Contents & Goals

Last Lecture:
• Behavioural Features
• State Machines Variation Points
• Inheritance in UML: concrete syntax
• Liskov Substitution Principle - desired semantics

This Lecture:
• Educational Objectives:
  • What's the Liskov Substitution Principle?
  • What is late/early binding?
  • What is the subset, what the uplink semantics of inheritance?
  • What’s the effect of inheritance on LSCs, State Machines, System States?
  • What’s the idea of Meta-Modelling?
• Content:
  • Two approaches to obtain desired semantics
  • The UML Meta Model

Easy: Static Typing

Given:

\[ C_1 \xrightarrow{f} D_1 \]

\[ C_2 \xrightarrow{f} F \]

Wanted:

• \( x > 0 \) also well-typed for \( D_1 \)
• assignment \( itsC_1 := itsD_1 \) being well-typed
• \( itsC_1.f(0), itsC_1 \lnot F \) being well-typed (and doing the right thing).

Approach:
• Simply define it as being well-typed, adjust system state definition to do the right thing.

Notions (from category theory):
• invariance,
• covariance,
• contravariance.

We could call, e.g. a method, sub-type preserving, if and only if it
• accepts more general types as input (contravariant),
• provides a more specialised type as output (covariant).

This is a notion used by many programming languages—and easily type-checked.
Late Binding

What transformer applies in what situation?

(Early compile-time binding.)

f not overridden in D
C f(): Int
D C 0
someC

f overridden in D
C f(): Int
D f(): Int
value of someC/someD

someC -> f() C::f()
f
C::f()
u1

someD -> f() D::f()
f
D::f()
u2

– 21 – 2014-02-05 – Slatebind –

Late Binding in the Standard and Programming Languages

• In the standard, Section 11.3.10, “CallOperationAction”:
  “Semantic Variation Points
  The mechanism for determining the method to be invoked as a
  result of a call operation is unspecified.”

[OMG, 2007b, 247]

• In C++,
  • methods are by default “(early) compile-time binding”,
  • can be declared to be “late binding” by keyword “virtual”,
    • the declaration applies to all inheriting classes.

• In Java,
  • methods are “late binding”;
  • there are patterns to imitate the effect of “early binding”

Exercise: What could have driven the designers of C++ to take that approach?

Note: late binding typically applies only to methods, not to attributes.
(But: getter/setter methods have been invented recently.)

Back to the Main Track: “...tell the difference...” for UML

With only early binding...

• we’re done (if we realize it correctly in the framework).
• then if we’re calling method f of an object u,
  • which is an instance of D with C ⪯ D
  • via a C-link,
  • then we (by definition) only see and change the
    C-part.
  • We cannot tell whether u is a C or an D instance.
  Sowe immediately also have behavioural/dynamic subtyping.

Difficult: Dynamic Subtyping

C f(Int): Int
D f(Int): Int

• C::f and D::f are type compatible, but D is not necessarily
  a subtype of C.

• Examples: (C++)
  int C::f(int)
  {
    return 0;
  }
  
  int D::f(int)
  {
    return 1;
  }
  
  int C::f(int)
  {
    return (rand() % 2);
  }
  
  int D::f(int x)
  {
    return (x % 2);
  }

Sub-Typing Principles Cont’d

In the standard, Section 7.3.36, “Operation”:
“Semantic Variation Points
[...] When operations are redefined in a specialization, rules regarding
invariance, covariance, or contravariance of types and preconditions
determine whether the specialized classifier is substitutable for its more
general parent. Such rules constitute semantic variation points with
respect to redefinition of operations.”

[OMG, 2007a, 106]

• So, better: call a method sub-type preserving, if and only if it
  (i) accepts more input values (contravariant),
  (ii) on the old values, has fewer behaviour (covariant).

Note: This (ii) is no longer a matter of simple type-checking!

• And not necessarily the end of the story:
  • One could, e.g. want to consider execution time.
  • Or, like [Fischer and Wehrheim, 2000], relax to “fewer observable
    behaviour”, thus admitting the sub-type to do more work on inputs.

Note: “testing” differences depend on the granularity of the semantics.

• Related: “has a weaker pre-condition,” (contravariant),
  “has a stronger post-condition.” (covariant).
Note: the old setting coincides with the special case where
\( \forall v \in v \cdot \triangleright C \). The greatest context of a variable \( x, v \) :: \( C \) is given by
expression \( \sigma \cdot v \). This is, for completeness, a more general case than
the usual one. The idea is that (by late binding) only the state machine of the most
•
restricted class shall be set to a copy of the original one.

Example has been applied to all OCL expressions and all actions.

Recall Preliminaries: Expression Normalisation

Domain Inclusion Structure

Ensuring Sub-Typing for State Machines
Construct a method call expression, which is applied to an instance of

\[ \text{Example.} \]

\[ \sigma(\text{expr}, v) \]

\[ \text{Let } \] k \[ \text{ be a } UML \text{ model, and } \] ν \[ \text{ be a structure.} \]

\[ \text{Late-binding: transformers (DomainInclusion)} \]

\[ \text{Nonlate-binding: semanticsofMethodCalls} \]

\[ \text{Well-formed method call expression, and} \]

\[ \text{Transformers (DomainInclusion)} \]

\[ \text{SatisfyingOCLConstraints (DomainInclusion)} \]

\[ \text{Well-formed method call expression, and} \]

\[ \text{Transformers (DomainInclusion)} \]

\[ \text{SatisfyingOCLConstraints (DomainInclusion)} \]
The provided text is a page from a document discussing concepts related to Uplink Structure, System State, and Typing in the context of model checking and state machines.

- **Exercise:** Have an experiment to test the domain of identities.
- **Idea:** Continue with the existing definition of structure, system state, and typing.
- **Definition:** The structure remains unchanged, e.g., rewrite (C₁ > C₂, ξ) = ([1, 2], (ς, σ)) in the context (E, σ_u, u), for each pair (ς, σ_u) where $ς$ ∈ $U$, i.e., disjoint sub-classes shall execute super-class's state machine (unless overridden).
- **Note:** Apply a different pre-processing to make appropriate use of that instance.
- **Definition:** If no unique smallest class exists, then there exists (by definition) a stable instance (ς, ε, σ_u), ∈ $U$, i.e., disjoint sub-classes shall execute super-class's state machine (unless overridden).
not type-compatible!

\[d \mapsto \sigma \mapsto \tau \mapsto \]

more precise, the values of the attributes 

\(\sigma \mapsto \rightarrow \)

\(\end{align}

\(c \cdot \)

\(d \rightarrow \)

\(d \mapsto \ast \)

\(\cup\)

\(\sigma \mapsto \rightarrow \)

\(c \cdot \)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)

\(\cup\)

\(d \mapsto \ast \)
**IdentityDowncast with Uplink Semantics**

- **Recall (C++)**:
  ```
  D d;
  C∗ cp = &d;
  D∗ dp = (D∗) cp;
  ```

- **Problem**: We need the identity of the D whose C-slice is denoted by cp.

- **One technical solution**:
  - Give up disjointness of domains for one additional type comprising all identities, i.e. have all ∈ CV/CBW, (all) = ∪ CV/BW (C) =  .
  - In each ⪯-minimal class have associations "most spec" pointing to most specialised slices, plus information of which type that slice is.
  - Then downcast means, depending on the most spec type (only finitely many possibilities), going down and then up as necessary, e.g. switch (most spec type) { case C: dp = cp -> most spec -> uplink Dn -> uplink D1 -> uplink D; ...

**DomainInclusion vs. Uplink Semantics**

- **Note**: The uplink semantics views inheritance as an abbreviation:
  - We only need to touch transformers (create)—and if we had constructors, we didn't even need that (we could encode the recursive construction of the upper slices by a transformation of the existing constructors.)
  - So:
    - Inheritance doesn't add expressive power.
    - And it also doesn't improve conciseness so dramatically.

- **As long as we're "early binding", that is...**

**Meta-Modelling: Idea and Example**

- **Meta-Modelling is one major prerequisite for understanding** the standard documents [OMG, 2007a, OMG, 2007b], and the MDA ideas of the OMG.
  - The idea is simple:
    - If a modelling language is about modelling things, and if UML models are and comprise things, then why not model those in a modelling language?
  - In other words: Why not have a model M_U such that the set of legal instances of M_U is the set of well-formed (!) UML models.

- **For example, let's consider a class.**
  - A class has (on a superficial level)
    - a name,
    - any number of attributes,
    - any number of behavioural features.
  - Each of the latter two has
    - a name and
    - a visibility.
  - Behavioural features in addition have
    - a boolean attribute isQuery,
    - any number of parameters,
    - a return type.
  - Can we model this (in UML, for a start)?

**Meta-Modelling: Example**
Meta-modelling has already been used for UML 1.x. For UML 2.0, the request for proposals (RFP) asked for a separation of concerns: Infrastructure and Superstructure.

- One reason: sharing with MOF (see later) and, e.g., CWM.
- Core UML Profiles
- ClassBox, StateBox, TransitionLine, …
- Superstructure (abstract syntax)
- Infrastructure (with semantics)
- Diagram Interchange
- Node, Edge, …

Meta-Modelling: Principle

Class name: Str
Property name: Str
Type name: Str

CV: Z → C: Property name = v: Type name = Z

Class, Object, Action, Filmstrip
Package, Snapshot
Class, State, Transition, Flow, …

Superstructure (concrete syntax)

Diagram Interchange

Node, Edge, …

Meta-Model
Model
Instance

Example: Metamodelling in UML diagrams

Figure 0-1
Overview of architecture

Figure 7.3 - Root diagram of the Kernel package

Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure

Figure 13.6 - Common Behavior

Figure 7.4 - Diagram Interchange
This is a derived union.

Well-formedness as Constraints in the Meta-Model

If we have an object diagram

\[ \text{An object diagram} \equiv (O, M) \]

where \( O \) is the set of objects and \( M \) is the set of meta-models, then we can consider the following:

- The set \( O \) is a subset of the meta-models.
- The set \( M \) is a subset of the meta-models.
- The objects in \( O \) satisfy the constraints of \( M \).

\[ \text{instances} \subseteq \text{meta-model} \]

The other way around:

\[ \text{instances} = \{ \text{Object} \mid \text{Object} \in \text{meta-model} \} \]

Constraints in the object diagram:

- Objects must be instances of the meta-model.
- Relationships between objects must be well-formed.

Well-formedness as Constraints in the Meta-Model
A classifier is a namespace whose members can include features. Classifier is an abstract metaclass.

The inheritedMember association is derived by inheriting the inheritable members of the parents.

The query parents() gives all of the immediate ancestors of a generalized Classifier.

A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is a specific Classifier of the same classifier.

The query conformsTo() gives true for a classifier that defines a type that conforms to another. This is used, for example, "Namespace (from Kernel)" on page 99.

The precise lifecycle semantics of aggregation is a semantic variation point.

Additional Operations

- Classifier::conformsTo(other: Classifier): Boolean;
- Classifier::maySpecializeType(): Boolean;
- Element::ownedElement
- Element::redefinedElement
- Namespace::member
- Shape::area
- Shape::height
- Shape::shape
- Shape::width
- Shape::xWin
- RedefinableElement::redefinedElement
- Element::redefinedElement
- Element::ownedElement
- Shape::area
- Shape::height
- Shape::shape
- Shape::width

ClassA::width is an attribute of type Integer.

ClassA::area is a derived attribute with type Integer. It is marked as read-only.

Figure 7.30 - Examples of attributes
Open Questions...

- Now you've been "tricked" again. Twice.
- We didn't tell what the "modelling language" for meta-modelling is.
- We didn't tell what the "is-instance-of" relation of this language is.

Idea: have a minimal object-oriented core comprising the notions of class, association, inheritance, etc. with "self-explaining" semantics.

- This is MetaObjectFacility (MOF), which (more or less) coincides with UML Infrastructure [OMG, 2007a].

So: things on metalevel

- M0 are object diagrams/system states
- M1 are words of the language UML
- M2 are words of the language MOF
- M3 are words of the language ...

Other approach:

- Define a generic, graph-based "is-instance-of" relation.
  - Object diagrams (that are graphs) then are the system states — not only graphical representations of system states.
  - If this works out, good: We can easily experiment with different language designs, e.g. different flavours of UML that immediately have a semantics.
  - Most interesting: also for a generic definition of behaviour within a closed modelling setting, but this is clearly still research, e.g. [Buscherm"ohle and Oelerink, 2008].

Meta-Modelling: (Anticipated) Benefits

Benefits Overview

- We'll (superficially) look at three aspects:
  - Benefits for Modelling Tools.
  - Benefits for Language Design.
  - Benefits for Code Generation and MDA.

Benefits for Modelling Tools

- The meta-model M$_UML$ of UML immediately provides a data-structure representation for the abstract syntax (∼ for our signatures).
  - If we have code generation for UML models, e.g. into Java, then we can immediately represent UML models in memory (because each MOF model is in particular a UML model).
- There exist tools and libraries called MOF-repositories, which can generically represent instances of MOF instances (in particular UML models). And which can often generate specific code to manipulate instances of MOF instances according to the MOF instance.

Benefits for Modelling Tools Cont'd

- And not only in memory, if we can represent MOF instances in files, we obtain a canonical representation of UML models in files, e.g. in XML.
  - →XML Metadata Interchange (XMI)
  - Note: Apriori, there is no graphical information in XMI (it is only abstract syntax like our signatures) →OMG Diagram Interchange.
  - Note: There are slight ambiguities in the XMI standard. And different tools by different vendors oftentimes seem to lie at opposite ends of the scale of interpretation. Which is surely a coincidence. In some cases, it's possible to fix things with, e.g., XSLT scripts, but full vendor independence is today not given. Plus XMI compatibility doesn't necessarily refer to Diagram Interchange.
  - To re-iterate: this is generic for all MOF-based modelling languages such as UML, CWM, etc. And also for Domain Specific Languages which don't even exist yet.
Benefits: Overview

We'll superficially look at three aspects:

- Benefits for Modelling Tools.
- Benefits for Language Design.
- Benefits for Code Generation and MDA.

Benefits for Language Design

Recall: we said that code-generators are possible "readers" of stereotypes.

For example, (heavily simplifying) we could introduce the stereotypes Button, Toolbar, ...

to instruct the modelling tool to use special pictures for stereotypes — in the meta-data (the abstract syntax), the stereotypes are clearly present.

Instruct the code-generator to automatically add inheritance from Gtk::Button, Gtk::Toolbar, etc. corresponding to the stereotype.

Et voilà: we can model Gtk-GUIs and generate code for them.

Another view:

UML with these stereotypes is a new modelling language: Gtk-UML.

Which lives on the same meta-level as UML (M2).

It's a Domain Specific Modelling Language (DSL).

One mechanism to define DSLs (based on UML, and "within" UML): Profiles.

For each DSL defined by a Profile, we immediately have

- in-memory representations,
- modelling tools,
- file representations.

Note: here, the semantics of the stereotypes (and thus the language of Gtk-UML) lies in the code-generator. That's the first "reader" that understands these special stereotypes. (And that's what's meant in the standard when they're talking about giving stereotypes semantics).

One can also impose additional well-formedness rules, for instance that certain components shall all implement a certain interface (and thus have certain methods available). (Cf. [Stahland-Völter, 2005].)

One step further:

Nobody hinders us to obtain a model of UML (written in MOF), throw out parts unnecessary for our purposes, add (integrate into the existing hierarchy) more adequate new constructs, for instance, contracts or something more close to hardware as interrupt or sensor or driver, and maybe also stereotypes.

→ a new language standing next to UML, CWM, etc.

Drawback: the resulting language is not necessarily UML anymore, so we can't use proven UML modelling tools.

But we can use all tools for MOF (or MOF-like things). For instance, Eclipse EMF/GMF/GEF.
Recall that we said that, e.g. Java code, can also be seen as a model. So code-generation is a special case of model-to-model transformation; only the destination looks quite different.

Note: Code generation needn't be as expensive as buying a modelling tool with full-fledged code generation.

If we have the UML model (or the DSL model) given as an XML file, code generation can be as simple as an XSLT script. "Can be" in the sense of "There may be situations where a graphical and abstract representation of something is desired which has a clear and direct mapping to some textual representation."

In general, code generation can (in colloquial terms) become arbitrarily difficult.

Example: Model and XMI

```xml
<?xml version='1.0' encoding='UTF-8' ?><XMI xmi.version='1.2' xmlns:UML='org.omg.xmi.namespace.UML' timestamp='Mon Feb 02 18:23:12 CET 2009'>
  <XMI.content>
    <UML:Model xmi.id='...'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id='...' name='SensorA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name='pt100'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Class xmi.id='...' name='ControllerA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name='65C02'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Class xmi.id='...' name='UsbA'>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name='NET2270'/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Association xmi.id='...' name='in' />
        <UML:Association xmi.id='...' name='out' />
      </UML:Namespace.ownedElement>
    </UML:Model>
  </XMI.content>
</XMI>
```