Introduction to OO and UML

Analysis and Design of Embedded Systems and OO*

Object-Oriented Programming

Jan Bendtsen
Automation and Control
Lecture Outline

- About this course
- Introduction to the object-oriented part of the course
- Why OO*?
- Introduction to UML
- A bit about the essential OO concept: Classes
- Common relationships between objects
- UML and design
About this course

- **Purpose:**
  - To obtain knowledge about *Object-oriented methods* and *realtime systems*
  - To be able to use relevant tools to analyze and design *realtime* and *embedded systems*

- **Goals:**
  - To be able to document insight into these concepts
  - To be able to use them along with the accompanying theories and methods for analysis and design of various problem areas/solutions
  - To be able to account for the connection between the objects being examined and the relevant methodologies
About this course - contents

- Realtime systems
  - Processes and threads
  - Inter-process communication
  - Basic realtime issues
  - The concepts of operating systems and realtime kernels
- Scheduling
  - Analysis of realtime systems
  - Scheduling theory
  - Embedded systems
About this course - contents

- Object oriented analysis and design
  - OO Basics and UML
  - Objects and classes
  - Interfaces and Inheritance
  - Events and Exceptions
  - Threads and Concurrency
  - Java networking
About this course - structure (?)
Introduction to the Object-Oriented part of the Course

Assumptions

- You know C
- You know about compilers, interpreters, etc.
- You have experience with IDEs and/or text editors
- You do not know anything about OO* or Java

After the course, you should

- Know some basic OO concepts and be able to model them in UML
- Be able to write your own applications/applets
- Have a little experience with client/server communication in Java
- Know where to look for more information
Introduction to the Object-Oriented part of the Course

Literature:

Suns Java Tutorial
available online at
http://java.sun.com/docs/books/tutorial/index.html

Course homepage:

Find it via ESN's homepage – Elektronik & IT 5th sem.!
- will be updated with new links, slides etc. as they become available – make sure to check before each lecture
Why OO*?

- **OO* : Object-Oriented Analysis, Design and Programming**
- An attempt to model real-world phenomena and things in a way that can be mapped to a “computer-friendly” representation
- Structured programming is often too “rigid” for large-scale systems
- Facilitates good programming practices: *modularity, code re-use* and *abstraction*
- The basis for *Design Patterns*
Why OO*?

- **Data Interpretation** – data is associated with a given object and thus “makes sense” in a given context.

For instance, the variable

```c
int numPoints = 6;
```

should represent different things depending on whether it is used in a graphics context or a spline interpolation routine.
Why OO*?

- Graphics: Polygon #1 has 8 vertices; polygon #2 has 4 vertices etc.

- Spline interpolation:
  
  ```
  for (i = 0; i < numPoints; i++) {
      // Calculate interpolation
  }
  ```
Why OO*?

- **Data Interpretation** – we want to associate data and operations. Operations that make sense for one kind of data does not necessarily make sense for another.

- For the C-minded: *An object is like a struct with functions attached.*

- Thus, the function call compute() can have a different meaning for a cubic spline interpolation and the estimation of the area of a 8-vertex polygon.
Why OO*?

- **Data Encapsulation** – Hide information that the user does not need to interact with and allow various objects of the same type to have individual states.

- **Inheritance** – Allow specialization of more general data types/classes; E.g., a Student may be a Human, which in turn is a Mammal, which in turn is an Animal etc.

- **Polymorphism** – the ability to hide several different kinds of objects behind the same interface; E.g., triangles, squares, block diagrams and glyphs can all be *drawn*, because they have geometric *shapes*.

- **Message Passing** – like in real life, objects tend to interact with each other in an essentially *asynchronous* manner.
Why OO*? - an Example

- Name: SISO filter
- Attributes: sample time, input channel, output channel, dynamics
- Operations (e.g.): setSampleTime(Ts), setPolesAndZeros(p, z), getTimeResponse(input), getFrequencyResponse()
- Hidden implementation details: pole-zero/polynomial representation, initial value, Nyquist frequency etc.
Unified Modeling Language is a *formalism* for specifying OO design

- It offers model *notation*, not a methodology for design
- It is a standard – most OO* tools use it
- It provides a good way of documenting design
- It helps developers communicate
- It helps finding errors (related to design, at least)
- *It is NOT, in itself, a design method*
Overview of UML

- Use Case Diagrams
- Class Diagrams
- Interaction Diagrams
- State Diagrams
- Activity Diagrams
- Package Diagrams
- Deployment Diagrams

<table>
<thead>
<tr>
<th>Actors</th>
<th>Usage, involvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>Association, inheritance, aggregation</td>
</tr>
<tr>
<td>Objects</td>
<td>Messages, lifelines</td>
</tr>
<tr>
<td>States</td>
<td>Transitions, behaviors</td>
</tr>
<tr>
<td>Activities</td>
<td>Guards, synchronization</td>
</tr>
<tr>
<td>Packages</td>
<td>Software dependencies</td>
</tr>
<tr>
<td>Processors and Nodes</td>
<td>Component dependency</td>
</tr>
</tbody>
</table>
Relationship btw. Diagram Types

Use Case Diagrams

Activity Diagrams
- Work-flows
- Inter-class behavior
- Structuring

Interaction Diagrams
- Scenarios
- Structures
- Interaction btw. objects
- Intra-class behavior

Class Diagrams

Structures

Package and Deployment Diagrams

State Diagrams
Use Case Diagrams

- Show how users and other actors interact with the OO system
- *Use cases* are used to describe activities that the system must be able to perform – essential services to be provided by the system
- They should be related to one or more actors in some way
- Examples of actors:
  - users
  - other computers
  - resources (e.g., log files)
  - timed events
- Use case diagrams are also used to *delimit* the system
- Each use case is accompanied by a *use case description*
**Use Case Description example**

- **View Telemetry data:**
  A user specifies a set of data (scientific data, satellite household data, alarms, etc.) that he/she wants to view from a list. Then he/she specifies a start and/or end time. The system retrieves the specified data from persistent storage and presents it via GUI. In case start or end times are left unspecified by the user, default values are substituted automatically.
Use Case Diagram example
Classes

- Classes are *definitions* of groups of objects with the same attributes and operations.
- They act as “blueprints” for the actual objects – e.g., SISO discrete-time filters.
- The closest concept in C is:
  ```c
  typedef struct {
      int numPoints;
      point[] points;
  } polygon;
  ```
- Individual objects of a given class are called *instances* of that class; they can be created and subsequently destroyed.
- In some (rare) cases, objects of a class cannot be created; such classes are called *abstract*. 
Classes

DiscreteSISOFilter

+ sampleTime
+ inputVector
+ outputVector
- Apolynomial
- Bpolynomial
- internalState

+ DiscreteSISOFilter()
+ setSampleTime()
+ setPoles()
+ setZeros()
+ getTimeResponse()
+ getFreqResponse()
- computeResponse()
- checkLimits

Class name
Public attributes
Private attributes
Constructor
Public operations
Private operations
Classes

<table>
<thead>
<tr>
<th>DiscreteSISOFilter</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ sampleTime</td>
</tr>
<tr>
<td>+ inputVector</td>
</tr>
<tr>
<td>+ outputVector</td>
</tr>
<tr>
<td>- Apolynomial</td>
</tr>
<tr>
<td>- Bpolynomial</td>
</tr>
<tr>
<td>- internalState</td>
</tr>
<tr>
<td>+ DiscreteSISOFilter()</td>
</tr>
<tr>
<td>+ setSampleTime()</td>
</tr>
<tr>
<td>+ setPoles()</td>
</tr>
<tr>
<td>+ setZeros()</td>
</tr>
<tr>
<td>+ getTimeResponse()</td>
</tr>
<tr>
<td>+ getFreqResponse()</td>
</tr>
<tr>
<td>- computeResponse()</td>
</tr>
<tr>
<td>- checkLimits</td>
</tr>
</tbody>
</table>

- Individual objects are instances of a class; they represent the actual “things”
- Objects are *instantiated* by invoking the class' *constructor*
- Two different objects of the same class must have different identifiers (variable names) and may have different *attribute* (field) values
- The field values represent the *state* of the individual object
Operations (methods) represent ways to interact with the individual objects

Methods are *invoked*, often with *arguments*.

Typically, a method affects only the object itself. Invoking the same method of two different objects of the same class may yield vastly different results

*Abstract methods* do not require an object to be instantiated (e.g. the main method in Java)

NB: A constructor is not a method
### Classes

<table>
<thead>
<tr>
<th>DiscreteSISOFilter</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ sampleTime</td>
</tr>
<tr>
<td>+ inputVector</td>
</tr>
<tr>
<td>+ outputVector</td>
</tr>
<tr>
<td>- Apolynomial</td>
</tr>
<tr>
<td>- Bpolynomial</td>
</tr>
<tr>
<td>- internalState</td>
</tr>
<tr>
<td>+ DiscreteSISOFilter()</td>
</tr>
<tr>
<td>+ setSampleTime()</td>
</tr>
<tr>
<td>+ setPoles()</td>
</tr>
<tr>
<td>+ setZeros()</td>
</tr>
<tr>
<td>+ getTimeResponse()</td>
</tr>
<tr>
<td>+ getFreqResponse()</td>
</tr>
<tr>
<td>- computeResponse()</td>
</tr>
<tr>
<td>- checkLimits</td>
</tr>
</tbody>
</table>

- Fields and methods have *access specifiers* associated with them
- Private (-): can only be accessed from within the object
- Public (+): can be accessed from everywhere
- Package (Java): can be accessed from objects belonging to the same package
- Protected (Java): Like Private, but allows access from inherited classes as well
Common Relationships between Classes

**Aggregation**
“A has a B” relationship – e.g., all DiscreteSISOFilter objects contain two Polynomial objects, Apolynomial and Bpolynomial
Common Relationships between Classes

**Association**
“A knows B” relationship – e.g., a DiscreteSISOFilter object might be connected with up to two Channel objects (input and output)

DiscreteSISOFilter objects can invoke methods in Channel objects
Common Relationships between Classes

**Inheritance**
“A is a B” relationship – e.g., a ButterworthFilter is a particular kind of DiscreteSISOFilter with a certain frequency response
Class Diagrams

- Used to show important class features
- Constructed based on use case diagrams and descriptions
- They show
  - Class names
  - Fields (attributes), methods (operations) and access specifiers
  - Inheritance, aggregation and association
  - Implementation of interfaces
- They demonstrate how the objects identified in use cases are organized (on the class level) relative to each other; e.g., which objects need to know each other
- Arguably the most important diagram type
Class Diagram Example

FirstClass
- attr1: ArrayList
- attr2: int
- method1(a: int, str: String): void
- method2(): String

SubClass
- method3(): Point2D
From the Use Case...

- **View Telemetry data:**
  The user specifies a set of data (scientific data, satellite household data, alarms, etc.) that he/she wants to view from a list. Then he/she specifies a start and/or end time. The system retrieves the specified data from persistent storage and presents it via GUI. In case start or end times are left unspecified by the user, default values are substituted automatically.

**Nouns:** Potential class or attribute candidates
TelemetryDataList Class Diagram

- **TelemetryDataList**
  - `location`: Coordinate3D
  - `velocity`: Coordinate3D
  - `attitude`: Quaternion
  - `startTime`: Calendar
  - `endTime`: Calendar
  - `defaultStartTime`: Calendar
  - `defaultEndTime`: Calendar
  - `getNames()`: String[]
  - `getValues(name: String)`: String[]
  - `setStart(startT: Calendar)`: void
  - `setEnd(endT: Calendar)`: void

- **Coordinate3D**
  - `x`: Double
  - `y`: Double
  - `z`: Double

- **UserInterfaceWindow**

- **DataManager**

- **FileReader**

- **Calendar**
Interaction Diagrams

- Once the classes are in place, it is time to think about interaction between them.
- Interaction diagrams are used when you want to model the behavior of several objects in a use case.
- They demonstrate how objects collaborate in a time-oriented manner.
- They also show when some objects are created or destroyed as a sequence of events or method invocations.
- They can be divided into Sequence Diagrams and Collaboration Diagrams (although they show essentially the same information).
Sequence Diagram Example

1. Request Telemetry data
2. create
3. create
4. create
5. getSelection()
6. setSelection()
7. readData()
8. Return data
9. getData()

User selects

Data is presented via GUI (not shown)
State Diagrams

- Used to indicate the state an object/program is currently in
- State diagrams demonstrate the behavior of an individual object through one or several use cases of the system
- State diagrams do not always require state diagrams
- A bit similar to old-fashioned flowcharts:
  - Initial state
  - Do something
  - When something is satisfied, switch to a new state
  - etc.
State Diagram Example

Processing Telemetry Request
  do/Create list and telemetry data objects

Initializing
  do/Initialize variables

Waiting

User request
User commits choice

Reading File
  do/Create and start file reader object

Showing Data
  do/Pass data to GUI

Reading done
Initialization done
Symbols in State and Activity Diagrams

- **States** – oblong boxes that indicate the stable states of the object between events

- **Transitions** – solid arrows showing potential changes from one state to the next (or to the same state)

- **Events** – text on the transitions (before '/') showing the incoming event (call to the object interface) causing a transition to occur

- **Conditions** – a boolean statement in [ ] that states whether a given event can occur (aka. Guards)

- **Actions** – text on the transitions (after '/') defining the objects' response to the transition between states

There may also be various extra symbols that define specific state-centric functionality
Activity Diagrams

- Activity diagrams describe the “workflow” behavior of the whole system.
- Similar to state diagrams, but not limited to the behavior of individual objects.
- They show the flow of activities within and outside the system, in particular parallel activities:
  - *Forks* are used to indicate when several activities occur at the same time.
  - *Merges* are used to indicate when conditional behaviors of branches end.
User request received

Handle request to view current location

Check user authorization

[user not authorized]

Sat Com established

[=user authorized]

DB Com established

Obtain global data

Obtain current data from satellite

Aggregate data

User feedback
Deployment diagrams show

- Computing nodes in the system
- Communication links
- Software components that run on said nodes

They are typically created for distributed systems (e.g., client-server architectures) to ensure that the correct programs are run on the correct machines.
OO Development Using UML

Requirement specification
Use case Diagrams
(basic behavior, system delimitation)
OO Development Using UML

Class Diagrams
/software structure/

Basic design
Elaboration
OO Development Using UML

Interaction Diagrams (scenarios)

Basic design
Elaboration
Activity Diagrams
(Workflow, Interclass behavior)
State Diagrams (intra-class behavior)
/**
 * TelemetryDataList
 **/

import java.util.ArrayList;
package org.studentsat.telemetry;

public class TelemetryDataList extends ArrayList implements Serializable {
    private Coordinate3D coordinates;
    private Coordinate3D velocity;
    private Quaternion attitude;

    public String[] getNames() {
        return coordinates.getAxisNames();
    }

    private void compute(int numCoords) {
        // compute method implementation
    }
}
There are many tools available for UML design and development

Typical features include

- Drag-and-drop diagram creation
- Support for (at least) Use Case, Class and Interaction diagrams – but often more
- Automatic template code generation
- (sometimes) reverse engineering of existing Java code

In this course we support

- Eclipse (ver. 3.4) *Modeling Tools* (http://www.eclipse.org)
- EUML2 (http://www.soyatec.com/euml2/)

Exercises for today: install these tools and try them out