Lecture 3: Specifications and Testing

CS 2110

September 1, 2026

Today's Learning Outcomes

11. Identify the pre-conditions, post-conditions, and side effects of a method and use these to write detailed Javadoc specifications.

13. Write comprehensive unit tests for a method given only its specifications and signature.

15. Explain the process of *test-driven development* and identify its potential benefits.

16. Differentiate between *black-box* and *glass-box* testing and identify scenarios where each is appropriate.

Client and Implementer Roles

Large, long-lasting software systems require coordination between developers.

Different people write different parts of the code, which must later be integrated.

A single developer interacts with a code module (a class, package, or **method**) from one (or both) of two possible perspectives:

1. Client

2. Implementer

Method Specifications

What information does a client need to (properly) use a method in their code?

Pre-conditions and Post-conditions

Pre-conditions: Properties that must be true when a method is called

Post-conditions: Properties that will be true when a method returns **if** the pre-conditions were met.

Specifying Pre-conditions

```
1  /**
2   * Returns `true` if the character in `str` at index `i` is uppercase; otherwise
3   * returns `false`. Requires that _____________________________.
4   */
5  static boolean uppercaseAt(String str, int i) { ... }
```

Handling Pre-condition Violations

When the client violates a method's pre-conditions, it does not offer any guarantees about its behavior. **Any** behavior satisfies its specifications!

Coding Demo: Defensive Programming

(Bad) Pre-condition Violation Handling

Throwing a "more appropriate" exception

```java
/** Returns `true` if the character in `str` at index `i` is uppercase, otherwise
 *  returns `false`. Requires that `0 <= i < str.length()`. */
static boolean uppercaseAt(String str, int i) {
  if (i < 0 || i >= str.length()) {
    throw new IllegalArgumentException("`i` must be between 0 and `str.length()-1`.");
  }
  return Character.isUpperCase(str.charAt(i));
}
```

"Correcting" the inputs

```java
static boolean uppercaseAt(String str, int i) {
  return Character.isUpperCase(Math.clamp(i, 0, str.length() - 1));
}
```

Unit Testing

Unit tests check whether a method/class *conforms to its specifications*.

Many unit tests follow the same basic structure:

1.
2.
3.

Writing many separate unit tests for each method/class allows us to more easily diagnose issues as we code.

Coding Demo: JUnit

Good Unit Test Suites

Soundness:

Coverage:

Testing Methods with Side Effects

```
1  /** Zeroes out all entries before and including the first instance of `key` in `arr`
2   *  and returns the index where `key` was found. If `key` is not present in `arr`,
3   *  then all indices of `arr` are zeroed out, and `arr.length` is returned. */
4   static int zeroThrough(int[] arr, int key) { ... }
```

Underspecification

Sometimes, there can be multiple different return values (or side effect behaviors) that all conform to a method's specifications.

In this case, we say that the method is **underspecified**.

Testing with Underspecification

```
1  /** Returns an index `i` with `0 < i < a.length-1` corresponding to a local
2    * maximum of array `a`, meaning `a[i] > a[i-1]` and `a[i] > a[i+1]`.
3    * Returns 0 if `a` does not  contain a local maximum. */
4  static int findLocalMax(int[] a) { ... }
```

```
@Test
void testMultipleMaxima() {
  int[] a = {10, 12, 15, 13, 16, 14, 11};
  int i = findLocalMax(a);


}
```

```
@Test
void testMultipleMaxima() {
  int[] a = {10, 12, 15, 13, 16, 14, 11};
  int i = findLocalMax(a);


}
```

Test-Driven Development

= Use specs to write unit tests **first**, then use tests to guide development.

Many Benefits:

But coming up with tests can be tricky… It takes some practice.

Black Box vs. Glass Box Testing

Black Box Testing: Tests are written without access to the source code. These tests are based on only specs, not any implementation details.

Glass Box Testing: Use implementation details to inform test suite.