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

Red
Green
Orange

Red
Green
Orange

Red
Green
Orange

LTOOD/OOAD – Verified Software Systems
Activity Diagrams

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

Legend:
A delegates task 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**

<table>
<thead>
<tr>
<th>Choose site</th>
<th>Transition</th>
<th>Survey house</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State</strong></td>
<td><strong>Transition</strong></td>
<td><strong>State</strong></td>
</tr>
</tbody>
</table>
Transitions

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

```
<table>
<thead>
<tr>
<th>Start state</th>
<th>Action or Activity</th>
<th>Transition</th>
<th>Final state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chose site</td>
<td></td>
<td></td>
<td>Survey House</td>
</tr>
</tbody>
</table>
```

*LTOOD/OOD – Verified Software Systems*
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
index := lookup(e)
```

```
Activity state
Work up bill
Construct House
```

LTOOD/OOSE – Verified Software Systems
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**
  - Check material
- **Condition**
  - Build house
  - Order material
  - Find drinks

**Keyword**
- [found water] [else]
Merge

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

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

Plan house construction

Conditional Concurrency

Fork

A

B

Find site

Clear Finance

Assign Architect

[Non-standard house]

{joinSpec = ... and B>500}

Join

Build house
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>Evaluate Impact</td>
<td>Register Bug</td>
<td>Fix Bug</td>
</tr>
<tr>
<td>Revise Plan</td>
<td></td>
<td>Test Fix</td>
</tr>
<tr>
<td>[ priority = 1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Object Flow

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

- Object flow is combined with control flow.

```
Control flow

Ordering

b : Order [new]

Process order

b : Order [processed]
```

Object flow

Object
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.
Sending and Receiving Signals

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

Diagram:

1. Wake Up
2. Turn on Coffee Pot
3. Get Cups
4. Coffee Done
5. Drink Coffee
6. Coffee Pot

Legend:
- **Send signal**
- **Receive signal**
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**: Description of use case behavior
- **Interaction Diagrams**: Definition of classes, objects, associations, messages
- **Workflow presentation**: Refinement of single object behavior
- **Class Diagrams**: Definition of classes, objects, associations, messages
- **Activity Diagrams**: Workflow presentation
- **State Diagrams**: Refinement of single object behavior

Legend: A delegates task to B
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

```
rotate(30.0, origin)
```

**sender object**

```
a:A
```

**name of the message**

```
rotate(30.0, origin)
```

**message**

```
b:Point
```

**receiver object**

```
b:Point
```

**parameter**

```
rotate(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**
- **object** (anonymous)
- **:TicketDB**
- **:Account**

Creation: create
Activation: activation
Life Line: life line
Result: result
Message: message
Bonus: bonus
Debit: debit
Recursive Activation: recursive activation
Destruction: destruction

Example:
- create(): Order
- reserve(date, count)
- debit(cost, o)
- bonus()
Sequence Diagram: Details

- **create(x)** from **o1:C1** to **o3:C3**
- **call(x)** from **o3:C3** to **o2:C2**
- **doit(y)** from **o2:C2** to **o3:C3**
- **doit(z)** from **o4:C4**
- **forking** from **o1:C1** to **o3:C3**
- **concurrent activation** from **o4:C4**
Sequence Diagram: Asynchronous Interaction

{b-a < 1 sec}

constraint

named point in time

message with duration (slanted)

activation

:Caller

a

lift receiver

b

dial tone

c

type(4)

...”

c

type(2)

d

bell tone

e

:Agency

Message
(without duration)

bell signal

:Callee

end bell tone

f

end bell tone

lift receiver

date

end bell signal

active object
(frame bold)
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>

Semantics:
- **display(x, y)**: Displays the values of x and y.
- **find(name)**: Searches for the element with the given name.
- **invert(rect)**: Inverts the rectangle.
- **condition [x<4]**: Applies the subsequent message when x is less than 4.
- **iteration**: Indicates a loop or repetition.
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.
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)`

```
create(init)  active  destroy / send master.destroyed(self)
```

```
start state
```

```
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 → Lock
- guard: candle removed
- action: key turned

Wait → Open
- guard: candle removed
- event: key turned

Lock → Open
- event: candle in
- action: key turned
- event: candle out
- action: key turned

safe closed
key turned [candle in] / open safe
key turned [candle out] / release killer rabbit
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.
Sending Signals

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

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

```
«send» amAlive(self)
```

```
receiving object
```

```
central
```

```
active
```

```
active
```

```
alternative representation
```

```
send signal
```
Composite States

- **active**
  - card inserted
  - abort
  - done

- **inspection**
  - ↓[ok]
  - transfer(x)

- **choice**
  - entry / read card
  - exit / eject card

- **transaction**

- **wait**
  - abort

- **composite state**
  - outgoing transition from every sub state
  - key in the machine

- **sub state**
  - transition from one sub state
  - transition in sub state

- **start state**

LTOOD/OOAD – Verified Software Systems 41
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
      - test ports
    - software test
      - test modules
  - test devices
  - test application

- **join**
  - operating
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

Example: SAP R/3
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
A **package diagram** shows the coherence (dependencies) between different packages of the system.

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

- aggregation
- nesting
- name lists

```
« subsystem »
Order System

Order  Position

« subsystem »
Marketing

subpackage

Order System

Customer

class

classes

visibility

« subsystem »
Marketing

+ Order
+ Customer
- Position

stereotype
```
### 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>
<tbody>
<tr>
<td>Map the total system on functions and subfunctions, starting with the use case.</td>
<td>n Collect classes into subsystems.</td>
</tr>
<tr>
<td></td>
<td>n Build layers of subsystems.</td>
</tr>
<tr>
<td></td>
<td>n <strong>Abstraction</strong>: Concentrate on essentials.</td>
</tr>
<tr>
<td></td>
<td>n <strong>Locality</strong>: Group together related components (data and algorithm).</td>
</tr>
<tr>
<td></td>
<td>n <strong>Hiding</strong>: Restrict the visibility of details, so that only those parts of a system that need to know the details have access to them.</td>
</tr>
</tbody>
</table>
Interfaces in Architecture

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

![Deployment Diagram](image)
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**
  
  - component

  ![Diagram of component and related objects]

  - Dictionary
  - Spelling Checker
  - Synonym Table

- **component with identity**

  - myLexicon: Dictionary
    - o : Options
    - u : User Lexicon

  ![Diagram of component with identity]

  - 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 of node, component, and object relationships]
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
  - Introduction:
  - Examples:
    http://www.daimi.au.dk/PetriNets/introductions/aalst