Software Engineering I (02161)

Week 7: Sequence Diagrams, Class Diagrams II, Layered Architecture

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2017

Contents

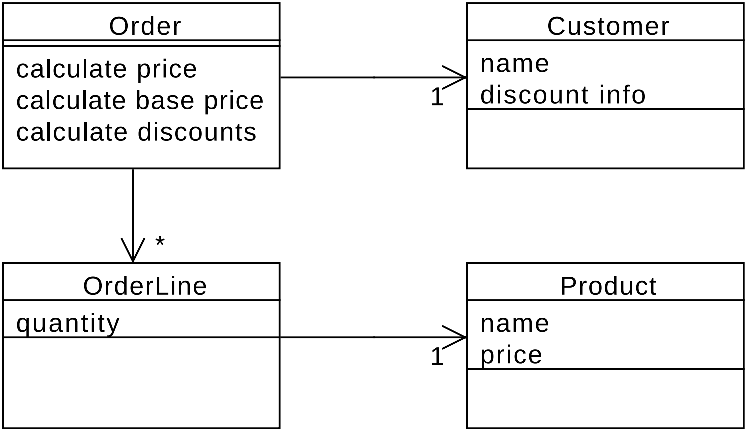
Recap

- How to get from requirements to the design?
  - Execute the use case scenarios / user stories
  - Different techniques: ”in your head”, Class Responsibility Collaboration (CRC) cards, Class diagrams + Sequence diagrams
- Communicating your design:
  - Class diagrams

Contents

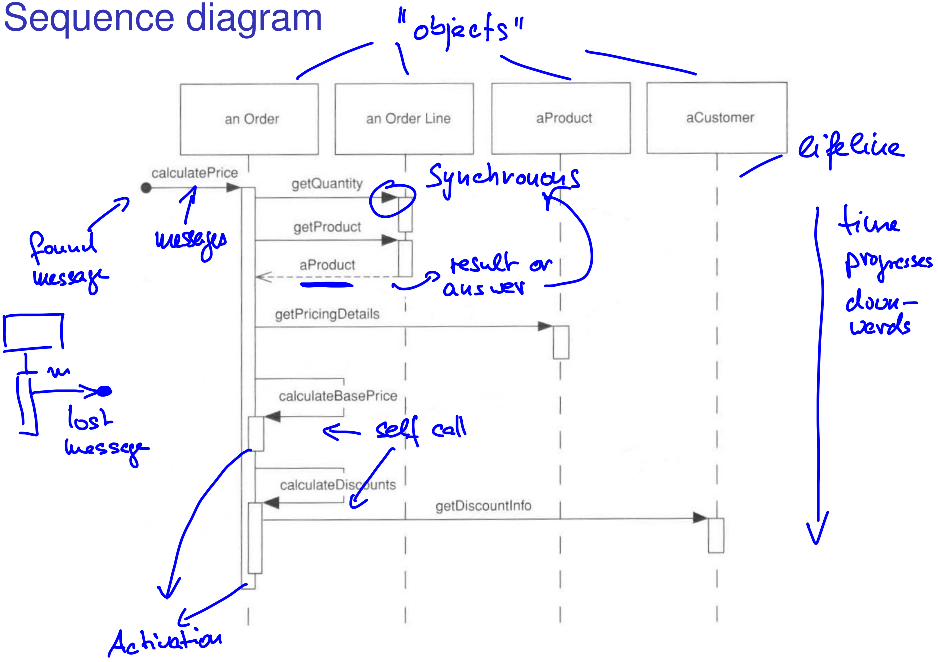
Sequence Diagram: Computing the price of an order

- Class diagram



- Problem:
  - What are the operations doing?

Sequence diagram



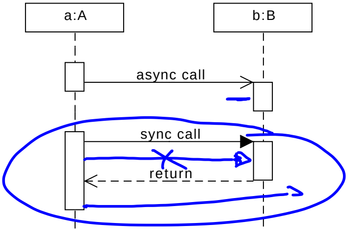
Synchronous– vs Asynchronous Calls:

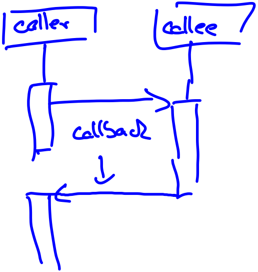
Synchronous

```
b.m(4);
c.n(...) // Starts after m has returned
```

Asynchronous

```
// (new Thread(){ public void run() {b.m(4);}}).start();
new Thread(() -> {b.m(4);}).start(); // Using Lambdas from Java 8
c.n(...) // Starts immediately after m has been called
```



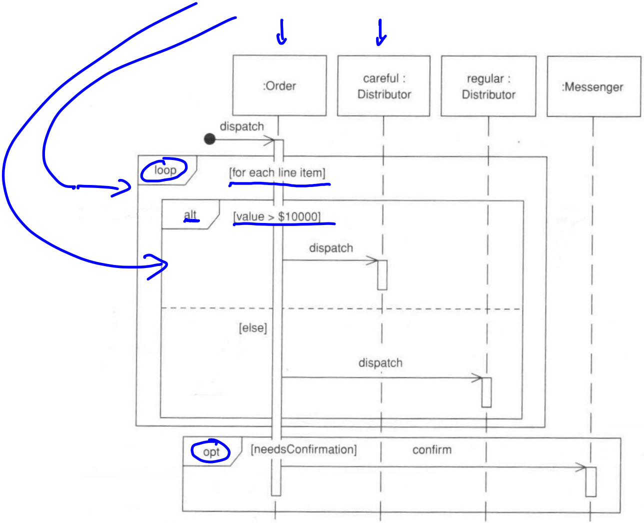


Interaction Frames Example

Realising an algorithm using a sequence diagram

```
public void dispatch() {
  for (LineItem lineItem : lineItems) {
    if (lineItem.getValue() > 10000) {
      careful.dispatch();
    } else {
      regular.dispatch();
    }
  }
  if (needsConfirmation()) {
    messenger.confirm();
  }
}
```

Realisation with Interaction Frames



Use Interaction Frames sparingly

Interaction Frame Operators I

Operator	Meaning
alt	Alternative multiple fragments; only the one whose condition is true will execute (Figure 4.4).
opt	Optional; the fragment executes only if the supplied condition is true. Equivalent to an alt with only one trace (Figure 4.4).
par	Parallel; each fragment is run in parallel.
loop	Loop; the fragment may execute multiple times, and the guard indicates the basis of iteration (Figure 4.4).
~~region~~ critical	Critical region; the fragment can have only one thread executing it at once.
neg	Negative; the fragment shows an invalid interaction.
ref	Reference; refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
sd	Sequence diagram; used to surround an entire sequence diagram, if you wish.

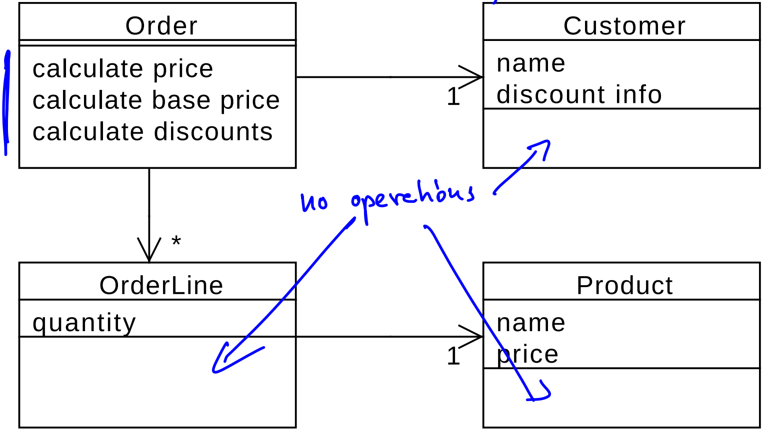
Usages of sequence diagrams

- Show the exchange of messages of a system
  - i.e. show the execution of the system
  - in general only, *one* scenario
  - with the help of interaction frames also several scenarios
- For example use sequence diagrams for
  - Designing (c.f. CRC cards)
  - Visualizing program behaviour

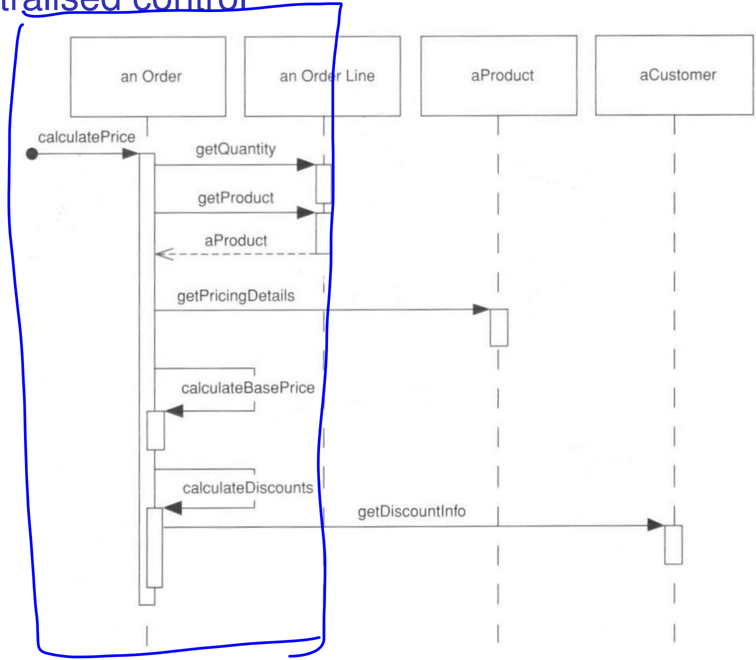
Contents

Centralized control

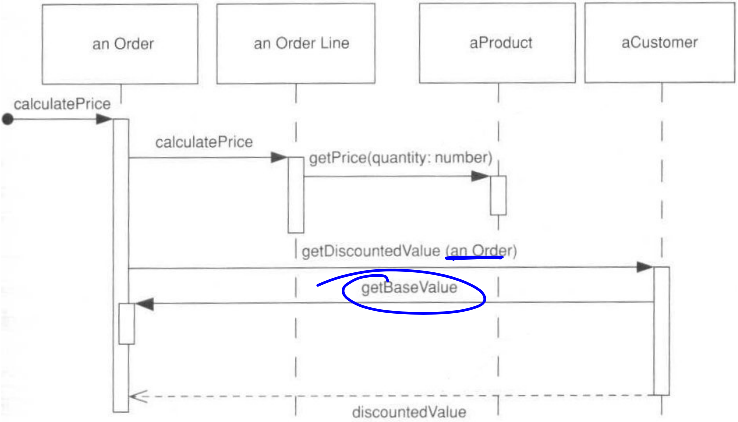
Code smell: Data objects.
Objects without methods



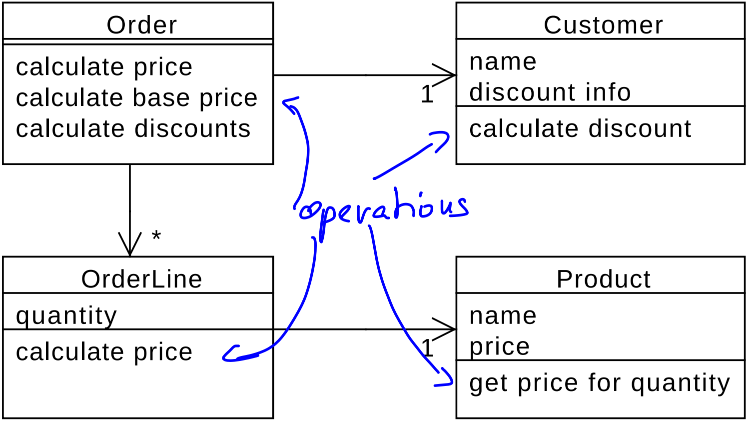
Centralised control



Distributed control



Distributed Control: Class diagram



Centralized vs Distributed control

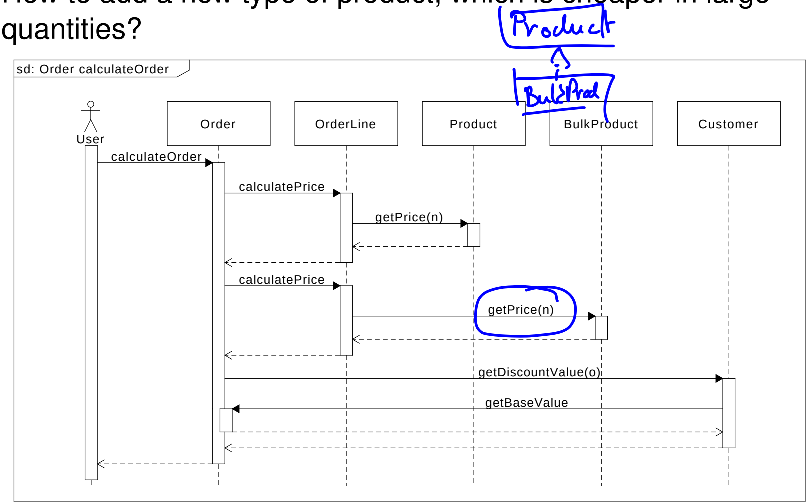
- Centralized control
  - **One method**
  - Data objects
  - $\rightarrow$ procedural programming language
- Distributed control
  - Objects **collaborate**
  - Objects = data *and* behaviour
  - $\rightarrow$ Object-orientation
- Advantage
  - Easy to adapt
  - $\rightarrow$ Design for *change*

Design for Change

How to add a new type of product, which is cheaper in large quantities?

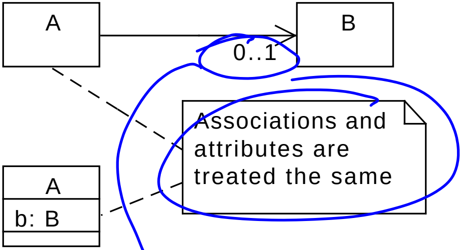
Design for Change

How to add a new type of product, which is cheaper in large quantities?



Contents

Implementing Associations: Cardinality 0..1

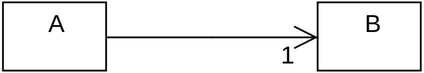


- Field can be null

```
public class A {

   private B b;

   public B getB() {
     return b;
   }

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

null or an object of type B

Implementing Associations: Cardinality 1



- Field may not be null

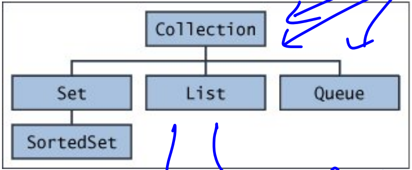
```
public class A {

   private B b = new B(); // 1st way of doing it

   public A(B b) { this.b = b;} // 2nd way

   public B getB() {                  // 3rd way
     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

interface

HashSet

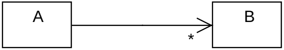


TreeSet

ArrayList/Vector
LinkedList

find implementation
class

Implementing Associations: Cardinality *

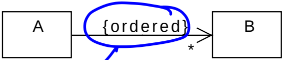


Default: Unordered, no duplicates

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

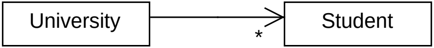
foreach (B b : a.getBs)

code works for empty lists and non empty lists




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

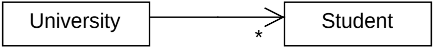
Encapsulation problem: getStudents




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

students.add(hans)
students.remove(olaf)

Encapsulation problem: getStudents

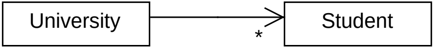



```
University dtu = new University("DTU");
..
Set<Student> students = dtu.getStudents();

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

Solution: getStudents returns an unmodifiable set

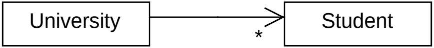
```
public void Set<Student> getStudents() {
   return Collections.unmodifiableSet(students);
}
```

Encapsulation problem: setStudents




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

Encapsulation problem: setStudents

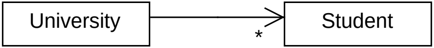



```
University dtu = new University("DTU");
..
Set<Student> students = new HashSet<Student>();
dtu.setStudents(students);

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

Solution: setStudents copies the set

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

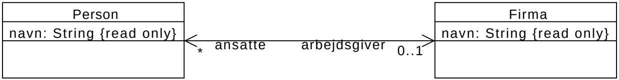
Solution: How to change the association?



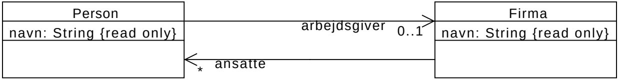

```
public class University {

   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

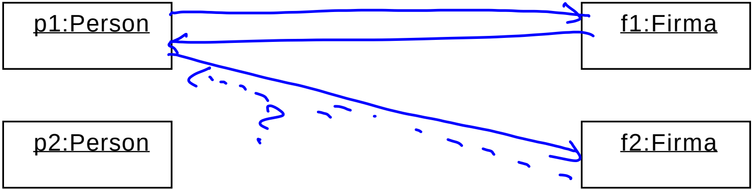


Implemented as two *uni-directional* associations

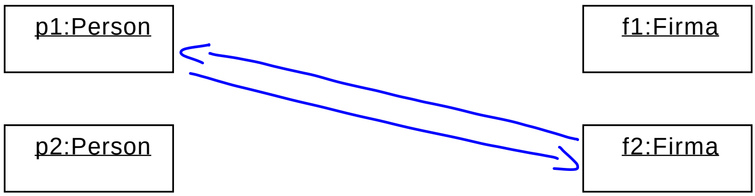


→ Problem of referential integrity

Referential Integrity

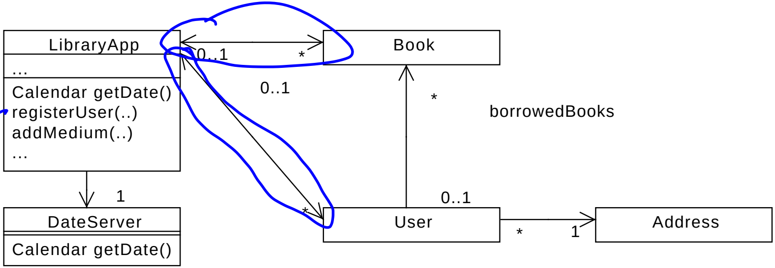


Referential Integrity: setArbejdsgiver



Referential Integrity: addAnsatte

Library application



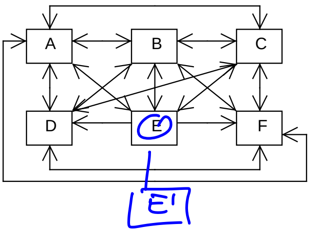

```
public void registerUser(User u) {
    users.add(u);
    u.setLibraryApp(this);
}
```

important to remember

Contents

Low Coupling

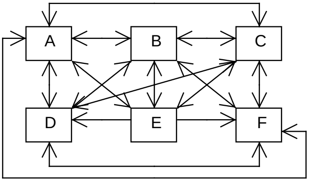
High coupling



A lot of systems depend on one or more systems

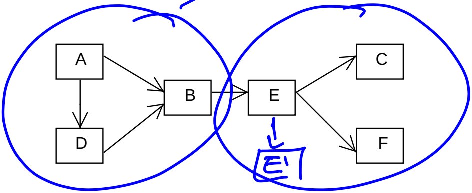
Low Coupling

High coupling



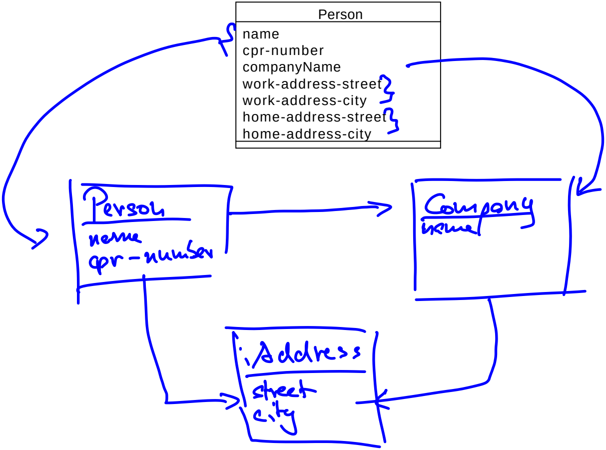
Low coupling

high cohesión within clusters



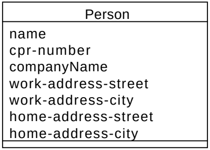
High Cohesion

Low Cohesion

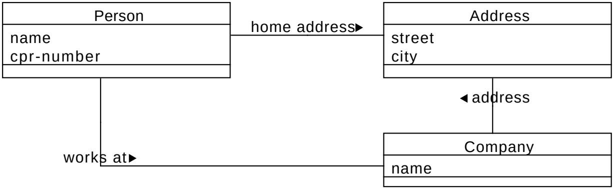


High Cohesion

Low Cohesion



High Cohesion



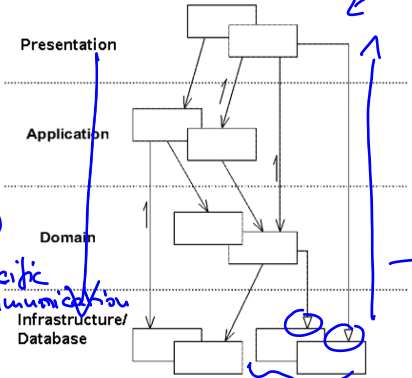
Layered Architecture

high cohesion

no business logic

application specific

business logic



specific communication

low coupling

non-specific communication

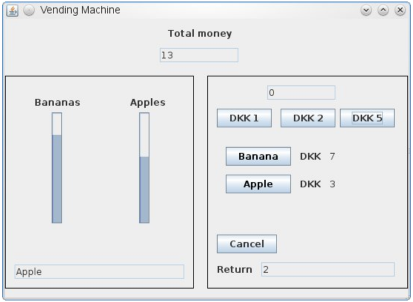
services

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

Example Vending Machine

Two different presentation layers; same application layer

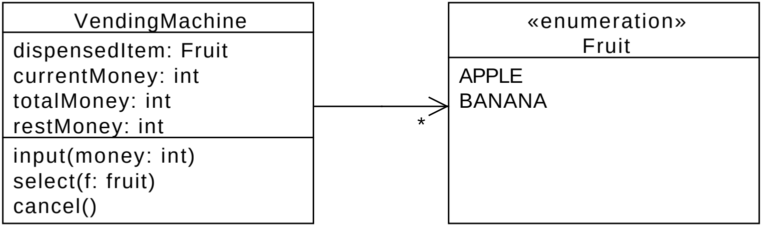
- Swing GUI



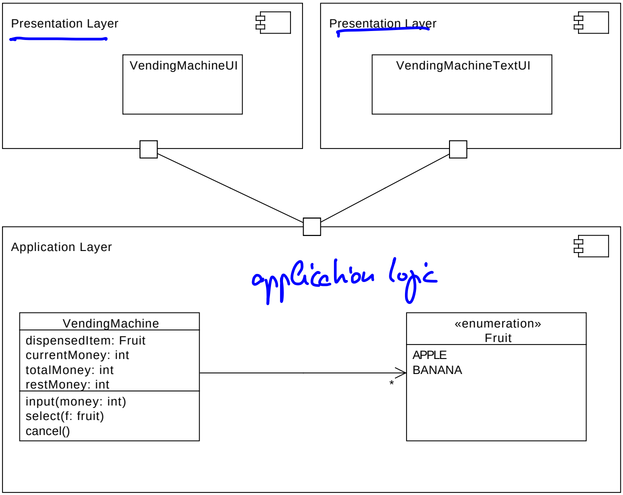
- Command line interface

```
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
```

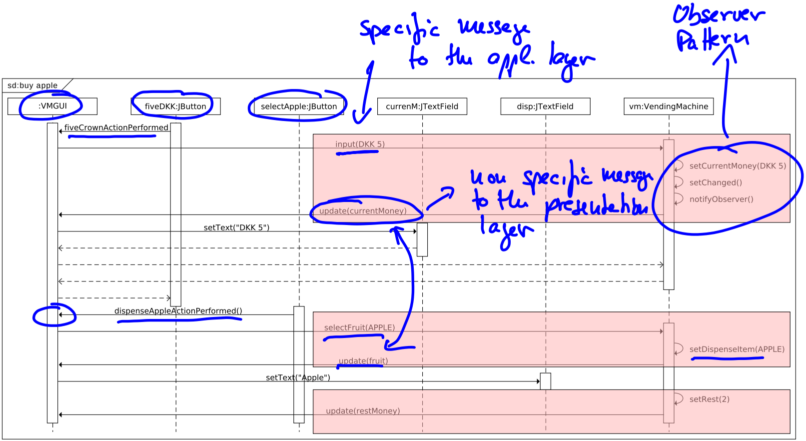
Application Layer



Architecture



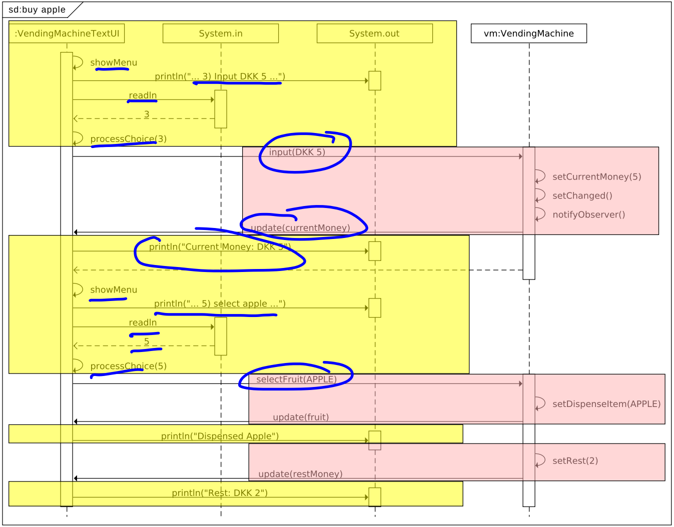
Presentation Layer: Swing GUI



Presentation Layer: Swing GUI

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

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

Presentation Layer: Command Line Interface



Advantages of the separation

1. Presentation layer easily changed
2. Additional presentation layers can be added easily without having to reimplement the business logic
   - mobile app in addition to desktop and Web application
3. Automatic tests: test the application and domain layer: test ”under the GUI”

- Exam project: You should model the application and domain layer with your class diagrams and sequence diagrams

Next Week

- State machines
- Example of a GUI and persistency layer for the library application
- Version control