Information System Infrastructure II

Analysis & Design

Hans-Arno Jacobsen
jacobsen@eecg.toronto.edu
www.eecg.toronto.edu/~jacobsen

Outline

• Modeling
• UML
• Class diagrams
• Sequence diagrams

Today's lecture draws its material from:

The Unified Modeling Language User Guide, Grady Booch,
James Rumbaugh, Ivar Jacobson. Addison Wesley.

Thanks to Charles Zhang and Gao Dapeng for preparing an earlier version of these slides.
Learning objective

Our objective is to learn about
- what modeling of software-intensive systems means
- “speak” a common and standard modeling language ("moderately well"), i.e., we have to practice speaking this language
- UML is this modeling language; we focus on
  - overall understanding of what this language involves and how it works
  - specifically we want to be able to use and understand
    - class diagrams
    - sequence diagrams

Modeling

Why model?

What is a model?
System Modeling

- “Developing a model [...] prior to its construction or renovation is as essential as having a blueprint for large building.” - [The Unified Modeling Language User Guide]
- Why?
  - Easy to understand and to evolve through visualization
  - Provides guidance to software construction.
  - Essential for communication among project teams and to assure architectural soundness.

The five views of an architecture “=” the blueprints of the software

- To understand the architecture of a system the following complementary and interlocking views are needed
  - a use case: exposes requirement of a system
  - a design view: vocabulary of problem & solution space
  - a process view: distribution of system's process & threads
  - an implementation view: physical realization
  - a deployment view: system engineering issues
- These views may have structural, as well as, behavioral aspects.
Unified Modeling Language

History of UML
- Started in 1994 at Rational by Grady Booch, Jim Rumbaugh and Ivar Jacobsen.
- Standardization effort initiated by OMG(Object Management Group) in 1995.
- Adopted as a standard by OMG in 1997.
- UML is based on an object oriented paradigm

UML - Unified Modeling Language
- standard language for writing software blueprints
- UML may be used to
  - visualize
  - specify
  - construct
  - document
    ... the artifacts of a software-intensive system
- UML primarily intended for software-intensive systems, used for domains such as:
  - enterprise information systems, banking and financial services, telecommunication, transportation, defense, aerospace, retails, medical electronics, healthcare, scientific computing, distributed and web-based systems ...
What is UML?

A visual language consists of the following elements

- Structural elements including shapes representing class, interface, collaboration, use case, active class, component and node.
- Behavioral elements including graphical notations representing messages and state transitions of the system.
- Grouping elements including only one primary kind of grouping element, namely, package.
- Annotational elements: Note

UML is for documenting

- Software is not just raw executable code, it is, among other:
  - requirements, architecture description, design, source code, project plans, cost models, test cases, test scripts, prototypes, releases, versions
- UML
  - documentation of system's architecture
  - expressing requirements and tests
  - models activities of project planning and release management
A conceptual model of the UML

- to understand UML, understand
  - basic **building blocks**
  - **rules** for putting building blocks together
  - **common mechanisms** that apply throughout UML
- basically, like any other language (artificial or natural)

Building blocks of UML

- **Things**
  - abstractions that are first class citizens in a model
- **Relationships**
  - tie these **things** together
- **Diagrams**
  - group interesting **collections of things** together
Things in UML

- Structural things
- Behavioral things
- Grouping things
- Annotational things

These are the basic object-oriented building blocks of UML. They are used to write well-formed models.

Structural things

- classes
- interfaces
- collaborations
- uses cases
- active classes
- components
- nodes
Class

In UML, a class is rendered as a rectangle.  

<table>
<thead>
<tr>
<th>Class</th>
<th>Car</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Make : char</td>
</tr>
<tr>
<td>- Mileage : unsigned long</td>
</tr>
<tr>
<td>- isRegistered : bool = false</td>
</tr>
</tbody>
</table>

Class Attributes

- **Attributes**
  - Attributes represent properties of the class. A class can have any number of attributes. Attributes can be expressed in the following format:
    - `[Visibility Adornment] AttributeName : [Attribute Type]`
- **Visibility** defines whether it can be accessed from other classes.
- **UML** defines three levels of visibility:
  - **Public** “+” Any outside class is allowed to access.
  - **Protected** “#” Any descendants are allowed to access
  - **Private** “-” Only the class itself can access
Class Operations

Operations
- An operation represents an action associated with that particular class.
- An operation often causes changes to the properties, but not always.
- Operations are expressed as the following:

[Visibility Adornment] MethodName (argumentname: Type...) : returnType

<table>
<thead>
<tr>
<th>a.k.a. operation or method signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Drive() : void</td>
</tr>
<tr>
<td>+Price() : unsigned long</td>
</tr>
<tr>
<td>+Fill(in gas : unsigned int) : void</td>
</tr>
</tbody>
</table>

Interfaces
- collection of operations that specify service of class
- externally visible behavior of that class
- may represent part or all of the behavior of class
- defines operation signature, but not implementation

Analogies:
- Java interface classes
- OMG IDL interfaces
- abstract base classes
- public members of C++ class
- APIs - Application Programming Interfaces
Interfaces cont.’d.

- interfaces rarely stands alone
- attached to class that realizes it

Collaborations

- name a conceptual chunk in system architecture encompassing static & dynamic aspects
- names a society of classes, interfaces, and other elements that belong together, i.e., implement a functionality that is bigger than the sum of its parts
- use them to model architecturally significant parts in system

Example: distributed information system involving several DBMS
- user’s perspective updates look like single atomic operations
- internally, fare more complex
- the internal transaction management logic
  - includes several classes, which themselves use other classes ...

transaction mgmt.
Use cases

- serves the requirements engineering phase
- the interaction of users (actors) with the system
- focus on requirements
  - how is the system used, how does one interact with it
  - implementation is not addressed
- description of set of sequence of action that system performs that yield and observable result to a particular actor
- often captured as a collaboration

Place order

Active classes

<table>
<thead>
<tr>
<th>EventMngr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>listen()</td>
</tr>
<tr>
<td>flush()</td>
</tr>
</tbody>
</table>

- the same as classes, except
  - its objects own a process or thread, i.e., may initiate control activity
  - its object represent behavior that is concurrent to other elements represented in the model
Component

- physical, replaceable part of a system (as opposed to logical component)
- COM+ components, Beans, CORBA Components
- physical packaging of otherwise logical elements (classes, interfaces etc.)

Nodes

- physical element that exists at run time
- represents computational resource contains memory and processing capabilities
- a set of components may reside on node, may migrate from node to node ...
Behavioral things

- dynamic parts of UML
- represent behavior over time and space
- “verbs” of model that represent behavior over time and space, as opposed to structural things that represent the “nouns” of the model

- two primary behavior things
  - messages
  - states

Messages

- an interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose
- e.g., a protocol
  - visit of foreign ministers and statesman
  - TCP protocol, UDP protocol, RMI protocol
  - request/respose

```
<table>
<thead>
<tr>
<th>client</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>display</td>
<td></td>
</tr>
</tbody>
</table>

account.deposit(a)  blocked  computing
successful
```

(c) 2002 Hans-Arne Jacobsen (jacobson@eecg.toronto.edu) MIE491f
States

- state machines are comprised of states and transitions
- change of state is triggered by an action
- e.g., coke machine, ATM machine, process execution cycle in operating system

Waiting

Grouping things

- organizational parts of UML models
- means to structure a UML model
- decompose it into parts
- only one primary kind, that is packages
- only exists at development time
Packages

- groups things
- may contain structural things, behavioral things, even other packages
- packages are conceptual, they only exist at development time
- compare to components which exist at run time

Notes

```c
int foo(int a){
    return a*a;
}
```

- notes are the annotational things
- explain, illustrate, highlight, and describe UML models
- represent comments
Relationships in the UML

- Classes do not stand alone. Most of them collaborate with others in a number of ways.
- The following relationships are most important in UML:
  - Dependencies
  - Associations
  - Generalizations
  - Realizations

Generalization

- Derived Type
  - let us suppose that a truck is a kind of car, we show the class Truck as a sub-type - a child of class Car - as the following.

```
Car

  ↖
  ↗

Truck
```
Dependency

- A dependency is a **using relationship** that states that a change in specification of one thing may affect another thing that uses it, but **not necessarily the reverse**.
- Graphically rendered as the following:

```
   Car
    ◆ Fill(g : Gas) : void
        →
    Gas
```

- If the definition of **Gas** changes, say price, the behaviour of **Car** is affected. **Not the other way around**.

---

Association

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another.
- Binary association involves two classes. Association involves more than two classes are called n-ary associations.
- Graphically rendered as a solid line connecting different classes.

```
0 .. 1
employer

#
employee
```
Association cont.'d.

- every employer may have many employees
- every employee may have 0 or 1 employer
- employer and employee are roles of the association

\[
\begin{array}{ccc}
0 \ldots 1 & \ast & 0 \ldots 1 \\
\text{employer} & \ast & \text{employee}
\end{array}
\]

Adornments of Association

- Name
  - An association can have a name that describes the nature of the association

- Role
  - Class has a specific role that it plays in the association relationship

(c) 2002 Hans-Arne Jacobsen (jacobsen@eceg.toronto.edu) MIE491F
Adornments of Association

Multiplicity

In many situations, we need to state how many objects are involved in the association.

- Multiplicity can state as exactly one(1), zero or one(0..1), many(0..*), one or more(1..*)
- When multiplicity is declared as one end of association, it is saying that there are that number of objects for each object at the opposite end.

Example: The following multiplicity states that, each person can own one or more cars. Each car can only have one owner.

```
Person  1  \  1..*  Car  \\
        \       /    \\
        owner     property
```

Adornments of Association

Aggregation

- Used to model a “whole/part” relationship, in which one class represents a larger thing, which consists of smaller things.
- Example:
  Suppose our car only consists of 4 doors and 4 tires. The following diagram illustrates the composition of class car

```
Car  1  \  \\
     \  \   \\
     4 \  \\
     Doors  4
```

Difference between empty diamond and filled diamond is that filled diamonds represent the integral aggregation, meaning the Door, typically specifically made for a car cannot exist by itself. Tire, however, can be installed on other cars, thus can exist by itself.
Diagrams in the UML

- Class diagrams
- Object diagrams
- Use case diagrams
- Sequence diagrams
- Collaboration diagrams
- State chart diagrams
- Activity diagrams
- Component diagrams
- Deployment diagrams

The bold faces items comprise our interest in this course.

Class Diagram

- Class diagram specifies the structure of a (software) system
- The following is a diagram of an Java applet program for "Hello World"

```java
public void paint(Graphics g) {
    g.drawString("Hello World!", 10, 10);
}
```
Sequence Diagram

Why sequence diagram?

- Class diagram defines the static structure of the program.
- Sequence diagram defines the dynamic behaviour of the program.
- Sequence diagram defines the interactions among classes.

Example:

- The following sequence diagram illustrates the interaction between the driver class and the car class

Use Case Diagram

- A use case diagram is a diagram that shows a set of use case and actors and their relationships.
- A use case is a system feature, an indication of a way of using the system - a requirement.
- For example, the following diagram is a simple diagram for Car.

(c) 2002 Hans-Arne Jacobsen (jacobsen@ee.ece.utoronto.ca) MIE491f

(c) 2002 Hans-Arne Jacobsen (jacobsen@ee.ece.utoronto.ca) MIE491f
UML Case Study

Suppose we want to build a geometric analysis tool, which computes perimeters and areas of squares and circles. We use case diagram to illustrate the functionality of the program from the point of view of users.

We use class diagram to model objects in the program. Objects are nouns in the problem statement. We establish a class hierarchy as the following.

```java
<<abstract>>
shape
+getPerimeter():double
+getArea():double

Square
- width: double
+ getPerimeter(): double
+getArea(): double
+Square(double)

getPerimeter{
    return 2*pi*width;
}
getArea{
    return pi * width* width;
}

Circle
- radius: double
+ getPerimeter(): double
+getArea(): double
+Circle(r : double)

Return 4*width
```

(c) 2002 Hans-Arne Jacobson (jacobson@eeg.toronto.edu) MIE491f
Our main class is called Geometric Tool, which has a dependency relationship with the shape class.

```
Geomeric Tool
  + OutputPerimeter(Shape s): void
  + OutputArea(Shape s): void

System.out.println(s.getPerimeter());
System.out.println(s.getArea());
```

To ease management, we can group all classes into a package called Geometric Tool, which depends on the system package of Java.

```
Geomeric Tool
  System
```

Sequence Diagram can be used to illustrate use cases. For example, the following sequence diagram shows the process of computing areas of circles.

```
: GeometricTool : Shape : Circle : OutputStream

OutputArea

OutputArea

Result: double

Println(result)
```

(c) 2002 Hans-Arne Jacobsen (jacobsen@eeec.utoronto.edu) MIE491f
Component Diagram gives the physical view of the program. In our case, suppose we package all our classes into one executable component called Geometric Tool. The following diagram shows the runtime requirement of the tool.

![Component Diagram](image_url)