CS3505/5020
Software Practice II

Project #4: Review progress
  Use cases
  Class diagrams
Concept: Requirements

- Requirements are not easy to define in the context of software
  - Certainly, formal definitions are often complex.

- In this case, we view requirements as the following
  - Guidance on the purpose of software, from the viewpoint of users and other stakeholders
    - Stakeholder: Anyone with interest in the system
  - Name some stakeholders
  - When you talk requirements you must mention stakeholders or you aren’t talking about requirements 😊
Requirements

- Some spilt requirements into two major types:
  - Functional
    » Functionality for the user/stakeholder
  - Non-functional
    » High-level concerns that don’t directly affect users
    » For example, the use of a certain UI toolkit
Requirements

- A different, somewhat non-orthodox view on requirements
  - Don’t distinguish between functional and non-functional if possible
  - Every requirement should be linked to a user or stakeholder (who wants this)
  - That is, why specify a non-functional requirement if the stakeholder is not linked to it?
    » Either the stakeholder doesn’t care in which case it doesn’t belong in the system
    » Or the stakeholder does care and it is a functional requirement

- Requirements are typically a textual document
  - Modeling of requirements provides a higher level view (abstract)
  - Modeling of requirements allows more succinct expression of some concepts
Requirements and Use Cases

● We will get more and more into what a use case is a bit later, but for now, note that:
  – Use-cases are a way of representing requirements.

● A lot of the software world has embraced use cases as “THE” way to express requirements
  – Some use a mix of ideas
    » Formal specification to completely informal documents
Concept: Architecture

- If requirements is hard to define, then architecture is just as hard (if not more so)
- Architecture provides a major view of the organization of software in terms of:
  - Architectural elements and their connections
  - Grouping of these elements into components and interfaces
  - Groups of components into subsystems, and subsystems into systems
Architecture

• Often, the hardest part of architecture is
  – Choosing the proper set of architectural elements,
    » Making new elements when needed
  – Using those elements on a proper level of abstraction (not too high, not too low)
  – Minimizing coupling, maximizing reuse potential in components, subsystems.

• The software practice courses:
  – Expose you to several sample architectures (instead of you having to invent your own)
  – See how to build solutions given a particular architecture
Requirements and Architecture

● Can have a great impact on the success or failure of a project
  – Good requirements, architecture can make a project much more likely to succeed (but it still can fail)
  – Bad requirements, architecture makes a project much less likely to succeed (but it still can succeed)
  – Successful development requires focus, effort at every stage.

● Strongly related to goals software process
  – If you develop software successfully, can you repeat the process or was it a fortunate accident?
Where We are Headed

● As a system architect / team lead you would
  – Be responsible for specifying the requirements
  – Lead in architecture
  – Lead or assist in design

● As a designer you would
  – Consume requirements
  – Assist in architecture
  – Lead in design

● As a developer you would
  – Consume requirements
  – Consume architecture
  – Assist in design

● So, our goal
  – Motivate the need for writing requirements
  – Expose you to writing requirements
  – Analyze and critique requirements and architecture
Use Cases

- Originated with Ivar Jacobsen
  - “Object-Oriented Software Engineering” 1992

- An important reference: “Writing Effective Use Cases” by Alistair Cockburn

- Designed to capture the high-level requirements of a system

- Describes WHAT a system should do
  - User focus
  - Gives scope to the project

- Use cases are not “functional decomposition”
  - That is, think about how you might build the system, decompose it, and describe each piece
  - That is for design and implementation
You can’t think like a programmer when writing them
- This makes it really hard for us programmer types to write use cases

Use cases should not say **HOW** something is done
- There are other tools for **HOW**, that we’ll see later

Use cases are about stakeholder and their interaction
- You do not talk about what stakeholders do outside of the system unless it is relevant to the system

Use cases are not strictly OO
- But they have been adopted fairly widely by the OO community
How do you do design if you are given a set of use cases?

Briefly - Use an object-oriented design process:
1. Take a use case
2. Walk through the use case and look for a class (object)
   - Caution – they may be implied!
3. When you identify one, write it down, figure out responsibilities
4. Repeat with all of the use cases

Use UML to draw your class diagrams.
Object Oriented Design (OOD)

- OOD is the technique used to architect software with groups of classes that interact with one another to solve a problem.
- To qualify as an OOD, a few requirements need to be met.
- The three fundamental principles are essential for an OOD to exist:
  - Classes (abstraction and encapsulation)
  - Inheritance
  - Polymorphism
- Common OO abstractions: packages, exceptions, streams, threads, components and events (asynchronous notifications), and communicators (sockets).
UML and Classes

- UML defines a visual representation of structure
- Everything that is defined in this has meaning
  - Just like when writing code, you use () or , or + to represent something
  - Same is true in UML
- Be careful of the types of arrows, boxes, etc. that you use
- Be careful of where you put other symbols
- Remember that there is a syntax defined for UML and that it can be automatically parsed
Classes in UML

- Classes describe objects
  - Behaviour (member function signature / implementation)
  - Properties (attributes and associations)
  - Association, aggregation, dependency, and inheritance relationships
  - Multiplicity and navigation indicators
  - Role names
**UML Class Review**

- **Class name**
- **Data members** (attributes)
  - -x: double
  - -y: double
  - -z: double
  - -n: int
- **Instance methods**
  - +name()
  - +method1(:double):double
  - +method2():bool
  - +classMethod()

These compartments are optional. But if you need methods, then you have to have at least empty data member compartment.

**Arguments**

**Return types**

**Important** – if something isn’t specified, then that doesn’t necessarily mean that it isn’t there.
The top compartment contains the class name

Abstract classes have italicized names

Abstract methods also have italicized names

Classifiers and stereotypes are used to identify groups of classes, e.g. «interface>> or «persistent>> (storable) or other class groups
**Class Attributes Review**

Attributes are the instance and class data members.

Class data members (underlined) are shared between all instances (objects) of a given class (think static).

Data types shown after ":".

Visibility shown as:
- `+` public
- `-` private
- `#` protected

Attribute Model:

```
visibility name : type multiplicity = default {property-string}
```

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>-instanceDataMember: type</td>
</tr>
<tr>
<td>-classDataMember: type</td>
</tr>
<tr>
<td>+Name()</td>
</tr>
<tr>
<td>+Name(:Name)</td>
</tr>
<tr>
<td>+operation()</td>
</tr>
<tr>
<td>+classMethod</td>
</tr>
</tbody>
</table>
Class Operations Review

Operations are the class methods with their argument and return types.

Public (+) operations define the class interface.

Class methods (underlined) have only access to class data members, no need for a class instance (object).

Parameter List – direction name: type = default
Direction Model – direction – in, out, inout

Operations Model
visibility name (parameter-list) : return-type {property-string}
Visibility

- + (public)
  - Anyone can access
- - (private)
  - No one can access
- # (protected)
  - Subclasses can access
Multiplicity - 1

- Indicates how many objects may fill the property
  - 1 – exactly one
  - 0 .. 1 – may or may not have one
  - * – zero or more
  - n .. m – from n to m where n < m

- Default is [1]
  - But it is best to not use default even if value is 1 to make it clear

- Question – if you indicate that something is a n..m multiplicity, then how is this realized?
UML Stereotype vs. UML property

- **Stereotype**, e.g., <<utility>>
  - Represents a new modeling construct

- **Property**, e.g., {abstract} or {readonly}
  - Represents a characteristic of an existing UML construct, for example a class can be abstract, or immutable

![Diagram](image-url)
Template Classes

Generic classes depending on parameterized types

Type parameter(s)

vector

+size(): int
+push_back(:T)
+operator[](::int):T

Operations compartment as usual, but may have type parameter instead of concrete type
Unary Association Review

A knows about B, but B knows nothing about A

void doSomething() {
    myB.service();
}

Arrow shows direction of association in direction of dependency

These two diagrams are identical in what they mean.
Compare Attribute and Association

These are identical
Binary Association Review

Binary association: both classes know each other

```
void doSomething() {
    myB.service();
}
```

```
void operation() {
    myA.service();
}
```

Usually "knows about" means a pointer or reference
Other methods possible: method argument, tables, database, ...
Implies dependency cycle
Association Details

Association name gives details of association
Name can be viewed as verb of a sentence
Arrow helps indicate directionality

A
+doSomething()

Uses

B
-myBs
1..*
+service()
+operation()

Names at association ends
explain "roles" of classes (objects)
An association that connects a class to itself is called a self association.

- Example: A Company has Employees.
- A single manager is responsible for up to 10 workers.
Dependency Review

- **Weak relationships (also transitory):**
  - Class A simply knows of class B
    » E.g., a method in A creates and immediately returns an object of class B

- **That is:** a change in one may force changes in the other although there is no explicit association between them.
Generalization (Inheritance) Review

Base class or super class

Arrow shows direction of dependency

Derived class or subclass

public class B : A {
    ...
}

A

- myX: double
  + setX(:double)
  + getX():double

B

+ operation()