Software Engineering
Prof. Dr. Bertrand Meyer
March 2007 – June 2007

Lecture 15: Modeling with UML

What is modeling?

Building an abstraction of reality
> Abstractions from things, people, and processes
> Relationships between these abstractions

Abstractions are simplifications
> They ignore irrelevant details
> They represent only the relevant details
> What is relevant or irrelevant depends on the purpose of the model

Draw complicated conclusions in the reality with simple steps in the model

Example 1: cat

Example 2: street map

Example 3: atom models in physics

Bohr model
> Nucleus surrounded by electrons in orbit
> Explains, e.g., spectra

Quantum physics
> Position of electrons described by probability distribution
> Takes into account Heisenberg’s uncertainty principle

Why model software?

Software is getting increasingly more complex
> Windows 2000: ~40 millions lines of code
> A single programmer cannot manage this amount of code in its entirety

Code is not easily understandable by developers who did not write it

We need simpler representations for complex systems

Modeling is a means for dealing with complexity
What is a good model?

Intuitively: A model is good if relationships, which are valid in reality R, are also valid in model M.

Definition Interpretation I: \( R \rightarrow M \)

- Mapping of real things in reality \( R \) to abstractions in model M
- Relationship between abstractions in M
- Relationship between real things in R

In a good model this diagram is commutative.

Models of models of models ...

Software development is transformation of models

- \( f_R \): Requirements Elicitation
- \( f_{M1} \): Analysis
- \( f_{M2} \): System Design

Modeling the Real World

- Problem domain
- Abstraction
- Modeling Method
- Continents
- Countries
- Oceans
- Their positions

Modeling example: data modeling

- Bank client
- Tuple of
  - Address
  - Asset class
  - At least one account

Modeling example: object modeling

- Bank client
- Object with
  - Data
  - Operations

The Unified Modeling Language, UML

- UML is a modeling language
  - Using text and graphical notation
  - For documenting specification, analysis, design, and implementation

Importance
- Recommended OMG (Object Management Group) standard notation
- De facto standard in industrial software development

Alternative: Business Object Notation (BON)
- Mainly used in the Eiffel community
**Bit of history**

In 1994-95 Rational Software Corporation hires the Three Amigos:
- Jim Rumbaugh (OMT)
- Grady Booch (Booch method)
- Ivar Jacobson (OOSE method)

In 1996 Rational decides to unify different methods and an international consortium is organized.

In 1997 first version gets standardized (UML 1.0).

Many minor revisions to fix bugs and fill gaps in semantics

In 2004 UML 2.0 gets standardized
- Attempt to define precise semantics
- Current standard

**UML notations**

- Use case diagrams – requirements of a system
- Class diagrams – structure of a system
- Interaction diagrams – message passing
  - Sequence diagrams
  - Collaboration diagrams
- State and activity diagrams – actions of an object
- Implementation diagrams
  - Component model – dependencies between code
  - Deployment model – structure of the runtime system
- Object constraint language (OCL)

**System models**

1. What are the transformations? → Functional Model
   - Create scenarios and use case diagrams
   - Talk to client, observe, get historical records
2. What is the structure of the system? → Object Model
   - Create class diagrams
   - Identify objects, associations and their multiplicity, attributes, operations
3. What is its behavior? → Dynamic Model
   - Create sequence diagrams
   - Show senders, receivers, and sequence of events
   - Create state diagrams (for the interesting objects)

**Dominance of models**

- **Object model**
  - The system has classes with nontrivial states and many relationships between the classes

- **Dynamic model**
  - The model has many different types of events: input, output, exceptions, errors, etc.

- **Functional model**
  - The model performs complicated transformations (e.g., computations consisting of many steps)
**Software Engineering, lecture 15: Modeling in UML**

### UML use case diagrams

**Actors**

- **Actors** represent roles, that is, a kind of user of the system.

**A use case** represents a sequence of interaction for a kind of task.

**Actor is potentially involved in the task**

**System boundaries**

- Client

### Actors

**An actor models an external entity** which communicates with the system.

- **Kind of user**
- **External system**
- **Physical environment**

**An actor** has a unique name and an optional description.

- Client: A person in the train
- GPS satellite: An external system that provides the system with GPS coordinates

### Use case

**A use case represents a kind of task** provided by the system as an event flow.

**A use case consists of**

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

**Use case example: Withdraw**

**Initiating actor:** Client

**Entry condition**

- Client has opened a bank account with the bank and
- Client has received a bank card and PIN

**Exit condition**

- Client has the requested cash or
- Client receives an explanation from the Bankomat about why the cash could not be dispensed

### Use case example: Withdraw event flow

**Actor steps**

1. Authenticate
2. Client selects “Withdraw CHF”
3. Client enters amount
4. Bankomat displays options
5. Bankomat queries amount
6. Bankomat returns bank card
7. Bankomat outputs specified amount in CHF

**System steps**

- Bankomat displays options
- Bankomat queries amount
- Bankomat returns bank card
- Bankomat outputs specified amount in CHF

**Details of authentication**

- Anything missing?

### Reusing use cases

**<<include>> stereotype** to include use cases:

- Reusing common functionality, no duplicates

**<<include>>**

- Withdraw
- Load Cash Card
- Authenticate
- Transfer
Separating variant behavior

Normal case specifies point at which the behavior may diverge (extension point)
Extending case specifies condition under which the special case applies (as entry condition)

Withdraw event flow revisited

Actor steps
System Steps
1. Authenticate (use case Authenticate)
2. Bankomat displays options
3. Client selects "Withdraw CHF"
4. Bankomat queries amount
5. Client enters amount
6. Bankomat returns bank card
7. Bankomat outputs specified amount in CHF
(ext. point: Refuse Withdrawal)

Withdraw event flow revisited

Use case Refuse Withdrawal

Entry Condition:
Entered amount higher than money in account
Exit Condition:
Error message is displayed
System Steps
1. Bankomat displays error message that entered amount is higher than available on account

Generalization and specialization

Factor out common (but not identical) behavior
Child use cases
- Inherit behavior and meaning of the parent use case
- Add or override some behavior
Flow of event:
- Details in textual description of parent use case
- Children describe only how they differ from parent

How to write a use case (summary)

Name of use case
Actors
- Description of Actors involved in use case
Entry condition
- "This use case starts when..."
Flow of events
- Free form, informal natural language
Exit condition
- "This use case terminates when..."
Exceptions
- Describe what happens if things go wrong
Special requirements
- Nonfunctional requirements, constraints
System models

Functional model
  Use case diagrams
Object model
  Class diagrams
Dynamic model
  Sequence diagrams
  State diagrams

Noun-Verb Analysis (Abbott's Textual Analysis)

Use cases represent an external view of the system
No correlation between use cases and classes inside system

Do a textual analysis of problem statement
Take the flow of events and find participating objects in use cases and scenarios
- Nouns are good candidates for classes
- Verbs are good candidates for operations

First create Analysis Object Model
During detailed design refine to implementation classes

Classes

A class encapsulates state (attributes) and behavior (operations)
- Each attribute has a type
- Each operation has a signature
The class name is the only mandatory information

More on classes

Valid UML class diagrams

<table>
<thead>
<tr>
<th>TarifSchedule</th>
<th>zone2price : Table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>getZones() : Enumeration</td>
</tr>
<tr>
<td></td>
<td>getPrice( Zone ) : Price</td>
</tr>
</tbody>
</table>

Corresponding BON diagram

- No distinction between attributes and operations (uniform access principle)

Associations

A link represents a connection between two objects
- Ability of an object to send a message to another object
- Object A has an attribute whose value is B
- Object A creates object B
- Object A receives a message with object B as argument

Associations denote relationships between classes

Optional label
<table>
<thead>
<tr>
<th>Person</th>
<th>works at</th>
</tr>
</thead>
<tbody>
<tr>
<td>Company</td>
<td></td>
</tr>
</tbody>
</table>

Optional roles
| Employee | employer |

Multiplicity of associations

The multiplicity of an association end denotes how many objects the source object can reference
- Exact number: 1, 2, etc. (1 is the default)
- Arbitrary number: * (zero or more)
- Range: 1..3, 1..*

1-to-1 association

| City 1 is capital of | 1 Country |

1-to-many association

| Polygon | 3..* Point |
Association: example

Problem statement:
A stock exchange lists many companies. Each company is uniquely identified by a ticker symbol.

Diagram does not express that ticker symbols are unique

Qualified associations

For each ticker symbol, a stock exchange lists exactly one company

Qualified associations reduce the multiplicity of associations

Navigability

Associations can be directed

Person knows about Company

Company knows about Person

Person and Company know about each other

Aggregation

Aggregation expresses a hierarchical part-of ("has-a") relationship

Used for documentation purposes only

"Think of it as a modeling placebo." – Jim Rumbaugh

Composition

Composition expresses a strong aggregation

Component cannot exist without aggregate
Component may have only one owner

Analogous to concept of Expanded types in Eiffel

Generalization and specialization

Generalization expresses a kind-of ("is-a") relationship

Generalization is implemented by inheritance

Generalization simplifies the model by eliminating redundancy
Stereotypes and conventions

UML provides stereotypes to attach extra classifications

- **<<Entity>>**
  - Account

- **<<Boundary>>**
  - Terminal

- **<<Control>>**
  - Withdrawal

Naming conventions help to distinguish kinds of objects (stereotypes lost during code generation)

- **<<Entity>>**
  - Account

- **<<Boundary>>**
  - Terminal_Boundary

- **<<Control>>**
  - Withdrawal_Control

UML packages

A package is a UML mechanism for organizing elements into groups
- Usually not an application domain concept
- Increase readability of UML models

Decompose complex systems into subsystems
- Each subsystem is modeled as a package

Avoid ravioli models

Don’t put too many classes into the same package: 7 ± 2 (or even 5 ± 2)

Put taxonomies on a separate diagram

System models

- **Functional model**
  - Use case diagrams

- **Object model**
  - Class diagrams

- **Dynamic model**
  - Sequence diagrams
  - State diagrams

Overview

Object model describes structure of system
Dynamic model describes behavior
Purpose: Detect and supply operations (methods) for the object model

- We look for objects that are interacting and extract their “protocol”
  - Sequence diagrams

- We look for objects that have interesting behavior on their own
  - State diagrams
UML sequence diagrams

**Actors and objects:**
- Columns
- Lifelines: dashed lines
- Messages: arrows

**Creatations and destructions:**
- Creation denoted by a message arrow pointing to the object
- In garbage collection environments, destruction can be used to denote the end of the useful life of an object

**Nested messages:**
- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations

**From use cases to sequence diagrams:**
- Sequence diagrams are derived from flows of events of use cases
- An event always has a sender and a receiver
  - Find the objects for each event
- Relation to object identification
  - Objects/classes have already been identified during object modeling
  - Additional objects are identified as a result of dynamic modeling

**Bankomat example: Withdraw event flow**

**Actor steps**
1. Authenticate (use case Authenticate)
2. Bankomat displays options
3. Client selects "Withdraw CHF"
4. Bankomat queries amount
5. Client enters amount
6. Bankomat returns bank card
7. Bankomat outputs specified amount in CHF (ext. point: Refuse Withdrawal)

**System steps**
2. Bankomat displays options
4. Bankomat queries amount
5. Client enters amount
6. Bankomat returns bank card
7. Bankomat outputs specified amount in CHF (ext. point: Refuse Withdrawal)
This diagram shows only the successful case. Exceptional case (Refuse Withdrawal) could go either on another diagram or could be incorporated to this one.

Sequence diagrams show main scenario and “interesting” cases:
- interesting: exceptional or important variant behavior
- Need not draw diagram for every possible case
- would lead to too many diagrams

Impact on object model
For each object that receives an event there is a public operation in the associated class:

- **Entity** objects are accessed by control and boundary objects
- **Entity** objects should never access boundary or control objects

Recommended layout of sequence diagrams
1st column: Actor who initiated the use case
2nd column: Boundary object
3rd column: Control object that manages the rest of the use case

Heuristics for sequence diagrams
Creation of objects:
- Control objects are created at the initiation of a use case
- Boundary objects are often created by control objects

Access of objects:
- Entity objects are accessed by control and boundary objects
- Entity objects should never access boundary or control objects
  - Easier to share entity objects across use cases
  - Makes entity objects resilient against technology-induced changes in boundary objects
**Stair structure**

The dynamic behavior is distributed

- Each object delegates some responsibility to other objects
- Each object knows only a few of the other objects and knows which objects can help with a specific behavior

**Fork or stair?**

Object-oriented supporters claim that the stair structure is better

- The more the responsibility is spread out, the better

Choose the stair (decentralized control) if

- The operations have a strong connection
- The operations will always be performed in the same order

Choose the fork (centralized control) if

- The operations can change order
- New operations are expected to be added as a result of new requirements

**Sequence diagrams summary**

Sequence diagrams represent behavior in terms of interactions

Complement the class diagrams (which represent structure)

Useful

- To find missing objects
- To detect and supply operations for the object model

**System models**

**Functional model**

Use case diagrams

**Object model**

Class diagrams

**Dynamic model**

Sequence diagrams

State diagrams

**State-dependent behavior**

Objects with extended lifespan often have state-dependent behavior

- Typical for control objects
- Less often for entity objects
- Almost never for boundary objects

Examples

- Withdrawal: has state-dependent behavior
- Account: has state-dependent behavior (e.g., locked)
- Display: does not have state-dependent behavior

State-dependent behavior is modeled only if necessary

**Events, actions, and activities**

**Event:** Something that happens at a point in time

- Typical event: Receipt of a message
- Other events: Change event for a condition, time event

**Action:** Operation in response to an event

- Example: Object performs a computation upon receipt of a message

**Activity:** Operation performed as long as object is in some state

- Example: Object performs a computation without external trigger
UML state diagrams

State diagram relates events and states for a class
Often called "state chart" or "state chart diagram"

Example 1: states of copy objects

Implementation has to take care of unexpected messages, e.g., return in state "on shelf"
- Specify precondition
- Report an error, throw an exception

Example 2: states of book objects

Events can have different effects depending on guard conditions
Some state diagrams do not have end markers

Example 3: ticket vending machine

An abstraction of the attribute values of an object
A state is an equivalence class of all those attribute values and links that do not need to be distinguished as far as the control structure of the class or the system is concerned

Example: State of a book
- A book is either borrowable or not
- Omissions: bibliographic data
- All borrowable books are in the same equivalence class, independent of their author, title, etc.

Nested state diagrams

Activities in states can be composite items that denote other state diagrams
Sets of substates in a nested state diagram can be denoted with a superstate
- Avoid spaghetti models
- Reduce the number of lines in a state diagram
Example: superstate

State diagram vs. sequence diagram

State diagrams help to identify
  - Changes to an individual object over time

Sequence diagrams help to identify
  - The temporal relationship between objects
  - Sequence of operations as a response to one or more events

Practical tips for dynamic modeling

Construct dynamic models only for classes with significant
dynamic behavior
  - Avoid “analysis paralysis”

Consider only relevant attributes
  - Use abstraction if necessary

Look at the granularity of the application when deciding on
actions and activities

Reduce notational clutter
  - Try to put actions into superstate boxes (look for
identical actions on events leading to the same state)

UML is not Enough

Contracts in UML

Urs is married to Sile, Sile is married to Beat, and Beat is
not married at all

A valid instantiation of the class diagram!

Associations describe relations between classes
UML is not Enough (cont’d)

Urs is married to Sile, who is only eleven

A valid instantiation of the class diagram!

Class diagrams do not restrict values of attributes

Expressing Contracts

Natural language

- Advantage: Easy to understand and use
- Disadvantage: Ambiguous

Mathematical notation

- Advantage: Precise
- Disadvantage: Difficult for normal customers

Contract language

- Formal, but easy to use
- Examples: Eiffel, JML

Object Constraint Language – OCL

The contract language for UML

Used to specify

- Invariants of objects
- Pre- and postconditions of operations
- Guards (for instance, in state diagrams)

Special support for

- Navigation through UML class diagram
- Associations with multiplicities

Form of OCL Invariants

Constraints can mention

- self: the contextual instance
- Attribute and role names
- Side-effect free methods (stereotype <<query>>)  
- Logical connectives
- Operations on integers, reals, strings, sets, bags, sequences
- Etc.

OCL Invariants: Example

A savings account has a non-negative balance

Checking accounts are owned by adults

OCL Invariants: Contexts

Checking accounts are owned by adults

Accounts are owned by adults

Customers are adults
Collections

OCL provides three predefined collection types

- Set, Sequence, Bag

Common operations on collections

- `size()` - Number of elements in the collection
- `includes(object)` - True iff the object is an element
- `isEmpty()` - True iff collection contains no elements
- `exists(expression)` - True iff expression is true for at least one element
- `forAll(expression)` - True iff expression is true for all elements

Generating Collections

Explicitly enumerating the elements

- Set `{1, 7, 16}`
- `self.accounts`

By navigating along 1:n associations

- Navigation along a single 1:n association yields a Set
- Navigation along a single 1:n association labeled with the constraint `{ ordered }` yields a Sequence

Collections
- Account
- (ordered) Customer
- amount
- accounts
- age

Example: Multiplicity Zero or One

Person

- `spouse` with multiplicity `0..1`

```
context Person
inv:
spouse->size() = 1 implies
age >= 16 and spouse.spouse = self and spouse <> self
```

Example: Composite Pattern

Leaf

- A composite is the parent of its components
- A component is contained in its parent composite

```
context Composite
inv:
children->forAll(c | c.parent = self)

context Component
inv:
parent->size() = 1 implies
parent.children->includes(self)
```

Contracts in Eiffel: Method Specifications

Method precondition

- Must be true before the method is executed

Method postcondition

- Must be true after the method terminates
- Old expressions is used to refer to values of the pre-state

```
class interface ACCOUNT feature
withdraw(a: INTEGER) is
require a >= 0;
ensure GetBalance() = old(GetBalance()) - a;
end
```

Example: Quantification and Type Information

```
context Customer
inv:
age <= 18 implies
accounts->forAll(a | a.oclIsKindOf(SavingsAccount))
```

Example: Subtype relation

```
context Customer
inv:
age <= 18 implies
accounts->forAll(a | a.oclIsKindOf(SavingsAccount))
```

Example: Contract constraint

```
context Customer
inv:
age <= 18 implies
accounts->forAll(a | a.oclIsKindOf(SavingsAccount))
```
Pre- and Postconditions in OCL

```
context Account::Withdraw( a: int )
pre: a >= 0
post: GetBalance() = GetBalance@pre() - a
```

**result** is used to refer to return value

Pre- and postconditions can be named (like in Eiffel)

Suffix @pre is used to refer to prestate values

Alternative Notation

Contracts can be depicted as notes in diagrams
- Stereotypes instead of keywords inv, pre, post

```
Account
Amount: int
AccountId: int

Deposit( a: int )
Withdraw( a: int )
GetBalance( ): int

<<invariant>>
AccountId >= 0

<<precondition>>
amount >= 0

GetBalance() = GetBalance@pre() - a
```