AIMS Embedded Systems Programming
MT 2017
Design with the Unified Modeling Language (UML)

Daniel Kroening

University of Oxford, Computer Science Department

Version 1.0, 2014
Outline

Model-Driven Architecture
Overview of UML
Class Diagrams
Object Diagrams
Use Case Diagrams
Activity Diagrams
State Machine Diagrams
Sequence Diagrams
Model-Driven Architecture (MDA)

Model-Driven Architecture

Key idea:

1. Build high-level model for the software system
2. Gradually refine into actual program while maintaining a layered architecture

MDA is forward engineering: the specification comes before the implementation
Technology Independence

Requirement: Preserve investment in an application

- as platforms proliferate
- while the existing platforms change

Solution: Isolate information and processing logic from technology specifics

- Build platform-independent models
- Refine these models to specific platforms
- Maintain this separation
Raising the Level of Abstraction

- Not new. Has been done before.
  - Programming Languages
    - (hardwired circuitry, punch cards, assembler, PASCAL/C, OO, ...)
  - Operating systems, data bases, ...

- Interesting: well-established for two sides of the problem
  - WYSIWYG GUI builders
  - Data modeling
  - → hand coding no longer predominates

- Goal: higher productivity
MDA Models

- **Business Model** (computation-independent)
- **Analysis Model** (platform-independent)
- **Design Model** (platform-dependent)
- **Code**

**Roles:**
- Business Analyst
- Architect/Designer
- Developer/Tester
What is UML?

- The **Unified Modeling Language** (UML) is a standardized language for describing software.
- **Graphical notation** for all constructs.
- Focus on object-oriented systems.
- Dominating language for modelling application software (2nd only to English).
History

- Grady Booch, Ivar Jacobson, and James Rumbaugh, at the time employed at Rational Software:
  
  **1990s: UML 1.x**

- November 1997: UML became a standard, defined by the *Object Management Group* (OMG)

- September 2004: UML 2.0
Why?

1. You are very likely to see this on the job!

2. It’s *the* standard for OO design

3. (Some) formalization possible, enables tool-support
UML Diagram Types

Structure
- Diagram
  - Class
  - Composite Structure
  - Object
  - Component
  - Deployment
  - Package

Behavior
- Interaction
  - Use Case
  - State Machine
  - Activity

THIS COURSE
- Sequence
  - Communication
  - Interaction
  - Overview
  - Timing
Disadvantages of UML

- Semantics is vague: meaning of diagrams is not always well-defined

- Some historical, inherited complexity

- Huge (main standard document has 732 pages), and not very accessible

- Tools assign different meanings to the same diagram
A Class Diagram...

- describes the **types** of the objects, and **relationships** among objects

---

<table>
<thead>
<tr>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>↗ name: String</td>
</tr>
<tr>
<td>↗ college: College</td>
</tr>
<tr>
<td>↗ friends: Person[0..*]</td>
</tr>
<tr>
<td>↗ courses: Course[0..*]</td>
</tr>
<tr>
<td>↘ supervisor: Academic</td>
</tr>
<tr>
<td>↗ enroll(Course c): void</td>
</tr>
<tr>
<td>↗ drop_out(Course c): void</td>
</tr>
</tbody>
</table>

- Classes have a **name** and two more compartments
- 1st compartment: fields
- 2nd compartment: methods
- Classes belong to a **package**.
Packages

These form a namespace.

The arrow denotes “contained in”
### Class Fields

These are variables, and hold the state of the object.

- **Fields have a**
  - name, and a
  - type.

- **The type may be**
  - a class,
  - or something “predefined”.
  - Arrays are also possible using *multiplicities* – we will discuss the syntax later.

- Static fields are **underlined**.

<table>
<thead>
<tr>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
</tr>
<tr>
<td>college: College</td>
</tr>
<tr>
<td>friends: Person[0..*]</td>
</tr>
<tr>
<td>courses: Course[0..*]</td>
</tr>
<tr>
<td>supervisor: Academic</td>
</tr>
<tr>
<td>enroll(Course c): void</td>
</tr>
<tr>
<td>drop_out(Course c): void</td>
</tr>
</tbody>
</table>
Initial Values

<table>
<thead>
<tr>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI: Real = 3.14159265  {readOnly}</td>
</tr>
<tr>
<td>e: Real = 2.71828183  {readOnly}</td>
</tr>
<tr>
<td>speed_of_light: Real = 299792458</td>
</tr>
<tr>
<td>Plancks_constant: Real = 6.62606896E-34</td>
</tr>
</tbody>
</table>

- Fields may have an initial value
- Syntax:

  field_name : field_type = value

- These are assigned upon creation of the object
  → constructor
Methods

<table>
<thead>
<tr>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>_size: Integer</td>
</tr>
<tr>
<td>size(): Integer</td>
</tr>
<tr>
<td>concat(s: String): String</td>
</tr>
<tr>
<td>substring(lower: Integer, upper: Integer): String</td>
</tr>
<tr>
<td>toInteger(): Integer</td>
</tr>
<tr>
<td>toReal(): Real</td>
</tr>
</tbody>
</table>

- Operations that have read/write access to the fields
- Defined by
  - name,
  - parameter list,
  - return type.
- It is possible to define operators as well.
## Modifiers: Example

<table>
<thead>
<tr>
<th>Patient</th>
</tr>
</thead>
<tbody>
<tr>
<td>*-dateOfBirth : Date</td>
</tr>
<tr>
<td>*+treatment : String</td>
</tr>
<tr>
<td>*#illness : String</td>
</tr>
<tr>
<td>*+name : String</td>
</tr>
<tr>
<td>*~GP : Number</td>
</tr>
<tr>
<td>*+updateCondition( condition : ConditionCode, Notes : String ) : void</td>
</tr>
<tr>
<td>*#changeName( newName : String ) : boolean</td>
</tr>
<tr>
<td>*#printPrescription() : boolean</td>
</tr>
<tr>
<td>*+notifyGP() : void</td>
</tr>
</tbody>
</table>

+ public  – private

# protected  ~ package
Relationships between Classes

- Generalization
- Association
- Aggregation
- Composition
Generalization

Taxonomic relationship between a more general description and a more specific one that extends it.

- OO: this is inheritance.
- denoted by an arrow line with an empty arrowhead from subclass to superclass.
- Often: read as “is a”.

D. Kroening: AIMS Embedded Systems Programming MT 2017
Generalization: Example I

Person
- dateOfBirth : Date
- name : String
- changeName(...) : boolean

Patient
- treatment : String
- illness : String
- GP : Number
- updateCondition(...) : void

Doctor
- specialities: Speciality[0..*]
This is multiple inheritance!¹

¹Read Bertrand Meyer, “Harnessing multiple inheritance”, for more on this.
Association

- “Connection” between classes (may involve more than two)
- Class X uses/references class Y
- This connection may have several properties:
  - Name
  - Navigability
  - Multiplicity
  - Role names, visibility
- Denoted with a solid line
Properties of Associations

- **Name**
  - Typically a verb
  - Describes nature of connection

- **Navigability**
  - Establishes the direction of the relation
  - Denoted with an arrow head
  - May be bidirectional or unspecified

- **Multiplicity**
  - Establishes how many objects participate in the relation
  - Typical: 0, 1, 0..1, 1..*, 0..* (same as *)
  - Default: 1

- **Role names, visibility**
  - Think of these as extra members
Association: Example I

Customer 0..* subscribes 0..* Magazine
+subscriber +subscribed magazine
Association: Example II

D. Kroening: AIMS Embedded Systems Programming MT 2017
Association: Example III (Navigability)

```
Student

name: String

college: College

Mark
```
n-Ary Relations

Some cases associations involve *more than two* classes

- A student takes zero or more courses, and possibly has grades
- A course has at least one student, and possibly has grades (for the students)
- A grade may have been obtained by at least one student, and may be assigned to some courses
n-Ary Relations: Another Example

Team

name: String

2

Score

1

Match

stadium: String

1..*

D. Kroening: AIMS Embedded Systems Programming MT 2017
Aggregation

- Special case of association
- Read as “has a”
- Must not involve more than two classes
- Not a container relationship – no implications on lifetime
- Drawn with empty diamond
Aggregation: Example

Professor \( \xrightarrow{\text{teaches}} \) Course

+teacher +course

0..*
Composition

- Special case of aggregation

- This is a container relationship: once the “owner” is destroyed, so is the object contained therein

- “Owner” must have multiplicity 0..1 or 1
- Drawn with a filled diamond
Composition: Example

Triangle \rightarrow 3 \rightarrow Point
Templates

- Classes may have *template parameters*
- Corresponds to C++ templates or Java generics
- Build an instance with `Vector<int, 3>`
Object Diagrams

- These are instances of classes
- Look like class diagrams, but the name is underlined
- The second compartment contains initializations

<table>
<thead>
<tr>
<th>S1: Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>name='John Smith'</td>
</tr>
</tbody>
</table>
Behavior Diagrams

- So far, we only saw **Structure Diagrams**

- No behavior on those defined so far – even the methods are somewhat mysterious

- We will now look at **Behavior Diagrams:**
  - Use Case Diagrams
  - Activity Diagrams
  - State Machine Diagrams
  - Sequence Diagrams
Use Case Diagrams

A Use Case Diagram...

is a description of system behavior with respect to **external entities** such as human users or other systems.
Use Case Diagrams

There are

▶ use cases (ovals): these correspond to behavior.

▶ actors (stick figures)

These diagrams are really informal.
Use Case Diagrams: Use Case Extension

Use cases can be specialized/extended

This is like inheritance – same notation
Use Case Diagrams: Actor Extension

- Actors can be specialized/extended
- This is also like inheritance
Activity Diagrams

- There is a notion of an active control location (the nodes)
- Change in semantics from UML 1.x to 2.x!
- UML 2.0: Rounded boxes are activities, semantics similar to Petri Nets
Activity Diagrams: Building Blocks

- Initial node
- Activity final node
- Flow final node
- Conditional branch (somewhat redundant)
- Merge (also redundant)
- Fork: concurrency!
- Join: process synchronization
Semantics of Activity Diagrams

- Idea: pass tokens around
- Forks replicate tokens, join nodes generate tokens if there are enough incoming tokens.
- Formalization: class assignment
Activity Diagrams: Branches and Guards

- Guards should not overlap – the ordering of evaluation is not defined
- The guards should be complete
- One of the edges may have a guard else
A State Machine Diagram...

is a graphical representation of a finite state machine.

- These are a variant of Harel’s statecharts (1987)
- Very popular in control engineering (automotive, aerospace, ...)
- Also see Martha’s *Executable Biology* course
State Machine Diagrams: Building Blocks

- Initial / Junction
- Terminate
- Entry Point / Exit Point / History (hierarchy)
- Choice (somewhat redundant)
- Fork
- Join
- Transition with trigger, guard, and action
State Machine Diagrams: Warmup Example

Twiddle thumbs

\[ i=0 \]

\[ [i<10] / i++ \]

\[ [i>=10] \]
Transitions in State Machine Diagrams

Trigger, ... [ Guard ] / Action

- Each part is optional
- A **Trigger** is typically an event identifier with arguments:
  \[ \text{Event (Recv-Arguments \ldots)} \]
  If there is more than one trigger: OR-semantics
- The **Guard** is a Boolean expression
- An **Action** can be
  - some assignment, function call,
  - a send-event command. Typical syntax:
    \[ \text{Event (Send-Arguments)} \]
Branches and Guards in State Machine Diagrams

- Guards should not overlap – the ordering of evaluation is not defined by the standard

- The guards need not be complete: you just stay in the current state until a transition becomes enabled
Events in State Machine Diagrams

- Idea: there is a finite set of events; events may be parameterized, e.g., button(1)

- Events may be external or internal

- There is a queue of events that have occurred
- The ordering of dequeuing is not defined

- The processing of an event must be finished before any other event is processed (run-to-completion)
Events in State Machine Diagrams: Example

Those plentiful “Hang up” edges are annoying

What about dialing more than one digit?
Internal Transitions in State Machine Diagrams

- **Internal transitions** are triggered only if the state containing them is active
- May have triggers, guard, action
- They fire without leaving/re-entering the state
Internal Actions

States may have special internal transitions defined using the following prefixes:

- **entry**: executed when a state is entered
- **exit**: executed when a state is left
- **do**: ongoing activity while in the state
State machines may be nested, which yields hierarchy

Transitions may cross hierarchy boundaries

Advantage: May avoid many transitions

Hierarchy may be used to model concurrency
Hierarchy in State Machine Diagrams: Example

Phone off the hook

- Dialtone
  - Dialing
    - Dial(d)
    - GoodNr
    - WrongNr
    - Timeout
    - RecBusy
    - Connected
  - Busy
    - Wrong Nr
    - RecPickup
  - Connected
  - Pick up
  - Timeout
  - Good Nr
  - Wrong Nr
  - Rec Busy
  - Connected

Hang up
Hierarchy in State Machine Diagrams: Example

- Phone off the hook
- Dialtone
- Dialing
- Ringing
- Busy
- Connected
- Wrong Number
- WrongNr
- GoodNr
- Timeout
- RecBusy
- RecPickup
- Dial(d)
- Hang up

D. Kroening: AIMS Embedded Systems Programming MT 2017
State Machine Diagrams: History

- We have seen that hierarchy is useful to model exceptions
- But how about recovery?
  - Sometimes it is helpful to resume what you interrupted.
- This is modelled by means of a **history node**.
State Machine Diagrams: History

- A transition to a history state restores the state that the chart was in when it was left.

- This is meant to be used as means to resume an activity after an exception or the like.

- There may be one transition outgoing from a history state – this goes to the default state.

- There is also a deep history state:
  - Denoted by H*  
  - Also restores the state of any sub-charts, e.g., within “Sleep” or “Eat”.
Precedence Rules for Transitions

▶ What if multiple transitions are enabled? Which one is taken?

▶ There is only one rule:

**UML Transition Precedence Rule**

A transition out of a composite state \( s \) has a *lower* precedence than any transition out of any sub-state of \( s \).

▶ This applies to UML State Machine Diagrams – Statechart dialects have all sort of additional rules.
Concurrency in State Machine Diagrams

- You can use \( \text{\includegraphics[width=2cm]{triangle}} \) to obtain concurrent threads of execution

- There is an alternative, which is easier to manage

- There are both \text{and-states} and \text{or-states} in state charts

- \text{or-states} are what you are used to:
  
you either “Eat” OR “Sleep” OR “Play”…

- \text{and-states} allow you to “Eat” AND “Play”
Concurrent threads denoted by means of “swimlanes”
- Composite activity ends once all composed activities end
A Sequence Diagram...
defines the behavior of objects by describing the messages they pass.

- Horizontal axis: the objects or actors
- Vertical axis: time
Sequence Diagrams: Building Blocks

Mike: Person

Object with type and lifeline

i++

Synchronous message

i++

Asynchronous message

return

Return from method

Control
Notation for Messages

1: hello

Messages can be numbered

2: buy("tuna sandwich")

Messages can have parameters

Messages can be self-referential
Synchronous vs. Asynchronous Messages

**Synchronous:** The caller *waits* for the completion of the execution of the operation

**Asynchronous:** The caller *does not wait* for the completion of the execution of the operation, but instead continues immediately
Lifetime vs. Control

A
B
foo(1,2,3)

A
B
foo(1,2,3)

D. Kroening: AIMS Embedded Systems Programming MT 2017
Sequence Diagrams: Webshop Example

Client -> Webshop

```
BUY(card, product)
```

Webshop -> Mastercard

```
AUTHORIZE(card, amount)
```

Mastercard -> Webshop

```
RESULT(status)
```

Webshop -> Client

```
BUILD_PAGE(status)
```
Semantics of Sequence Diagrams

- A sequence diagram defines a partial ordering on the time an event (send/receive) occurs.

- Rules (causal order):
  1. Send before matching receive
  2. Receive or send before send of same process
  3. Two receives on the same process sent from the same process

WARNING:

No other guarantees provided, even if suggested by diagram!
Races in Sequence Diagrams

Let $\prec_v$ be the visual ordering, and $\prec_c$ the causal ordering.

A diagram has a race iff there exists a trace $M_1, \ldots, M_n$ with

$$M_1 \prec_c M_2 \ldots \prec_c M_n$$

but not

$$M_1 \prec_v M_2 \ldots \prec_v M_n$$
Sequence Diagrams: Webshop Example Again

Client

```
''BUY(card, product)"

''PLEASE_WAIT''
```

Webshop

```
''AUTHORIZE(card, amount)"
```

Mastercard

```
''RESULT(status)"

''BUILD_PAGE(status)"
```

D. Kroening: AIMS Embedded Systems Programming MT 2017

69
Combining Sequence Diagrams

Sequence diagrams can be combined or integrated into other diagrams by adding a frame.
Combining Sequence Diagrams

These are called High-level Message Sequence Charts (HMSCs)

Does this one have a race?