Software Lifecycle Activities

<table>
<thead>
<tr>
<th>Requirements Elicitation</th>
<th>Requirements Analysis</th>
<th>System Design</th>
<th>Object Design</th>
<th>Implementation</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case Model</td>
<td>Application Domain Objects</td>
<td>SubSystems</td>
<td>Implementation Domain Objects</td>
<td>Source Code</td>
<td>Test Cases</td>
</tr>
</tbody>
</table>

Use Case Model

Definition: Object Modeling

**Main goal**: Find the important abstractions

What happens if we find the **wrong abstractions**?

- **Iterate** and correct the model

**Steps** during object modeling:

1. Class identification (based on the fundamental assumption that we can find abstractions)
2. Find the attributes
3. Find the methods
4. Find the associations between classes (Order of steps is secondary, only a heuristic)

Do UML associations have direction?

If you want to make A a client, and B a server, you can make the association unidirectional. The arrowhead points to the server role:

- Class A (the "client") accesses class B ("the server").
- B is also called **navigable**

Do UML associations have direction?

An association between two classes is by default a bi-directional mapping.

- Class A can access class B and class B can access class A
- Both classes play the agent role.

Aggregation

- Models "part of" hierarchy
- Useful for modeling the breakdown of a product into its component parts (sometimes called bill of materials (BOM) by manufacturers)
- UML notation: Like an association but with a small diamond indicating the assembly end of the relationship.
Aggregation vs. Inheritance

Both associations describe trees (hierarchies).
- Aggregation tree describes "part-of" relationships (also called and-relationship)
- Inheritance tree describes "kind-of" relationships (also called or-relationship)

Aggregation relates instances (involves two or more different objects)

Inheritance relates classes (a way to structure the description of a single object)

Other Associations

"Uses":
- A subsystem uses another subsystem (System Design)

"Contains":
- Sometimes called "spatial aggregation"
- ... contains ...
- Example: A UML package contains another UML package

Parent/child relationship:
- ... is father of ...
- ... is mother of ...

Seniority:
- ... is older than ...
- ... is more experienced than ...

Object Types

Entity Objects:
- Represent the persistent information tracked by the system (Application domain objects, "Business objects")

Boundary Objects:
- Represent the interaction between the user and the system

Control Objects:
- Represent the control tasks performed by the system

Roles

- A role name is the name that uniquely identifies one end of an association.
- A role name is written next to the association line near the class that plays the role.
Roles

When do you use role names?
- Necessary for associations between two objects of the same class
- Also useful to distinguish between two associations between the same pair of classes

When do you not use role names?
- If there is only a single association between a pair of distinct classes, the names of the classes serve as good role names

Example of Role

Problem Statement:
A person assumes the role of repairer with respect to another person, who assumes the role of inspector with respect to the first person.

Roles in Associations

Client Role:
- An object that can operate upon other objects but that is never operated upon by other objects.

Server Role:
- An object that never operates upon other objects. It is only operated upon by other objects.

Agent Role:
- An object that can both operate upon other objects and be operated upon by other objects. An agent is usually created to do some work on behalf of an actor or another agent.

Qualification

The qualifier improves the information about the multiplicity of the association between the classes. It is used for reducing 1-to-many multiplicity to 1-1 multiplicity.

Example: Without qualification, a directory has many files. A file belongs to only one directory.

With qualification: A directory has many files, each with a unique name.

Software Lifecycle Activities

What is this?
Erroneous State (“Error”)

Algorithmic Fault

Examples of Faults and Errors

- Faults in the Interface specification
  - Mismatch between what the client needs and what the server offers
  - Mismatch between requirements and implementation
- Algorithmic Faults
  - Missing initialization
  - Branching errors (too soon, too late)
  - Missing test for nil
- Mechanical Faults (very hard to find)
  - Documentation does not match actual conditions or operating procedures
- Errors
  - Stress or overload errors
  - Capacity or boundary errors
  - Timing errors
  - Throughput or performance errors

How do we deal with errors and faults?

Verification?
Dealing with Errors

Verification:
- Assumes hypothetical environment that does not match real environment
- Proof might be buggy (omits important constraints; simply wrong)

Modular redundancy:
- Expensive

Declaring a bug to be a “feature”
- Bad practice

Patching
- Slows down performance

Testing (this lecture)
- Testing is never good enough

Another View on How to Deal with Errors

Error prevention (before the system is released):
- Use good programming methodology to reduce complexity
- Use version control to prevent inconsistent system
- Apply verification to prevent algorithmic bugs

Error detection (while system is running):
- Testing: Create failures in a planned way
- Debugging: Start with unplanned failures
- Monitoring: Deliver information about state. Find performance bugs

Error recovery (recover from failure once the system is released):
- Modular redundancy
- Recovery blocks
Some Observations

It is impossible to completely test any nontrivial module or any system.

- **Theoretical limitations**: Halting problem
- **Practical limitations**: Prohibitive in time and cost

Testing can only show the presence of bugs, not their absence (Dijkstra).

Testing Takes Creativity

Testing often viewed as dirty work.

To develop an effective test, one must have:

- Detailed understanding of the system
- Knowledge of the testing techniques
- Skill to apply these techniques in an effective and efficient manner

Testing is done best by independent testers.

- We often develop a certain mental attitude that the program should behave in a certain way when in fact it does not.
- Programmers often stick to the data set that makes the program work.
- "Don't mess up my code!"
- A program often does not work when tried by somebody else.
- Don't let this be the end-user.

Fault Handling Techniques

Fault Handling Techniques

- Fault Avoidance
- Fault Detection
- Fault Tolerance

- Design Methodology
- Reviews
- Configuration Management
- Testing
- Debugging

- Modular Redundancy

Component Testing

**Unit Testing:**

- Individual subsystem
- Carried out by developers
- Goal: Confirm that subsystem is correctly coded and carries out the intended functionality

**Integration Testing:**

- Groups of subsystems (collection of classes) and eventually the entire system
- Carried out by developers
- Goal: Test the interfaces among the subsystems

System Testing

**System Testing:**

- The entire system is tested
- Carried out by developers
- Goal: Determine if the system meets the requirements (functional and global)

**Acceptance Testing:**

- Evaluates the system delivered by developers
- Carried out by the client. May involve executing typical transactions on site on a trial basis
- Goal: Demonstrate that the system meets customer requirements and is ready to use

Implementation (Coding) and testing go hand in hand.

Unit Testing

**Informal:**

- Incremental coding

**Static Analysis:**

- Hand execution: Reading the source code
- Walk-Through (informal presentation to others)
- Code Inspection (formal presentation to others)
- Automated Tools checking for
  - syntactic and semantic errors
  - departure from coding standards

**Dynamic Analysis:**

- Black-box testing (Test the input/output behavior)
- White-box testing (Test the internal logic of the subsystem or object)
- Data-structure based testing (Data types determine test cases)