6. Design Phase:
Modeling of System Behavior
Modeling of System Behavior

- How do systems behave?
- Modeling system behavior is so much harder than modeling system structure
- Use cases ⇒
  - scenarios/processes → interaction diagrams
  - workflows → activity diagrams
Specification of Behavior

- Computer Science developed several models for behavior specification
  - logic-oriented models: predicate transformers on pre- and post-conditions
  - graph-oriented models:
    - Petri nets
    - state machines
    - activity diagrams

- Goals
  - Specification of state changes of an object (or an interaction) triggered by an external event or a received signal.
  - Definition of protocols, i.e., legal sequences of operations of a class or an interface.
Petri Net Example: Traffic Light

example from [1]
Petri Net Example in Action
Activity Diagrams

- Elements
  - States
  - Actions
  - Transitions
  - Branches, Merge
  - Concurrency, Synchronization
  - Swimlanes, Object Flow
  - Signals
Role of Activity Diagrams in UML

Legend:
A delegates task
A \(\rightarrow\) B

to B

Use Case Diagrams
Visualization of high-level reaction to events

Activity Diagrams
Workflow presentation
Refinement of timing and sequence

State Diagrams
Interaction Diagrams
Activity Diagrams

- The states are action states or activity states and the transitions are fired (triggered) by the termination of activities.
- In activity diagrams the concept of state does not refer to a static situation, but to named clusters of acts.

Example: Building a house

- Choose site
- Survey house

State

Transition
Transitions

- Transitions describe how to get from one state into another.
- A transition is executed when the previous action or activity is terminated.
Action States and Activity States

- An **action state** in an activity diagram describes an atomic change of a system’s state without temporary fine structure, e.g., an operation call or the calculation of a value. Action states cannot be decomposed.

- An **activity state** describes an enduring activity which can be interrupted and typically is described by a submachine (activity diagram).

![Action state](Switch light on)

![Activity state](Work up bill

index := lookup(e)

Construct House)
Activity States: Actions

- In activity states actions can be executed at the beginning (entry point) or end (exit point) of an activity.

**Point in time**

- Construct House
  - entry / call architect
  - exit / pay architect

**Action**

- Drive Car
  - entry / radio on
  - do / steer car
  - exit / radio off
Branches

- A **branch** appears, if a decision is to be made and depending on a condition several following actions/activities are to be initiated.

```
Branch
[complete]
Condition
[found water]
```

```
Check material
Order material
Build house
Find drinks
```

```
[complete]
[else]
```
Merge

- A **merge** merges optional branches in activity diagrams.

```
<table>
<thead>
<tr>
<th>Check material</th>
</tr>
</thead>
<tbody>
<tr>
<td>[incomplete]</td>
</tr>
<tr>
<td>[incomplete]</td>
</tr>
<tr>
<td>Order material</td>
</tr>
<tr>
<td>[complete]</td>
</tr>
<tr>
<td>Build house</td>
</tr>
<tr>
<td>Merge</td>
</tr>
</tbody>
</table>
```
Concurrency

- Actions and Activities can be executed concurrently and re-synchronized.

Fork

A

B

Find site

Clear Finance

Assign Architect

Plan house construction

[Non-standard house]

\{\text{joinSpec} = \ldots \text{ and } B > 500\}

Join

Build house

Conditional Concurrency
Dynamic Concurrency

- An activity is executed concurrently multiple times where the number of concurrent activities depends on a list of arguments.

Order processing

Proof availability of the ordered goods

Repeatedly executed activity
Synchronization of Concurrent Activities

- Concurrent activities can be synchronized by a synchronization state. The state contains implicit a counter representing the number of waiting activities. The counter can be limited or unlimited. Inherited from State Machines.
Swimlanes

- **Swimlanes** structure actions and activities and show them clustered by execution unit (object or a class, mostly concurrent to other actions/activities).

<table>
<thead>
<tr>
<th>Management</th>
<th>Support</th>
<th>Engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Evaluate Impact</td>
<td>Fix Bug</td>
</tr>
<tr>
<td></td>
<td>Revise Plan</td>
<td>Test Fix</td>
</tr>
<tr>
<td>[ priority = 1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Support</td>
<td>Register Bug</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Release Fix</td>
<td></td>
</tr>
</tbody>
</table>

**Swimlane**
Object Flow

- An **object flow** shows objects with their states being generated or used by actions or activities in activity diagrams.

```
Object flow
```

```
Object state
```

```
Object
```

```
Ordering
```

```
b : Order [new]
```

```
Process order
```

```
b : Order [processed]
```
Object and Control Flow

- Object flow is combined with control flow.

```
Ordering

b : Order [new]

Process order

b : Order [processed]

Control flow

Object

Object flow
```
Deferred Events

- Events which cannot be reacted upon in the current action, can be **deferred** (queued).
- Deferred events can later lead to a reaction. Other events expire if they are not consumed immediately.

```
Process order
Goods not in stock / defer
```

```
Proof availability
Goods not in stock / re-order goods
```
Sending and Receiving Signals

- Signals can be sent during a sequence and objects can wait for a signal.

- Send signal
- Receive signal

Wake Up

Turn on Coffee Pot

Get Cups

Coffee Done

Drink Coffee

Coffee Pot
On Interaction Modeling

- Need to model dynamic aspects of interaction between objects:
  - specifies details of system functionality;
  - functionality is realized as interaction (*message passing*) between objects;
  - uses the elements specified in structure diagrams (e.g. instances of classes).

- Individual interaction diagrams usually show a much smaller part of the system than static diagrams.
Role of Interaction Diagrams in UML

Use Case Diagrams

Interaction Diagrams

Class Diagrams

Activity Diagrams

State Diagrams

Legend:
A delegates task to B

Description of use case behavior

Definition of classes, objects, associations, messages

Workflow presentation

Refinement of single object behavior
Interaction Diagrams in UML

- **Interaction diagrams** are a series of diagrams describing the behavior of an object-oriented system with object interaction.
  - A **sequence diagram** stresses the temporal sequence of object interactions.
  - A **collaboration diagram** stresses the objects taking part in the object interaction.

- For the description the following terms are used:
  - **message** (asynchronous **signal** and synchronous **method call**)
  - **activation** of an object
  - **active object**
Conceptual Modeling of Interactions

- **sender object**: a:A
- **message**: rotate(30.0, origin)
- **receiver object**: b:Point
- **name of the message**: rotate
- **parameter**: (30.0, origin)
- **object reference**
Synchronous (Procedural) Interaction

- Synchronous interaction
  - *message* sent between objects
  - sending object must be *activated* and activates the receiving object. The sender object blocks (i.e., waits) until control returns.

- Asynchronous interaction
  - objects send *signals*
  - *sender stays active* as well but does not block (i.e., continues)
  - *receiver* consumes signal and is activated *in parallel*
Sequence Diagram: Synchronous Interaction

- **actor**
- **object**
  - creation
  - activation
  - destruction
- **life line**
- **object** (anonymous)
  - reserve(date,count)
  - debit(cost,o)
  - bonus()
  - recursive activation
- **TicketDB**
- **Account**
  - create()
Sequence Diagram: Details

- Creates an object `o3:C3` via `create(x)`
- Calls method `call(x)`
- Performs operation `doit(y)`
- Initiates concurrent activation

- Forking of objects

`o1:C1`, `o2:C2`, `o3:C3`, `o4:C4`
Sequence Diagram: Asynchronous Interaction

:Caller

| a | lift receiver |
| b | dial tone |
| c | type(4) |
| d | ... |
| e | type(2) |
| f | end bell tone |

:Agency

| active object (frame bold) |
| Message (without duration) |

:Callee

| bell signal |
| lift receiver |
| end bell signal |

Constraint: \{b-a < 1 sec\}

Named point in time: message with duration (slanted)

Activation
Collaboration Diagram: Example

Sequence numbers denote chronology of events.
- Simple scheme: just number messages
- Decimal scheme: nested numbering of messages where call-stack depth is visible
# Message Labels

<table>
<thead>
<tr>
<th>Message</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>2: display(x, y)</td>
<td>Simple message with sequence number</td>
</tr>
<tr>
<td>1.2.1: p=find(name)</td>
<td>nested call with decimal sequence number</td>
</tr>
<tr>
<td>[x&lt;4] 2: invert(rect)</td>
<td>conditional message (x less than 4) with simple sequence number</td>
</tr>
<tr>
<td>3.1*: update</td>
<td>iteration</td>
</tr>
<tr>
<td>3.2*[i=1..8]: draw(v[i])</td>
<td>iteration</td>
</tr>
</tbody>
</table>
State Diagrams

- Elements:
  - States
  - Transitions
  - Events
  - Composition
  - Concurrency
  - Branch
  - History
Intra-Class Behaviour: State Machines

- Most objects keep an internal state.
  - It usually cannot change in arbitrary ways.
  - There needs to be a precise model that makes explicit which state changes are legal.
  - Well-known concept for this: state machines.

- A state machine ...
  - specifies **states** and **state transitions** of an object
  - specifies possible **actions** in each states
  - is attached to exactly one class
  - is inherited by subclasses (but a concept of the refinement is not defined precisely)

- A state machine thus defines a **protocol** (i.e., legal sequences of operations) for its class.
Elements of State Machines

- **State:**
  - in a specific *state* only certain actions are possible
  - a state is a phase in the life cycle of an object

- **Transition:**
  - concrete actions are associated with *state transitions*

- **Event:**
  - transitions and their actions are triggered by *events*

- ** Guards:**
  - *guards* may prevent transitions from taking place
  - guards are modeled by boolean conditions.
States and State Diagrams

- A **state diagram** is a graph consisting of
  - **states**
    - simple states
    - composite states (states refined by nested state diagrams)
  - **state transitions** connecting the states.
Start and Final States

- Need well defined beginning and end of life-cycle.
- Start state: State transition is executed immediately during the creation of the object.
- Only possible event: create(parameter)

start state

create(init)

active

destroy / send master.destroyed(self)

final state
Transitions

- Transitions connect two states
- Transition can include a **triggering event**, a **guard** and **actions** to be executed.
- Transitions without event and guard are executed **immediately** (possibly after passing through all sub states).

... I decided to use a controller for a secret panel in a Gothic castle. In this castle, I want to keep my valuables in a safe that's hard to find. So to reveal the lock to the safe, I have to remove a strategic candle from its holder, but this will reveal the lock only while the door is closed. Once I can see the lock, I can insert my key to open the safe. For extra safety, I make sure that I can open the safe only if I replace the candle first. If a thief neglects this precaution, I'll unleash a nasty monster to devour him
Example

- **Wait**
  - guard
  - event: candle removed
    - [door closed] / reveal lock
  - action: key turned
    - [candle out] / release killer rabbit

- **Lock**
  - safe closed
  - Open
    - key turned [candle in] / open safe
Time and Change Events

- A **change event** occurs if a specific constraint is fulfilled. The transition is made as soon as the constraint expression evaluates to true.
- A **time event** appears after the expiration of a time period.

```
active  \<change event\> \when\ NumberOfUsers = 0 \<time event\> \after\ 10\ minutes \<change event\> backup
```
Sending Signals

- Signals can be sent to other objects during a transition.

```
after 10 min. / send central.amAlive(self)
```
Composite States

composite state

start state

active

inspection

choice

transaction

outgoing transition from every sub state

transition from one sub state

key in the machine

transition in sub state

wait

card inserted

abort

done

card inserted

entry / read card

exit / eject card

transfer(x)

[ok]
Concurrent Sub States

- In a state several sequences of sub states described by state machines can be performed concurrently.
Concurrent Sub States: Alternative

fork

installation

final test

hardware test

software test

test ports -> test devices

test modules -> test application

operating

join
Architecture Design Models

- An **architecture model** (structure model) is a model of a data processing system describing the static structure of the components of a system.

- For large systems you should model this explicitly.

- **Examples:**
  - network topology (hardware)
  - function tree (software)
  - deployment diagram, package diagram (software)
  - module diagram (hard/software)
  - organization chart in a company model
Example: SAP R/3

- Flexible three-tier client/server-architecture
  - Presentation
  - Application
  - Database
  - distributed presentation
  - remote database access
  - three-layer client/server
  - multi-layer cooperative client/server

LTOOD/OOD - (c) STS
Specification Models in UML

- Architecture diagrams have these elements:
  - Package Diagrams:
    - Classes, Interfaces & Packages
    - Nested Packages
    - Dependencies
  - Deployment Diagrams:
    - Nodes & Components

- Specification models for architecture design:
  - **logical** structures: packages with package diagrams; subsystems; interfaces
  - **physical** structures: components in component diagrams; nodes; deployment diagrams
Package Diagram

- A **package diagram** shows the coherence (dependencies) between different **packages** of the system.

![Package Diagram](image)

- Package A
- Package B
- Package C
- **Client**
- **Supplier**

**Dependencies are not transitive!**
Alternatives for the representation of the content of packages are:

- aggregation
- nesting
- name lists
Subsystems

- Problem: How do I break down a large system into smaller systems?

<table>
<thead>
<tr>
<th>Functional Decomposition</th>
<th>OO Packages</th>
</tr>
</thead>
</table>
| Map the total system on functions and subfunctions, starting with the use case. | - Collect classes into subsystems.  
- Build layers of subsystems.  
- **Abstraction**: Concentrate on essentials.  
- **Locality**: Group together related components (data and algorithm).  
- **Hiding**: Restrict the visibility of details, so that only those parts of a system that need to know the details have access to them. |
Interfaces in Architecture

- Interfaces let you specify the outward appearance.
- They are important in architecture diagrams, because they show how system boundaries are crossed.

```
uses-relation
Openable
interface
```

```
class or component fulfilling the interface
```
Deployment Diagram

- A deployment diagram shows the configuration of a node at runtime as well as the components (instances) and objects residing on it.

- Components not existing as runtime object (instance) should appear in component diagrams only.
Components (1)

- A **component** is a physical, replaceable part of a system containing an implementation which realizes a set of interfaces.

- Components have two aspects:
  - **Code**: A component consists of code, components can contain or use components.
  - **Identity**: A component can have identity and state represented by objects. An object which wants to use services of the component must specify the instance. (e.g., *Bean* reference, DLL *handle*, process, CORBA-*IOR*,...)

**Example**: Spelling checker as component: Identity and state by user dictionary, different versions and languages.
Components (2)

- Notation
  - Dictionary
  - Spelling Checker
  - Synonym Table

- Component with identity
  - myLexicon: Dictionary
    - o:Options
    - u:User Lexicon
    - interfaces
    - object in component
Node

- A **node** is a physical entity existing at runtime
  - represents processing resources
  - has storage and computation capacity
  - On a node objects and components can be live
- Nodes can be computers, humans, or other devices or machines

![Diagram example]

- **mainServer**
  - `c:CustomerDB`
  - `hw :Customer`

- **workstation**
  - `hw :Customer`

- **node**
- **component on node**
- **migrated object**
- **object on node**
Discussion

- Components ...
  - are conceptually and functionally larger than a class.
  - unite behavior and collaboration of a group of classes (see collaboration diagrams)
  - are independent of other components but usually collaborate with them
  - are replaceable with other realizations of the same interface
  - are fundamental building blocks of component-based architectures
  - can be built recursively. A system can be a component on the next higher inspection level.
References

- Petri Nets
  - Examples: http://www.daimi.au.dk/PetriNets/introductions/aalst