Software Engineering I (02161)

Week 7

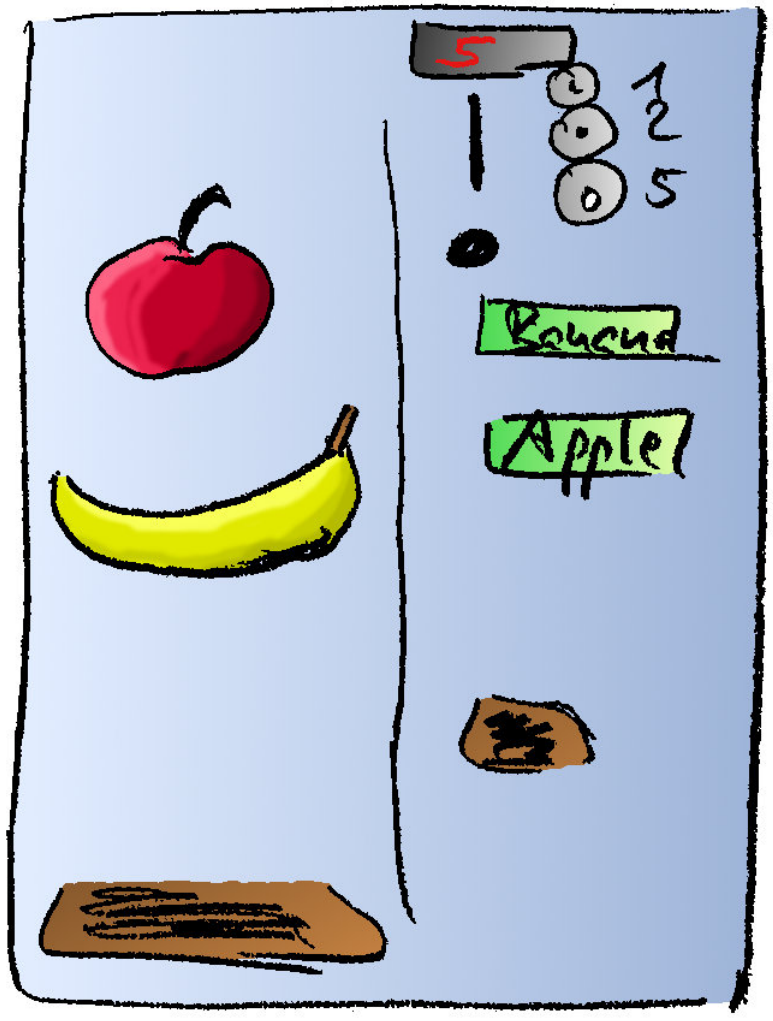
Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2015

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

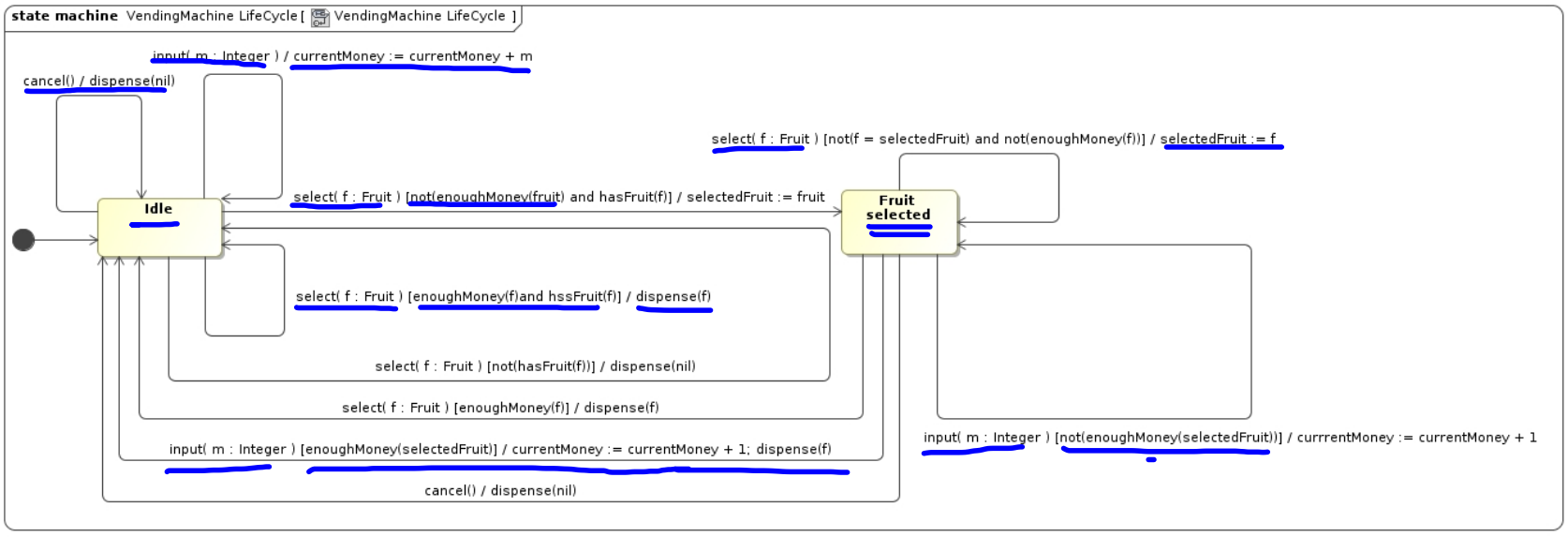
State transition table

event	guard	state	state
		Idle (I)	*Fruit f selected and not enough money (F(f))*
Select fruit f	*enough money for f*	dispense f and rest money → I	dispense f and rest money → I
Select fruit f	*not enough money for f*	→ F(f)	→ F**(f)**
Select fruit f	*no fruits of type f available*	return money → I	return money → I
Input money	*enough money for fruit f*	add money to current money → I	dispense f and rest money → I
Input money	*not enough money for f*	add money to current money → I	add money to current money → F(f)
cancel		return current money → I	return current money → I

- Easy to check for completeness: Does every state implement a reaction to every event?
- Easy to describe behavior: finite number of events and states

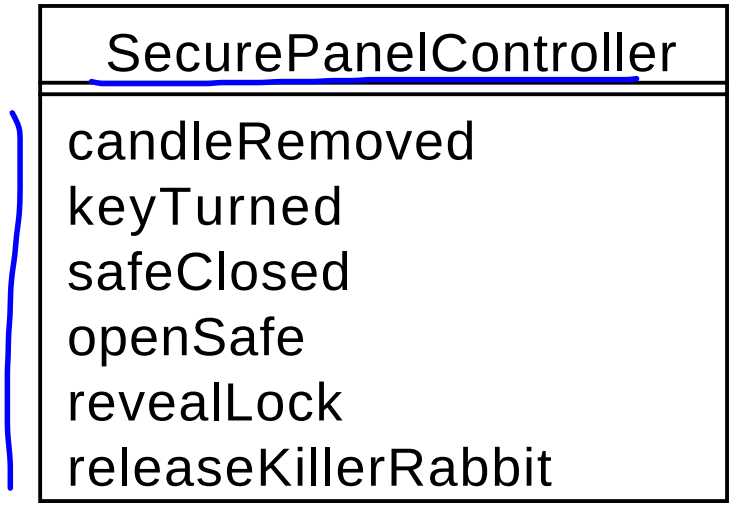
→ Good for these type of situations. For example, embedded systems

UML State Machines

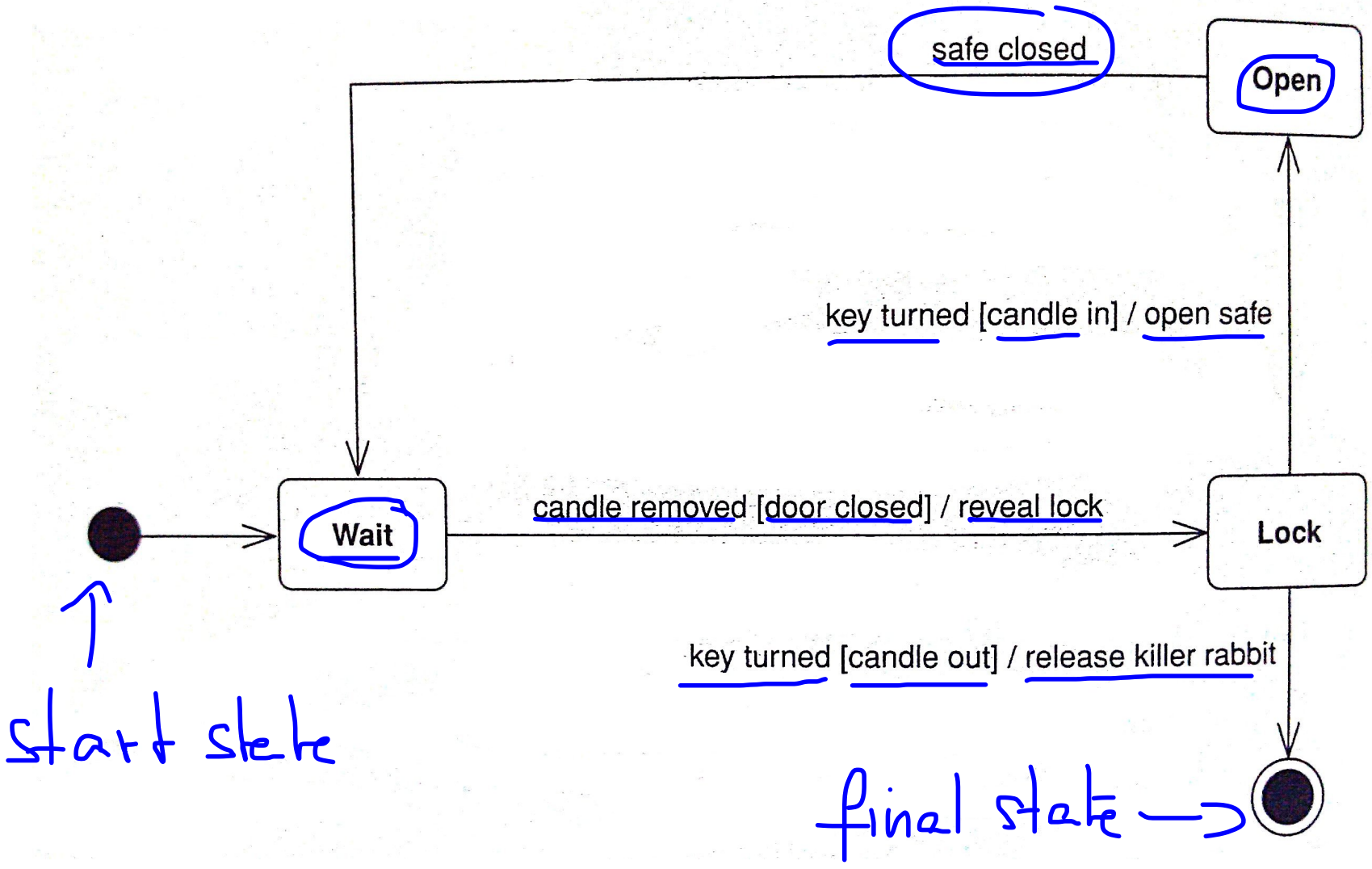


Example: Safe

- 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

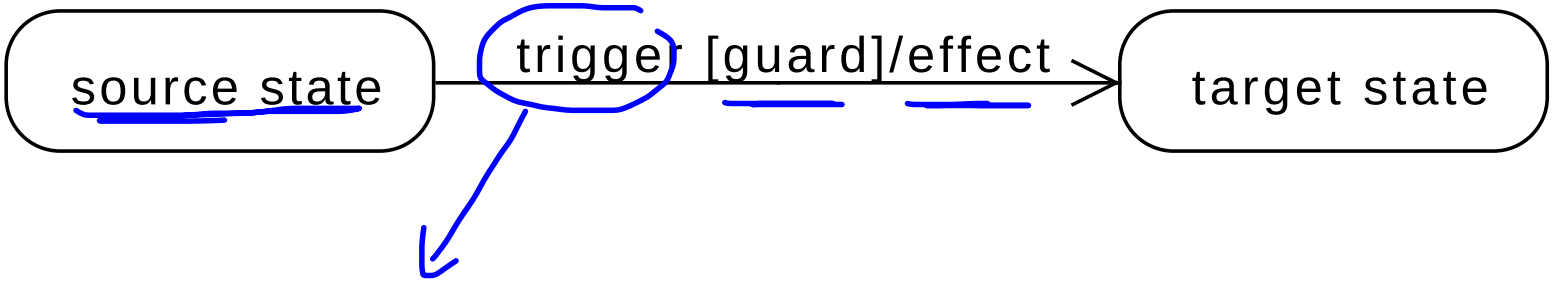


Example: Safe



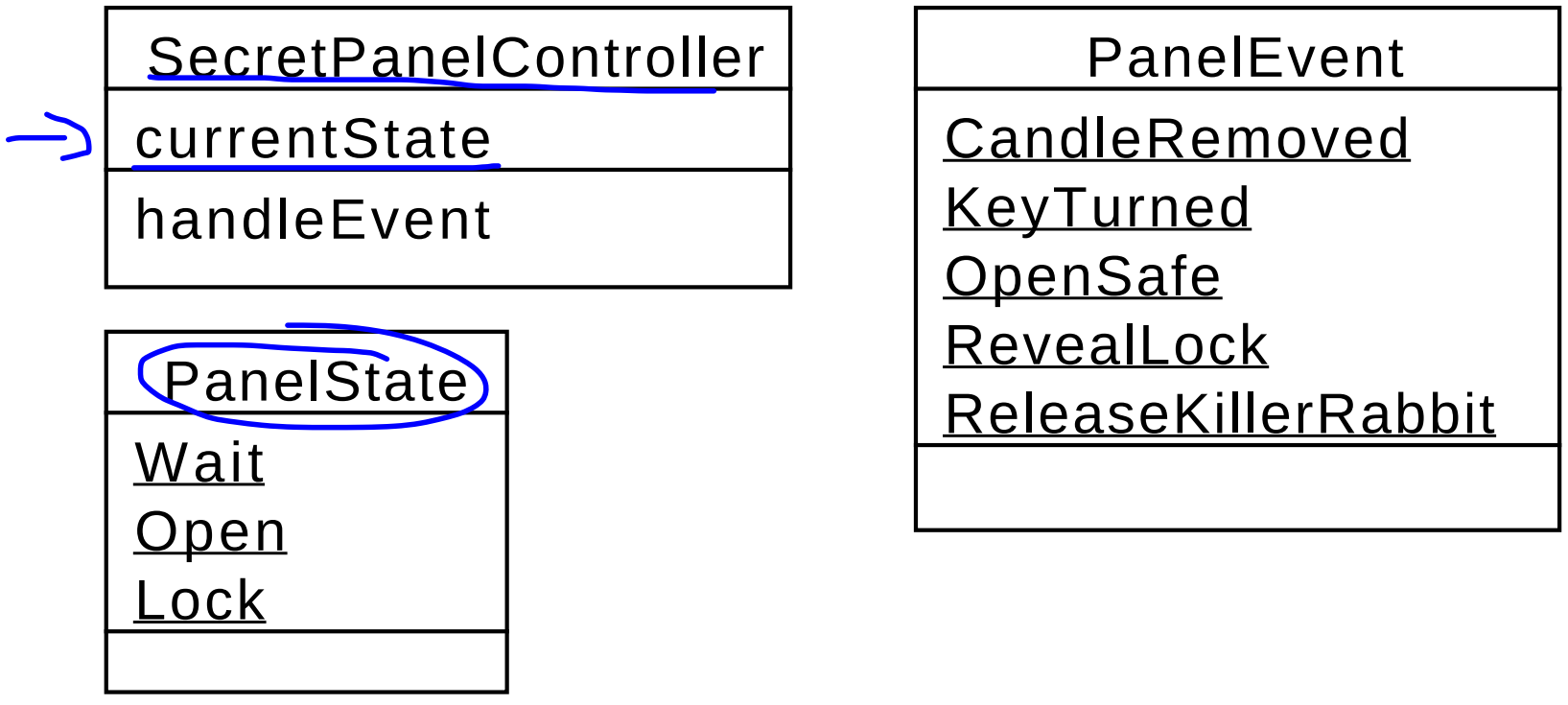
Transitions

- General form



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

Implementation 1: Class diagram

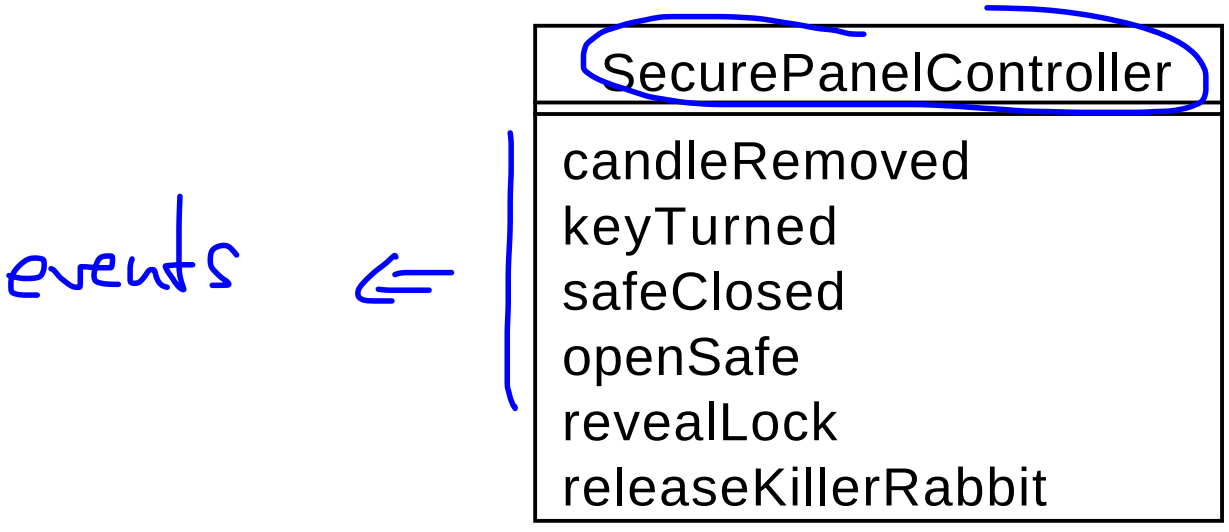


Implementation 1

```
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;
    }
   }
 }
}
```

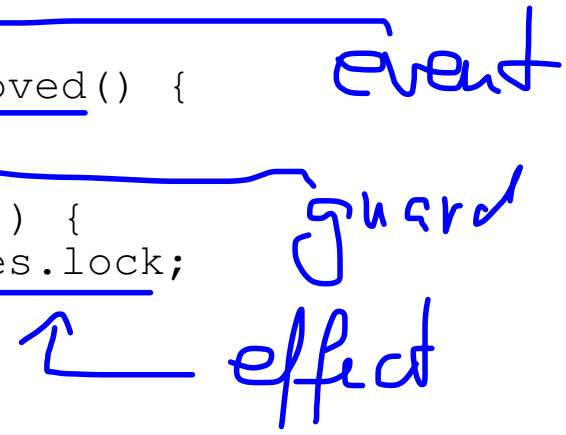
guard

Implementation 2: Class diagram

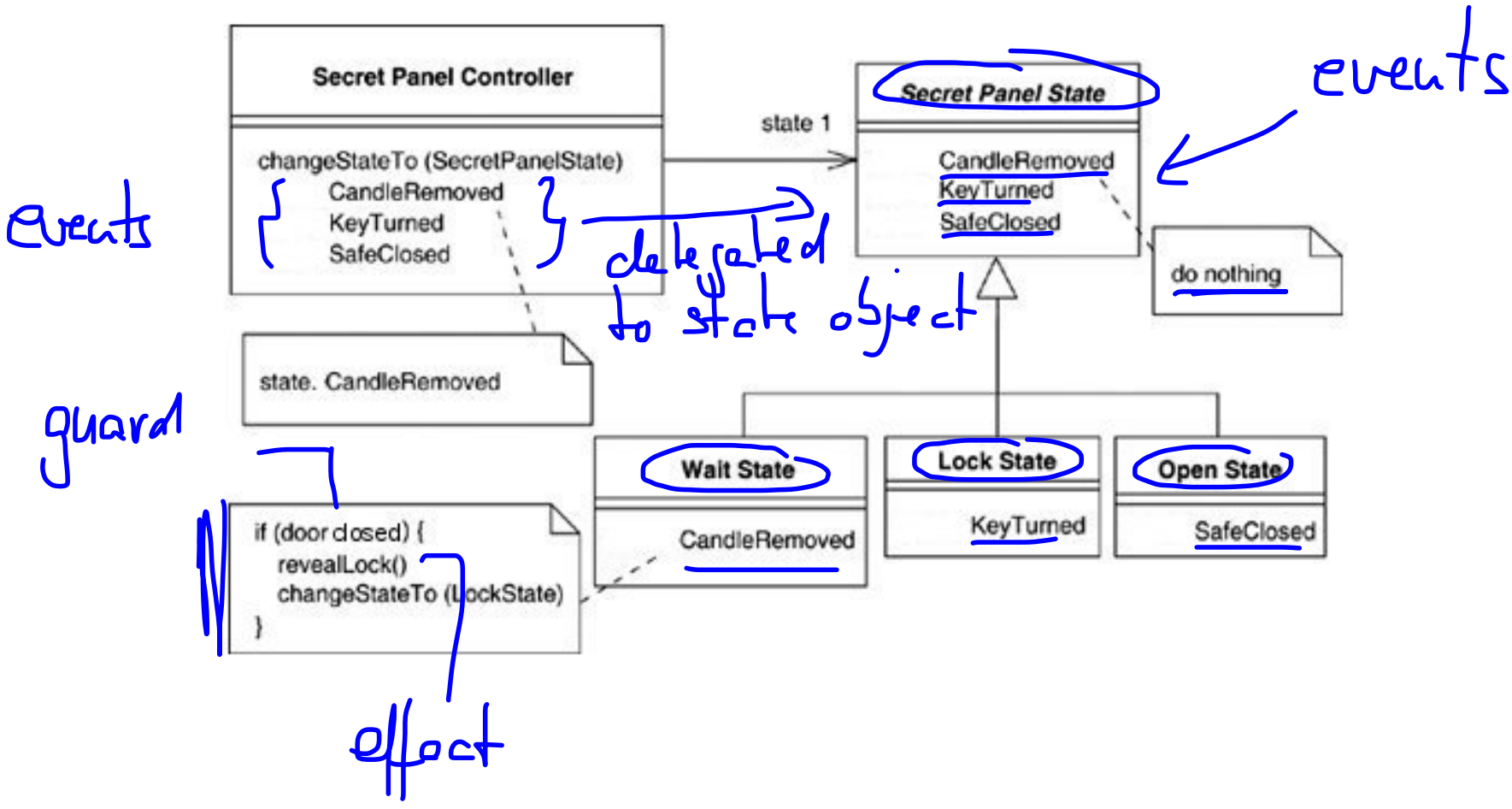


Implementation 2

```
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;
      }
   } ... }
```



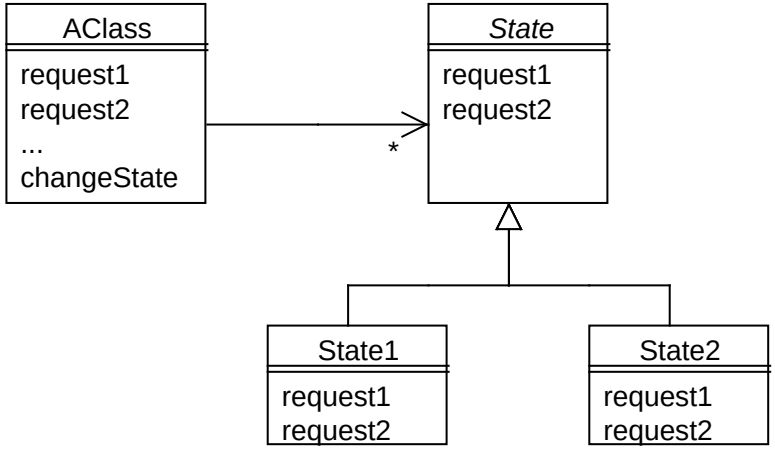
Implementation 3: Using the state pattern

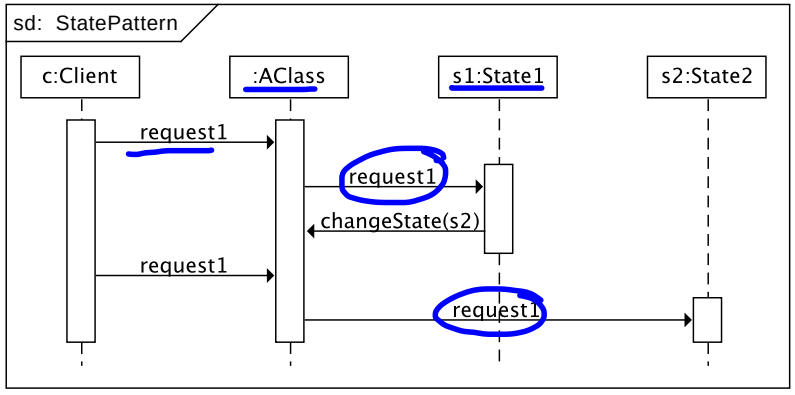


State Pattern

State Pattern

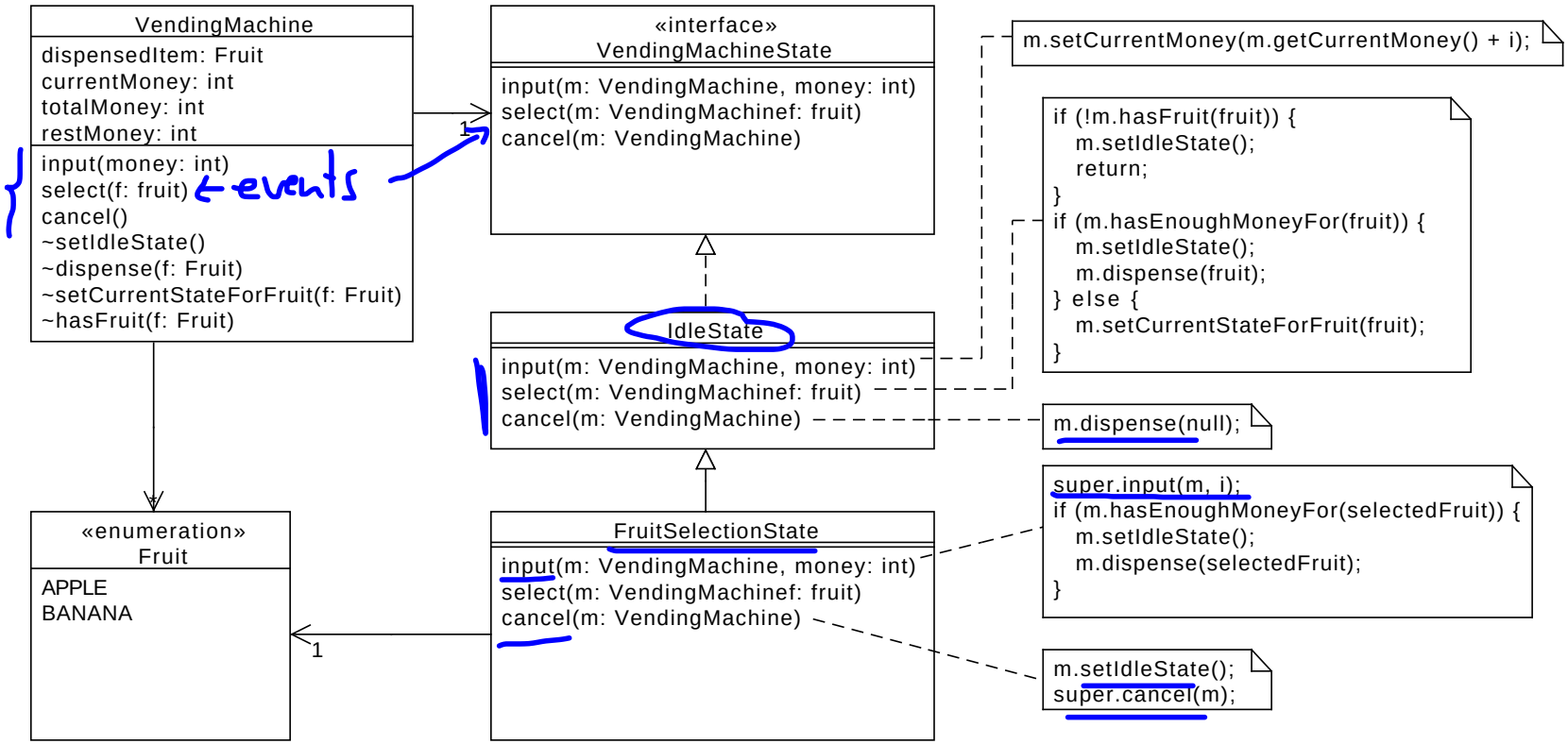
”Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.” Design Pattern book





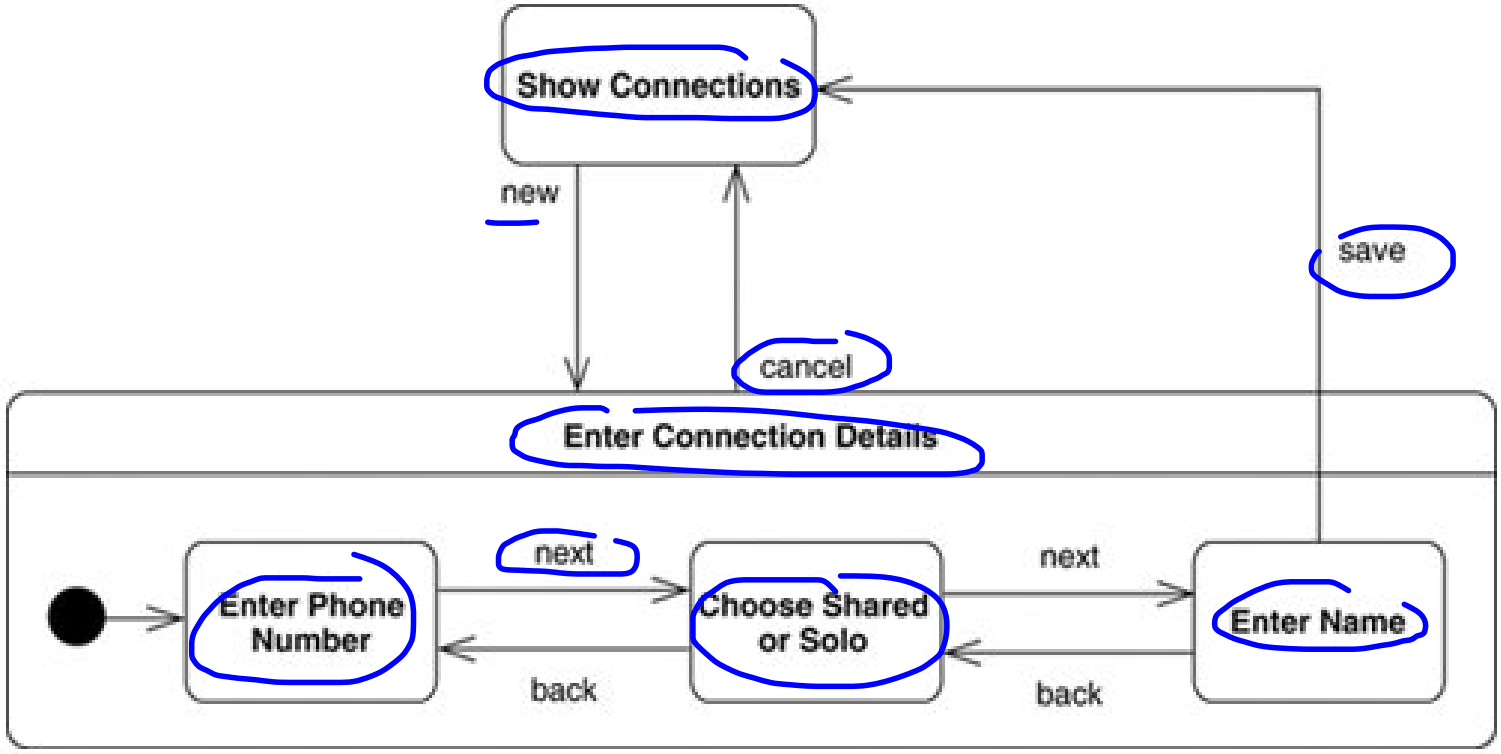
Vending machine Implementation

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

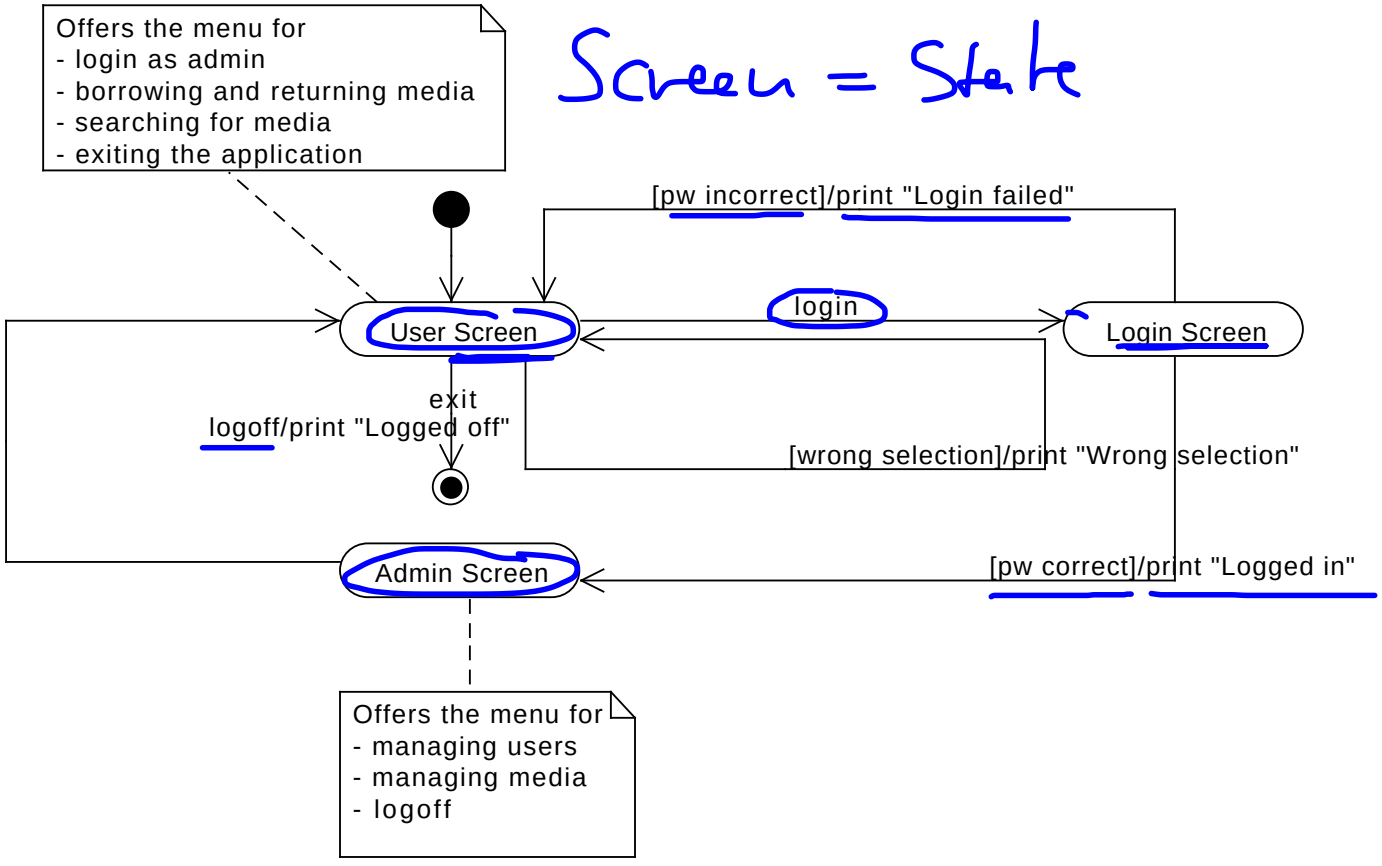
Logged in.

Admin Screen

0) Logoff

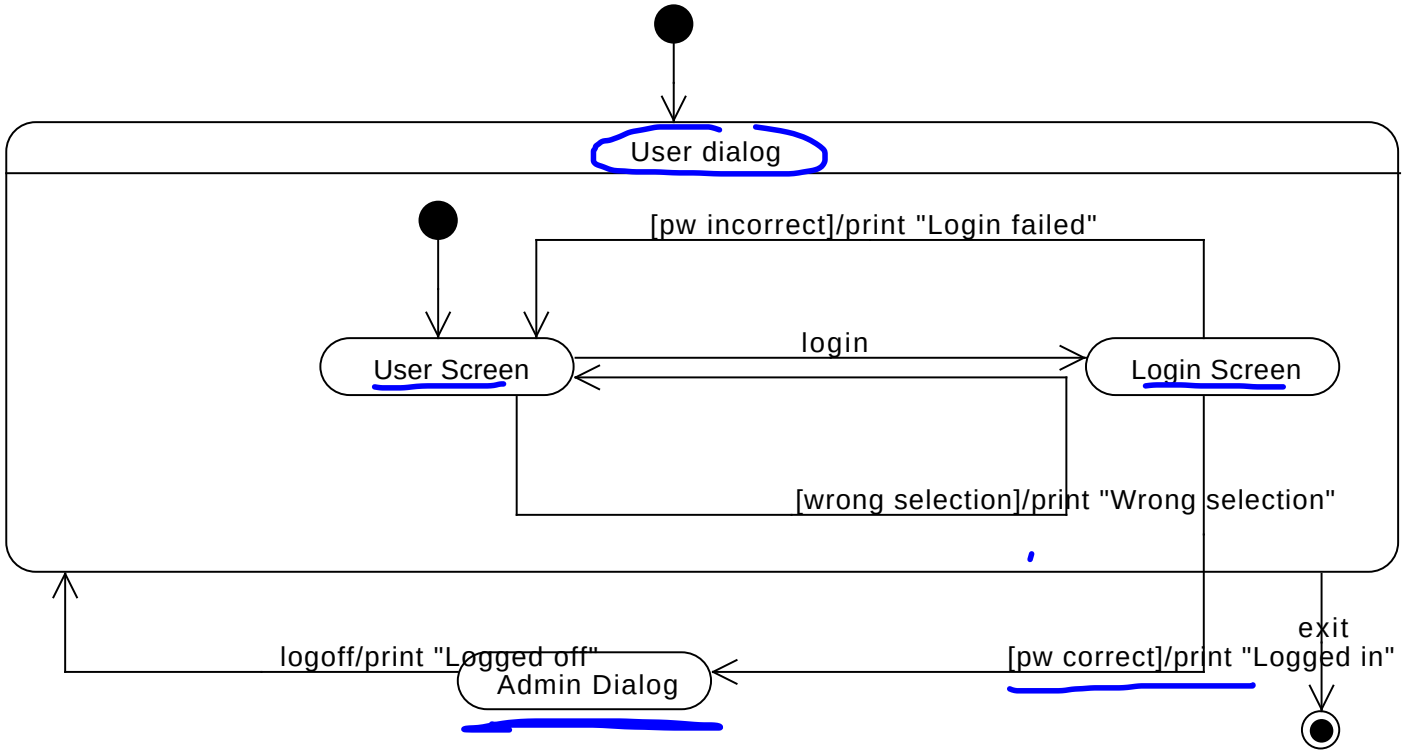
0
Logged off.

Example Library Application



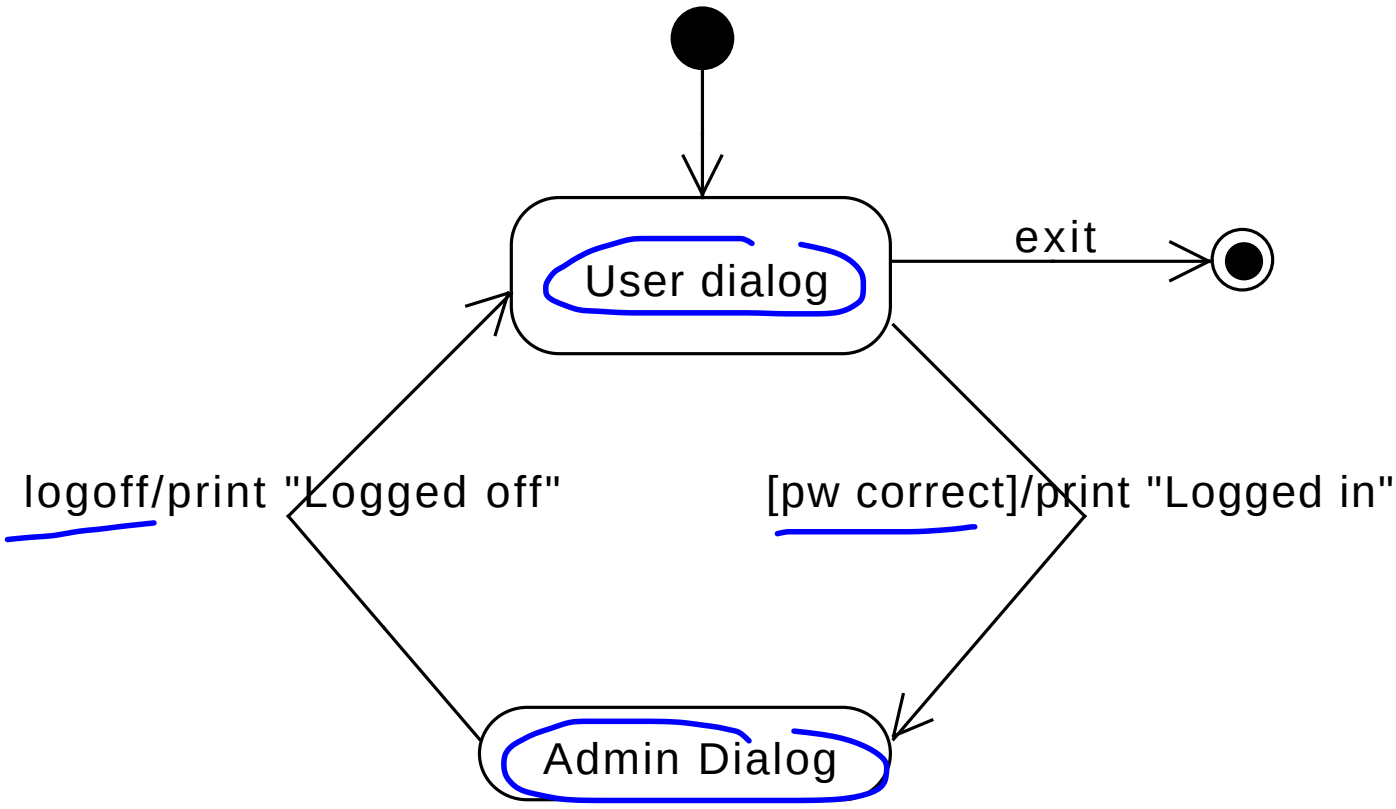
Library App: Focus on user dialog

Use state UserDialog to group the user screen activities



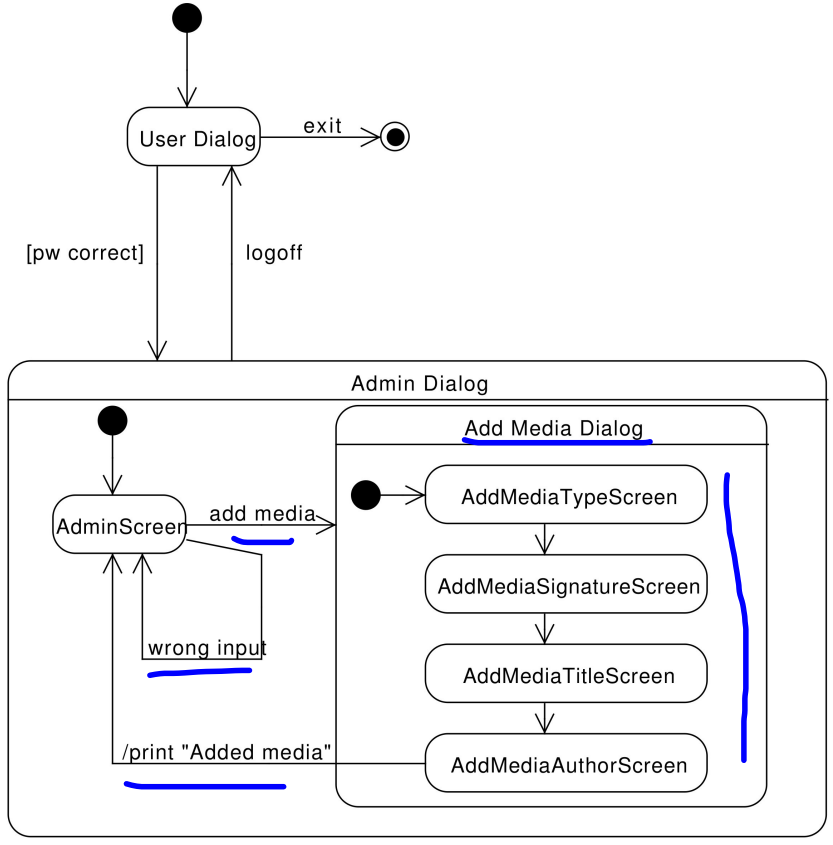
Library App: Overview

Focus on the sequence of dialogs instead of screens

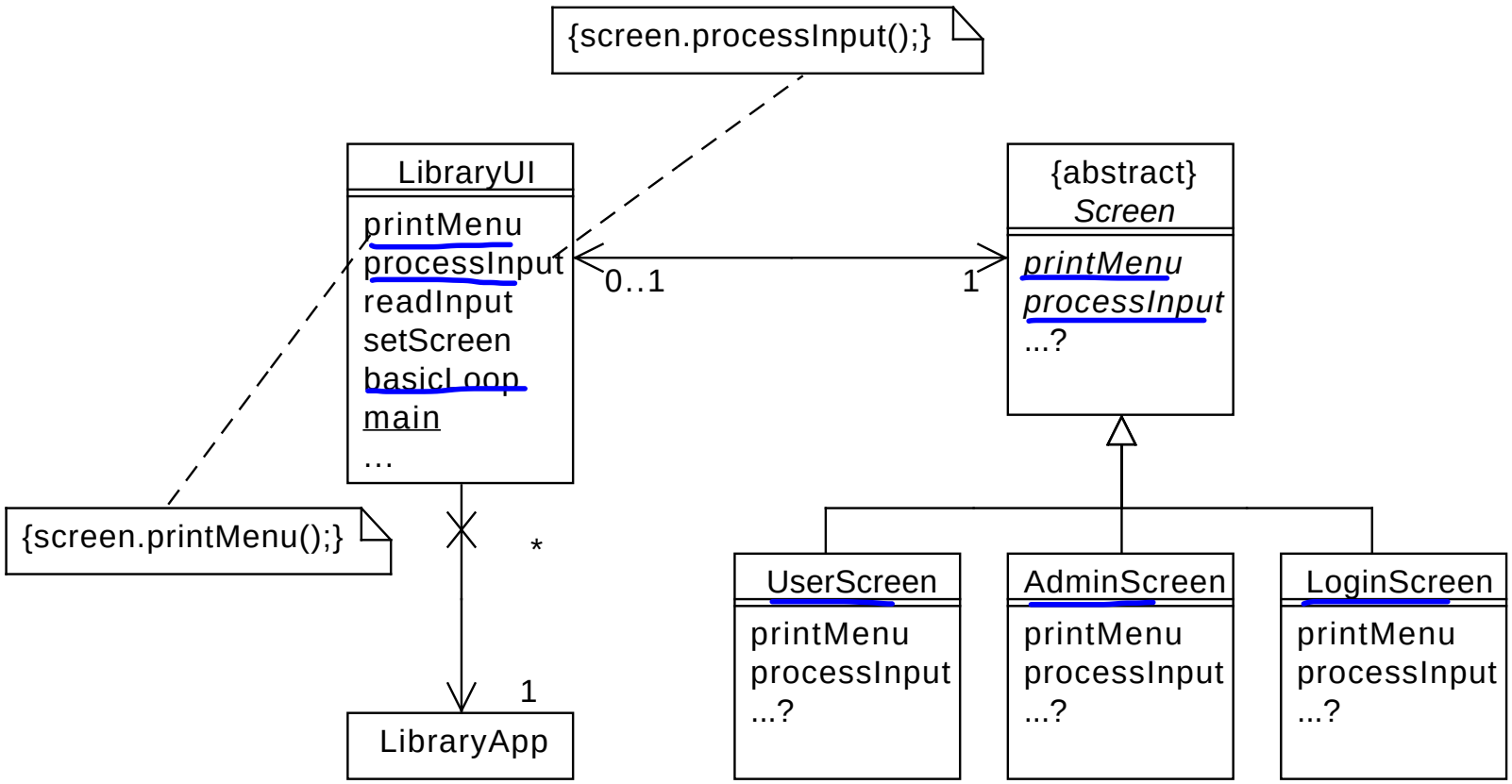


Library App: Focus on admin dialog

Use state AdminDialog to group the admin screen activities



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);
}
```

Basic loop

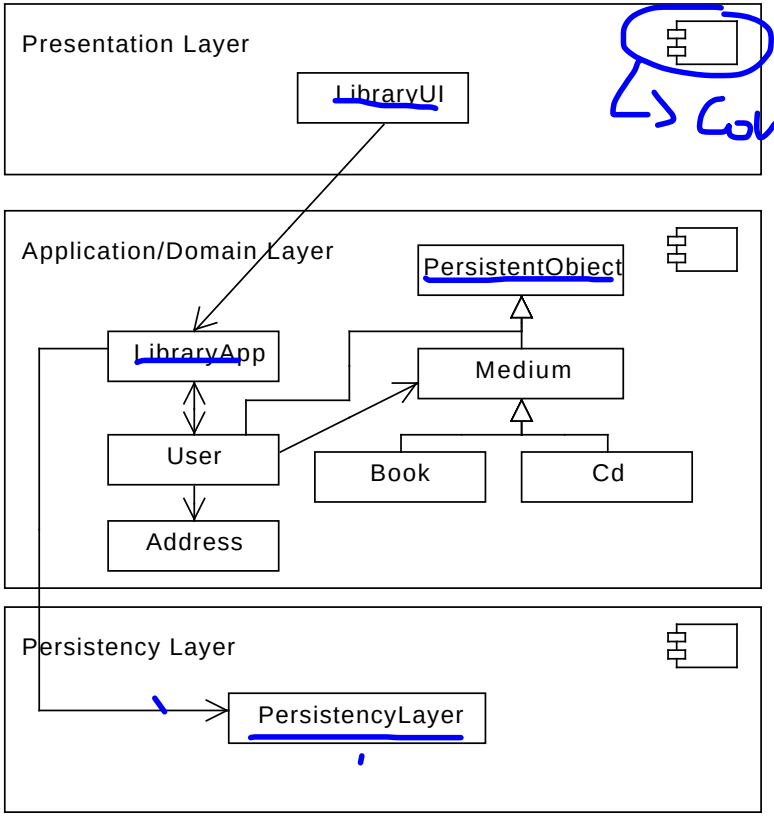
```
public void basicLoop(BufferedReader in, PrintWriter out)
      throws IOException {
  String selection;
  do {
    printMenu(out);
    selection = readInput(in);
  } while (!processInput(selection, out));
}

public void printMenu(PrintWriter out) throws IOException {
   screen.printMenu(out);
}

public boolean processInput(String input, PrintWriter out)
                                                  throws IOException {
   return screen.processInput(input,out);
}
```

Library App user interface exercise (programming exercise 5)

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

Contents

Layered Architecture: Persistency Layer for the library application



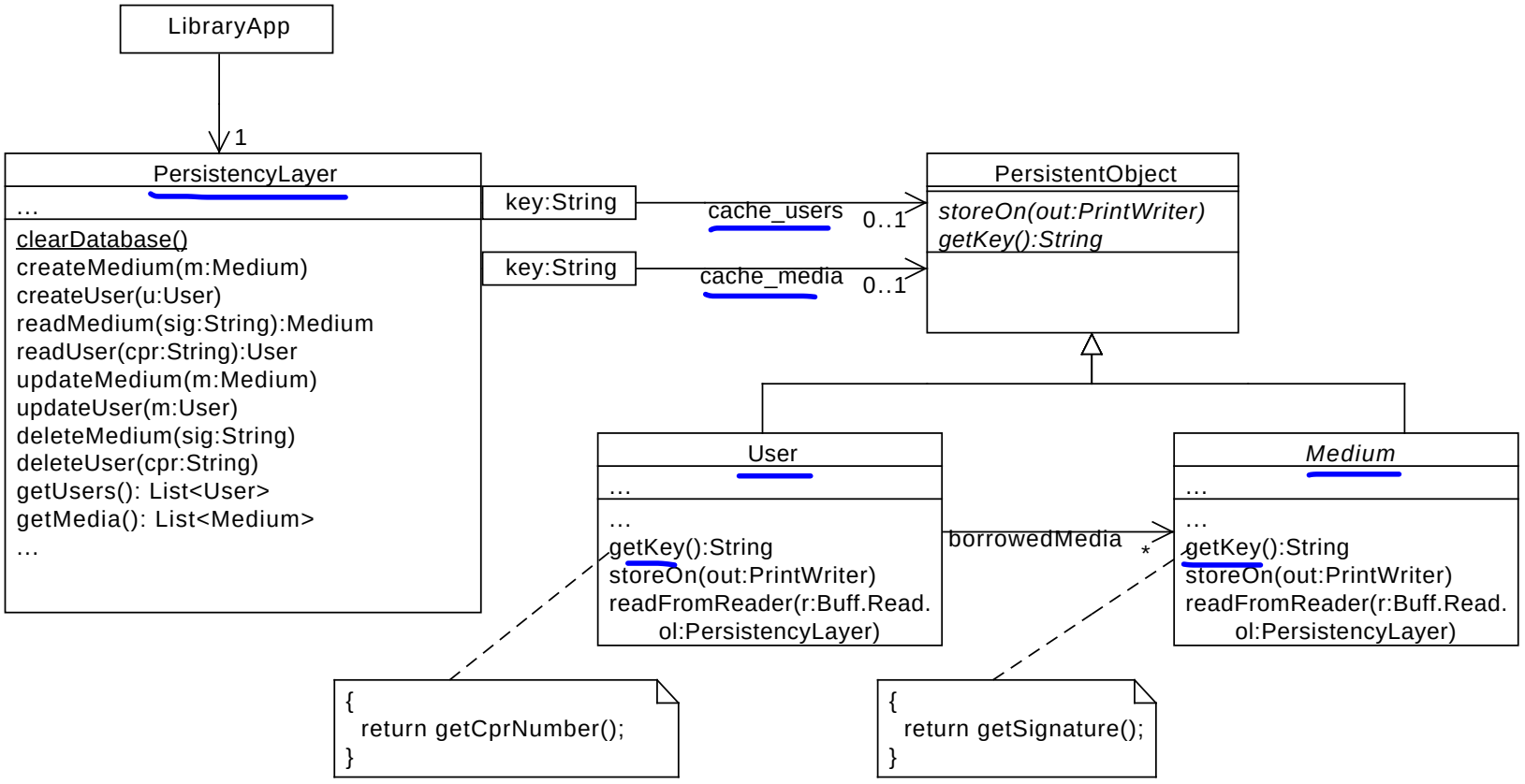
- Data stored in two files `users.txt` & `media.txt`; address has no file
- A book

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

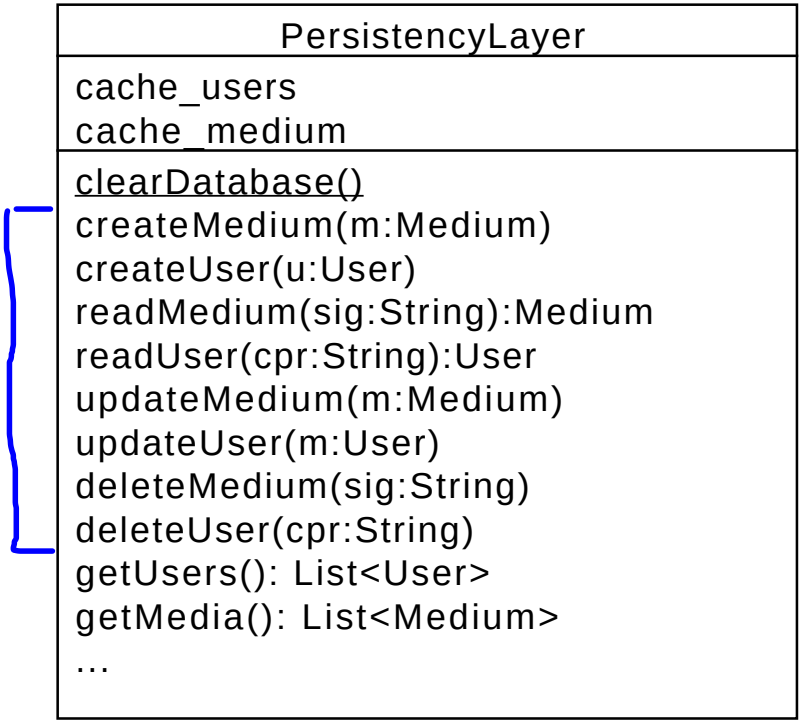
- A user

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

Persistency Layer



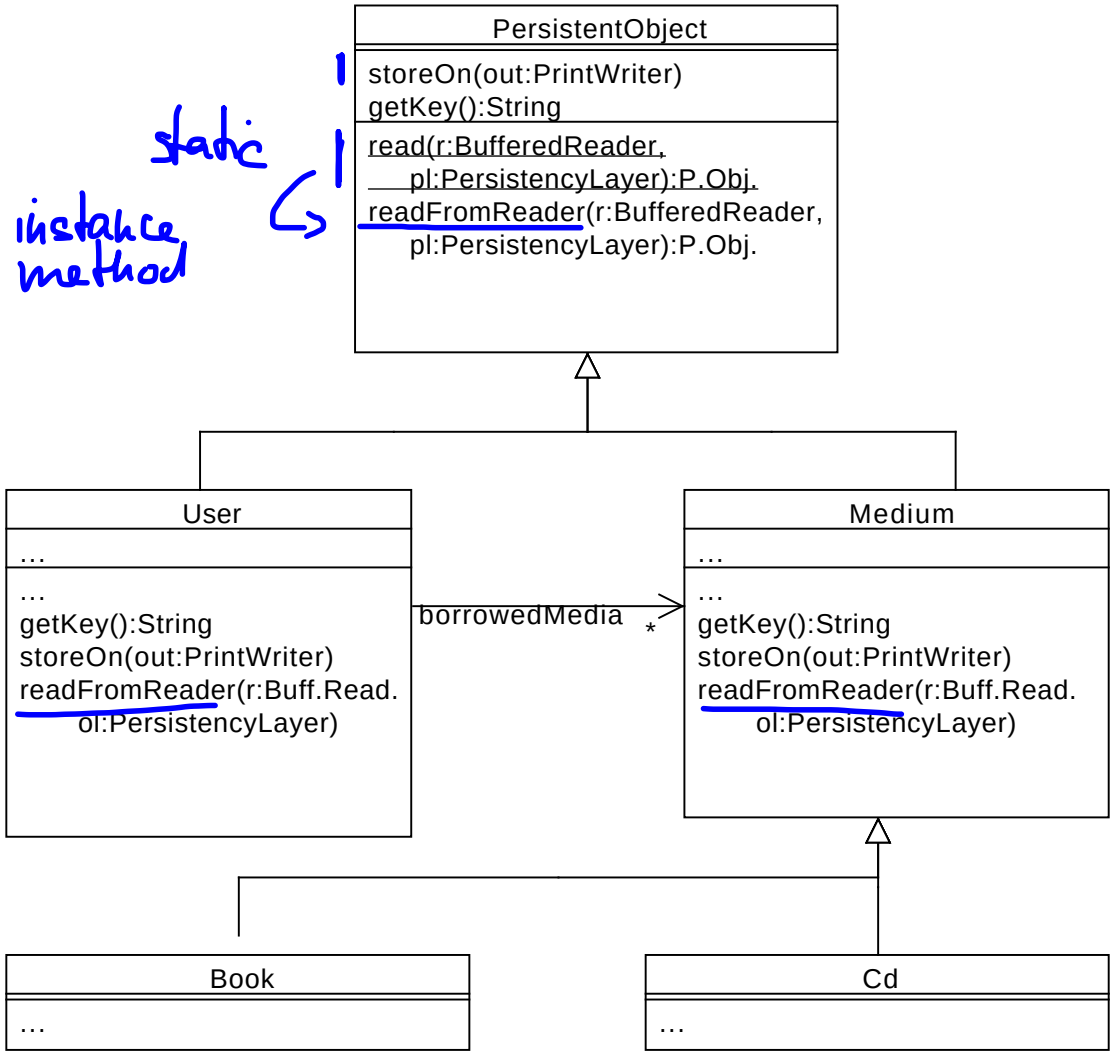
Layered Architecture: Persistency Layer for the library application



- CRUD: Create, Read, Update, Delete
- clearDatabase
  - removes the text files to create an empty database
  - Used with tests in @Before methods: Independent tests
- createMedium/User: appends a new record to the corresponding file
- updateMedium/User: copy all entries in a new file; replace the old entry with the new entry on copying
- deleteMedium/User: do the same as updateMedium/User, but don't copy the object to be deleted

→ What is the complexity of
createMedium/User, $O(1)$
updateMedium/User, $O(n)$
deleteMedium/User? $O(n)$

Reading/Writing User and Media objects



Reading User and Media objects

```
public class PersistentObject {
 ...
  public static PersistentObject read(BufferedReader in,
      PersistencyLayer pl) throws IOException {
    String type = in.readLine();
    PersistentObject po = null;
    if (type.equals("dtu.library.app.User")) {
      po = new User();
    } else if (type.equals("dtu.library.app.Book")) { ... }
    if (po != null) { po.readFromReader(pl, in);}
    return po.readFromRader(reader,pl);
  }
 ...
}
```

- Delegate the initialization of the created object to the newly created object
  - → No access to the instance variables of the object
  - → The object knows best which data it needs

```
PersistentObject po = new User();
po.readFromReader(pl,in);
```

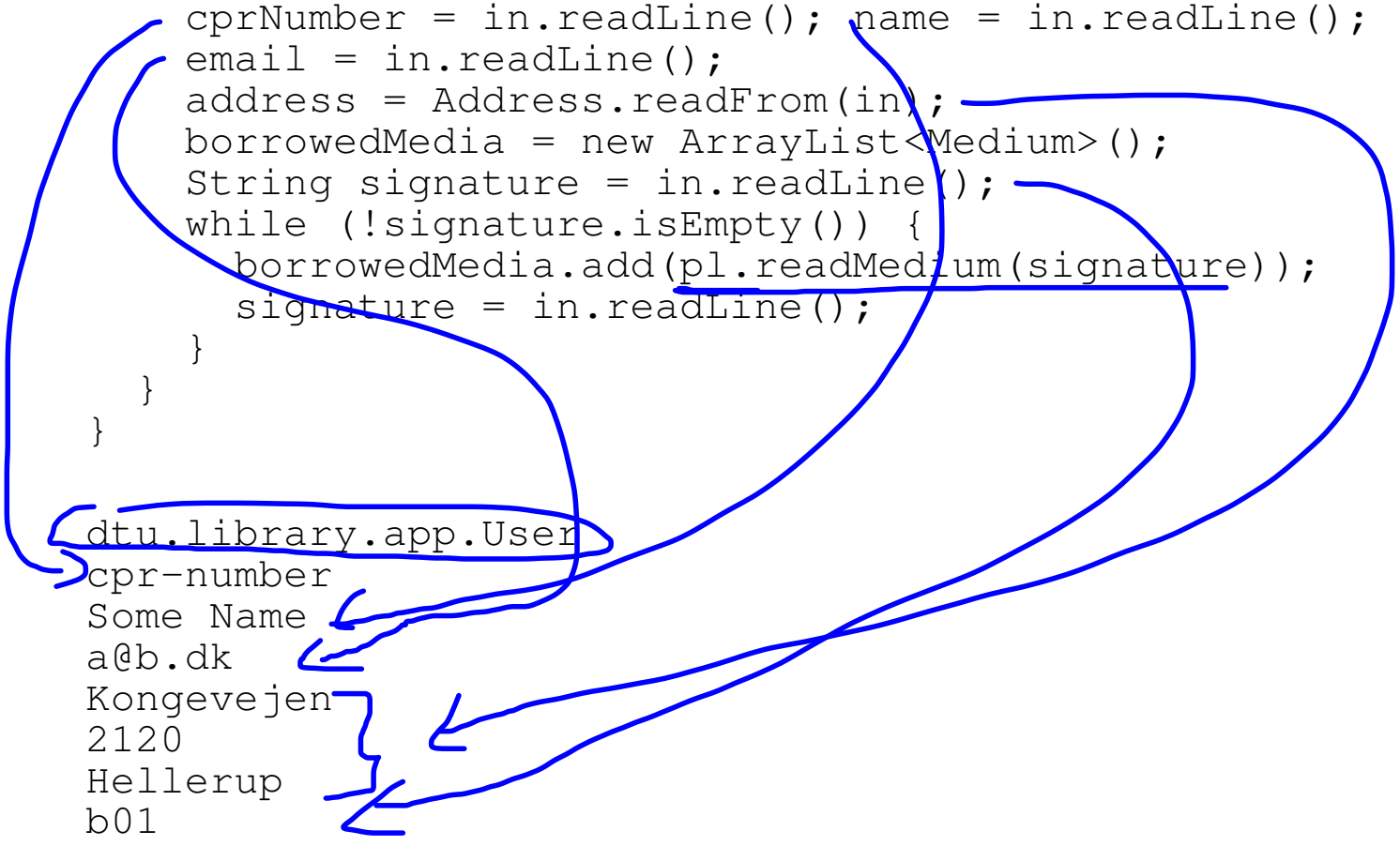
- instead of

```
PersistentObject po = new User();
po.cprNumber(in.readLine());
po.name(in.readLine());
...
```

Class User

```
public class User {
  ...
  public void readFromReader(PersistencyLayer pl, BufferedReader in)
      throws IOException {
    cprNumber = in.readLine(); name = in.readLine();
    email = in.readLine();
    address = Address.readFrom(in);
    borrowedMedia = new ArrayList<Medium>();
    String signature = in.readLine();
    while (!signature.isEmpty()) {
      borrowedMedia.add(pl.readMedium(signature));
      signature = in.readLine();
    }
  }
}
```

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



Use of Files

Writing files

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

Reading files

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

Deleting and renaming files

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

Tests for the integration

```
@Before
public void setUp() throws Exception {
  libApp = new LibraryApp();
  PersistencyLayer.clearDatabase();
  libApp.adminLogin("adminadmin");
  Address address = new Address("Kongevejen", 2120, "Hellerupl");
  user = new User("cpr-number", "Some Name", "a@b.dk", address);
  libApp.register(user);
  b = new Book("b01", "some book title", "some book author");
  c = new Cd("c01", "some cd title", "some cd author");
  libApp.addMedium(b);
  libApp.addMedium(c);
}
```

```
@Test
public void testBorrowing() throws Exception {
  user.borrowMedium(b);
  user.borrowMedium(c);
  PersistencyLayer pl = new PersistencyLayer();
  User user1 = pl.readUser(user.getCprNumber());
  assertEquals(2, user1.getBorrowedMedia().size());
  Utilities.compareUsers(user, user1);
}
```

Implementation in LibraryApp

```
public void borrowMedium(Medium medium) throws BorrowException {
  if (medium == null)
    return;
  if (borrowedMedia.size() >= 10) {
    throw new TooManyBooksException();
  }
  for (Medium mdm : borrowedMedia) {
    if (mdm.isOverdue()) {
      throw new HasOverdueMedia();
    }
  }
  medium.setBorrowDate(libApp.getDate());
  borrowedMedia.add(medium);
  try {
    libApp.getPersistencyLayer().updateUser(this);
  } catch (IOException e) {
    throw new Error(e);
  }
}
```

appl. Lgc

persist.

Issues: Object identity

```
PersistencyLayer pl = new PersistencyLayer();
User user1 = pl.readUser("12345");
User user2 = pl.readUser("12345");
assertNotSame(user1,user2) // ?
assertSame(user1,user2)    // ?
```

false

true

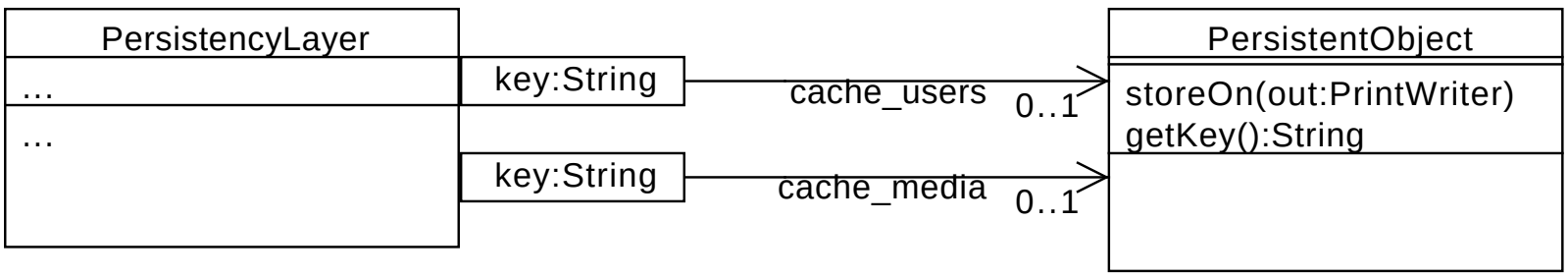
Should be the same objects

Solution: Qualified Associations / Maps

```
Map<String,PersitentObject> cacheUsers =
                              new HashMap()<String,PersistentObject>
Map<String,PersitentObject> cacheMedia =
                              new HashMap()<String,PersistentObject>
```

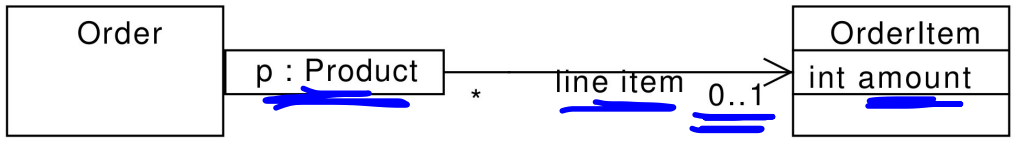
UML Notation

qualified association




```
public User readUser(String key) {
   if (cacheUsers.contains(key)) { return cacheUsers.get(key); }
   User user = readObjectFromFile(String key);
   if (user != null) { cacheUsers.put(key,user); }
   return user;
}
```

Qualified Assocations I

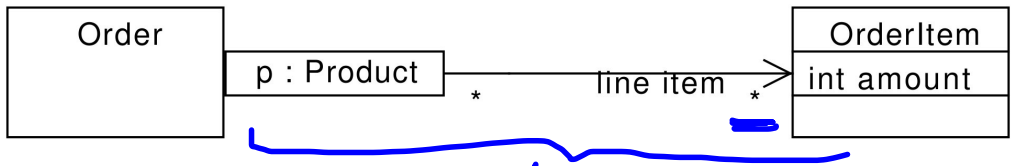


- A qualified association is an association, where an object is associated to another object via a qualifier (a third object)
- An **Order** has an **OrderItem** for each **product**
- If the multiplicity is ≤ 1 then an order has at most one list item for each product
  - $\rightarrow$ This is usually implemented by a **map** or **dictionary** mapping products to order items

```
public class Order {
   private Map<Product,OrderItem>
      listItem = new HashMap<Product,OrderItem>()
   ...
}
```

Qualified Associations II

- If the multiplicity is *, then several order items may be associated to a product



- Then the map has to return a collection for each product

```
public class Order {
   private Map<Product,Collection<OrderItem>>
      listItems = new HashMap<Product,Collection<OrderItem>>()
   ...
}
```

Map<K,V> Interface

- Dictionary (table): keys of type *K*, values of type *V*
- Implementation class: HashMap<K,V>
- Operations
  - `m.put(aK,aV)`
  - `m.get(aK)`
  - `m.containsKey(aK)`
- Properties
  - *aK* is not a key in *m*

    ```
    assertFalse(m.containsKey(aK));
    assertNull(m.get(aK));
    ```
  - Value *aV* is added with key *aK* to *m*

    ```
    m.put(aK,aV);
    assertTrue(m.containsKey(aK));
    assertSame(aV,m.get(aK));
    ```

Exercise tasks (programming exercise 6):

1) Implement the persistency layer (tests provided)
2) Intergrate persistentcy layer in the library application (tests have to be written)

- Additional information
  `http://www2.imm.dtu.dk/courses/02161/2015/slides/pe_persistency.pdf`