Software Engineering
Spring Semester 2008

Slides: Based on KSE06 - With kind permission of Peter Müller

Lecture 16+17: 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: $R \rightarrow M$

\[
\begin{array}{ccc}
M & \xrightarrow{f_M} & M \\
\uparrow & & \uparrow \\
R & \xrightarrow{f_R} & R
\end{array}
\]

$I$: Mapping of real things in reality $R$ to abstractions in model $M$

$f_M$: Relationship between abstractions in $M$

$f_R$: Relationship between real things in $R$

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

Models of models of models ...

I: Requirements Elicitation
I₁: Analysis
I₂: System Design

R → M → M₁ → M₂ → R

fᵣ
fₘ
fₘ₁
fₘ₂
Modeling the Real World

- Continents
- Countries
- Oceans
- Their positions
- …
Modeling example: data modeling

Bank client

ER-Diagram

Client
- Address
- Asset class

possesses

1

Account
- Balance
- Account No.

n

Abstraction

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

Modeling Method

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

Modeling Method

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

Modeling Method

Tuple of
- Address
- Asset class
- At least one account
Modeling example: object modeling

Abstraction

Object with

- Data
- Operations

UML Class Diagram

Bank client

Account

<table>
<thead>
<tr>
<th>Address</th>
<th>1</th>
<th>Client</th>
<th>1</th>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Asset class</td>
<td></td>
<td>Balance</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Account No.</td>
</tr>
</tbody>
</table>
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:
- UML mostly criticized for lack of 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)
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)
System models

Functional model
  Use case diagrams

Object model
  Class diagrams

Dynamic model
  Sequence diagrams
  State diagrams
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
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
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
3. Client selects “Withdraw CHF”
5. Client enters amount

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

Details of authentication
Exceptional cases

Anything missing?
<<include>> stereotype to include use cases:
reusing common functionality, no duplicates
Separating variant behavior

<<extend>> stereotype to provide special case

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

System Steps
2. Bankomat displays options
4. Bankomat queries amount
6. Bankomat returns bank card
7. Bankomat outputs specified amount in CHF *(ext. point: Refuse Withdrawal)*

Referring to included use case
Listed as extension point

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. 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
The set of all use cases specify the \textbf{complete functionality} of the \textit{system} and its \textit{environment}.
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
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>TarifSchedule</td>
<td></td>
<td>getPrice( Zone ) : Price</td>
</tr>
<tr>
<td>zone2price : Table</td>
<td></td>
<td>getZones( ) : Enumeration</td>
</tr>
<tr>
<td>getPrice( Zone ) : Price</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More on classes

Valid UML class diagrams

<table>
<thead>
<tr>
<th>TarifSchedule</th>
<th>TarifSchedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>zone2price</td>
<td></td>
</tr>
<tr>
<td>getPrice</td>
<td></td>
</tr>
<tr>
<td>getZones()</td>
<td></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
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
         City
         `-> Country
```

1-to-many association

```
Polygon   3..* 3..*
     `-> Point
```

City is a capital of Country, and Polygon is composed of multiple Points.
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.

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

- Special form of association
- Objects can simultaneously be part of several aggregates

Used for documentation purposes only

- No formal information
- Use with care!

"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

Aggregation and composition can be documented like other associations

- Multiplicity, label, roles
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>&lt;&lt;Entity&gt;&gt;</th>
<th>&lt;&lt;Boundary&gt;&gt;</th>
<th>&lt;&lt;Control&gt;&gt;</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>&lt;&lt;Entity&gt;&gt;</th>
<th>&lt;&lt;Boundary&gt;&gt;</th>
<th>&lt;&lt;Control&gt;&gt;</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 ± 2 (or even 5 ± 2)
Put taxonomies on a separate diagram

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

Savings Account
- Withdraw()

Checking Account
- Withdraw()

Mortgage Account
- Withdraw()
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

- Activations: narrow rectangles
- Actors and objects: columns
- Lifelines: dashed lines
- Messages: arrows

Time

- :Client
- :Terminal

- insertCard( )
- insertPIN( )
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
Bankomat example: Withdraw event flow

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

**System Steps**
2. Bankomat displays options
4. Bankomat queries amount
6. Bankomat returns bank card
7. Bankomat outputs specified amount in CHF (ext. point: Refuse Withdrawal)
<<Entity>>: Account

<<Entity>>: Client

<<Boundary>>: Terminal

select ( wthdrCHF )

<<Control>>: Withdrawal

initWthdr ( cur )

queryAmount( )

select ( option )

select ( amount )

wthdr ( cur )

check( amount, cur )

okay

withdraw( amount, cur )

dispense( amount, cur )

displayConfimation( )

ejectCard( )

taken

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]

[else]

increase()

decrease()
Impact on object model

For each object that receives an event there is a **public operation** in the associated class.

Identify additional objects and classes

- In the example: sink for dispense message (CashDispenser)
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

1st column:
- Client

2nd column:
- <<Boundary>>: Terminal
- <<Boundary>>: Display
- <<Entity>>: Account

3rd column:
- <<Control>>: Withdrawal
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
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**
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
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

- **Idle**
  - [ticket dispensed]
  - entry / clear balance

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

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

- **ExactlyPaid**
  - [change dispensed]
  - do / dispense ticket
  - [change = 0]

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

Superstate

Idle
entry / clear balance

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

TicketSelected
selectTicket( tkt )
[ change = 0 ]

ExactlyPaid
do / dispense ticket
[ change = 0 ]

OverPaid
do / dispense change
[ change > 0 ]

[ ticket dispensed ]
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 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)
Contracts in UML
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
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

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

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

```ocl
class Person
  inv:
    self.age >= 0

context Person
```

The context is an instance of a class in the UML diagram
Declares an invariant
A boolean constraint
OCL Invariants: Example

A savings account has a non-negative balance

Checking accounts are owned by adults

context SavingsAccount inv:
  self.amount >= 0

context CheckingAccount inv:
  self.owner.age >= 18
OCL Invariants: Contexts

Checking accounts are owned by adults

\[ \text{context CheckingAccount inv:} \]
\[ \text{self.owner.age} \geq 18 \]

Accounts are owned by adults

\[ \text{context Account inv:} \]
\[ \text{self.owner.age} \geq 18 \]

Customers are adults

\[ \text{context Customer inv:} \]
\[ \text{self.age} \geq 18 \]
## 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

\[
\text{Set \{ 1, 7, 16 \}}
\]

\[
\text{self.accounts}
\]

\[
\begin{array}{ccc}
\text{Account} & \ast & \text{Customer} \\
\text{amount} & \text{accounts} & \text{age}
\end{array}
\]

\{ ordered \}
**Example: Multiplicity Zero or One**

<table>
<thead>
<tr>
<th>Person</th>
<th>spouse</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0..1</td>
</tr>
<tr>
<td>age</td>
<td></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 <= 18 implies
accounts->forall( a | a.oclIsKindOf( SavingsAccount ) )

∀a∈accounts: a.oclIsKindOf( SavingsAccount )
Example: Composite Pattern

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
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)
Alternative Notation

Contracts can be depicted as notes in diagrams

- **Stereotypes instead of keywords** \( \text{inv}, \text{pre}, \text{post} \)

<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>

- **Precondition**
  \( a \geq 0 \)

- **Invariant**
  \( \text{AccountId} \geq 0 \)

- **Postcondition**
  \( \text{GetBalance()} = \text{GetBalance}@\text{pre}() - a \)