Principles of Software Construction: Objects, Design, and Concurrency

Invariants, immutability, and testing

Charlie Garrod          Chris Timperley
Administrivia

• Homework 4a due Thursday at 11:59 p.m.
  – Mandatory design review meeting before the homework deadline
• Final exam is Monday, December 9th, 1–4pm
Outline

• Class invariants and defensive copying
• Immutability
• Testing and coverage
• Testing for complex environments
Class invariants

• Critical properties of the fields of an object
• Established by the constructor
• Maintained by public method invocations
  – May be invalidated temporarily during method execution
Safe languages and robust programs

• Unlike C/C++, Java language safe
  – Immune to buffer overruns, wild pointers, etc.
• Makes it possible to write robust classes
  – Correctness doesn’t depend on other modules
  – Even in safe language, requires programmer effort
Defensive programming

• Assume clients will try to destroy invariants
  – May actually be true (malicious hackers)
  – More likely: honest mistakes

• Ensure class invariants survive any inputs
  – Defensive copying
  – Minimizing mutability
This class is not robust

public final class Period {
    private final Date start, end; // Invariant: start <= end

    /**
     * @throws IllegalArgumentException if start > end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.after(end))
            throw new IllegalArgumentException(start + " > " + end);
        this.start = start;
        this.end   = end;
    }

    public Date start() { return start; }
    public Date end()   { return end; }
    ...
    // Remainder omitted
}
The problem: Date is mutable

*Obsolete as of Java 8; sadly not deprecated even in Java 11*

// Attack the internals of a Period instance
Date start = new Date();  // (The current time)
Date end   = new Date();  //   "     "      "
Period p = new Period(start, end);
end.setYear(78);   // Modifies internals of p!
The solution: defensive copying

// Repaired constructor - defensively copies parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end   = new Date(end.getTime());
    if (this.start.after(this.end))
        throw new IllegalArgumentException(start + " > "+ end);
}
A few important details

- Copies made before checking parameters
- Validity check performed on copies
- Eliminates window of vulnerability between validity check & copy
- Thwarts multithreaded TOCTOU attack
  - Time-Of-Check-To-Time-Of-U

// BROKEN - Permits multithreaded attack!
public Period(Date start, Date end) {
    if (start.after(end)) {
        throw new IllegalArgumentException(start + " > " + end);
    }
    // Window of vulnerability
    this.start = new Date(start.getTime());
    this.end   = new Date(end.getTime());
}
Another important detail

• Used constructor, not clone, to make copies
  – Necessary because Date class is nonfinal
  – Attacker could implement malicious subclass
    • Records reference to each extant instance
    • Provides attacker with access to instance list

• But who uses clone, anyway? [EJ Item 11]
Unfortunately, constructors are only half the battle

// Accessor attack on internals of Period
Period p = new Period(new Date(), new Date());
Date d = p.end();
p.end.setYear(78); // Modifies internals of p!
The solution: more defensive copying

// Repaired accessors - defensively copy fields
public Date start() {
    return new Date(start.getTime());
}
public Date end() {
    return new Date(end.getTime());
}

Now Period class is robust!
Summary

• Don’t incorporate mutable parameters into object; make defensive copies
• Return defensive copies of mutable fields…
• Or return unmodifiable view of mutable fields
• **Real lesson – use *immutable* components**
  – Eliminates the need for defensive copying
Outline

• Class invariants and defensive copying
• **Immutability**
• Testing and coverage
• Testing for complex environments
Immutable classes

• **Class whose instances cannot be modified**
• Examples: String, Integer, BigInteger, Instant
• How, why, and when to use them
How to write an immutable class

• Don’t provide any mutators
• Ensure that no methods may be overridden
• Make all fields final
• Make all fields private
• Ensure security of any mutable components
public final class Complex {
    private final double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Getters without corresponding setters
    public double realPart() {
        return re;
    }
    public double imaginaryPart() {
        return im;
    }

    // minus, times, dividedBy similar to add
    public Complex plus(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
}
@Override public boolean equals(Object o) {
    if (!(o instanceof Complex)) return false;
    Complex c = (Complex) o;
    return Double.compare(re, c.re) == 0 &&
           Double.compare(im, c.im) == 0;
}

@Override public int hashCode() {
    return 31 * Double.hashCode(re) + Double.hashCode(im);
}

@Override public String toString() {
    return String.format("%d + %di", re, im);
}
Distinguishing characteristic

• Return new instance instead of modifying
• *Functional programming*
• May seem unnatural at first
• Many advantages
Advantages

• Simplicity
• Inherently Thread-Safe
• Can be shared freely
• No need for defensive copies
• Excellent building blocks
Major disadvantage

- **Separate instance for each distinct value**
- Creating these instances can be costly
  
  ```java
  BigInteger moby = ...;  // A million bits long
  moby = moby.flipBit(0);  // Ouch!
  ```

- Problem magnified for multistep operations
  - Well-designed immutable classes provide common multistep operations
    - e.g., `myBigInteger.modPow(exponent, modulus)`
  - Alternative: mutable companion class
    - e.g., `StringBuilder` for `String`
When to make classes immutable

• **Always, unless there's a good reason not to**
• Always make small “value classes” immutable!
  – Examples: Color, PhoneNumber, Unit
  – Date and Point were mistakes!
  – Experts often use long instead of Date
When to make classes mutable

• Class represents entity whose state changes
  – Real-world - BankAccount, TrafficLight
  – Abstract - Iterator, Matcher, Collection
  – Process classes - Thread, Timer

• If class must be mutable, *minimize mutability*
  – Constructors should fully initialize instance
  – Avoid reinitialize methods
Outline

• Class Invariants
• Immutability
• Testing and coverage
• Testing for complex environments
Why do we test?
Testing decisions

• Who tests?
  – Developers who wrote the code
  – Quality Assurance Team and Technical Writers
  – Customers

• When to test?
  – Before and during development
  – After milestones
  – Before shipping
  – After shipping

• When to stop testing?
Test driven development (TDD)

- **Write tests before code**
- Never write code without a failing test
- Code until the failing test passes

From *Growing Object-Oriented Software* by Nat Pryce and Steve Freeman
http://www.growing-object-oriented-software.com/figures.html

@sebrose http://cucumber.io
Why use test driven development?

• Forces you to think about interfaces early
• Higher product quality
  – Better code with fewer defects
• Higher test suite quality
• Higher productivity
• It’s fun to watch tests pass
TDD in practice

• Empirical studies on TDD show:
  – May require more effort
  – May improve quality and save time

• Selective use of TDD is best

• Always use TDD for bug reports
  – Regression tests
Testing decisions

- **Who tests?**
  - Developers who wrote the code
  - Quality Assurance Team and Technical Writers
  - Customers

- **When to test?**
  - Before and during development
  - After milestones
  - Before shipping
  - After shipping

- **When to stop testing?**
How much testing?

• You generally cannot test all inputs
  – Too many – usually infinite
  – Limited time and resources

• But when it works, exhaustive testing is best!
What makes a good test suite?

• Provides high confidence that code is correct
• Short, clear, and non-repetitious
  – Prefer smaller, more-directed tests
  – More difficult for test suites than regular code
  – Realistically, test suites will look worse
• Can be fun to write if approached in this spirit
Black-box testing

- **Look at specifications, not code**
- Test representative cases
- Test boundary conditions
- Test invalid (exception) cases
- Don’t test unspecified cases
White-box testing

• Look at specifications **and** code
• Write tests to:
  – Check interesting implementation cases
  – Maximize branch coverage
Code coverage metrics

- Method coverage – coarse
- Branch coverage – fine
- Path coverage – too fine
  - Cost is high, value is low
  - (Related to cyclomatic complexity)
- ...
Coverage metrics: useful but dangerous

- **Can give false sense of security**
- **Examples of what coverage analysis could miss**
  - Data values
  - Concurrency issues – race conditions, etc.
  - Usability problems
  - Customer requirements issues
- **High branch coverage is not sufficient**
Summary: Test suites – ideal and real

• Ideal test suites would
  – Uncover all errors in code
  – Test “non-functional” attributes such as performance and security
  – Minimum size and complexity

• Real test Suites
  – Uncover some portion of errors in code
  – Have errors of their own
  – Are nonetheless priceless
Automated Test Generation
Fuzz Testing

• Also known as *random input testing, torture testing*
• Try “random” inputs, as many as you can
  – Choose inputs to tickle interesting cases
  – Knowledge of implementation helps here
• **Seed random number generator so tests repeatable**
• Successful in some domains (parsers, file processing, ...)
  – But, many tests execute similar paths
  – Generally hard to reach certain program states
  – Often finds only superficial errors
Oracle Problem

How should my program behave for any given input?

```java
2468   /** Returns a BigInteger whose value is the greatest common divisor of
2469   * (this.abs) and (abs(val)). Returns 0 if
2470   * (this.abs == 0 && val == 0).
2471   * @param val value with which the GCD is to be computed.
2472   * @return (this.abs GCD(abs(this), abs(val)))
2473   */
2474   public BigInteger gcd(BigInteger val) {
2475       if (val.signum == 0) {
2476           return this.abs();
2477       } else if (this.signum == 0)
2478           return val.abs();
2479
2480       BigInteger a = new BigInteger(this);
2481       BigInteger b = new BigInteger(val);
2482       BigInteger result = a.hardSgcd(b);
2483       return result.toBigInteger();
2484   }
```
A simple oracle: The program shouldn’t crash

+ No need to manually specify an oracle!
+ Relatively low engineering effort
- Limited to crashing bugs

American Fuzzy Lop (AFL)

https://domesticanimalbreeds.com/american-fuzzy-lop-rabbit-everything-you-need-to-know/
http://lcamtuf.coredump.cx/afl/
https://embed.cs.utah.edu/csmith/
Another alternative: Differential Testing

Use an existing, functionally-equivalent implementation as a reference. (E.g., a correct implementation with undesirable non-functional properties.)
Another alternative: Differential Testing

Alternatively, we can use an older, correct implementation.
No reference implementation? Property-based testing

Unit testing generally relies on checking concrete input-output examples. Property-based testing checks that certain \textit{properties} hold true for all possible inputs.

- Attempts to generates inputs that violate properties.
- Easier to specify than expected outputs!
- What properties should I check?

```java
@RunWith(JUnitQuickcheck.class)
public class StringProperties {
    @Property public void concatenationLength(String s1, String s2) {
        assertEquals(s1.length() + s2.length(), (s1 + s2).length());
    }
}
```

\url{https://github.com/pholser/junit-quickcheck}
EvoSuite: Automated Test Generation for Java

- Generates minimal, coverage-maximizing test suites.
- Uses dynamic specification inference to suggest assertions that can be used by those tests.

http://www.evosuite.org/evosuite/
Summary

• Automated test generation is not a panacea.
  – Can be difficult to reach “interesting” program states
  – Requires an oracle
  – Cheap to automatically generate tests, but expensive to maintain.

• But it is a useful technique!
  – Complements developer-written tests
  – Can be better at identifying certain bug classes
Outline

• Class invariants
• Immutability
• Testing and coverage
• Testing for complex environments
Problems when testing some apps

- **User-facing applications**
  - Users click, drag, etc., and interpret output
  - Timing issues

- **Testing against big infrastructure**
  - Databases, web services, etc.

- **Real world effects**
  - Printing, mailing documents, sensor noise, etc.

- Collectively comprise the *test environment*
Example – Tiramisu app

• Mobile route planning app
• Android user interface
• Backend uses live PAT data
Another example

- 3rd party Facebook apps
- Android user interface
- Backend uses Facebook data
Testing in real environments

```java
void buttonClicked() {
    render(getFriends());
}

List<Friend> getFriends() {
    Connection c = http.getConnection();
    FacebookApi api = new FacebookApi(c);
    List<Node> persons = api.getFriends("john");
    for (Node person1 : persons) {
        for (Node person2 : persons) {
            ...
        }
    }
    return result;
}
```
Eliminating Android dependency?

```java
@Test void testGetFriends() {
    ... // A Junit test
}

List<Friend> getFriends() {
    Connection c = http.getConnection();
    FacebookApi api = new FacebookApi(c);
    List<Node> persons = api.getFriends("john");
    for (Node person1 : persons) {
        for (Node person2 : persons) {
            ...
        }
    }
    return result;
}
```
That won’t quite work

- **GUI applications process many thousands of events**
- Solution: automated GUI testing frameworks
  - Allow streams of GUI events to be captured, replayed
- These tools are sometimes called *robots*
The more general case: Record and replay

https://github.com/SeleniumHQ/selenium
https://netflix.github.io/pollyjs/#/
https://wiki.ros.org/rosbag
Eliminating Facebook dependency?

```java
@Test void testGetFriends() {
    ... // A JUnit test
}

List<Friend> getFriends() {
    FacebookApi api = new MockFacebook(c);
    List<Node> persons = api.getFriends("john");
    for (Node person1 : persons) {
        for (Node person2 : persons) {
            ...
        }
    }
    return result;
}
```
That won’t quite work!

• **Changing production code for testing unacceptable**
• Problem caused by *constructor* in code
• Instead of constructor, use special *factory* that allows alternative implementations
• Use tools to facilitate this sort of testing
  – *Dependency injection* tools, e.g., Dagger, Guice, Spring
  – Mock object frameworks such as Mockito
Fault injection

- Mocks can emulate failures such as timeouts
- Allows you to verify the robustness of system against faults that you can’t generate at will

https://github.com/mrwilson/byte-monkey
https://blog.probablyfine.co.uk/2016/05/30/announcing-byte-monkey.html
Advantages of using mocks

• Test code locally without large environment
• Enable deterministic tests (in some cases)
• Enable fault injection
• Can speed up test execution
  – e.g., avoid slow database access
• Can simulate functionality not yet implemented
• Enable test automation
Design Implications

• Think about testability when writing code
• When a mock may be appropriate, design for it
• Hide subsystems behind an interfaces
• Use factories, not constructors to instantiate
• Use appropriate tools
  – Dependency injection or mocking frameworks
Hardware differences matter...
More Testing in 15-313

Foundations of Software Engineering

• Manual testing
• Security testing, penetration testing
• Fuzz testing for reliability
• Usability testing
• GUI/Web testing
• Regression testing
• Property-based testing
• Differential testing
• Stress/soak testing
Conclusion

• To maintain class invariants
  – Minimize mutability
  – Make defensive copies where required

• Interface testing is critical
  – Design interfaces to facilitate testing
  – Write creative test suites that maximize power-to-weight ratio
  – Coverage tools can help gauge test suite quality

• Testing apps with complex environments requires added effort