Principles of Software Construction: Objects, Design, and Concurrency

Specifications and unit testing, exceptions

Christian Kästner **Vincent Hellendoorn**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Explicit over Implicit

Can anything go wrong with this?

```
int add(int a, int b) {
  return a + b;
}
```

Explicit over Implicit

Can anything go wrong with this?

```
int add(int a, int b) {
  return a + b;
}
```

How about this:

```
int divide(int a, int b) {
  return a / b;
}
```

Explicit over Implicit

Can anything go wrong with this?

```
int add(int a, int b) {
  return a + b;
}
```

How about this:

```
int divide(int a, int b) {
  return a / b;
}
divide(4, 3); // 1
```

Explicit over Implicit

Can anything go wrong with this?

```java
int add(int a, int b) {
  return a + b;
}
```

How about this:

```java
int divide(int a, int b) {
  return a / b;
}
divide(4, 3); // 1
divide(2, 0); // Exception
  java.lang.ArithmeticException: / by zero
```

Explicit over Implicit

BTW, harder to force in TS*:

```
function divide(a: bigint, b: bigint): bigint {
    return a / b;
}
divide(4n, 3n); // 1
divide(2n, 0n); // RangeError: Division by zero
```

*Compile with: `--target es2020`

Explicit over Implicit

Most real-world code has a **contract**.

- It might not be obvious to you!
- This is why we:
  - Encode specifications
  - Test
  - Use exceptions
- Imperative to build systems that scale

Today

1. Exception Handling
2. Unit Testing
3. Specifications

Exceptions

- Inform caller of problem by transfer of control
  - They split control-flow into a “normal” and an “erroneous” branch
  - Compare “`if/else`”
- Semantics
  - Propagates up the call stack until exception is caught, or main method is reached
    - So, it can terminate the program!
- Where do exceptions come from?

Exceptions

Just try:

```
String read(String path) {
  return Files.lines(Path.of(path))
              .collect(Collectors.joining("\n"));
}
```

Handling Exceptions

```
String read(String path) {
  try {
     return Files.lines(Path.of(path))
              .collect(Collectors.joining(“\n”));
  }
  catch (IOException e) {
    // implement fall-back behavior.
  }
}
```

Handling Exceptions

```
String read(String path) throws IOException {
  return Files.lines(Path.of(path))
             .collect(Collectors.joining(“\n”));
}
```

Benefits of exceptions

- You can't forget to handle common failure modes
  - Explicit > implicit
  - Compare: using a flag or special return value
- Provide high-level summary of error
  - Compare: core dump in C/C++
- Improve code structure
  - Separate normal code path from exceptional
  - Error handling code is segregated in catch blocks
- Ease task of writing robust, maintainable code

Exception Handling

Undeclared vs. Declared

```
int divide(int a, int b) {
  return a / b;
}
```

```
String read(String path) throws
                        IOException {
  return Files.lines(Path.of(path))
     .collect(Collectors.joining("\n"));
}
```

Exception Handling

Undeclared vs. Declared

```
int divide(int a, int b) {
  return a / b;
}
```

```
String read(String path) throws
                          IOException {
  return Files.lines(Path.of(path))
     .collect(Collectors.joining(“\n”));
}
```

Unchecked vs. Checked

```
divide(4, 3); // Compiles
                 fine
```

```
read(“test.txt”); // Unhandled
  exception: java.io.IOException
```

Exception Handling

Handling unchecked exceptions is not enforced by the compiler

These are quite common

- E.g., all exceptions in C++
- In Java: any exception that extends Error or RuntimeException

Exception Handling

Handling unchecked exceptions is not enforced by the compiler

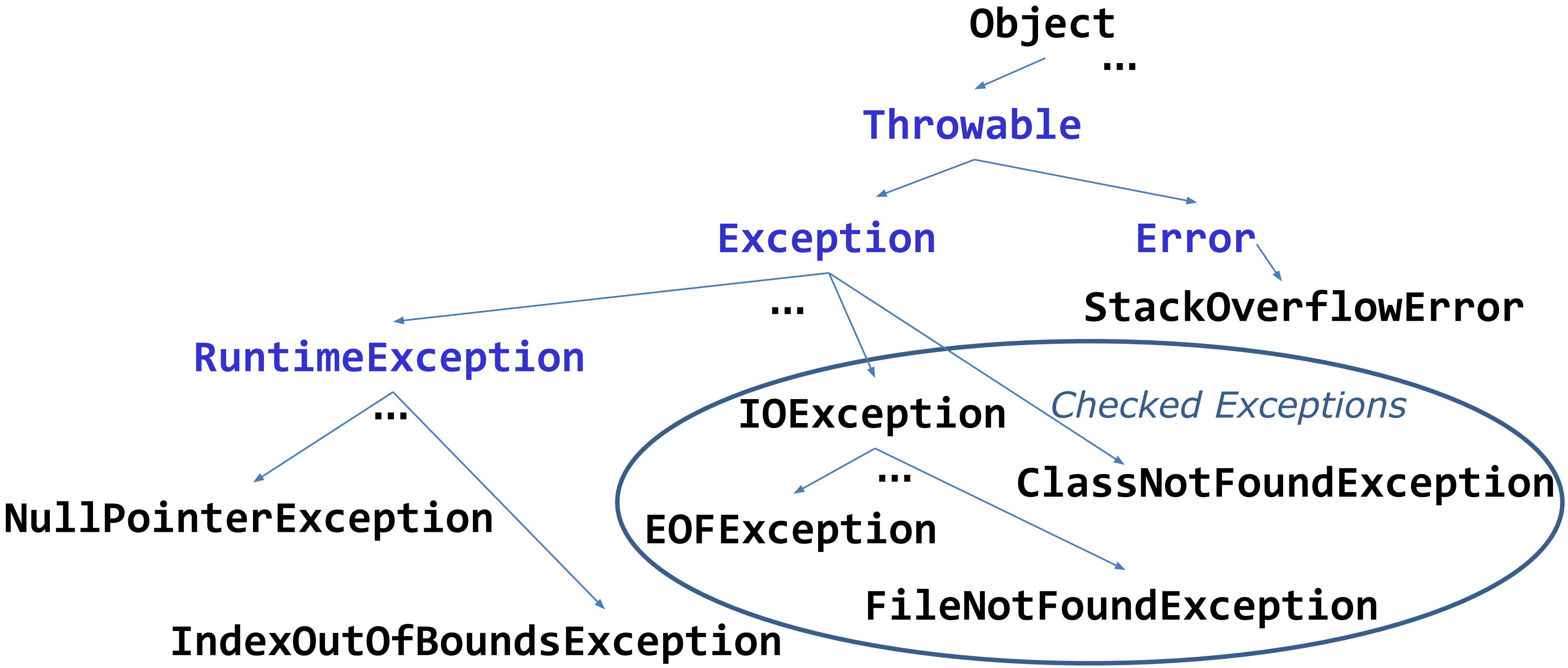
These are quite common

- E.g., all exceptions in C++
- In Java: any exception that extends Error or RuntimeException
  - E.g.:

```
int divide(int a, int b) throws ArithmeticException {
  return a / b;
}
divide(4, 3); // Compiles fine
```

  - **Note:** we don't typically declare unchecked exceptions.

Java's exception hierarchy (messy)



Design choice: checked vs. unchecked

- Unchecked exception
  - Programming error, other unrecoverable failure
- Checked exception
  - An error that every caller should be aware of and handle
- Special return value (e.g., null from Map.get)
  - Common but atypical result (not erroneous!)
- Do not use error codes – too easy to ignore
- Avoid null return values
  - Never return null instead of zero-length list or array

Defining & using Exception Types

```
class BufferBoundsException extends Throwable {
  public BufferBoundsException(String message) {
    ...
  }
}

void atIndex(int[] buff, int i) throws CustomException {
  if (buff.length <= i)
    throw new BufferBoundsException("...");
  return buff[i];
}
```

Exception Handling

- It's still wise to guard for "obvious" unchecked exceptions

```
if (arr.length > 10)
  return arr[10];
```

- Or explicitly signal the problem, recall:

```
if (buff.length <= i)
  throw new BufferBoundsException("...");
return buff[i];
```

- Why is this better than letting the index fail?

Exception Handling

- It's still wise to guard for "obvious" unchecked exceptions

```
if (arr.length > 10)
  return arr[10];
```

- Or explicitly signal the problem, recall:

```
if (buff.length <= i)
  throw new BufferBoundsException("...");
return buff[i];
```

- Why is this better than letting the index fail?
  - `BufferBoundsException` can be a checked exception!
  - Which forces someone to handle it
  - Here, we declared: `atIndex(int[] buff, int i) throws BufferBoundsException`
  - So every calling method must handle it, or throw it on

Guidelines for using exceptions (1)

- Avoid unnecessary checked exceptions (EJ Item 71)
- Favor standard exceptions (EJ Item 72)
  - `IllegalArgumentException` – invalid parameter value
  - `IllegalStateException` – invalid object state
  - `NullPointerException` – null param where prohibited
  - `IndexOutOfBoundsException` – invalid index param
  - `IOException` -- and its subclasses, mostly for File-related actions
- Throw exceptions appropriate to abstraction (EJ Item 73)

Guidelines for using exceptions

- Document all exceptions thrown by each method
  - Unchecked as well as checked (EJ Item 74)
  - But don't *declare* unchecked exceptions!
- Include failure-capture info in detail message (Item 75)

```
throw new IlegalArgumentException(
    "Quantity must be positive: " + quantity);
```

Guidelines for using exceptions (2)

- Document all exceptions thrown by each method
  - Unchecked as well as checked (EJ Item 74)
  - But don't *declare* unchecked exceptions!
- Include failure-capture info in detail message (Item 75)

```
throw new IllegalArgumentException(
    "Quantity must be positive: " + quantity);
```

- Don't ignore exceptions (EJ Item 77)

```
try {
   processPayment(payment);
}
catch (Exception e) {  // BAD!
}
```

Cleanup

Exception handling often also supports cleaning up

```
openMyFile();
try {
  writeMyFile(theData); // This may throw an error
} catch(e) {
  handleError(e); // If an error occurred, handle it
} finally {
  closeMyFile(); // Always close the resource
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling

Manual Resource Termination

Is ugly and error-prone, especially for multiple resources

- Even good programmers usually get it wrong
  - Sun's Guide to Persistent Connections got it wrong in code that claimed to be exemplary
  - Solution on page 88 of Bloch and Gafter's Java Puzzlers is badly broken; no one noticed for years
- 70% of the uses of `close` **in the JDK itself** were wrong in 2008!
- Even the "correct" idioms for manual resource management are deficient

The solution: `try`-with-resources

Automatically closes resources!

```
try (DataInputStream dataInput =
        new DataInputStream(new FileInputStream(fileName))) {
    return dataInput.readInt();
} catch (IOException e) {
    ...
}
```

Exceptions Across Languages

Alas, try-with-resources does not exist in JS/TS

- Neither does ‘`throws`’

Exception structures differ radically across languages

- Most languages have ‘`try/catch`’ and ‘`throw`’
  - Some have ‘`finally`’
- Python has ‘`with`’ for resource management (since 2006)
  - C# has ‘`using`’
  - Java’s try-with-resources was added in 2011
- Go returns an error-typed value, to be checked for nullity

Exceptions Across Languages

Use what you have

- When possible, be explicit
  - Use the compiler to enforce, where possible
  - Pro-actively pre-empt corner-cases, where not
    - Unchecked exceptions, JS/TS
- Make exceptions part of your contract

Outline

1. Exception Handling
2. **Unit Testing**
3. Specifications

Testing

How do we know this works?

```
int isPos(int x) {
  return x >= 1;
}
```

Testing

How do we know this works?

```
int isPos(int x) {
  return x >= 1;
}

@Test
void testIsPos() {
  assertTrue(isPos(1));
}
```

Testing

Are we done?

Testing

How do we know this works?

Testing

Are we done?

```
int isPos(int x) {
  return x >= 1;
}

@Test
void testIsPos() {
  assertTrue(isPos(1));
}

@Test
void testNotPos() {
  assertFalse(isPos(-1));
}
```

Testing

How do we know this works?

Testing

Are we done?

```
int isPos(int x) {
  return x >= 0;  // What if?
}

@Test
void testIsPos() {
  assertTrue(isPos(1));
}

@Test
void testNotPos() {
  assertFalse(isPos(-1));
}
```

Testing

How do we know this works?

Testing

Are we done?

```
int isPos(int x) {
  return x >= 0;  // What if?
}

@Test
void test1IsPos() {
  assertTrue(isPos(1));
}

@Test
void test0IsNotPos() {
  assertFalse(isPos(0)); // Fails
}
```

Testing

How do we know a program is correct?

- In a perfect world (maybe): formal verification
  - Easy enough for proving that `isPos(x)` -- the implementation is the definition
  - Tedious, <u>cannot</u> be done automatically
- Hence, testing

Testing

- Execute the program with selected inputs in a controlled environment
  - Why is this related to contracts?

Testing

- Execute the program with selected inputs in a controlled environment
  - Why is this related to contracts?
  - Because we need to know what to test!

Testing

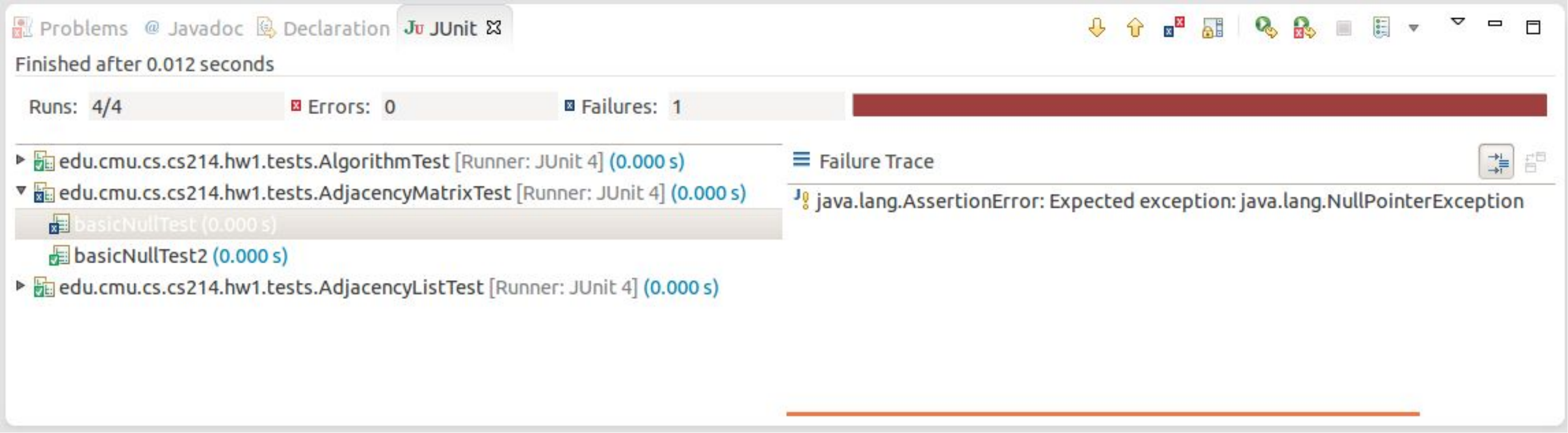
- Execute the program with selected inputs in a controlled environment
  - Why is this related to contracts?
  - Because we need to know what to test!
- Goals
  - Reveal bugs, so they can be fixed (primary goal)
  - Clarify the specification, documentation

Unit Tests

- For “small” units: methods, classes, subsystems
  - Unit is smallest testable part of system
  - Test the parts before assembling them
  - Intended to catch local bugs
- Typically (but not always) written by developers
- Many small, fast-running, independent tests
- Few dependencies on other system parts or environment
- Insufficient, but a good starting point

For Java: JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available, e.g., IntelliJ integration



For Java: JUnit

Syntax:

```
import static org.junit.Assert.*;

class PosTests {

  @Before
  void setUp() {
    // Anything you want to run
       before each test
  }

  @Test
  void test1IsPos() {
    assertTrue(isPos(1));
  }
}
```

For TS: Jest

- In particular, ts-jest
  - Many other options; your choice
- Requires a few files:
  - jest.config.js, to specify testing mode
  - package.json with (ts-)jest dependencies
- Provides useful features:
  - ‘test’, ‘expect’ (= ‘assert’)
  - ‘toBe’, ‘toEqual’
  - ‘fn’, for Mocking (later)

test > TS isPos.test.ts > ...

```
import { isPos } from "../src/isPos"

test('1 is positive', () => {
    expect(isPos(1)).toBe(true);
});

test('-1 is not positive', () => {
    expect(isPos(-1)).toBe(false);
});

test('0 is not positive', () => {
    expect(isPos(0)).toBe(false);
});
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

```
    at Object.<anonymous> (test/isPos.test.ts:12:19)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 2 passed, 3 total
Snapshots:   0 total
```

Writing Testable Code

- Think about testing when writing code
  - Unit testing encourages you to write testable code
- Modularity and testability go hand in hand
  - Same test can be used on multiple implementations of an interface!
- Test-Driven Development
  - A design and development method in which you write tests before you write the code
  - Writing tests can expose API weaknesses!

Run Tests Often

- You should only commit code that passses all tests…
- So run tests before every commit
- If test suite becomes too large & slow for rapid feedback
  - Run local package-level tests (“smoke tests”) frequently
  - Run all tests nightly
  - Medium sized projects often have thousands of test cases
- Continuous integration (CI) servers help to scale testing

Reflections on Testing

"Testing shows the presence, not the absence of bugs."

Edsger W. Dijkstra, 1969

"Functionality that can't be demonstrated by automated test simply don't exist."

Kent Beck

Boundary Value Testing

We cannot test for every integer.

Choose *representative* values:
1 for positives, -1 for negatives

And *boundary cases*: 0 is a likely candidate for mistakes

- Think like an attacker

```
int isPos(int x) {
  return x >= 0;  // What if?
}

@Test
void test1IsPos() {
  assertTrue(isPos(1));
}

@Test
void test0IsNotPos() {
  assertFalse(isPos(0)); // Fails
}
```

Outline

1. Exception Handling
2. Unit Testing
3. **Specifications**

Specifications

So what exactly do you test?

- What it claims to do: specification testing
- What it does: structural testing

What is a contract?

- Agreement between an object and its user
  - What object provides, and user can count on
- Includes:
  - Method signature (type specifications)
  - Functionality and correctness expectations
  - Sometimes: performance expectations
- **What** the method does, not **how** it does it
  - **Interface** (API), not **implementation**
- "Focus on concepts rather than operations"

Method contract details

- Defines method's and caller's responsibilities
- Analogy: legal contract
    - If you pay me this amount on this schedule…
    - I will build a room with the following detailed spec
    - Some contracts have remedies for nonperformance
- Method contract structure
    - Preconditions: what method requires for correct operation
    - Postconditions: what method establishes on completion
    - Exceptional behavior: what it does if precondition violated
- Defines correctness of implementation

How to Encode Specifications?

Formal frameworks exist, to capture pre- and post-conditions

- E.g., ‘`requires arr != null`’
- Useful for formal verification
- But rarely used
  - Takes a lot of effort, and doesn’t scale well

How to Encode Specifications?

More common: prose specification. Document:

- Every parameter
- Return value
- Every exception (checked and unchecked)
- What the method does, including
    - Primary purpose
    - Any side effects
    - Any thread safety issues
    - Any performance issues

How to Encode Specifications?

More common: prose specification. Document

- Every parameter
- Return value
- Every exception (checked and unchecked)
- What the method does, including
  - Primary purpose
  - Any side effects
  - Any thread safety issues
  - Any performance issues
- Do **not** document implementation details
  - Known as overspecification

Docstring Specification

```
class RepeatingCardOrganizer {
  ...







  public boolean isComplete(CardStatus card) {
    return card.getResults().stream()
      .filter(isSuccess -> isSuccess)
      .count() >= this.repetitions;
  }
}
```

Docstring Specification

```
class RepeatingCardOrganizer {
  ...
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card) {
    return card.getResults().stream()
      .filter(isSuccess -> isSuccess)
      .count() >= this.repetitions;
  }
}
```

Docstring Specification

```
class RepeatingCardOrganizer {
  ...
  /**
   * Checks if the provided card has been answered correctly the required number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least {@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card) {
    // IGNORE THIS WHEN SPECIFICATION TESTING!
  }
}
```

Docstring Specification

```
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card);

  // What is specified?
```

Docstring Specification

```
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card);

  // What is specified?
  // - Parameter type (no constraints)
```

Docstring Specification

```
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card);

  // What is specified?
  // - Parameter type (no constraints)
  // - Return constraints: “at least” this.repetitions correct answers
```

Docstring Specification

```
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card);

  // What is specified?
  // - Parameter type (no constraints)
  // - Return constraints: “at least” this.repetitions correct answers
  // So what do we test?
```

Docstring Specification

```
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card);

@Test
public void testIsCompleteSingleSuccess() {
  CardRepeater repeater = new RepeatingCardOrganizer(1); // Single repetition
  CardStatus cs = new CardStatus(new FlashCard("", ""));
  cs.recordResult(true); // Single Success
  assert???(repeater.isComplete(cs));
}
```

Docstring Specification

```
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card);

@Test
public void testIsCompleteSingleSuccess() {
  CardRepeater repeater = new RepeatingCardOrganizer(1); // Single repetition
  CardStatus cs = new CardStatus(new FlashCard("", ""));
  cs.recordResult(true); // Single Success
  assertTrue(repeater.isComplete(cs));
}
```

Docstring Specification

```
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card);

@Test
public void testIsNotCompleteSingleFailure() {
  CardRepeater repeater = new RepeatingCardOrganizer(1); // Single repetition
  CardStatus cs = new CardStatus(new FlashCard("", ""));
  cs.recordResult(false); // Single failure
  assertFalse(repeater.isComplete(cs));
}
```

Docstring Specification

```
class RepeatingCardOrganizer {
  ...
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card) {
    return card.getResults().stream()
      .filter(isSuccess -> isSuccess)
      .count() >= this.repetitions;
  }
}
```

We've now run this twice.
Are we done testing?

Specification vs. Structural Testing

You can test for different objectives

- Specification-based testing: test solely the specification
  - Ignores implementation, use inputs/outputs only
  - Cover all specified behavior
- Structural Testing: consider implementation
  - Optimize for various kinds of code coverage
    - Line, Statement, Data-flow, etc. -- More next week

Specification vs. Structural Testing

You can test for different objectives

- Structural Testing:
  - By some definitions, we are done. Full line coverage, branch coverage.
  - Rarely enough, but often adequate
- Specification Testing:
  - Do not rely on code; need to consider corner-cases
  - Think like an attacker

Specification vs. Structural Testing

```
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card) {
    return card.getSuccesses.get(0);  // <-- Bad, but passes both tests
  }
```

Outlook

Homework 2 is all about testing

- Specification-testing the FlashCard system
- Some structural testing as well
  - More next Tuesday, also on coverage, test-case design
- To be released fairly soon

Summary

- Being explicit about program behavior is ideal
  - Helps you detect bugs
  - Forces handling of special cases -- a key source of bugs
  - Increases transparency of your program's interface
- Specification comes in multiple forms
  - Explicit contracts, formal or informal
  - Compile-time signals, e.g. through exceptions
  - Testing helps clarify, often improve specifications
    - TDD takes this to the extreme
    - <u>You rarely know your code until you test it</u>