Principles of Software Construction: Class invariants, immutability, and testing

Josh Bloch  Charlie Garrod
Administrivia

• Homework 4a due **today**, 11:59 p.m.
• Design review meeting is **mandatory**
  – But we expect it to be really helpful
  – Feedback is a wonderful thing
• PSA – You have less than one week left to register to vote! Deadline is October 11!
Key concepts from Tuesday...

- Internal representations matter
  - The wrong representation can be toxic
- Code must be clean and concise
  - Repetition is toxic
- Good coding habits matter
Outline

• Class invariants and defensive copying
• Immutability
• Testing and coverage
• Testing for complex environments
• Implementation testing with assertions
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

// 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**

```java
// 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 parameter check and copy
• Thwarts multithreaded TOCTOU attack — Time-Of-Check-To-Time-Of-U

```java
// 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

```java
// 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
• Implementation testing with assertions
Immutable classes

• Class whose instances cannot be modified
• Examples: String, Integer, BigInteger
• 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
Immutable class example

```java
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; }

    // subtract, multiply, divide similar to add
    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
}
```
Immutable class example (cont.)

Nothing interesting here

```java
@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 as primitives
  - 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
• Implementation testing with assertions
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
Test driven development

• Write tests before code
• Never write code without a failing test
• Code until the failing test passes
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
How much testing?

• You generally cannot test all inputs
  – Too many – usually infinite
• 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
  – More difficult for test suites than regular code
  – Realistically, test suites will look worse
• Can be fun to write if approached in this spirit
Next best thing to exhaustive testing: *random inputs*

- Also know as *fuzz 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
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
Test suites – ideal and real

• Ideal test suites
  – 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
Outline

• Class invariants
• Immutability
• Testing and coverage
• Testing for complex environments
• Implementation testing with assertions
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, etc.

• Collectively comprise the test environment
Example – Tiramisu app

• Mobile route planning app
• Android UI
• Back end uses live PAT data
Another example

- 3rd party Facebook apps
- Android user interface
- Backend uses Facebook data
void buttonClicked() {
    render(getFriends());
}

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

```java
@Test void testGetFriends() {
    assert 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;
}
```
That won’t quite work

• GUI applications process *thousands* of events
• Solution: automated GUI testing frameworks
  – Allow streams of GUI events to be captured, replayed
• These tools are sometimes called *robots*
Eliminating Facebook dependency

@Test void testGetFriends() {
    assert getFriends() == ...;
}

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
• Use factory instead of constructor
• Use tools to facilitate this sort of testing
  – Dependency injection tools, e.g., Dagger, Guice
  – Mock object frameworks such as Mockito
Fault injection

- Mocks can emulate failures such as timeouts
- Allows you to verify the robustness of system
Advantages of using mocks

• Test code locally without large environment
• Enable deterministic tests
• 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
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
- Differential testing
- Stress/soak testing
Outline

• Class Invariants
• Immutability
• Test suites and coverage
• Testing for complex environments
• Implementation-testing with assertions
What is an assertion?

• Statement containing boolean expression that programmer believes to be true:
  
  ```
  assert speed <= SPEED_OF_LIGHT;
  ```

• Evaluated at run time – throws Error if false

• Disabled by default - no performance effect

• Typically enabled during development

• Can enable in the field when problems occur!
Syntax

AssertStatement:
assert $Expression_1$ ;
assert($Expression_1$, $Expression_2$) ;

• $Expression_1$ - asserted condition (boolean)
• $Expression_2$ - detail message of AssertionError
Why use assertions?

- Document & test programmer's assumptions
  - e.g., class invariants
- Verify programmer’s understanding
- Quickly uncover bugs
- Increase confidence that program is bug-free
- Asserts turn black box tests into white box tests
Look for “assertive comments”

```java
int remainder = i % 3;
if (remainder == 0) {
    ...
} else if (remainder == 1) {
    ...
} else { // (remainder == 2)
    ...
}
```
int remainder = i % 3;
if (remainder == 0) {
    ...
} else if (remainder == 1) {
    ...
} else {
    assert remainder == 2;
    ...
}
Use second argument for *failure capture*

```java
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert (i % 3 == 2, i);
        ...
}
```
Look for switch with no default

```java
switch(flavor) {
    case VANILLA:
        ...
        break;
    case CHOCOLATE:
        ...
        break;
    case STRAWBERRY:
        ...
}
```
Add an “assertive default”

switch(flavor) {
    case VANILLA:
        ...
        break;
    case CHOCOLATE:
        ...
        break;
    case STRAWBERRY:
        ...
        break;
    default:
        assert (false, flavor);
}
Do not use assertions for *public* preconditions

/**
 * Sets the refresh rate.
 * 
 * @param  rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0 or rate > MAX_REFRESH_RATE.
 */
public void setRefreshRate(int rate) {
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
        throw new IllegalArgumentException(...);
    setRefreshInterval(1000 / rate);
}
Do use assertions for *non-public* preconditions

```java
/**
 * Sets the refresh interval (which must correspond
 * to a legal frame rate).
 * @param interval refresh interval in ms
 */
private void setRefreshInterval(int interval) {
    assert interval > 0 && interval <= 1000, interval;
    ... // Set the refresh interval
}
```
Do use assertions for postconditions

/**
 * Returns BigInteger whose value is (this⁻¹ mod m).
 * @throws ArithmeticException if m <= 0, or this
 * BigInteger is not relatively prime to m.
 */
public BigInteger modInverse(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException(m + " <= 0");
    ... // Do the computation
    assert this.multiply(result).mod(m).equals(ONE);
    return result;
}
Complex postconditions

```c
void foo(int[] a) {
    // Manipulate contents of array
    ...

    // Array will appear unchanged
}
```
void foo(final int[] a) {
    class DataCopy {
        private int[] aCopy;
        DataCopy() { aCopy = (int[]) a.clone(); }
        boolean isConsistent() {
            return Arrays.equals(a, aCopy);
        }
    }
    DataCopy copy = null;
    assert (copy = new DataCopy()) != null;
    ... // Manipulate contents of array
    assert copy.isConsistent();
}
Caveat – asserts must not have side effects visible outside other asserts

Do this:

```java
boolean modified = set.remove(elt);
assert modified;
```

**Not this:**

```java
assert set.remove(elt);  //Bug!
```
Sermon: accept assertions into your life

• Programmer’s interior monologue:
  – “Now at this point, we know...”
• During, not after, development
• Quickly becomes second nature
• Pays big code-quality dividends
Conclusion

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

• Interface testing is critical
  – Design interfaces to facilitate testing
  – Coverage tools can help gauge test suite quality

• Use assertions to test implementation details
  – Asserts amplify the value of your interface tests