Chapter 7: Object Oriented Design

7.1 Software Development Activities

Wait, how did we end up in COMP-1712?

Software development is a complex process that goes beyond just writing code. As projects grow in size and complexity, proper planning becomes essential.

There are four core development activities:

- establishing the requirements
- creating a design
- implementing the design
- testing

These activities are not strictly sequential - they overlap and interact.

1. **Establishing the Requirements**
    - Defines *what* the program must accomplish (not *how* to do it)
    - Often developed iteratively in conjunction with the client.
    - Requirements are developed iteratively because it is difficult to establish detailed, unambiguous and complete requirements.
    - Usually documented in a *functional specification*
2. **Creating a Design**
    - Specifies *how* the program will meet requirements
    - An object-oriented design defines classes, objects, and their interactions
    - Changes are easier to make during design phase
    - Includes both high-level architecture and low-level method design
3. **Implementing the Design**
    - Translates design into actual code
    - Should be least creative phase - major decisions are made during design
    - Often overemphasized by newer programmers
4. **Testing**
    - Ensures program meets requirements and constraints
    - Involves multiple test runs with various inputs
    - Requires thorough result analysis

Validating Parameters

Methods often have restrictions on parameter values that need to be validated. For example:

- `Integer.parseInt(String arg)` needs argument in integer format
- `Math.sqrt(double x)` requires a non-negative number

- How to handle invalid parameters:
  - `Integer.parseInt` throws `NumberFormatException` for non-integer strings
  - `Math.sqrt` returns `NaN` for negative numbers
    - only possible because Math.sqrt returns a double, which can use the IEEE standard NaN special case we learned about in COMP-1113

Handling Exceptional Cases

When validating parameters, consider all possible combinations:

- Valid parameters: Process normally (including boundary cases like empty lists)
- Invalid parameters: Throw an exception
  - Use `IllegalArgumentException` for invalid arguments
  - Example: `throw new IllegalArgumentException("error message");`

Testing with Exceptions

JUnit 5 provides ways to test exception handling:

```
@Test

void testExpectedException() {

    Assertions.assertThrows(IllegalArgumentException.class,

        () -> { new RationalNumber2(2,0);} );

}
```

try-catch testing:

When testing exception handling with try-catch blocks, we can verify that exceptions are thrown under the expected conditions and handle them appropriately. This approach provides a more detailed view of the exception handling process compared to JUnit assertions, as we can include custom messages and multiple catch blocks if needed.

```
try {

    test1 = new RationalNumber2(2, 0);

    System.out.println("Creation test failed");

} catch (IllegalArgumentException ex) {

    System.out.println("Creation test worked");

}
```

Class and Object Identification

Key principles for identifying classes in object-oriented design:

- Classes can come from:
    - Existing class libraries
    - Previous projects (reuse)
    - New implementations

Objects are typically represented by nouns, while their services (methods) are represented by verbs.

7.2 Identifying Classes and Objects

When designing object-oriented software, we need to determine which classes will make up our program. This is a crucial first step in representing the solution's elements.

Finding Potential Classes

- Examine the problem carefully
- Review functional specifications if available
- Look for **nouns** in program requirements

Of course, not every noun becomes a class. Remember that classes represent groups of objects with similar behaviour. You must decide whether to represent each element as an object or a primitive representation

For example: should employee salary be a float in the Employee class or a separate Salary class? Either could be correct, depending on the program you're developing.

Support Classes

While identifying classes representing the 'nouns' of our program is important, we also need to consider classes that provide services or support functionality. These supporting classes are crucial for organizing code, maintaining separation of concerns, and ensuring the application runs smoothly even though they might not directly represent objects from the problem domain.

Don't Reinvent the Wheel

During real system development, some needed classes may already exist. These could be from the Java standard library, previous solutions, or third-party libraries - all forms of software reuse. Even if an existing class isn't an exact match, it may serve as a foundation for a new class.

Assigning Responsibilities

When identifying classes, always consider their responsibilities. Classes represent objects with behaviours defined by their methods. Class behaviours define program functionality, so we use **verbs** to name them. Early design stages don't require identifying all methods - focus on primary responsibilities and how they translate to methods. If no existing class is appropriate for a required behaviour, this can be a sign that a new class is needed.

7.3 Static Class Members

A class member marked with the `static` keyword belongs to the class itself. Static members are shared among all instances of the class and can be accessed without creating a class instance.

Static Variables

Static variables (aka class variables) are initialized when the class is first loaded, and shared across all instances of a class. Unlike instance variables where each object has its own copy, static variables have only one copy that belongs to the class itself.

```
public class Counter {

    private static int count = 0;  // Shared by all Counter objects



    public Counter() {

        count++;  // Increments the shared counter

    }

}
```

Constants (`final` variables) are often declared `static`, since there may as well be only one copy of an immutable value.

Static Methods

Static methods (aka class methods) are called through the class name rather than an object instance. We've already seen examples of static methods in the Math class. Recall how we did not need an instance of Math to use methods such as `Math.sqrt`:

```
Math.sqrt(27);  // Called directly through class name
```

Static methods cannot reference instance methods/variables within their class- only other static members. For example:

```
public class Example {

    private int instanceVar = 0;
    private static int staticVar = 0;


    public static void staticMethod() {
        staticVar = 1;     // OK - can access static variables
        instanceVar = 1;   // ERROR - cannot access instance variables
    }
}
```