Contents & Goals

Last Lecture:
• (Mostly) completed discussion of modelling structure.

This Lecture:
• Educational Objectives:
  - Capabilities for following tasks/questions.
  - Discuss the style of this class diagram.
  - What's the difference between reflective and constructive descriptions of behaviour?
  - What's the purpose of a behavioural model?
  - What does this State Machine mean? What happens if I inject this event?
  - Can you please model the following behaviour.

Content:
• For completeness: Modelling Guidelines for Class Diagrams
• Purposes of Behavioural Models
• Constructive vs. Reflective
• UML Core State Machines (first half)

OCL Constraints in (Class) Diagrams

Semantic of a Class Diagram

Let $\mathcal{BV}/\mathcal{BW}$ be a set of class diagrams. We say, the semantics of $\mathcal{BV}/\mathcal{BW}$ is the signature it induces and the set of OCL constraints occurring in $\mathcal{BV}/\mathcal{BW}$, denoted:

$$\mathcal{C}_2: \mathcal{BV}/\mathcal{BW} \Rightarrow \langle \mathcal{CB}(\mathcal{BV}/\mathcal{BW}), \text{Inv}(\mathcal{BV}/\mathcal{BW}) \rangle$$

Given a structure $\mathcal{BW}$ of $\mathcal{CB}(\mathcal{BV}/\mathcal{BW})$ (and thus of $\mathcal{BV}/\mathcal{BW}$), the class diagrams describe the system states $\Sigma_{\mathcal{BW}/\mathcal{CB}}$. Of those, some satisfy $\text{Inv}(\mathcal{BV}/\mathcal{BW})$ and some don't. We call a system state $\sigma \in \Sigma_{\mathcal{BW}/\mathcal{CB}}$ consistent if and only if $\sigma \models \text{Inv}(\mathcal{BV}/\mathcal{BW})$.

In pictures:

$\mathcal{BV}/\mathcal{BW} = \{ \text{CD}_1, \ldots, \text{CD}_n \}$

signature $\mathcal{CB}(\mathcal{BV}/\mathcal{BW})$

invariants $\text{Inv}(\mathcal{BV}/\mathcal{BW})$
A UML model is an image or pre-image of a software system. A set of class diagrams within invariants \( \text{Inv}(\text{BV/BW}) \) describes the structure of system states. Together with the invariants it can be used to state:

- **Pre-image**: Dear programmer, please provide an implementation which uses only system states that satisfy \( \text{Inv}(\text{BV/BW}) \).
- **Post-image**: Dear user/maintainer, in the existing system, only system states which satisfy \( \text{Inv}(\text{BV/BW}) \) are used.

(The exact meaning of "use" will become clear when we study behaviour—intuitively: the system states that are reachable from the initial system state(s) by calling methods or firing transitions in state-machines.)

**Example**: highly abstract model of traffic lights controller.

### TLCtrl

- `red`: Bool
- `green`: Bool
- `not(red and green)`

**Correct**

**SemanticsofOCLBooleanOperations**

<table>
<thead>
<tr>
<th>Boolean Operation</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>false</code></td>
<td><code>false</code></td>
</tr>
</tbody>
</table>

**A.2.2 Common Operations On All Types**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>not</code></td>
<td><code>true</code> if the input is <code>false</code>, <code>false</code> if the input is <code>true</code></td>
</tr>
</tbody>
</table>

**Addendum:** SemanticsofOCLBooleanOperations

**Object Constraint Language, v2.0**

**Table A.2** - Semantics of boolean operations

<table>
<thead>
<tr>
<th><code>b_1</code></th>
<th><code>b_2</code></th>
<th><code>b_1 and b_2</code></th>
<th><code>b_1 or b_2</code></th>
<th><code>b_1 xor b_2</code></th>
<th><code>b_1 implies b_2</code></th>
<th><code>not b_1</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>false</code></td>
<td><code>false</code></td>
<td><code>false</code></td>
<td><code>false</code></td>
<td><code>false</code></td>
<td><code>false</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>false</code></td>
<td><code>true</code></td>
<td><code>false</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>true</code></td>
<td><code>false</code></td>
<td><code>false</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>true</code></td>
<td><code>false</code></td>
<td><code>true</code></td>
<td><code>false</code></td>
</tr>
</tbody>
</table>

**DesignGuidelinesfor(Class)Diagram**

Be careful whose advice you buy, but, be patient with those who supply it.

Baz Luhrmann/Mary Schmich

(partly following [Ambler, 2005])

**Main and General Modelling Guidelines**

(admittedly: trivial and obvious)

Begoodtoyouraudience.

"Imagine you're given your diagram \( D \) and asked to conduct task \( T \).

- Can you do \( T \) with \( D \)? (semantics sufficiently clear? all necessary information available?...)
- Does doing \( T \) with \( D \) cost you more nerves/time/money... than it should? (syntactical well-formedness? readability? intention of deviations from standard syntax clear? reasonable selection of information? layout?...)

In other words:

- the things most relevant for \( T \), do they stand out in \( D \)?
- the things less relevant for \( T \), do they disturb in \( D \)?
Main and General Quality Criteria

• Q: When is a class diagram a good diagram?
  • A: If it serves its purpose/makes its point.

Examples for purposes and points and rules of thumb:

• Analysis/Design
  • Realizable, no contradictions
  • Abstract, focused, admitting degrees of freedom for (more detailed) design
  • Platform independent – as far as possible but not (artificially) far away

• Implementation/A
  • Close to target platform (C0, C1 are easy for Java, C are more expensive – other way around for RDB)

• Implementation/B
  • Complete, executable

• Documentation
  • Right level of abstraction: “If you’ve only one diagram to spend, illustrate the concepts, the architecture, the difficult part”
  • The more detailed the documentation, the higher the probability for regression “outdated/wrong documentation is worse than no documentation.”

General Diagramming Guidelines

[Ambler, 2005] (Note: “Exceptions prove the rule.”)

• 2.1 Readability
  • 1. – 3. Support readability of lines
  • 4. Apply consistently sized symbols
  • 9. Minimize the number of bubbles
  • 10. Include white space in diagrams
  • 13. Provide a notational legend

• 2.2 Simplicity
  • 14. Show only what you have to show
  • 15. Prefer well-known notation over exotic notation
  • 16. Large vs. small diagrams
    • 18. Content first, appearance second

• 2.3 Naming
  • 20. Set and (23. Consistently) follow effective naming conventions

• 2.4 General
  • 24. Indicate unknowns with question marks
  • 25. Consider applying color to your diagrams
  • 26. Apply color sparingly

Class Diagram Guidelines

[Ambler, 2005]

• 5.1 General Guidelines
  • 88. Indicate visibility only on design models (in contrast to analysis models)

• 5.2 Class Style Guidelines
  • 96. Prefer complete singular nouns for class names
  • 97. Name operations with strong verbs
  • 99. Do not model scaffolding code
    • [Except for exceptions]
ClassDiagram Guidelines

[103. Never Show Classes with Just Two Compartments]
[104. Label Uncommon Class Compartments]
[105. Include an Ellipsis (...) at the End of an Incomplete List]
[107. List Operations/Attributes in Order of Decreasing Visibility]

5.3 Relationships

[112. Model Relationships Horizontally]
[115. Model a Dependency When the Relationship is Transitory]
[117. Always Indicate the Multiplicity]
[118. Avoid Multiplicity " ∗ "]
[119. Replace Relationship Lines with Attribute Types]

5.4 Associations

[127. Indicate Role Names When Multiple Associations Between Two Classes Exist]
[129. Make Associations Bidirectional Only When Collaboration Occurs in Both Directions]
[131. Avoid Indicating Non-Navigability]
[133. Question Multiplicities Involving Minimums and Maximums]

5.6 Aggregation and Composition

exercises

Example: Modelling Games

Task: Game Development

Task: develop a video game.

Genre: Racing.

Rest: open, i.e. Degrees of freedom:

Exemplary choice: 2D-Tron

• simulation vs. arcade

• platform (SDK or not, open or proprietary, hardware capabilities...)

• graphics (3D, 2D, ...)

• number of players, AI

min. 2, AI open

• controller

open (later determined by platform)

• game experience

minimal: main menu and game...
Stocktaking...

Have:

- Meanstomodelthe structure ofthesystem.
  - Classdiagramsgraphically,conciselydescribesetsofsystemstates.
  - OCLexpressionslogicallystate constraints/invariantsonsystemstates.

Want:

- Meanstomodelbehaviourofthesystem.
  - Meanstodescribehowsystemstates evolveovertime,thatis,todescribesetsof sequences \( \sigma_0, \sigma_1, \ldots \in \Sigma \) ofsystemstates.

WhatCanBePurposesofBehaviouralModels?

(Wewilldiscussthisinmoredetail inLecture22.)

Example: Pre-Image

- Image (theUMLmodelissupposedtobetheblue-printforasoftwaresystem).

Adescriptionofbehaviourcouldservethefollowingpurposes:

- Require Behaviour.
  - "Systemdefinitelydoesthis" "This sequenceofinsertingmoneyandrequestingandgettingwatermustbe possible." (Otherwisethesoftwareforthevendingmachineiscompletelybroken.)

- Allow Behaviour.
  - "Systemdoessubsetofthis" "Afterinsertingmoneyandchoosingadrink,thedrinkisdispensed(ifinstock)." (Iftheimplementationinsistsontakingthemoneyfirst,that'safairchoice.)

- Forbid Behaviour.
  - "Systemneverdoesthis" "This sequenceofgettingboth,awaterandallmoneyback,mustnotbepos- sible." (Otherwisethesoftwareisbroken.)

Note:thelattertwoaretriviallysatisfied bydoingnothing...

Constructive vs. Reflective Descriptions

[Harel,1997]proposestodistinguishconstructiveandreflectivedescriptions:

- "Alanguageis constructive ifitcontributestothedynamicsemantics ofthemodel. Thatis,itsconstructscontaininformation neededin executingthemodelorintranslatingitintoexecutable code."
  - Aconstructivedescriptiontells how thingsare computed(whichcan thenbedesiredorundesired).

- "Otherlanguagesare reflective or assertive, andcan beused bythesystem modelertocapturepartsofthethinkingthatgointobuildingthe model–behaviorincluded–,toderiveand presentviewsofthemodel, staticallyorduringexecution,ortosetconstraintson behaviorin preparationforverification."
  - Areflectivedescriptiontells what shallorshallnotbe computed.

Note: Nosharp boundaries!
UML provides two visual formalisms for constructive description of behaviors:

- Activity Diagrams
- State-Machine Diagrams

We (exemplary) focus on State-Machines because

- somehow "practice proven" (in different flavors),
- prevalent in embedded systems community,
- indicated useful by [Dobing and Parsons, 2006] survey, and
- Activity Diagram's intuition changed (between UML 1.x and 2.x) from transition-system-like to petri-net-like...

---

Example state machine:

\[
\begin{align*}
s &= 1 \\
E &\{ n \neq \emptyset \} \\
\text{on} &\{ x \} \\
&= x + 1; \\
\text{on} &\{ \emptyset \} \\
&= 0
\end{align*}
\]