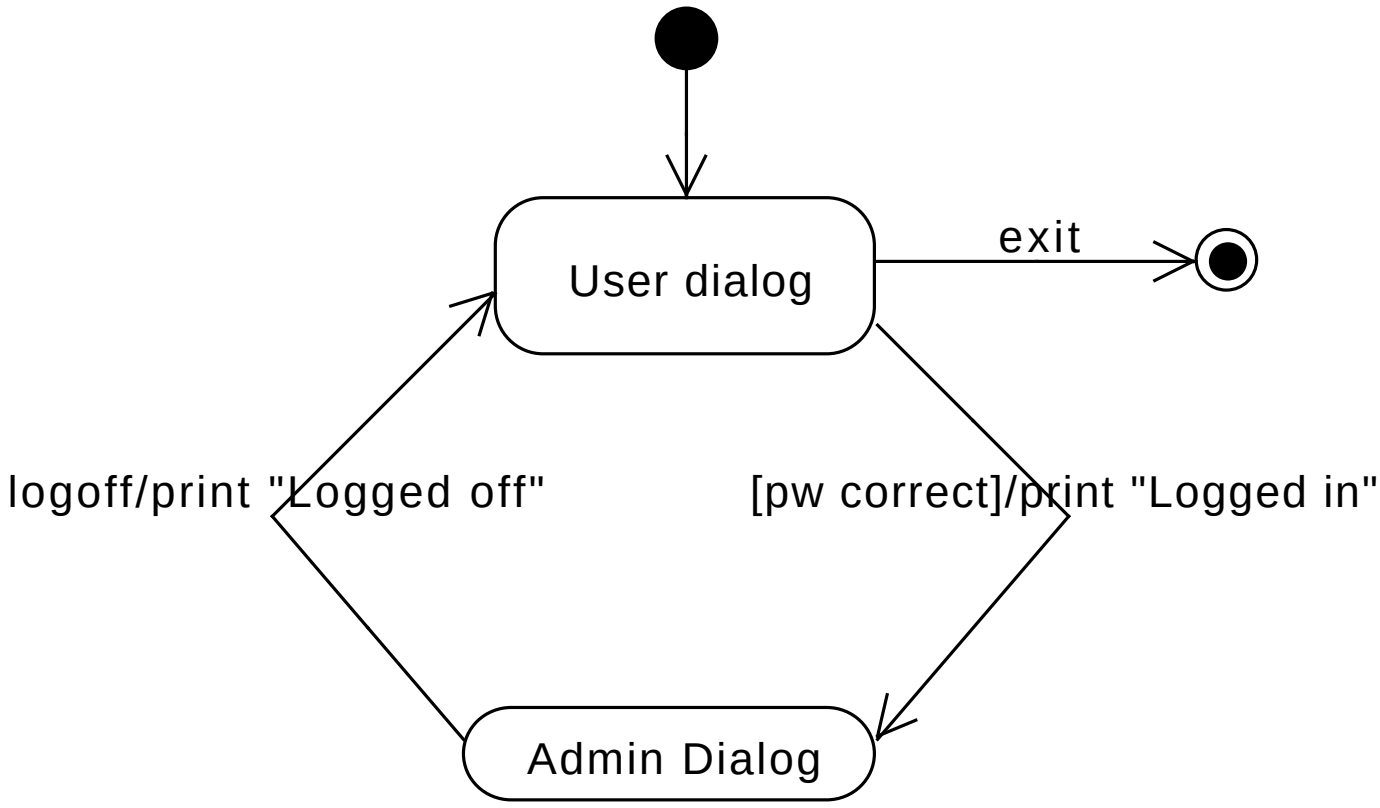
Example Library Application



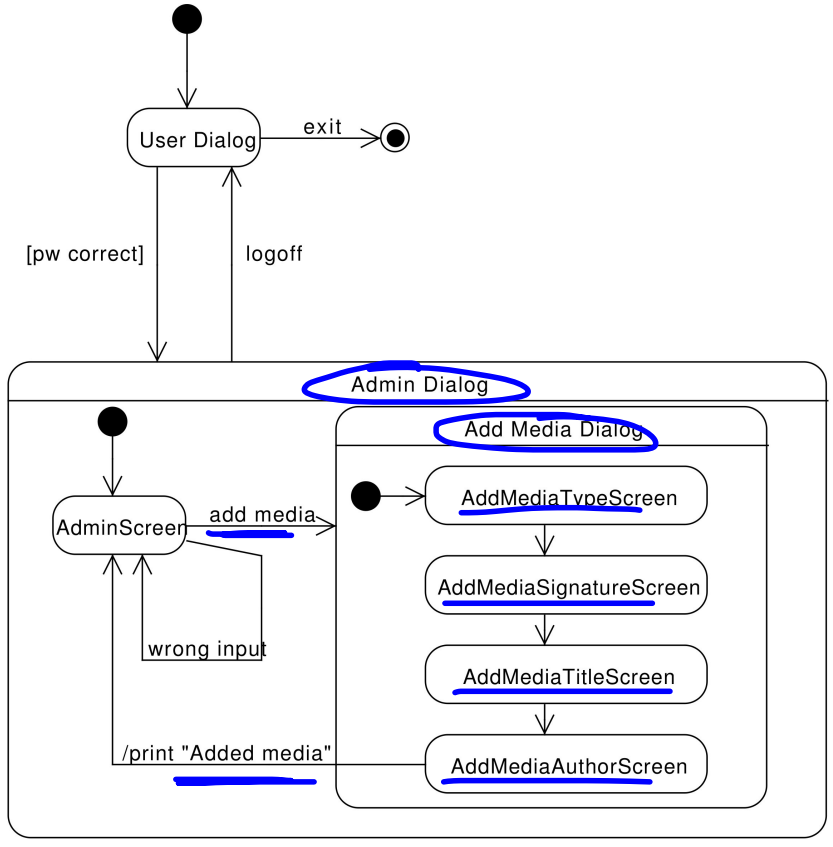
Library App


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

Library App


User dialog
exit
logoff/print "Logged off"
[pw correct]/print "Logged in"
Admin Dialog

Library App: Admin Dialog



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