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
Architecture vs. Software Engineering

- Problem
- Design
- Implementation
- Program
- Model
- Reverse engineering
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 $I: R \rightarrow M$

In a good model this diagram is commutative.
Software development is transformation of models

Models of models of models ...

M: Requirements Elicitation
M1: Analysis
M2: System Design
M: Functional Model
R: Object Model
R: Subsystem Decomposition

fM, fM1, fM2, fR
Modeling the Real World

Problem domain

- Continents
- Countries
- Oceans
- Their positions
- ...

Representation of model

Abstraction

Modeling Method

Model of problem
Modeling example: data modeling

Abstraction

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

Modeling Method

ER-Diagram

Client

- Address
- Asset class

possesses

1

n

Account

Balance

Account No.
Modeling example: object modeling

Object with
- Data
- Operations

UML Class Diagram

<table>
<thead>
<tr>
<th>Address</th>
<th>1</th>
<th>1</th>
<th>Client</th>
<th>1</th>
<th>1..</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asset class</td>
<td>Balance</td>
<td>Account No.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Unified Modeling Language, UML

- **Modeling language** = language for describing models (mostly models of software)
- **Model in UML** = graph
  - vertices = entities
  - edges = relations
- Models can be represented in different formats (e.g. graphical, xmi)
- **Diagrams** are graphical representation of parts of a model
The Unified Modeling Language, UML

Authors: The Three Amigos

- Grady Booch
- James Rumbaugh
- Ivar Jacobson

Importance

- Recommended OMG (Object Management Group) standard notation
- De facto standard in industrial software development
A bit of history (or why “unified”?)
Uses of UML

- **Specification:** the language is supposed to be simple enough to be understood by the clients.
- **Visualization:** models can be represented graphically.
  - Plain text < text with pictures < comics
- **Design:** the language is supposed to be precise enough to make code generation possible.
- **Documentation:** the language is supposed to be widespread enough to make your models understandable by other developers.
What UML is not about?

- **Programming language**
  - this would bound the language to a specific computing architecture
  - however code generation is encouraged
- **Software development process**
  - however the authors had their own process in mind: RUP (Rational Unified Process)
- **CASE tool specification**
  - however tools do exist: Sun, IBM Rose, Microsoft Visio, Borland Together e.t.c
Notation changes in UML 2.0

- One notation for all structural entities - rectangle with a **stereotype**:

  ![Component 1]

- Special notation for provided and required interfaces:

  ![Interface between Component 1 and Component 2]
Canonical diagrams in UML 2.0

- **Functional**
  - Use case diagram (requirements, client’s point of view)

- **Static structure**
  - Class diagram (classes and relationship between them)
  - Object diagram (relationship between objects at an interesting point in runtime)
  - Composite structure diagram (internal structure of a class)
  - Package diagram (packages and relationship between them)

- **Implementation diagrams**
  - Component diagram (physical components and relationship between them)
  - Deployment diagram (assigning components to nodes)
Canonical diagrams in UML 2.0

- **Behavioral**
  - State diagram (object lifecycle)
  - Activity diagram (= flowchart, algorithm description)
- **Interaction diagrams**
  - Sequence diagram (message passing, ordered in time)
  - Communication diagram (message passing)
  - Interaction overview diagram (= activity diagram with interaction diagrams in nodes)
  - Timing diagram (focus on timing constraints)
The three views

- **Functional:** What does the system do?
  - Interaction between the system and external entities
  - Diagrams: use case

- **Structural:** What does the system consist of?
  - Parts (modules) of the system and relationship between them
  - Static (no notion of time)
  - Diagrams: class, component, deployment

- **Behavioral:** How does the system work?
  - Notion of time or sequence of events/actions
  - Diagrams: state, activity, sequence, communication
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).
Aspects covered in lecture

Functional model

*Use case diagram* (requirements, client’s point of view)

Object model

*Class diagram* (classes and relationship between them)

Dynamic model

*Sequence diagram* (message passing, ordered in time)
*State diagram* (object lifecycle)

+ Object constraint language (OCL)
UML use case diagrams

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

**Actor** is potentially involved in the task.

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

**System boundaries**

**Withdraw**
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 Bancomat about why the cash could not be dispensed
Use case example: Withdraw event flow

**Actor steps**
1. Authenticate
3. Client selects “Withdraw CHF”
5. Client enters amount

**System Steps**
2. Bancomat displays options
4. Bancomat queries amount
6. Bancomat returns bank card
7. Bancomat outputs specified amount in CHF

Anything missing?

Details of authentication

Exceptional cases
<<include>> stereotype to include use cases: reusing common functionality, no duplicates
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

<table>
<thead>
<tr>
<th>Actor steps</th>
<th>System Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Authenticate <em>(use case Authenticate)</em></td>
<td>2. Bancomat displays options</td>
</tr>
<tr>
<td>5. Client enters amount</td>
<td>6. Bancomat returns bank card</td>
</tr>
<tr>
<td>7. Bancomat outputs specified amount in CHF <em>(ext. point: Refuse Withdrawal)</em></td>
<td></td>
</tr>
</tbody>
</table>

Referring to included use case

Listed as extension point
Use case Refuse Withdrawal

**Entry Condition:**
Entered amount higher than money in account

**Exit Condition:**
Error message is displayed

**System Steps**

1. Bancomat 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
Use case diagrams

- **Entities:**
  - actors
  - use cases

- **Relations:**
  - association between an actor and a use case
  - generalization between actors
  - generalization between use cases
  - dependencies between use cases

- **Comments:**
  - system boundaries
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
Aspects covered in lecture

Functional model

Use case diagram (requirements, client’s point of view)

Object model

Class diagram (classes and relationship between them)

Dynamic model

Sequence diagram (message passing, ordered in time)
State diagram (object lifecycle)

+ Object constraint language (OCL)
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
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>
</tr>
</thead>
<tbody>
<tr>
<td>zone2price</td>
</tr>
<tr>
<td>getPrice()</td>
</tr>
<tr>
<td>getZones()</td>
</tr>
</tbody>
</table>

Corresponding BON diagram

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

- Most widely used relation on class diagrams
- In general means that classes know about each other - their objects can send each other messages (call operations, read attributes)

**Special cases:**
- Class A has an attribute of type B
- Class A creates instances of B
- Class A receives a message with argument of type B

- Mostly are binary, but can be N-ary
- Can have different adornments that express additional information
**Association adornments (1)**

- **Name** (possibly with direction)
  
  ![Diagram with associations](Person works for Company)

- **Multiplicity** of an end - how many objects of the class take part in the relation
  
  - **1-to-1**
    
    ![Diagram with associations](City 1 is capital of 1 Country)
  
  - **1-to-many**
    
    ![Diagram with associations](Polygon 3..* Point)
  
  - **many-to-many**
    
    ![Diagram with associations](Person * works for * Company)
Association adornments (2)

- **Aggregation** - part-of relation between objects
  - an object can be part of multiple objects
  - part can be created and destroyed independently of the aggregate

  ![Diagram](Diagram.png)

- **Composition** - strong aggregation
  - an object can only be part of a single object
  - exists only together with the aggregate

  ![Diagram](Diagram.png)
Association adornments (3)

- **Role** of an end: name + interface

- **Navigability** of an end - whether the objects at this end can be accessed through this association
Relations in UML

- **Dependency** - changing the independent entity may influence the dependent one

- **Association** - entities are directly connected (e.g. aggregation)

- **Generalization** - an entity is a special case of another entity, they satisfy the substitution principle

- **Implementation** - an entity is an implementation of another entity (e.g. a class and an interface)
Association adornments (4)

- **Ordering** of an end - whether the objects at this end are ordered
- **Changeability** of an end - whether the set of objects at this end can be changed after creation
- **Qualifier** of an end - is an attribute that allows retrieving one object from the set at this end

![Diagram](image-url)
Generalization and specialization

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

Generalization is implemented by inheritance

- The child classes inherit the attributes and operations of the parent class

Generalization simplifies the model by eliminating redundancy
Stereotypes and conventions

UML provides stereotypes to attach extra classifications

<table>
<thead>
<tr>
<th><code>&lt;Entity&gt;</code></th>
<th><code>&lt;Boundary&gt;</code></th>
<th><code>&lt;Control&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>Terminal</td>
<td>Withdrawal</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th><code>&lt;Entity&gt;</code></th>
<th><code>&lt;Boundary&gt;</code></th>
<th><code>&lt;Control&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>Terminal_Boundary</td>
<td>Withdrawal_Control</td>
</tr>
</tbody>
</table>
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: $7 \pm 2$ (or even $5 \pm 2$)
Put taxonomies on a separate diagram

Account
- Amount
- AccountId
- Deposit()
- Withdraw()
- GetBalance()

Savings Account
- Withdraw()

Checking Account
- Withdraw()

Mortgage Account
- Withdraw()
Class diagrams

- **Entities**: 
  - classes (and interfaces)

- **Relations**: 
  - association between classes 
  - generalization between classes 
  - implementation between a class and an interface 
  - dependencies between classes 

---

**Diagram:**

- `IChief` with methods: `send_petition`
- `ISubordinate` with methods: `report`
- `Position` with methods: `occupy`, `free`, `request_report`, `send_petition`
- `Manager` with methods: `request_report`, `send_petition`
- `IPosition` with methods: `occupy`, `free`
- `Department` with methods: `<<instantiate>>`, `<<call>>`, `Report`
UML 2.0: “Chupa-chups” notation

**Entities:**
- classes (and interfaces)

**Relations:**
- association between classes
- generalization between classes
- implementation between a class and an interface
- dependencies between classes
Aspects covered in lecture

Functional model
  Use case diagram (requirements, client’s point of view)

Object model
  Class diagram (classes and relationship between them)

Dynamic model
  Sequence diagram (message passing, ordered in time)
  State diagram (object lifecycle)

+ Object constraint language (OCL)
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
Sequence diagrams

- **Entities:**
  - objects (including instances of actors)

- **Relations:**
  - message passing

- **Sugar:**
  - lifelines
  - activations
  - creations
  - destructions
  - frames

- Actors and objects: columns
- Lifeline: dashed line
- Messages: arrows
- Activation s: narrow rectangles
Nested messages

The source of an arrow indicates the activation which sent the message.
An activation is as long as all nested activations.
Creation and destruction

Creation is 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.
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
Bancomat example: Withdraw event flow

**Actor steps**

1. Authenticate  
   *(use case Authenticate)*

3. Client selects “Withdraw CHF”

5. Client enters amount

**System Steps**

2. Bancomat displays options

4. Bancomat queries amount

6. Bancomat returns bank card

7. Bancomat outputs specified amount in CHF  
   *(ext. point: Refuse Withdrawal)*
select (wthdrCHF)
initWthdr (cur)
queryAmount()
check(amount, cur)
withdraw(amount, cur)
dispense(amount, cur)
ejectCard()
taken
displayConfirmation()
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
Interaction frames

:Container

:Processor

:Item

loop
[for each item]

process()

alt
[value < 100]

increase()

[else]

decrease()
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

- **Actor:** Client
- **Boundary:** Terminal, Display
- **Control:** Withdrawal
- **Entity:** Account
The **dynamic behavior is placed in a single object**, usually a control object. It knows all the other objects and often uses them for direct queries and commands.
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
Interaction of the models: Modeling process

Modeling usage

Functional view

Structural view

Behavioral view

Modeling structure

Modeling behavior
Example: modeling the ATM

- **Modeling usage:** use case diagram
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 ATM about why the cash could not be dispensed
Withdraw event flow revisited

**Actor steps**

1. Authenticate  
   *(use case Authenticate)*
2. Client selects “Withdraw CHF”
3. Client enters amount

**System Steps**

1. ATM displays options
2. ATM queries amount
3. ATM returns bank card
4. ATM outputs specified amount in CHF  
   *(ext. point: Refuse Withdrawal)*
From use cases to sequence diagrams

Client

Terminal

Display

Account

select (withdraw CHF)

initWithdraw (cur)

:Withdrawal

queryAmount()

select (option)

withdraw (amount)

dispense(amount, cur)

taken
ejectCard()

check(amount, cur)

okay

withdraw (amount, cur)

displayConfimation(

From use cases to sequence diagrams
... and further to class diagrams

- Add a **class** for each object on the diagram
- For each object that receives an event add a **public operation** in the associated class

```
check (amount, cur)
withdraw (amount, cur)
```

**Identify additional classes** (e.g. for message arguments, messages with no receivers)
- Problem text analysis may help (nouns may correspond to **classes**, verbs – to **operations**)

```
:Account

check (int, Currency): boolean
withdraw (int, Currency)
```
Aspects covered in lecture

Functional model
   Use case diagram (requirements, client’s point of view)

Object model
   Class diagram (classes and relationship between them)

Dynamic model
   Sequence diagram (message passing, ordered in time)
   State diagram (object lifecycle)

+ Object constraint language (OCL)
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
State diagrams

- **Entities:**
  - states: name, activity, entry/exit action

- **Relations:**
  - transitions between states: event, condition, action

![State diagram with entities and relations](image)
State diagrams: example 1

<table>
<thead>
<tr>
<th>Copy</th>
<th>1..*</th>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>borrow return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>borrow return</td>
</tr>
</tbody>
</table>

Copy

On loan
entry / book.borrow()

On shelf
entry / book.return()

Book

Not borrowable

Borrowable

borrow() [ last copy ]

return() [ not last copy ]
Example 2: ticket vending machine

- **Idle** entry/clearbalance
- **CollectMoney**
  - insCoin(amount) / add to balance
  - [change < 0]
- **selectTicket(tkt)**
- **TicketSelected** entry / compute change
  - [change = 0]
- **ExactlyPaid**
  - do / dispense ticket
  - [change > 0]
- **OverPaid**
  - do / dispense change
  - [change < 0]
- [ticket dispensed]
State

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.
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
State diagrams: example composite state

TrafficLight

Off

TurnOn

On

TurnOff

Blinking

SwitchOn

Working

SwitchOff

after 30 sec

Red

after 3 sec

Yellow

Red

after 5 sec

Green

Yellow

after 45 sec
Example: superstate

**Idle**
- entry / clear balance

**CollectMoney**
- insCoin(amount) / add to balance
- [change < 0]

**TicketSelected**
- selectTicket(tkt)
- entry / compute change
- [change = 0]

**ExactlyPaid**
- do / dispense ticket
- [ticket dispensed]

**OverPaid**
- do / dispense change
- [change > 0]

**Superstate**
Expanding the superstate

Transitions from other states to the superstate enter the first substate of the superstate

Transitions to other states from a superstate are inherited by all the substates (state inheritance)
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

- Create **component** and **deployment** diagrams only for large, distributed systems
- Create **state** diagrams only for classes with complex, interesting behavior (usually classes representing entities from the problem domain or performing control)
- Create **activity** diagrams for complex algorithms and business processes (not for every operation)
- Create **sequence** and **communication** diagrams for nontrivial collaborations and protocols (not for every scenario)
- Don’t put too much information on a diagram
- Choose the level of abstraction and maintain it
Entities in UML

- **Structural**
  - **Class** - a description of a set of object with common **attributes** and **operations**
  - **Interface** - a set of **operations** (a service), provided by a class or component
  - **Actor** - an external entity that interacts with the system
  - **Use case** - a description of a set of scenarios (sequences of events and actions) that produce a result, significant for some actor
  - **Component** - physically replaceable artifact that provides a certain set of interfaces
  - **Node** - a hardware resource
Entities in UML

- **Behavioral**
  - **State** - a period in an object lifecycle, during which the object is satisfying some property, performing an activity or waiting for an event
  - **Activity** - a state, in which the object is doing some work (instead of just passively waiting for an event)

- **Grouping**
  - **Package** - a group of model elements (maybe including other packages)

- **Annotating**
  - **Note** - arbitrary text comment attached to a model
Relations in UML

- **Dependency** - changing the independent entity may influence the dependent one

- **Association** - entities are directly connected (e.g. aggregation)

- **Generalization** - an entity is a special case of another entity, they satisfy the substitution principle

- **Implementation** - an entity is an implementation of another entity (e.g. a class and an interface)
Aspects covered in lecture

Functional model
  Use case diagram (requirements, client’s point of view)

Object model
  Class diagram (classes and relationship between them)

Dynamic model
  Sequence diagram (message passing, ordered in time)
  State diagram (object lifecycle)

+ **Object constraint language (OCL)**
UML is not Enough

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.
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

\[ \text{spouse expresses "is married to"} \]

\[ \text{spouse: Person } \rightarrow \text{ Person} \]
\[ \text{spouse } = \text{spouse}^{-1} \]
\[ \text{spouse } \cap \text{id } = \emptyset \]

\[ \forall p : \text{Person}: p \in \text{dom}(\text{spouse}) \Rightarrow \]
\[ \text{spouse}(p) \in \text{dom}(\text{spouse}) \land \]
\[ p \neq \text{spouse}(p) \land \]
\[ p = \text{spouse}(\text{spouse}(p)) \]

\[ \text{spouse } /= \text{Void implies} \]
\[ \text{spouse } /= \text{Current and} \]
\[ \text{spouse}.\text{spouse } = \text{Current} \]
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.

The context is an instance of a class in the UML diagram

`context Person inv:
  self.age >= 0`

Declarations an invariant

A boolean constraint
A savings account has a non-negative balance

Checking accounts are owned by adults
## OCL Invariants: Contexts

- **Checking accounts are owned by adults**
  - **CheckingAccount** context
    - `context CheckingAccount inv:
      self.owner.age >= 18`

- **Accounts are owned by adults**
  - **Account** context
    - `context Account inv:
      self.owner.age >= 18`

- **Customers are adults**
  - **Customer** context
    - `context Customer inv:
      self.age >= 18`
# Collections

OCL provides three predefined collection types

- **Set**, **Sequence**, **Bag**

Common operations on collections

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size()</code></td>
<td>Number of elements in the collection</td>
</tr>
<tr>
<td><code>includes(object)</code></td>
<td>True iff the object is an element</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>True iff collection contains no elements</td>
</tr>
<tr>
<td><code>exists(expression)</code></td>
<td>True iff expression is true for at least one element</td>
</tr>
<tr>
<td><code>forAll(expression)</code></td>
<td>True iff expression is true for all elements</td>
</tr>
</tbody>
</table>
Generating Collections

Explicitly enumerating the elements

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

```
Set \{ 1, 7, 16 \}
```

```
self.accounts
```

```
Account

* \{ ordered \}

<table>
<thead>
<tr>
<th>accounts</th>
</tr>
</thead>
</table>

Customer

<table>
<thead>
<tr>
<th>age</th>
</tr>
</thead>
</table>
Example: Multiplicity Zero or One

<table>
<thead>
<tr>
<th>Person</th>
<th>spouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>0..1</td>
</tr>
</tbody>
</table>

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

self can be omitted
spouse used as set
spouse used as object
Example: Quantification and Type Information

context Customer inv:
age \leq 18 \implies accounts->forall( a \mid a.oclIsKindOf( SavingsAccount ) )

∀a∈accounts: a.oclIsKindOf( SavingsAccount )
A composite is the parent of its components.

A component is contained in its parent composite.

**context** Composite inv:

\[
\text{children}\to\forall\ (c \mid c.\text{parent} = \text{self})
\]

**context** Component inv:

\[
\text{parent}\to\text{size}(\ ) = 1 \text{ implies } \text{parent.children}\to\text{includes}(\ \text{self})
\]
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

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

Pre- and Postconditions in OCL

Context specifies method signature

result is used to refer to return value

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

Contracts can be depicted as notes in diagrams

- Stereotypes instead of keywords `inv`, `pre`, `post`

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amount: int</td>
</tr>
<tr>
<td>AccountId: int</td>
</tr>
<tr>
<td>Deposit( a: int )</td>
</tr>
<tr>
<td>Withdraw( a: int )</td>
</tr>
<tr>
<td>GetBalance( ): int</td>
</tr>
</tbody>
</table>

```
<<invariant>>
AccountId >= 0
```

```
<<precondition>>
a >= 0
```

```
<<postcondition>>
GetBalance( ) = GetBalance@pre( ) - a
```