Principles of Software Construction: Objects, Design, and Concurrency

Achieving and Maintaining Correctness in the Face of Change – Pt. 2

Josh Bloch Charlie Garrod

School of Computer Science

Outline

- Java programming basics - loose ends
- Contracts for interfaces
- Interface testing - JUnit and friends
- Class Invariants for implementations
- Implementation testing - assertions

Review: `Object` methods

- `equals` – true if the two objects are “equal”
- `hashCode` – a hash code for use in hash maps
- `toString` – a printable string representation

Review: `Object` implementations

- Provide *identity semantics*
- Must override if you want value semantics
- Overriding `toString` is easy and beneficial
- Last time I said it was hard to override `equals` and `hashCode`
- **I lied**

The `equals` contract

The equals method implements an **equivalence relation**. It is:

- **Reflexive**: For any non-null reference value x, x.equals(x) must return true.
- **Symmetric**: For any non-null reference values x and y, x.equals(y) must return true if and only if y.equals(x) returns true.
- **Transitive**: For any non-null reference values x, y, z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.
- **Consistent**: For any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) must return false.

The `equals` contract in English

- Reflexive - every object is equal to itself
- Symmetric - if a.equals(b) then b.equals(a)
- Transitive - if a.equals(b) and b.equals(c), then a.equals(c)
- Consistent - equal objects stay equal unless mutated
- “Non-null” - a.equals(null) returns false
- Taken together these ensure that equals is a global equivalence relation over all objects

`equals` Override Example

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public boolean equals(Object o) {
        if (!(o instanceof PhoneNumber))  // Does null check
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNumber == lineNumber
               && pn.prefix == prefix
               && pn.areaCode == areaCode;
    }
    ...
}
```

The hashCode contract

Whenever it is invoked on the same object more than once during an execution of an application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

The `hashCode` contract in English

- Equal objects **must** have equal hash codes
  - **If you override `equals` you must override `hashCode`**
- Unequal objects **should** have different hash codes
  - Take all value fields into account when constructing it
- Hash code must not change unless object mutated

hashCode override example

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public int hashCode() {
        int result = 17;  // Nonzero is good
        result = 31 * result + areaCode;   // Constant must be odd
        result = 31 * result + prefix;     //     "     "   "   "
        result = 31 * result + lineNumber; //     "     "   "   "
        return result;
    }    ...
}
```

For more than you want to know about overriding object methods, see *Effective Java* Chapter 2

The == operator vs. `equals` method

- For primitives you *must* use ==
- For object reference types
  - The == operator provides *identity semantics*
    - Exactly as implemented by `Object.equals`
    - Even if `Object.equals` has been overridden
    - This is seldom what you want!
  - **you should (almost) always use `.equals`**
  - Using == on an object reference is a bad smell in code
  - `if (input == "yes") // A bug!!!`

Pop quiz: what does this print?

```
public class Name {
    private final String first, last;
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first; this.last = last;
    }
    public boolean equals(Name o) {
        return first.equals(o.first) && last.equals(o.last);
    }
    public int hashCode() {
        return 31 * first.hashCode() + last.hashCode();
    }
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(
            s.contains(new Name("Mickey", "Mouse")));
    }
}
```

(a) `true`
(b) `false`
(c) It varies

What Does It Print?

(a) `true`

(b) `false`

(c) It varies

`Name` overrides **`hashCode`** but not **`equals`**!
The two **`Name`** instances are unequal.

Another Look

```
public class Name {
    private final String first, last;
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first; this.last = last;
    }
    public boolean equals(Name o) { // Accidental overloading
        return first.equals(o.first) && last.equals(o.last);
    }
    public int hashCode() {         // Overriding
        return 31 * first.hashCode() + last.hashCode();
    }
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(
          s.contains(new Name("Mickey", "Mouse")));
    }
}
```

How Do You Fix It?

Replace the overloaded `equals` method with an overriding `equals` method

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Name))
        return false;
    Name n = (Name)o;
    return n.first.equals(first) && n.last.equals(last);
}
```

With this change, program prints `true`

The Moral

- If you want to override a method:
  - Make sure signatures match
  - Use `@Override` so compiler has your back
  - *Do* copy-and-paste declarations (or let IDE do it for you)

Outline

- Java programming basics - loose ends
- Contracts for interfaces
- Interface testing - JUnit and friends
- Class Invariants for implementations
- Implementation testing - assertions

Contracts

- Agreement between an object and its user
- Includes
  - Method signature (type specifications)
  - Functionality and correctness expectations
  - Performance expectations
- **What the method does**, not how it does it.

Contracts and interfaces

- Implementations must adhere to an interface's contracts
- Polymorphism
  - Can have different implementations of the interface specification
  - Clients care about the interface, not the implementation

Contracts and methods

- States method's and caller's responsibilities
  - Analogy: legal contracts
    If you pay me $30,000, I will build a new room on your house
  - Helps to pinpoint responsibility
- Contract structure
  - **Precondition**: what method relies on for correct operation
  - **Postcondition**: what method establishes after correctly running
  - **Exceptional behavior:** what method does if precondition violated
- Mandates correctness with respect to specification
  - If the caller fulfills the precondition,
    the method will run to completion and fulfill the postcondition

Formal specifications

```
/*@ requires len >= 0 && array != null && array.length == len;
  @
  @ ensures \result ==
  @             (\sum int j;  0 <= j && j < len;  array[j]);
  @*/
int total(int array[], int len);
```

- Theoretical approach
  - Advantages
    - Runtime checks (almost) for free
    - Basis for formal verification
    - Assisting automatic analysis tools
  - Disadvantages
    - Requires a lot of work
    - Impractical in the large

Textual specifications - Javadoc

- Practical approach
- Document
  - Every parameter
  - Return value
  - Every exception (checked and unchecked)
  - What the method does, including
    - Purpose
    - Side effects
    - Any thread safety issues
    - Any performance issues
- Do **not** document implementation details

Javadoc - example for a method

```
/**
 * Returns the element at the specified position of this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant time.
 * In some implementations, it may run in time proportional to the
 * element position.
 *
 * @param index position of element to return; must be non-negative and
 *              less than the size of this list.
 * @return the element at the specified position of this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

postcondition

precondition

Outline

I. Java programming basics - loose ends
II. Contracts for interfaces
III. Interface testing - JUnit and friends
IV. Class Invariants for implementations
V. Implementation testing - assertions

Context

- **Information Hiding** means modules are independent, communicate only via APIs
- **Contracts** describe behavior of the APIs (without revealing implementation details)
- **Testing** helps gain confidence that modules behave correctly (with respect to contracts)

Semantic correctness

- Compiler ensures types are correct (type checking)
  - Prevents runtime “Method Not Found” and “Cannot add Boolean to Int” errors
- Static analysis tools (e.g., FindBugs) recognize certain common problems (*bug patterns*)
  - Possible `NullPointerExceptions` or Forgetting to close files
- How to ensure semantic correctness (adherence to contracts)?

Formal verification

- Use mathematical methods to prove correctness with respect to the formal specification
- Formally prove that all possible executions of an implementation fulfill the specification
- Manual effort; partial automation; not automatically decidable

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

Edsger W. Dijkstra, 1969

Testing

- Executing the program with selected inputs in a controlled environment
- Goals:
  - Reveal bugs (main goal)
  - Assess quality
  - Clarify the specification, documentation
  - Verify contracts

"Beware of bugs in the above code; I have only proved it correct, not tried it."

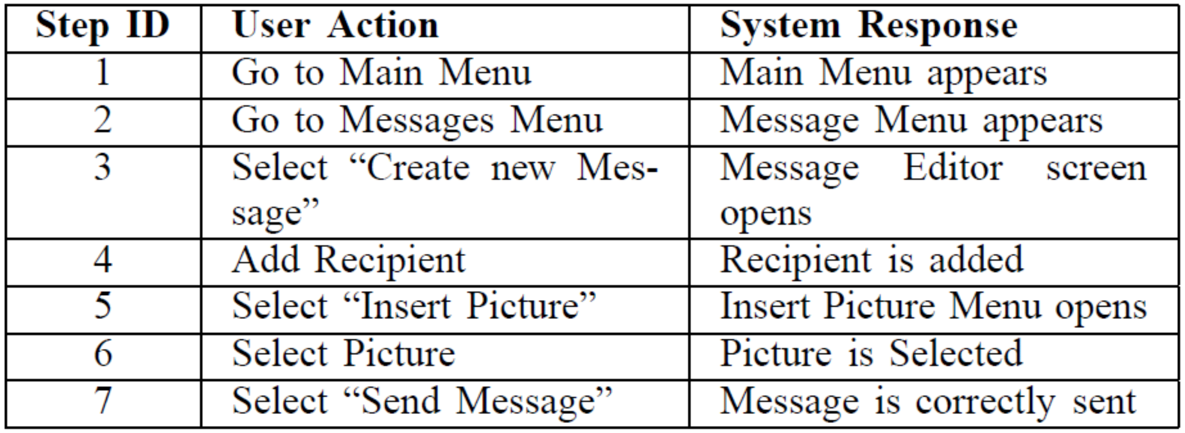
Donald Knuth, 1977

Who's right, Dijkstra or Knuth?

- They're both right!
- **Please see "Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken"**
  - Official "Google Research" blog
  - http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
- There is no silver bullet
  - Use all tools at your disposal

Manual testing?

GENERIC TEST CASE: USER SENDS MMS WITH PICTURE ATTACHED.



Step ID	User Action	System Response
1	Go to Main Menu	Main Menu appears
2	Go to Messages Menu	Message Menu appears
3	Select “Create new Message”	Message Editor screen opens
4	Add Recipient	Recipient is added
5	Select “Insert Picture”	Insert Picture Menu opens
6	Select Picture	Picture is Selected
7	Select “Send Message”	Message is correctly sent

- Live System?
- Extra Testing System?
- Check output / assertions?
- Effort, Costs?
- Reproducible?

Automate testing

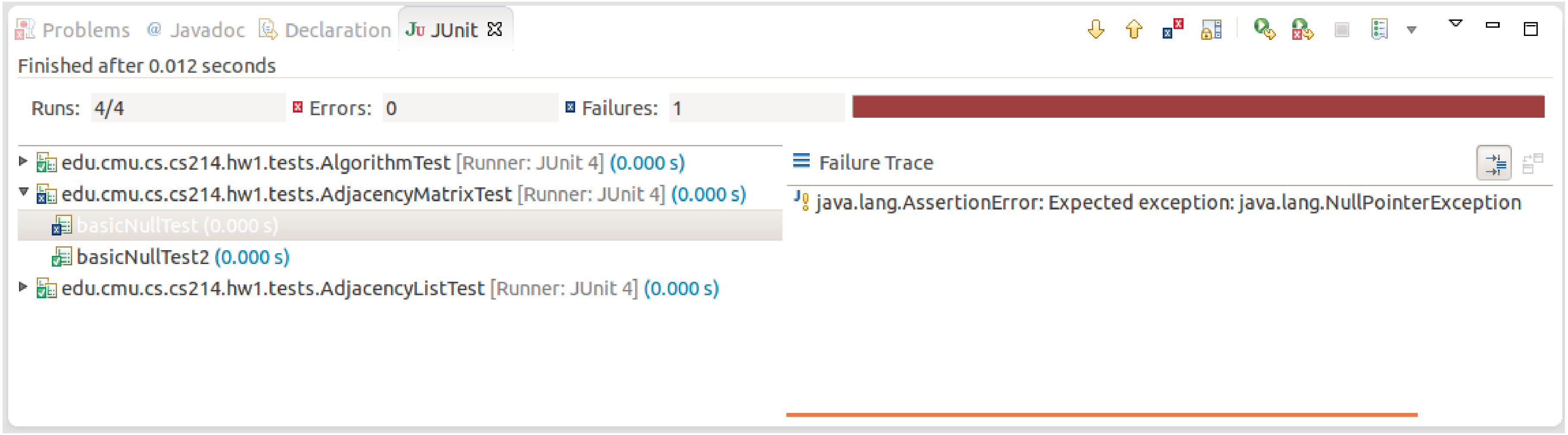
- Execute a program with specific inputs, check output for expected values
- Set up testing infrastructure
- **Execute tests regularly**

Unit tests

- Unit tests for small units: methods, classes, subsystems
  - Smallest testable part of a system
  - Test parts before assembling them
  - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Little dependencies on other system parts or environment
- Insufficient but a good starting point

JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available
- Can be used as design mechanism



```
Finished after 0.012 seconds
Runs: 4/4    Errors: 0    Failures: 1

edu.cmu.cs.cs214.hw1.tests.AlgorithmTest [Runner: JUnit 4] (0.000 s)
edu.cmu.cs.cs214.hw1.tests.AdjacencyMatrixTest [Runner: JUnit 4] (0.000 s)
    basicNullTest (0.000 s)
    basicNullTest2 (0.000 s)
edu.cmu.cs.cs214.hw1.tests.AdjacencyListTest [Runner: JUnit 4] (0.000 s)

Failure Trace
java.lang.AssertionError: Expected exception: java.lang.NullPointerException
```

Selecting test cases: common strategies

- Read specification
- Write tests for
  - Representative case
  - Invalid cases
  - Boundary conditions
- Write stress tests
  - Automatically generate huge numbers of test cases
- Think like an attacker
  - The tester's goal is to find bugs!
- How many test should you write?
  - Aim to cover the specification
  - Work within time/money constraints

JUnit conventions

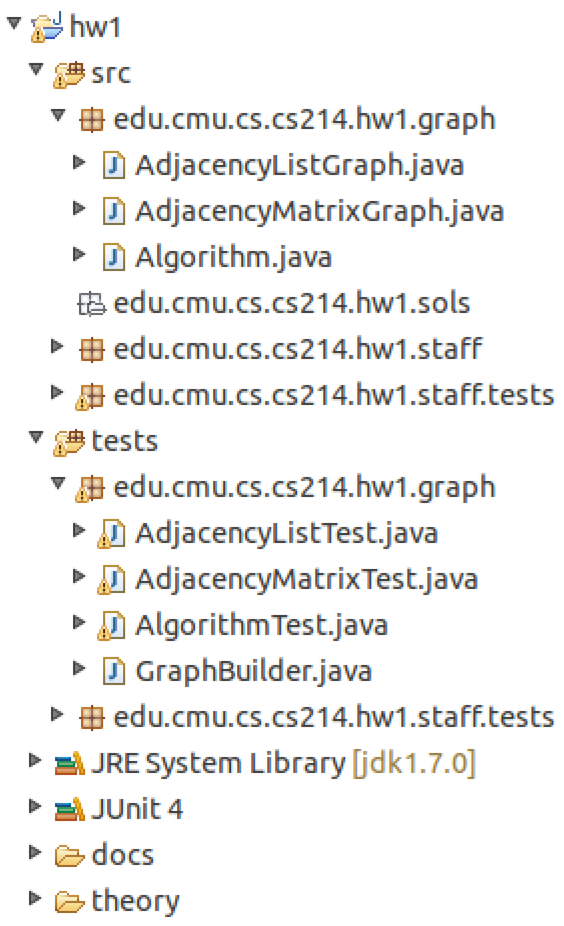
- TestCase collects multiple tests (in one class)
- TestSuite collects test cases (typically package)
- Tests should run fast
- Tests should be independent

- Tests are methods without parameter and return value
- AssertError signals failed test (unchecked exception)

- Test Runner knows how to run JUnit tests
  - (uses reflection to find all methods with @Test annotat.)

Test organization

- Conventions (not requirements)
- Have a test class XTest for each class X
- Have a source directory and a test directory
  - Store ATest and A in the same package
  - Tests can access members with default (package) visibility
  - Maven style: src/main/java and src/test/java



```
hw1
  src
    edu.cmu.cs.cs214.hw1.graph
      AdjacencyListGraph.java
      AdjacencyMatrixGraph.java
      Algorithm.java
    edu.cmu.cs.cs214.hw1.sols
    edu.cmu.cs.cs214.hw1.staff
    edu.cmu.cs.cs214.hw1.staff.tests
  tests
    edu.cmu.cs.cs214.hw1.graph
      AdjacencyListTest.java
      AdjacencyMatrixTest.java
      AlgorithmTest.java
      GraphBuilder.java
    edu.cmu.cs.cs214.hw1.staff.tests
  JRE System Library [jdk1.7.0]
  JUnit 4
  docs
  theory
```

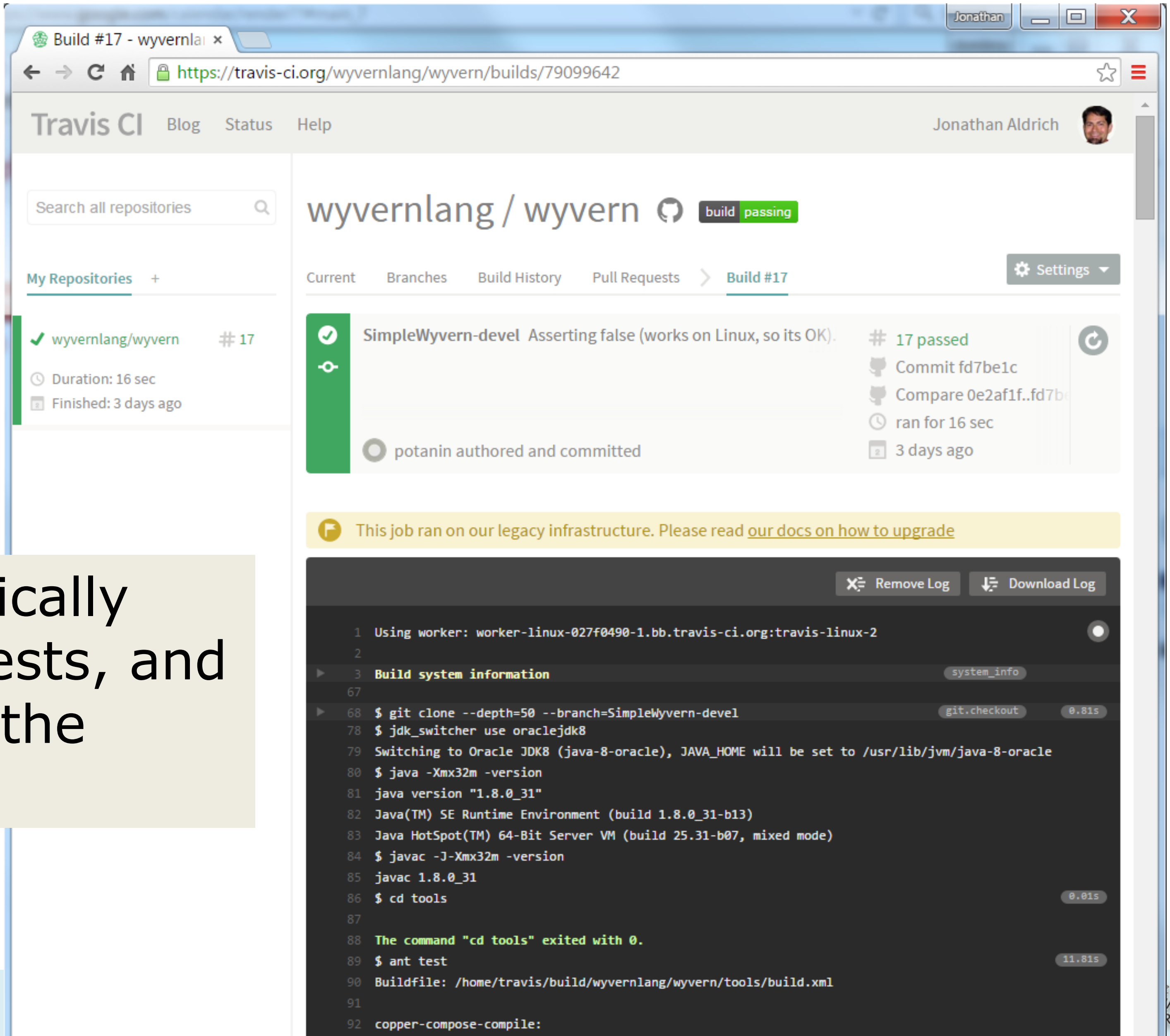
Testable code

- 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 frequently

- You should only commit code that is passing all tests
- Run tests before every commit
- If entire test suite becomes too large and slow for rapid feedback:
  - Run local package-level tests ("smoke tests“) frequently
  - Run all tests nightly
  - Medium sized projects easily have 1000s of test cases
- Continuous integration servers help to scale testing

Continuous integration - Travis CI



wyvernlang/wyvern #17
Duration: 16 sec
Finished: 3 days ago

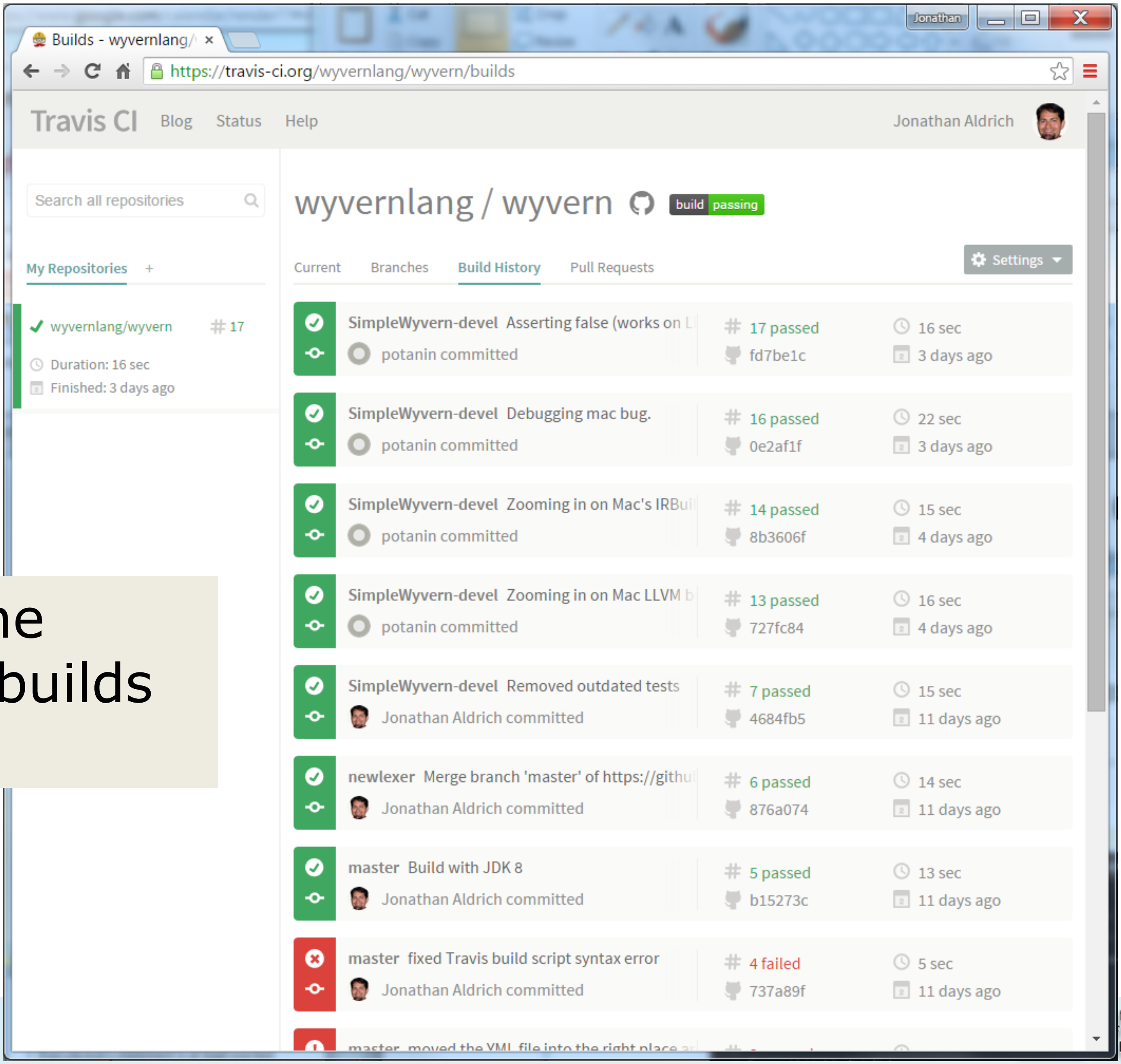
SimpleWyvern-devel Asserting false (works on Linux, so its OK).
potanin authored and committed
17 passed
Commit fd7be1c
Compare 0e2af1f..fd7b
ran for 16 sec
3 days ago

This job ran on our legacy infrastructure. Please read our docs on how to upgrade

```
Using worker: worker-linux-027f0490-1.bb.travis-ci.org:travis-linux-2

Build system information                                              system_info

$ git clone --depth=50 --branch=SimpleWyvern-devel                    git.checkout   0.81s
$ jdk_switcher use oraclejdk8
Switching to Oracle JDK8 (java-8-oracle), JAVA_HOME will be set to /usr/lib/jvm/java-8-oracle
$ java -Xmx32m -version
java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
$ javac -J-Xmx32m -version
javac 1.8.0_31
$ cd tools                                                                           0.01s

The command "cd tools" exited with 0.
$ ant test                                                                          11.81s
Buildfile: /home/travis/build/wyvernlang/wyvern/tools/build.xml

copper-compose-compile:
```

Automatically builds, tests, and displays the result

Continuous integration - Travis CI



Builds - wyvernlang/

https://travis-ci.org/wyvernlang/wyvern/builds

Travis CI Blog Status Help Jonathan Aldrich

Search all repositories

My Repositories +

✓ wyvernlang/wyvern # 17

Duration: 16 sec

Finished: 3 days ago

wyvernlang / wyvern build passing

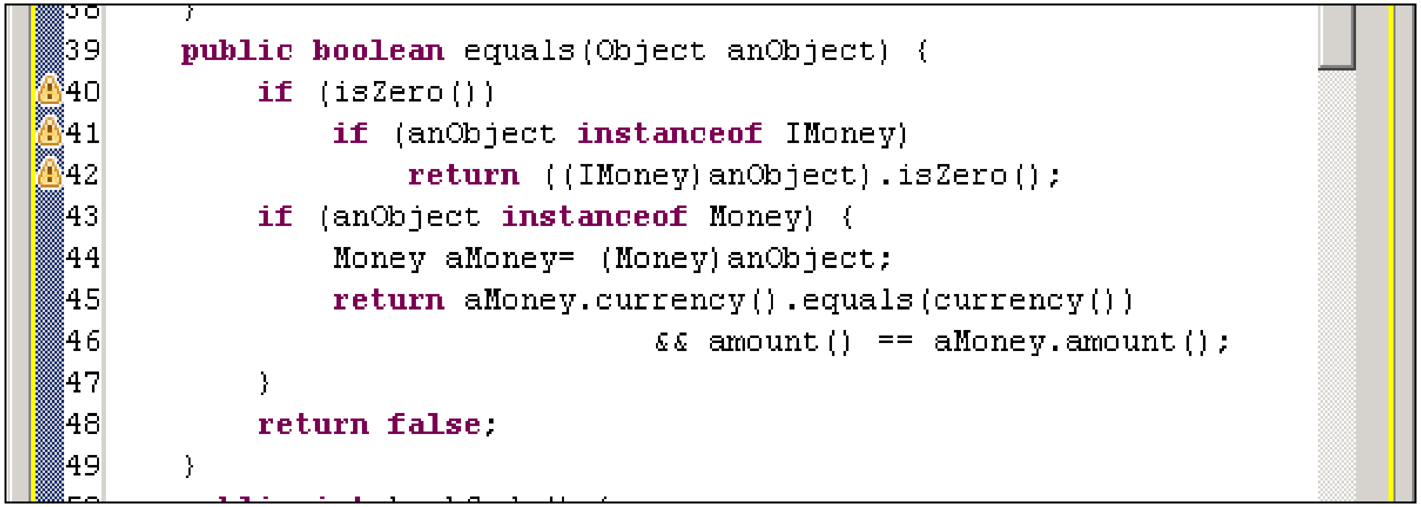
Current Branches Build History Pull Requests Settings

Branch	Message	Committer	Build	Commit	Duration	Finished
SimpleWyvern-devel	Asserting false (works on L	potanin committed	17 passed	fd7be1c	16 sec	3 days ago
SimpleWyvern-devel	Debugging mac bug.	potanin committed	16 passed	0e2af1f	22 sec	3 days ago
SimpleWyvern-devel	Zooming in on Mac's IRBui	potanin committed	14 passed	8b3606f	15 sec	4 days ago
SimpleWyvern-devel	Zooming in on Mac LLVM b	potanin committed	13 passed	727fc84	16 sec	4 days ago
SimpleWyvern-devel	Removed outdated tests	Jonathan Aldrich committed	7 passed	4684fb5	15 sec	11 days ago
newlexer	Merge branch 'master' of https://githu	Jonathan Aldrich committed	6 passed	876a074	14 sec	11 days ago
master	Build with JDK 8	Jonathan Aldrich committed	5 passed	b15273c	13 sec	11 days ago
master	fixed Travis build script syntax error	Jonathan Aldrich committed	4 failed	737a89f	5 sec	11 days ago
master	moved the YML file into the right place a					

Can see the results of builds over time

Outlook: statement coverage

- Trying to test all parts of the implementation
- Execute every statement, ideally



```
    public boolean equals(Object anObject) {
        if (isZero())
            if (anObject instanceof IMoney)
                return ((IMoney)anObject).isZero();
        if (anObject instanceof Money) {
            Money aMoney= (Money)anObject;
            return aMoney.currency().equals(currency())
                             && amount() == aMoney.amount();
        }
        return false;
    }
```

- Does this guarantee correctness?

Summary

- Testing
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now
  - Does not find all problems (unsound)
- Static Analysis
  - Analysis of all possible executions
  - Specific issues only with conservative approx. and bug patterns
  - Tools available, useful for bug finding
  - Automated, but unsound and/or incomplete
- Proofs (Formal Verification)
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - Practical for small programs, may scale up in the future
  - Sound and complete, but not automatically decidable

Outline

I. Java programming basics - loose ends
II. Contracts for interfaces
III. Interface testing - JUnit and friends
IV. Class Invariants for implementations
V. Implementation testing - 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 execution

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

Defensive copying

- 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

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*

```
// 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-Use

```
// 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?

Unfortunately, constructors are only half the battle

```
// Accessor attack on internals of Period
Date start = new Date();
Date end   = new Date();
Period p = new Period(start, 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

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

```
public final class Complex {
    private final float re, im;

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

    // Getters without corresponding setters
    public float realPart()      { return re; }
    public float 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.)

```
    public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof Complex)) return false;
        Complex c = (Complex)o;
        return (Float.floatToIntBits(re) ==
                Float.floatToIntBits(c.re)) &&
               (Float.floatToIntBits(im) ==
                Float.floatToIntBits(c.im));
    }
    public int hashCode() {
        int result = 17 + Float.floatToIntBits(re);
        result = 37*result + Float.floatToIntBits(im);
        return result;
    }
    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}
```

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

```
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

When to make classes immutable

- Always, unless there's a good reason not to
- Always make small “value classes” immutable!
  - Examples: `Color`, `PhoneNumber`, `Price`
  - `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

Summary

- Reuse objects where appropriate
  - Improves clarity and performance
- Make defensive copies where required
  - Provides robustness
- Write immutable classes
  - Simple, thread-safe, sharable and reusable

Outline

I. Java programming basics - loose ends
II. Contracts for interfaces
III. Interface testing - JUnit and friends
IV. Class Invariants for implementations
V. Implementation testing - assertions

Assertiveness Training

- What is an assertion
- Why use assertions
- How to use assertions

What is an assertion?

- Statement containing boolean expression that programmer believes to be true:

  `assert speed <= SPEED_OF_LIGHT;`
- Evaluated at run time - `Error` if `false`
- Disabled by default - no performance effect
- Typically enabled during development
- May be enabled in field!

Syntax

```
AssertStatement:
     assert Expression1 ;
     assert(Expression1, Expression2) ;
```

- *Expression*$_1$ - asserted condition (boolean)
- *Expression*$_2$ - detail message of `AssertionError`

Why use assertions?

- Document & test programmer's assumptions
- Verify programmer's understanding
- Quickly uncover bugs
- Increase confidence that program is bug-free
- They turn black box tests into white box tests

How to use assertions

- Sprinkle source code liberally with assertions
  - To enable in user code: `java -ea`
  - To enable in JRE libraries: `java -esa`
  - To enable everywhere: `java -ea –esa`
- Can enable at package or class granularity
  - See compiler documentation for details

Look for "assertive comments"

```
int residue = i % 3;
if (residue == 0) {
    ...
} else if (residue == 1) {
    ...
} else { // (residue == 2)
    ...
}
```

Replace with real assertions!

```
int residue = i % 3;
if (residue == 0) {
    ...
} else if (residue == 1) {
    ...
} else {
    assert residue == 2;
    ...
}
```

Use second argument for *failure capture*

```
if (i%3 == 0) {
    ...
} else if (i%3 == 1) {
    ...
} else {
    assert (i%3==2, i);
    ...
}
```

Look for switch with no default

```
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

```
/**
 * 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
}
```

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

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

    // Array will appear unchanged
}
```

Use assertions for complex postconditions

```
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();
}
```

Caveats

- No *side effects* visible outside other assertions

  Do this:

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

  not this:

  ```
  assert set.remove(elt);  //Bug!
  ```

- Watch out for infinite recursion

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

Summary

- Overriding `Object` methods demands care
- Check adherence to interfaces with unit tests
- Maintain invariants critical to correctness
  - Minimize mutability
  - Make defensive copies where required
- Use assertions to test class invariants, postconditions, and private preconditions