OO concepts

UML representation

- Objects, Classes, Messages/Methods
- Inheritance, Polymorphism, Dynamic Binding
- Abstract Classes, Abstract Coupling
Abstract Classes and Abstract Coupling
Why abstract classes?

- OO languages are used in many software projects in the same manner as module-oriented languages (Modula-2, Ada) for Lego-style building from separate parts:
  - Classes are a language construct for implementing modules/abstract data types
  - Such software modules can be adapted to new projects by sub-classing
- In order to achieve reusable software architectures, it is essential to employ a skillful combination of subclassing (and thus polymorphism) and dynamic binding in the form of abstract classes.
Abstract classes (I)

- Provide default (general) behavior
  - Only few methods are implemented

- Require subclasses to provide more specific behavior
  - Some method names and their parameters are fixed, but their implementation must be provided by subclasses

- Represent a standardization of the class interface for all subclasses
Abstract classes (II)

- The classes resulted from grouping common characteristics **do not reflect usually objects of the real world**, but an abstractization of them. Therefore one calls these classes abstract classes.

- A further reason for this naming is that **it does not makes sense to generate instances** from such classes. Abstract classes contain “dummy” implementations or no implementations (→ abstract methods) for some methods.
Abstract Coupling (I)

- Other classes are implemented based on abstract classes. The **coupling** between an abstract class A and another class B can take place in several ways:
  - B has an instance variable of the static type A
  - One or more methods of B have a parameter of the static type A
  - B accesses a global variable of the static type A
Abstract coupling (II)

- The classes coupled with an abstract class can work with objects of arbitrary subclasses of the abstract class \textbf{without change}. This is due to polymorphism and dynamic binding.
- The behavior of these components can thus be changed not by changing the components themselves, but by technically clean modifications of the behavior of abstract classes, which is done in their subclasses.

\begin{quote}
\textbf{Abstract classes + abstract coupling = the basis for OO Frameworks (semifinished designs).}
\end{quote}
The main issue is to find good abstractions so that other software components can be realized by building on the abstractions.

Abstract classes evolve typically only in interaction with the classes coupled with them.
Example: reservation system
Frameworks – static view

Framework adaptation

abstract classes (here each with one abstract method)

Framework classes

Payment

Promotion

FreqShopping

EMoney

CreditCard

Framework classes

abstract classes (here each with one abstract method)
Black-Box versus White-Box Framework types

vor der Adaptierung

nach der Adaptierung
Hands-on exercise

Web shop
Case Study Web Shop (I)

- Design a Web shop from which one can buy books over Internet.

- A component should be „the catalog“, which administers the books.
First design:

Katalog

- anzahlDerBuecher: int
  + buchHinzufuegen(buch:Buch): boolean
  + buchLoeschen(buch:Buch): boolean
  + findeBuch(name: String): Buch
  + anzahlDerBuecher(): int

Buch

- autor: String
- preis: float
- ISBNNummer: int
  + anzeigen(): void
  + schickeAn(kunde: Kunde): boolean
Case Study Web Shop (III)

- New requirement: Support also music CDs and computer games.
- The catalog must be extended accordingly.
Case Study Web Shop (IV)

- New design
Collaboration and Sequence Diagrams
Collaboration Diagram (I)

- In a Collaboration diagram there are only simple relationships between objects
  (__________)

- Optionally, the message flow between objects can be represented. (However, sequence diagrams are more suitable in this respect.)
  ______message, ...

Message is: [no:] method()
- method() is as given in the class diagram
- no is an optional number, which defines the order of the method calls.
Collaboration Diagram (II)

Example:

```
1/freieZimmer(Date,Date):Vector ~

:Hotel

1.1/listFrei(Date,Date):boolean ~

Zimmer 1:Hotelzimmer

1.2/listFrei(Date,Date):boolean

Zimmer 2:Hotelzimmer
```

Stefan Resmerita, WS2015
A sequence diagram essentially expresses the same semantics as a collaboration diagram, but it is usually easier to read.

Collaboration diagrams offer the advantage that additional information can be represented (e.g. relations between objects).

Collaboration diagrams can be transformed automatically into sequence diagrams.
Sequence Diagram (II)

Example:

```
Benutzer

Hotel

Zimmer 1
Hotelzimmer

Zimmer 2
Hotelzimmer

freieZimmer(Date,Date):Vector

listFrei(Date,Date):boolean

listFrei(Date,Date):boolean
```

Stefan Resmerita, WS2015
Case Study: Web Shop(I)

- Catalog dynamics:
  - How are products inserted into the catalog?
  - How does the sequence diagram look?
  - How does the collaboration diagram look?
Case Study: Web Shop(II)
Case Study: Web Shop(III)
Use Case Diagrams
Use Case Diagrams: First Artifact

- They help in:
  - Better understanding of requirements
  - Documentation of requirements

- Use case diagrams connect different modelling views of a system
Use Case diagrams (scenarios) represent an **important communication means**, by which the end users of a system and the developers exchange information.

Components of a Use Case diagram:
- System functions (Use Cases)
- Environment (Actors)
- Relations between Use Cases and Actors
Actors

UML representation:

Examples of actors:
- Students who register for courses
- External account system
- Receptionist that serves a hotel reservation system
A Use Case models a dialogue between an actor and the system. It describes which functionality the system offers to an actor.

UML representation:
Use Cases (II)

The following questions are helpful in defining use cases:

- What are the tasks of an actor?
- Will an actor produce information in the system, store it, change it, delete it, or just read it?
- Which use cases will produce, store, change, delete, or read information?
- Does an actor have to be informed about certain events in the system?
- Can all functional requirements be described with the use cases?
Example of the **short description** of a use case:

**Name:** *Student course registration*

This use case is started by a student. For a certain semester, a timetable can be read, changed, or deleted.

**Flow of Events:**
- Can be described in a text document
- Suggestion for a template:
  - Preconditions
  - Main flow and possible sub-flows
  - Alternative flows
Use Cases (IV)

- Example: Selection (by professors) of offered courses

- Pre conditions:
  The use case “Offering Courses” must be achieved before this use case begins.

- Main flow
  This use case begins when a professor logs in the course management system and enters his/her password. The system verifies whether the password is valid (E-1) and requests the professor to select the current term or a future term (E-2). Afterwards the professor selects the desired activity: Add, delete, read, print or terminate.
Use Case (V)

If adding is selected, the sub-flow S-1 is followed:

*Add a course offer* is performed.

...

- **Sub-flows**
  
  S-1: Add a course offer
  
  Course name and number can be entered by appropriate input fields
  
  (E-3): The system connects the professor with the offered course
  
  (E-4): The use case begins again.

...

- **Alternative flows**
  
  (E-1): A wrong name or password was entered. The user can again enter both or terminate the use case.
Use Case Diagram (I)

Use case diagrams show some or all actors and use cases, as well as relations between these entities.

Typically there are:

- A main use case diagram, which graphically depicts the most important actors and main functionality

- Further use case diagrams, e.g.:
  - A diagram that shows all use cases for a certain actor
  - A diagram that shows a use case and all its relations
Use Case Diagram (II)

Example:

- Student
  - LVA-Anmeldung
    - Auswahl von LVAs
      - Professor
    - LVA-Schema anfordern
  - Abrechnungssystem
Use Case Diagram (III)

- The “Uses” relationship shows that functionality in a use case is required in another use case.
- The “Extends” - relationship expresses optional behavior in a use case.

Both relations are represented by a dependence arrow and designated by **stereotyped names**.

In UML there is the so-called **Stereotype** concept, which allows extensions of the fundamental modeling elements. The names of stereotypes are given between `<<` and `>>`.

Stereotypes can be used to describe the relations between use cases.
Use Case Diagram (IV)

Example:

- Professor
- LVA-Schema anfordern
- Auswahl von LVAs
- Benutzerkennung prüfen

<<include>>
Hands-On Exercise (I)

- WebShop

  - Customer:
    - Browses the offer and selects a product
    - Pays with credit card or by bank transfer

  - Seller:
    - Introduces new products into the catalog
    - Removes old products from the catalog
Hands-On Exercise (II)
Hands-On Exercise (II)
Hands-On Exercise (II)

Kunde

Produkt wählen

Produkt kaufen

Registrierung
Hands-On Exercise (II)

Kunde

Produkt wählen

Produkt kaufen

Registrierung

Angebotsliste bearbeiten

Anbieter
Hands-On Exercise (II)
Hands-On Exercise (II)
Hands-On Exercise (II)
Registration: main flow

The use case begins when the user selects the registration option.

The system requests the user to fill out a form with its name, address, age, nickname and password (E-1).

Afterwards the system sends an e-mail to the user to indicate a successful registration.
Hands-On Exercise (IV)

- Alternative flows:
  - E-1: If the form is not correctly completed, the user is requested to fill out the corresponding fields.
  - E-1: If the nickname is already in use, the user is required to provide another nickname.
  - ...
CRC Cards
CRC-Cards (I)

- Which classes are used in order to model a scenario?
- How do these classes work together?
CRC-Cards (II)

- Class, Responsibility, Collaboration
- Beck and Cunningham, OOPSLA’89
  - Developed CRC-Cards in order to be able to descriptively teach the paradigm change from procedural to OO.
  - Direct introduction to the idea of Responsibility Driven Design (Wirfs-Brock 1990).
CRC-Cards (III)

- Pocket-size index Card
- Specifies:
  - Class name
  - Responsibilities
  - Collaborators
Example

<table>
<thead>
<tr>
<th>Hotel</th>
<th>Hotelzimmer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verwalte Kunden</td>
<td>Kunde</td>
</tr>
<tr>
<td>Verwalte Zimmer</td>
<td>Hotelzimmer</td>
</tr>
<tr>
<td></td>
<td>Belegungsplan</td>
</tr>
<tr>
<td></td>
<td>Erstelle Rechnungen</td>
</tr>
<tr>
<td></td>
<td>Date, Reservierung</td>
</tr>
<tr>
<td></td>
<td>Kunde, Date</td>
</tr>
</tbody>
</table>
Advantages

- Communication between designers
- From data containers to responsibilities
- Collaboration between classes is easier to understand.
- The card size determines a granularity of class description that enforces a high level specification of classes.
Packages
and
Package Diagrams
Packages

Packages are mainly used in order to group classes which belong together logically.

UML notation:
Packages (II)

Packages can be nested, in order to be able to better structure complicated architectures.

UML gives the option to list the names of the classes that belong to a package.
Package Diagrams

The following relations between packages can be defined:

- **Dependency:**
  
  It is used to express that classes in a package require classes of another package.

- **Generalization**

  It is used to show that the classes in a package fulfill contracts of the classes of the other package.
Example: E-Commerce Application
State-Transition Diagrams
Notation Elements

State Transition Diagrams show the dynamic behavior of a class instance or of a whole system

- **State symbol:**
  
  ![State symbol diagram]

- **Transition symbol:**
  
  ![Transition symbol diagram]
An action can be written as follows:

- **Method call**: e.g. `converter.ReadFile()`
- **Event triggering**: e.g. `DeviceFailure`
- **Begin activity**: e.g. `Start Converting`
- **Stop activity**: e.g. `Stop Converting`
Example

- **Controller in a greenhouse:**

  ![Controller Diagram]

  - Transition: temperature drop or rise / AdjustTemp()
  - Transition: define climate
  - Transition: terminate climate
  - Transition: sunset / LightOff()
  - Transition: sunrise / LightOn()
  - Transition: terminate climate
  - Transition: temperature drop or rise / AdjustTemp()
Additional notation (I)

- Actions can also be defined within a state:
  - If the system enters the state, if the system exits the state
    - Example: doHeating
  - Transitions can have attached conditions (guards), which are indicated in square brackets.
    - e.g. [restart time >= 5 minutes]
Additional notation (II)

- Conditions can contain also time limits:
  - `timeout (Heating, 30s)`: TRUE, if system is longer than 30 sec. in the state Heating

- States can be nested, if needed:
Additional Notation (III)

- State with history:
  - A state which contains sub-states may have a history mark
  - When the state is exited, the last active sub-state is remembered
  - When the state is re-entered, the last active sub-state is entered

- History is indicated with the decoration $\mathcal{H}$
Example: improve this!