Topic 4: Guidelines for Class Design

Part 3: Design by Contract, Defensive Programming, and Unit Testing (Ch. 3.6, 3.7)

Design by Contract

Vs Defensive Programming

Motivation

- What can go wrong with using the following?

```
double squareRoot(double n) {
    ... // compute x
    return x;
}
```

- So, why do your classes interact correctly?
- Your client code agrees to a contract

- Your classes check all arguments and operations for correctness

Design / Programming by Contract

- **Design / Programming by Contract**: Each method and class has a contract. Two perspectives:
  - Client code
  - Class

- **Precondition**: What the client ensures before calling the method.
- **Postcondition**: What the class ensures when method finishes.

Example:

- Consider the following implementation:

```
/**
 * Removes top element from the stack
 * @pre          stack is not empty
 * @post         stack is not full,
 * @post         top element removed,
 * @post         size decreased by one
 * @throws       NullPointerException if Stack empty
 */
public void pop(){
        elements.remove(0);
}
```

- It is the client's responsibility to ensure contract preconditions are not violated
  - i.e. must call `Stack.isEmpty()` before calling `pop()`

Defensive Programming

- **Defensive Programming**: A class is responsible for maintaining a correct state
  - All input values and actions are checked for correctness.
  - ie: prevent adding a duplicate element to a "set"
  - ie: prevent adding an element to a full array.
- Find bad inputs/actions and **fail fast**
  - Assertions

Defensive Programming (2)

- Assert (basics)
  - Usage: `assert condition;`
  - If the condition is false, halts the program (throws AssertionError)
- Example Statement:

```
assert age >= 0;
```

- Example Method:

```
public void pop() {
    assert !isEmpty();
    elements.remove(0);
}
```

Summary

- Should a square-root method check that the input is non-negative?
  - Design by Contract: that's the client's job!
  - Defensive Programming: client may call us with a bad value we should check.
- Benefit of Design by Contract
  - Removes duplicate validity checks - otherwise client & class check for valid values.
  - Duplicate checks make system more complicated.
- Benefit of Defensive Programming
  - Errors in calling code are caught quickly - Should use for all calls accessible by untrusted code.

Options for Error handling

1. Do Nothing
   - ie: sqrt() w/o any checking or documentation,
2. Check preconditions
   - Works best with language support.
   - ie: sqrt() w/o any checking, but with documentation
3. Fail fast
   - (assert) - Check for programmer errors
   - ie: sqrt() w/ assert
4. Raise exception
   - ie: sqrt() w/ exception
5. Return invalid indicator
   - ie: sqrt() w/ return -1
6. Correct the problem
   - Given incorrect input, try to correct it as best as possible.
   - ie: sqrt() w/ abs(x) call to make positive.

Asserts

Testing code sanity

Assertions

- Assert statements - Trigger a runtime error if a condition is false

- Example Usage

```
double rSquared = getCircleArea() / Math.PI;
assert rSquared >= 0;
double r = squareroot(rSquared);
```

Enabling Assertions

- Enabling Assertions
- Turned on/off at runtime by JVM. Use VM option

```
-ea     OR
-enableassertions
```

- In IntelliJ
  Run → Edit Configurations → VM Options
- (AssertDemo.java)

Assert Usage

- Assertions should check for "invalid" conditions, which should crash the program.
- Guide to using Asserts
  - Assert the expectations you place on programmers
    - ie: Calling pop() on a non-empty stack.
  - Don't assert things that could reasonably be false.
    - ie: Don't assert a user's input is > 0
    - Must check for and handle these errors.

Assert Usage (2)

- Do not assert conditions that would already cause runtime errors
  - ie: `assert array != null;`
- Use assertions to catch unanticipated cases.
  - ie: in a switch statement:

```
switch(productType) {
case prod.Software:
        // ...
        break;
case prod.Hardware:
        // ...
        break;
default:
        assert false;
}
```

Assert Usage (3)

- Don’t assert the impossible

```
int age = getUserAge();
if (age < 50) {
       // ...
} else if (age >= 50) {
       // ...
} else {
       assert false;
}
```

Assert Issues

- Too many assertion statements can be detrimental
  - slow down program
  - Complicate code
- Should not be used for runtime error handling
- Possible errors should be handled by exceptions
  - ie: file not found error

Unit Testing

JUnit

Unit Testing

- A unit test demonstrates the correctness of a single class
  - Tests class in isolation
- A common testing framework to use is **JUnit**
  - Contains a set of tools that can be incorporated into a test class
  - Each test case needs to be placed in a method whose name starts with `test`

```
import junit.framework.*;
public class TestClassName extends TestCase {
    public void testSomething() { ... }
    ... Write tests here ...
}
```

  - To compile the test class, we need to include the `junit.jar` file

```
javac -classpath .:junit.jar TestClassName.java
```