Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing classes

Java basics, functional correctness

Michael Hilton Bogdan Vasilescu

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- No Smoking
- Office Hours – check online calendar
- Homework 1 due Thursday 11:59 p.m.
- Everyone must read and sign the collaboration policy

Key concepts from last Thursday

Key concepts from last Thursday

- Infrastructure
- Introduction to Java
  - Syntax
  - Types
  - I/O
  - Iterators
  - Exceptions

Today

- Information hiding: Design for change, design for reuse
  - Encapsulation: Visibility modifiers in Java
  - Interface types vs. class types
- Functional correctness
  - JUnit and friends

Intro to Java

Git, CI

UML

GUIs

Static Analysis

Performance

GUIs

More Git

Design

Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Unit Testing, Design Patterns	Understanding the Problem	Concurrency Primitives, Synchronization
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies	Designing Abstractions for Concurrency
	Testing Subsystems	
	Design for Reuse at Scale: Frameworks and APIs	

Visibility modifiers in Java ("encapsulation")

- `private`: Accessible only from declaring class
- "package private": Accessible from any class in package
  - a.k.a. default access, no visibility modifier
- `protected`: Accessible from package and also from subclasses
- `public`: Accessible anywhere

Visibility modifier example

- Consider:

```
public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void translateBy(Point p) {
        x += p.x;
        y += p.y;
    }
}
```

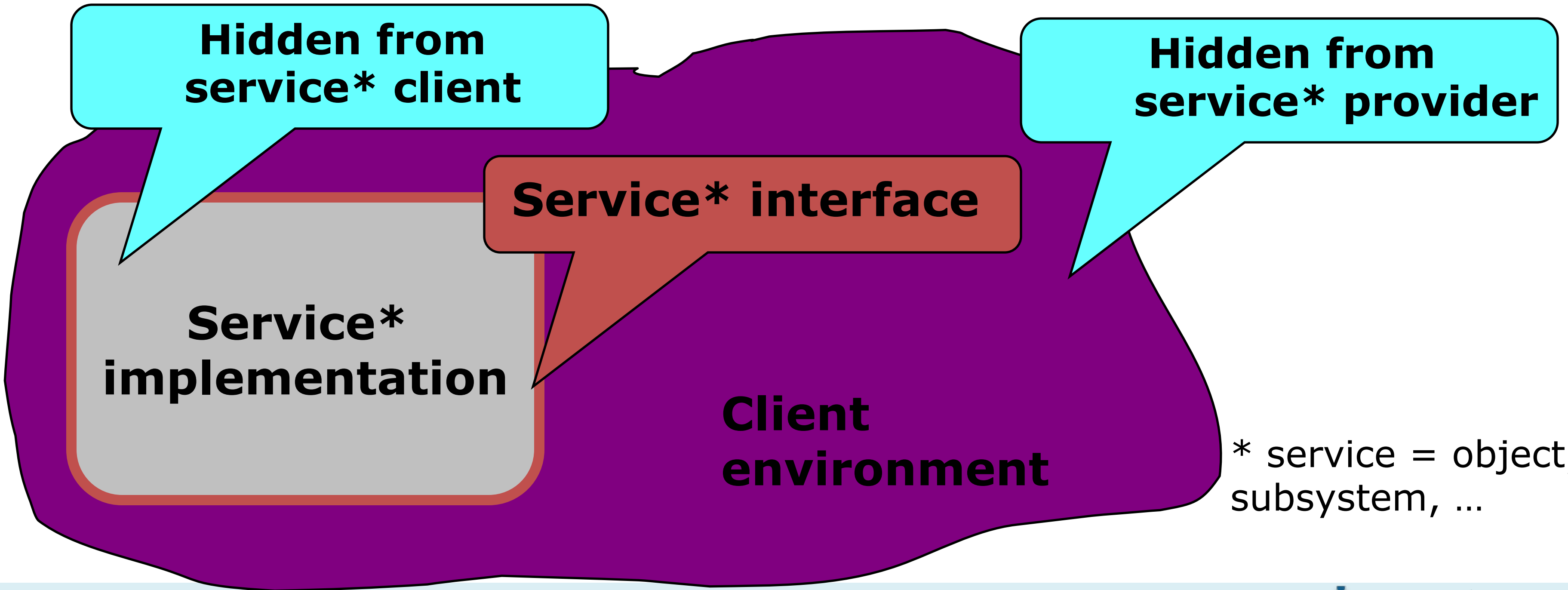
Visibility modifier example

- Consider:

```
public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void translateBy(Point p) {
        x += p.x; // This is OK.  p.x and p.y are
        y += p.y; // accessible from the Point class!
    }
    public double getX() { return x; }
    public double getY() { return y; }
}
```

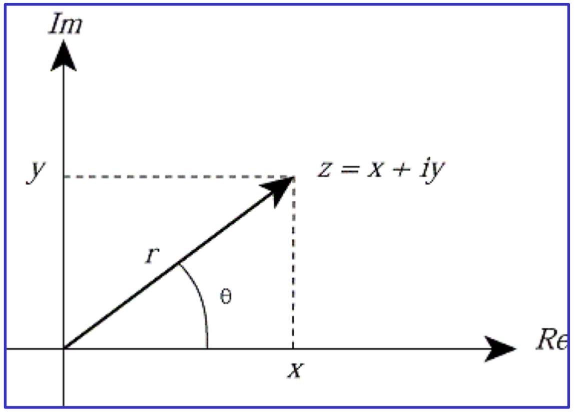
Fundamental Design Principle for Change: Information Hiding

- Expose as little implementation detail as necessary
- Allows to change hidden details later



* service = object, subsystem, …

A more complex example




```
public class Complex {
  private final double re;  // Real part
  private final double im;  // Imaginary part

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

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new Complex(re + c.re, im + c.im);
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

Using the `Complex` class

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new Complex(-1, 0);
        Complex d = new Complex(0, 1);

        Complex e = c.add(d);
        System.out.println(e.realPart() + " + "
                                + e.imaginaryPart() + "i");
        e = c. multiply(d);
        System.out.println(e.realPart() + " + "
                                + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it prints

```
-1.0 + 1.0i
 0.0 + -1.0i
```

Extracting an interface from our class

```java
public interface Complex {
    // No constructors, fields, or implementations!

    double realPart();
    double imaginaryPart();
    double r();
    double theta();

    Complex plus(Complex c);
    Complex minus(Complex c);
    Complex times(Complex c);
    Complex dividedBy(Complex c);
}
```

An interface defines but does not implement API

Modifying our earlier class to use the interface

```
public class OrdinaryComplex implements Complex {
  private final double re;  // Real part
  private final double im;  // Imaginary part

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

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

Modifying our earlier client to use the interface

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new OrdinaryComplex(-1, 0);
        Complex d = new OrdinaryComplex(0, 1);

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                          + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + "
                          + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it still prints

```
-1.0 + 1.0i
 0.0 + -1.0i
```

Interfaces permit multiple implementations

```
public class PolarComplex implements Complex {
  private final double r;      // Radius
  private final double theta;  // Angle

  public PolarComplex(double r, double theta) {
    this.r = r;
    this.theta = theta;
  }

  public double realPart()      { return r * Math.cos(theta) ; }
  public double imaginaryPart() { return r * Math.sin(theta) ; }
  public double r()             { return r; }
  public double theta()         { return theta; }

  public Complex plus(Complex c)      { ... } // Completely new impls
  public Complex minus(Complex c)     { ... }
  public Complex times(Complex c)     { ... }
  public Complex dividedBy(Complex c) { ... }
}
```

Interface decouples client from implementation

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new PolarComplex(Math.PI,   1);  // -1
        Complex d = new PolarComplex(Math.PI/2, 1);  //  i

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it STILL prints

```
-1.0 + 1.0i
 0.0 + -1.0i
```

Information hiding is more general than visibility

- Use interfaces to separate expectations from implementation
  - Create interfaces to define your API
  - Declare variables, arguments, and return values as interface type
    - Write API in terms of other interfaces, not implementations
- Do not publicly document implementation details

Information hiding facilitates change, promotes reuse

- Think in term of abstractions, not implementations
  - Abstractions are more likely to be reused
- Can change implementations more easily
  - Different performance
  - Different behavior
- Prevents bad programmer behavior, unnecessary dependencies

Other benefits of information hiding

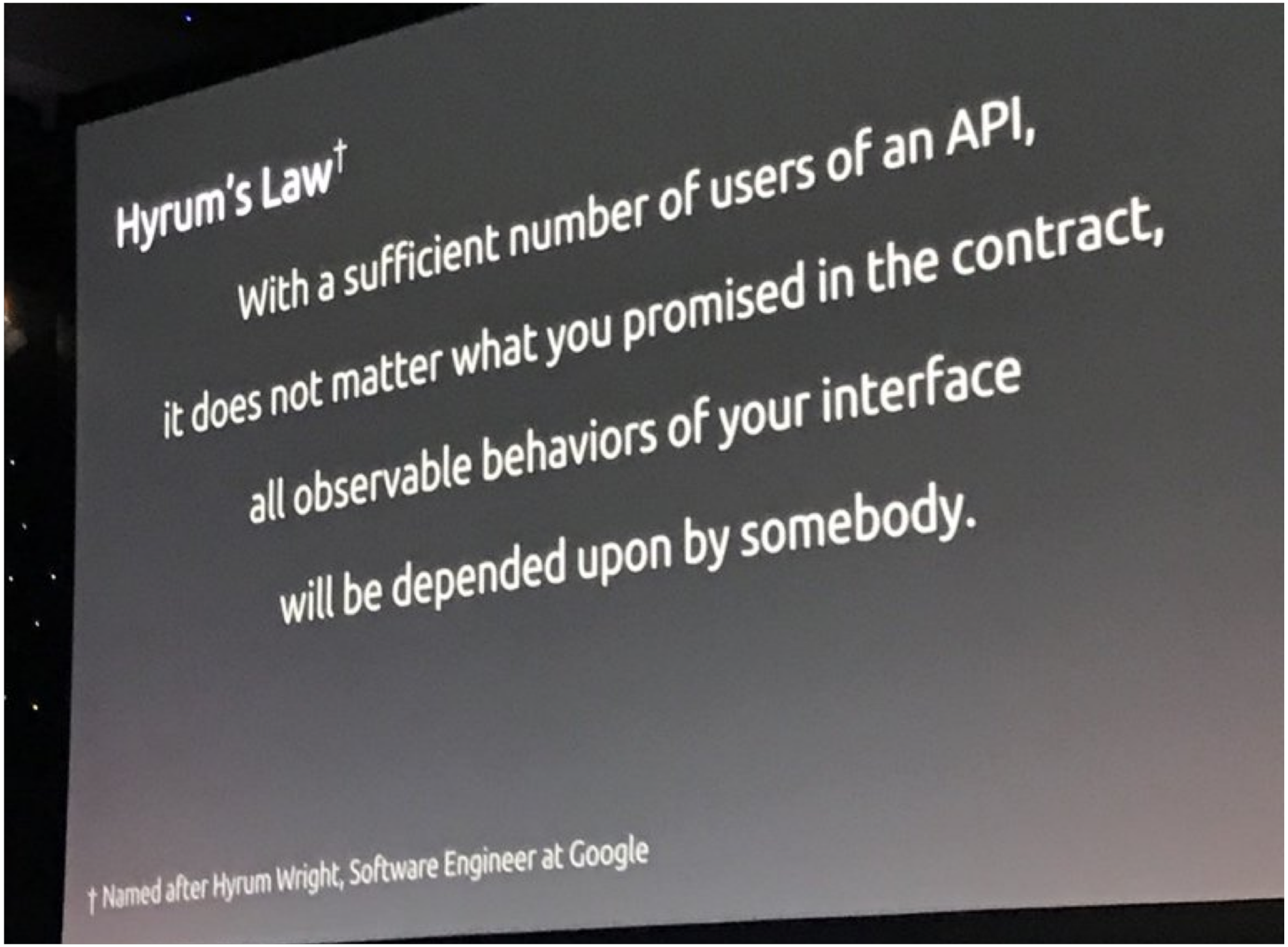
- Decoupled subsystems are easier to understand in isolation
- Speeds up system development
- Reduces cost of maintenance
- Improves effectiveness of performance tuning

Best practices for information hiding

- Carefully design your API
- Provide *only* functionality required by clients
  - *All* other members should be private
- You can always make a private member public later without breaking clients
  - But not vice-versa!

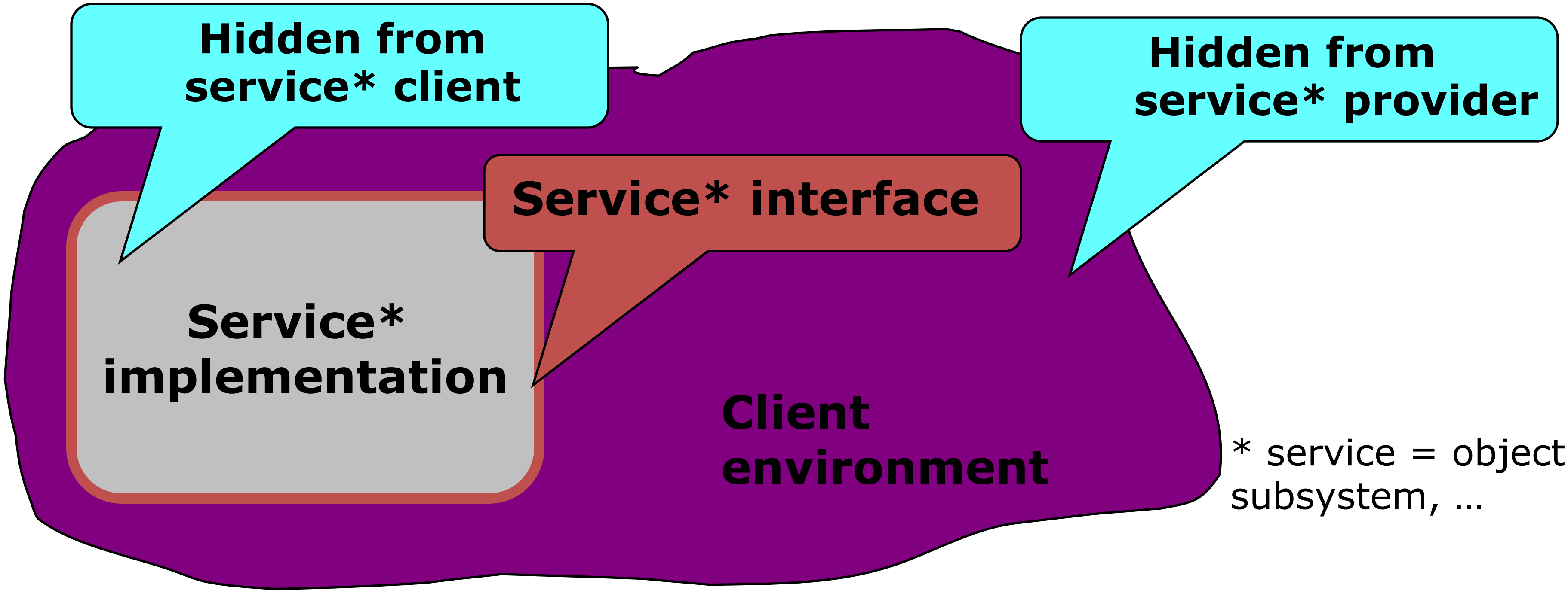
Hyrum's Law

Hyrum Wright
@hyrumwright
Infrastructure software engineer. Googler. Father. Occasional professor.



CONTRACTS
(BEYOND TYPE SIGNATURES)

Contracts and Clients



* service = object, subsystem, …

Contracts

- Agreement between provider and users of an object
- Includes
  - Interface specification (types)
  - Functionality and correctness expectations
  - Performance expectations
- What the method does, not how it does it
  - Interface (API), not implementation

Who’s to blame?

```
Algorithms.shortestDistance(g, “Tom”, “Anne”);

> ArrayOutOfBoundsException
```

Who’s to blame?

```
Algorithms.shortestDistance(g, “Tom”, “Anne”);

> -1
```

Who’s to blame?

```
Algorithms.shortestDistance(g, “Tom”, “Anne”);

> 0
```

Who's to blame?

```
class Algorithms {
    /**
     * This method finds the
     * shortest distance between to
     * verticies. It returns -1 if
     * the two nodes are not
     * connected. */
    int shortestDistance(…) {…}
}
```

Who’s to blame?

```
Math.sqrt(-5);

> 0
```


institute for SOFTWARE RESEARCH

Who's to blame?

```
/**
  * Returns the correctly rounded positive square root of a
  * {@code double} value.
  * Special cases:
  * <ul><li>If the argument is NaN or less than zero, then the
  * result is NaN.
  * <li>If the argument is positive infinity, then the result
  * is positive infinity.
  * <li>If the argument is positive zero or negative zero, then
  * the result is the same as the argument.</ul>
  * Otherwise, the result is the {@code double} value closest to
  * the true mathematical square root of the argument value.
  *
  * @param a a value.
  * @return the positive square root of {@code a}.
  * If the argument is NaN or less than zero, the result is NaN.
  */
public static double sqrt(double a) { …}
```

Textual Specification

public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.
- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len. Let *k* be the number of bytes actually read; these bytes will be stored in elements b[off] throughb[off+*k*-1], leaving elements b[off+*k*] through b[off+len-1] unaffected.
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- Throws:
  - IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
  - NullPointerException - If b is null.
  - IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off

Textual Specification

public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data f
attempt is made to read as ma
The number of bytes actually r
until input data is available, en
- If len is zero, then no bytes ar
attempt to read at least one by
end of file, the value -1 is retu
into b.
- The first byte read is stored in
on. The number of bytes read
bytes actually read; these byte
1], leaving elements b[off+*k*]
- In every case, elements b[0] through b[off] and
elements b[off+len] through b[b.length-1] are unaffected.

- Specification of return
- Timing behavior (blocks)
- Case-by-case spec
  - len=0 → return 0
  - len>0 && eof → return -1
  - len>0 && !eof →return >0
- Exactly where the data is stored
- What parts of the array are not affected

- Throws:
  - IOException - If the first byte c
or if the input stream has been
  - NullPointerException - If b is n
  - IndexOutOfBoundsException -
than b.length - off

- Multiple error cases, each with a precondition
- Includes "runtime exceptions" not in throws clause

Specifications

- Contains
  - Functional behavior
  - Erroneous behavior
  - Quality attributes (performance, scalability, security, …)
- Desirable attributes
  - Complete
    - Does not leave out any desired behavior
  - Minimal
    - Does not require anything that the user does not care about
  - Unambiguous
    - Fully specifies what the system should do in every case the user cares about
  - Consistent
    - Does not have internal contradictions
  - Testable
    - Feasible to objectively evaluate
  - Correct
    - Represents what the end-user(s) need

Functional Specification

- States method's and caller's responsibilities
- Analogy: legal contract
  - If you pay me this amount on this schedule…
  - I will build a with the following detailed specification
  - 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 what it means for impl to be correct

Functional Specification

- States
- Analog
  - If y
  - I wil … detailed specification
  - Som … ntracts 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 what it means for impl to be correct

What does the implementation have to fulfill if the client violates the precondition?

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);
```

Advantage of formal specifications:

* runtime checks (almost) for free
* basis for formal verification
* assisting automatic analysis tools

JML (Java Modelling Language) as specifications language in Java (inside comments)

Disadvantages?

Runtime Checking of Specifications with Assertions

```
/*@ requires len >= 0 && array.length == len
   @ ensures \result ==
   @               (\sum int j;  0 <= j && j < len;  array[j])
   @*/
float sum(int array[], int len) {
    assert len >= 0;
    assert array.length == len;
    float sum = 0.0;
    int i = 0;
    while (i < len) {
        sum = sum + array[i]; i = i + 1;
    }
    assert sum …;
    return sum;
}
```

Enable assertions with -ea flag, e.g.,

`> java -ea Main`

Specifications in the real world
Javadoc

```
/**
 * 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

Exceptional behavior

Javadoc contents

- 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

Write a Specification

- Write
  - a type signature,
  - a textual (Javadoc) specification, and
  - a formal specification

for a function **slice(list, from, until)** that returns all values of a list between positions <from> and <until> as a new list

Reminder: Formal specification

```
/*@ 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);
```

Reminder: Javadoc specification

```
/**
 * Returns …
 * @param index position of element …
 * @return the element at the specified posi
 * @throws IndexOutOfBoundsException if the
 *         ({@code index < 0 || index >= thi
 */
E get(int index);
```

Contracts and Interfaces

- All objects implementing an interface must adhere to the interface's contracts
  - Objects may provide different implementations for the same specification
  - Subtype polymorphism: Client only cares about interface, not about the implementation

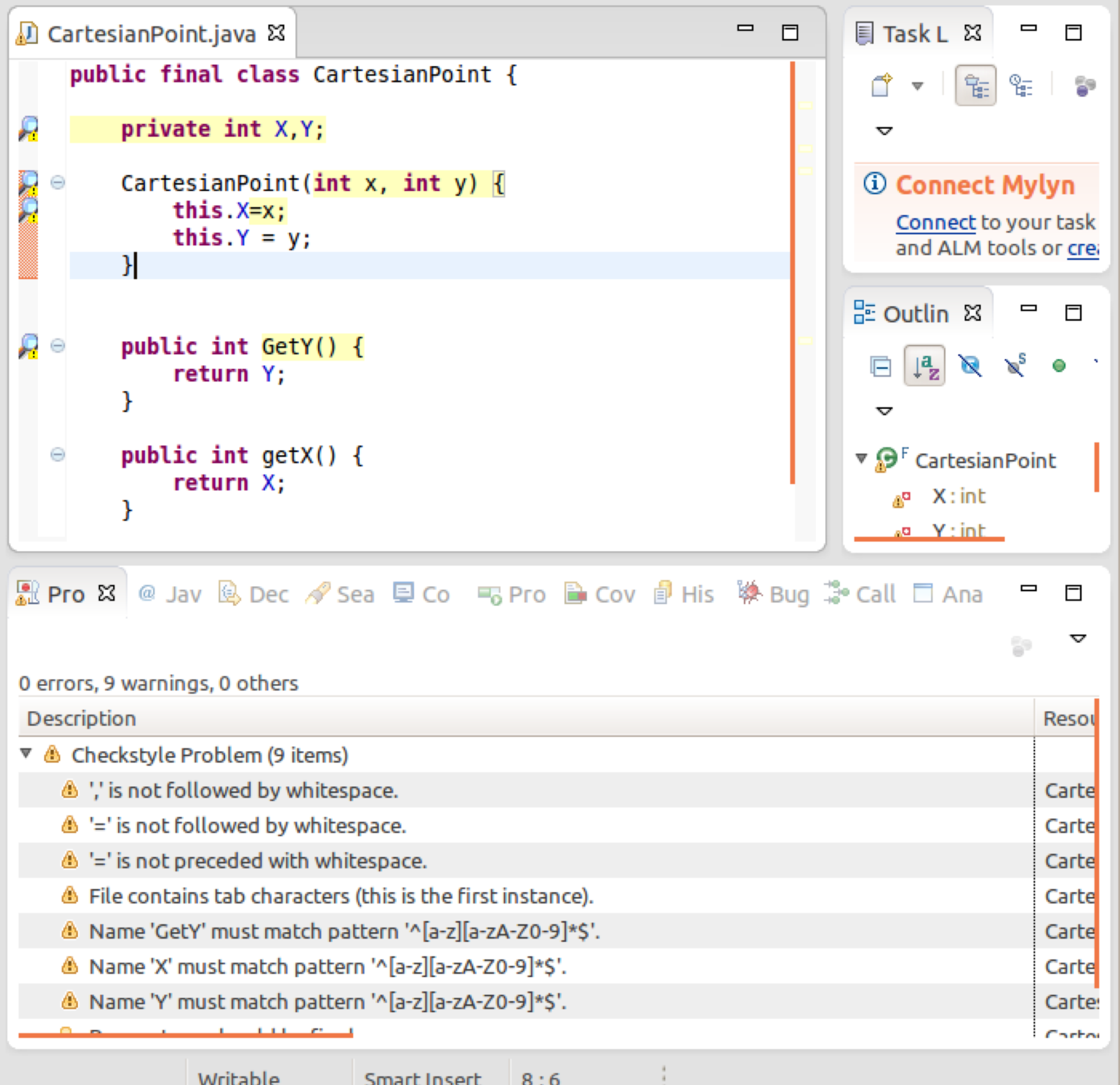
p.getX() s.read()

=> **Design for Change**

Functional correctness

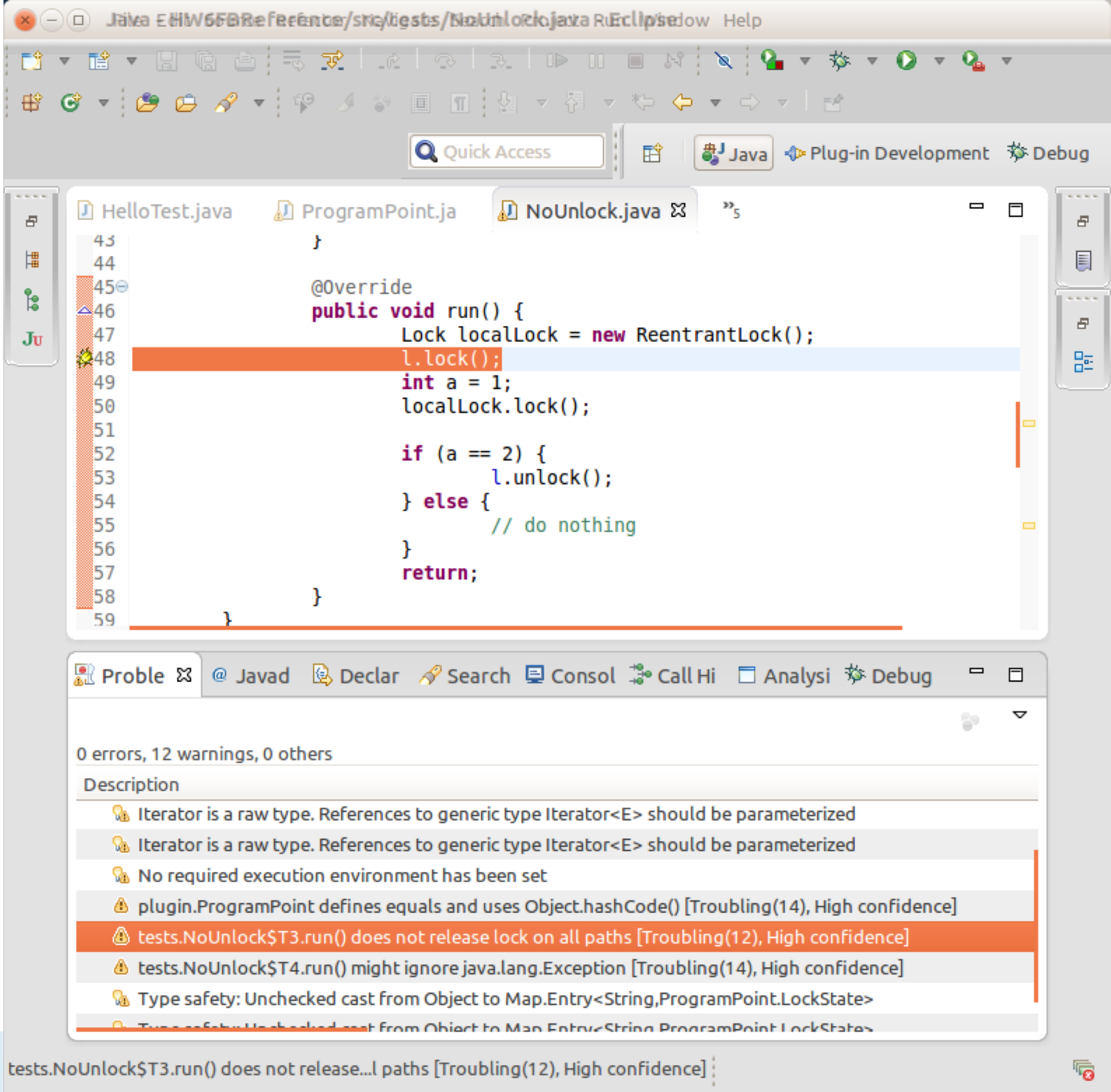
- Compiler ensures types are correct
- Static analysis tools recognize common problems ("bug patterns")
- …

CheckStyle



CartesianPoint.java
public final class CartesianPoint {
private int X,Y;
CartesianPoint(int x, int y) {
this.X=x;
this.Y = y;
}
public int GetY() {
return Y;
}
public int getX() {
return X;
}
Task L
Connect Mylyn
Connect to your task and ALM tools or
Outlin
CartesianPoint
X : int
Y : int
Pro
Jav
Dec
Sea
Co
Pro
Cov
His
Bug
Call
Ana
0 errors, 9 warnings, 0 others
Description
Resou
Checkstyle Problem (9 items)
',' is not followed by whitespace.
'=' is not followed by whitespace.
'=' is not preceded with whitespace.
File contains tab characters (this is the first instance).
Name 'GetY' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Name 'X' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Name 'Y' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Writable
Smart Insert
8 : 6

SpotBugs



Quick Access
Java
Plug-in Development
Debug
HelloTest.java
ProgramPoint.ja
NoUnlock.java
@Override
public void run() {
Lock localLock = new ReentrantLock();
l.lock();
int a = 1;
localLock.lock();
if (a == 2) {
l.unlock();
} else {
// do nothing
}
return;
}
}
Proble
Javad
Declar
Search
Consol
Call Hi
Analysi
Debug
0 errors, 12 warnings, 0 others
Description
Iterator is a raw type. References to generic type Iterator<E> should be parameterized
Iterator is a raw type. References to generic type Iterator<E> should be parameterized
No required execution environment has been set
plugin.ProgramPoint defines equals and uses Object.hashCode() [Troubling(14), High confidence]
tests.NoUnlock$T3.run() does not release lock on all paths [Troubling(12), High confidence]
tests.NoUnlock$T4.run() might ignore java.lang.Exception [Troubling(14), High confidence]
Type safety: Unchecked cast from Object to Map.Entry<String,ProgramPoint.LockState>
tests.NoUnlock$T3.run() does not release...l paths [Troubling(12), High confidence]

Functional correctness

- Compiler ensures types are correct
- Static analysis tools recognize common problems ("bug patterns")
- Formal verification
  - Mathematically prove code matches its specification
- Testing
  - Execute program with select inputs in a controlled environment
- …

Formal verification vs. testing?

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

Donald Knuth, 1977

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

Edsger W. Dijkstra, 1969

Formal verification vs. testing?

Consider `java.util.Arrays.binarySearch`:

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
         else if (midVal > key)
            high = mid - 1;
         else
            return mid; // key found
     }
     return -(low + 1);  // key not found.
}
```

Formal verification vs. testing?

Consider `java.util.Arrays.binarySearch`:

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
         else if (midVal > key)
            high = mid - 1;
         else
            return mid; // key found
     }
     return -(low + 1);  // key not found.
}
```

Fails if
`low + high` > MAXINT (2^{31} - 1)
Sum overflows to negative value

Comparing strategies for correctness

- 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

Which strategies to use in your project?

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 or a testing system?
- How to check output / assertions?
- What are the costs?
- Are bugs reproducible?

Automate testing

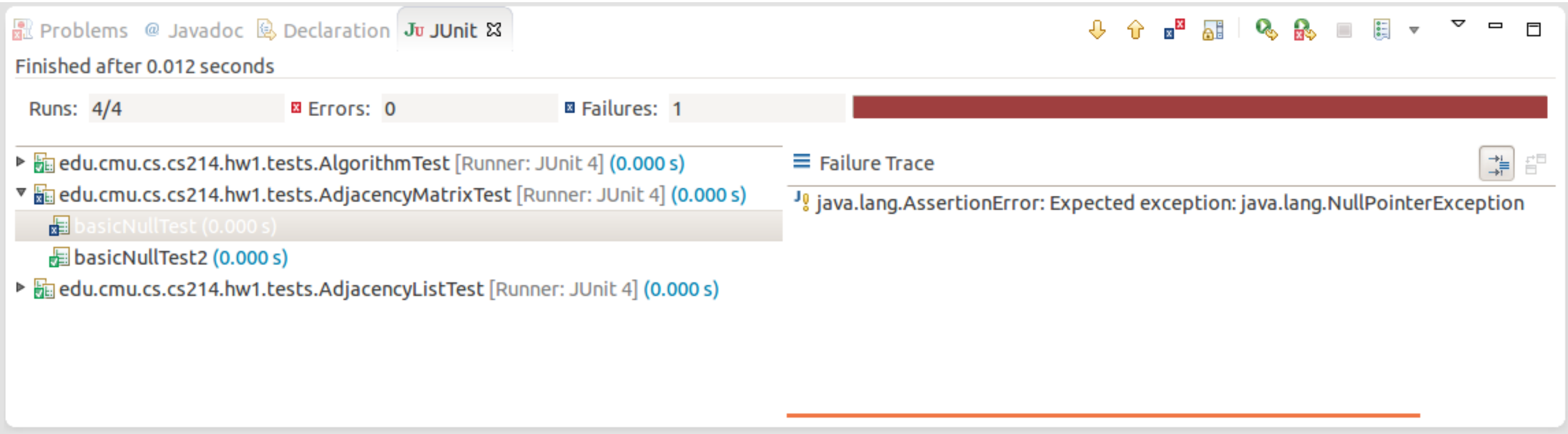
- Execute a program with specific inputs, check output for expected values
- Set up testing infrastructure
- Execute tests regularly
  - After *every* change

Unit testing

- 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
- Few dependencies on other system parts or environment

JUnit

- A popular, easy-to-use, unit-testing framework for Java



A JUnit example

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
        @Test
        public void testSanityTest(){
                Graph g1 = new AdjacencyListGraph(10);
                Vertex s1 = new Vertex("A");
                Vertex s2 = new Vertex("B");
                assertEquals(true, g1.addVertex(s1));
                assertEquals(true, g1.addVertex(s2));
                assertEquals(true, g1.addEdge(s1, s2));
                assertEquals(s2, g1.getNeighbors(s1)[0]);
        }

        @Test
        public void test….

        private int helperMethod…
}
```

Selecting test cases

- Write tests based on the specification, for:
  - Representative cases
  - Invalid cases
  - Boundary conditions
- Write stress tests
  - Automatically generate huge numbers of test cases
- Think like an attacker
- Other tests: performance, security, system interactions, …

A testing example

```
/**
 * computes the sum of the first len values of the array
 *
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the first len array values
 * @throws NullPointerException if array is null
 * @throws IndexOutOfBoundsException if len > array.length
 * @throws IllegalArgumentException if len < 0
 */
int partialSum(int array[], int len);
```

A testing example

```
/**
 * computes the sum of the first len values of the array
 *
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the first len array values
 * @throws NullPointerException if array is null
 * @throws IndexOutOfBoundsException if len > array.length
 * @throws IllegalArgumentException if len < 0
 */
int partialSum(int array[], int len);
```

- Test negative length

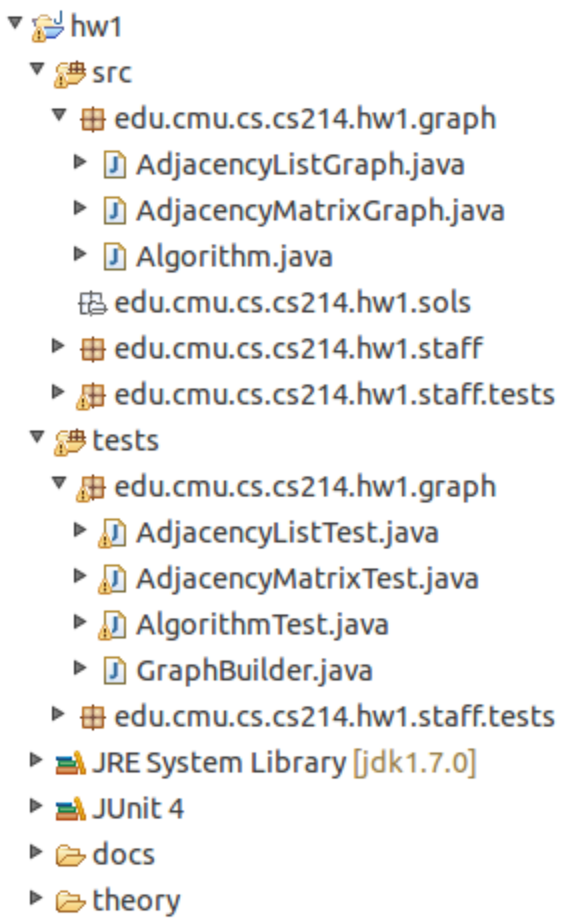
A testing example

```
/**
 * computes the sum of the first len values of the array
 *
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the first len array values
 * @throws NullPointerException if array is null
 * @throws IndexOutOfBoundsException if len > array.length
 * @throws IllegalArgumentException if len < 0
 */
int partialSum(int array[], int len);
```

- Test negative length
- Test length > array.length
- Test length == array.length
- Test small arrays of length 0, 1, 2
- Test null array
- Test long array
- Stress test with randomly-generated arrays and lengths

Test organization conventions

- Have a test class `FooTest` for each public class Foo
- Separate source and test directories
  - `FooTest` and Foo in the same package



Testable code

- Think about testing when writing code
  - Modularity and testability go hand in hand
- Same test can be used on all implementations of an interface!
- Test-driven development
  - Writing tests before you write the code
  - Tests can expose API weaknesses

Writing testable code

```
//700LOC
public boolean foo() {
    try {
        synchronized () {
            if () {
            } else {
            }
            for () {
                if () {
                    if () {
                        if () {
                            if ()?
                            {
                                if () {
                                    for () {
                                    }
                                }
                            }
                        } else {
                            if () {
                                for () {
                                    if () {
                                    } else {
                                    }
                                    if () {
                                    } else {
                                        if () {
                                        }
                                    }
                                    if () {
                                        if () {
                                            if () {
                                                for () {
                                                }
                                            }
                                        }
                                    } else {
                                    }
                                }
                            } else {
                            }
                        }
                    }
                }
            }
```

Unit testing as a design mechanism:

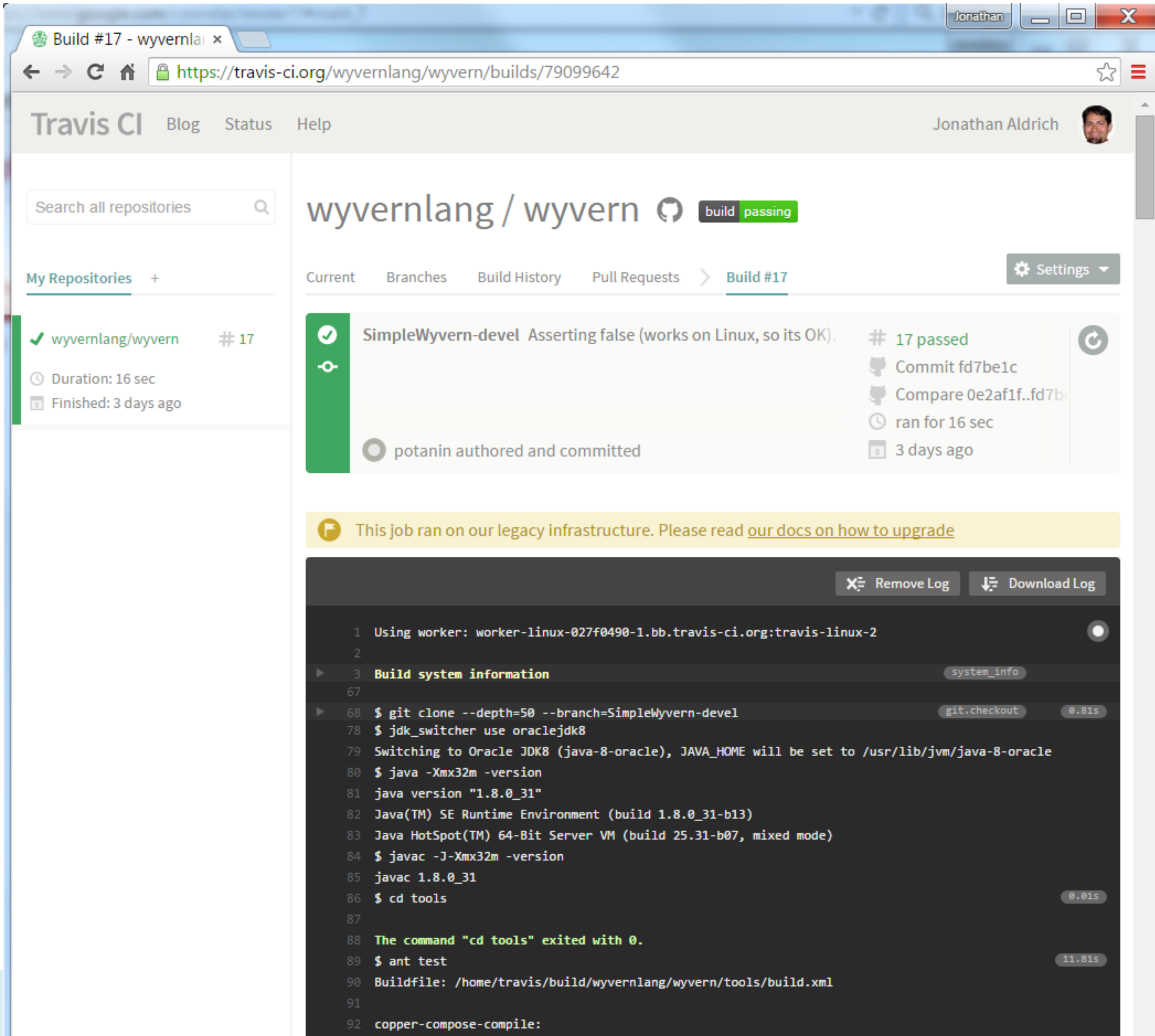
- Code with low complexity
- Clear interfaces and specifications

Source: http://thedailywtf.com/Articles/Coding-Like-the-Tour-de-France.aspx

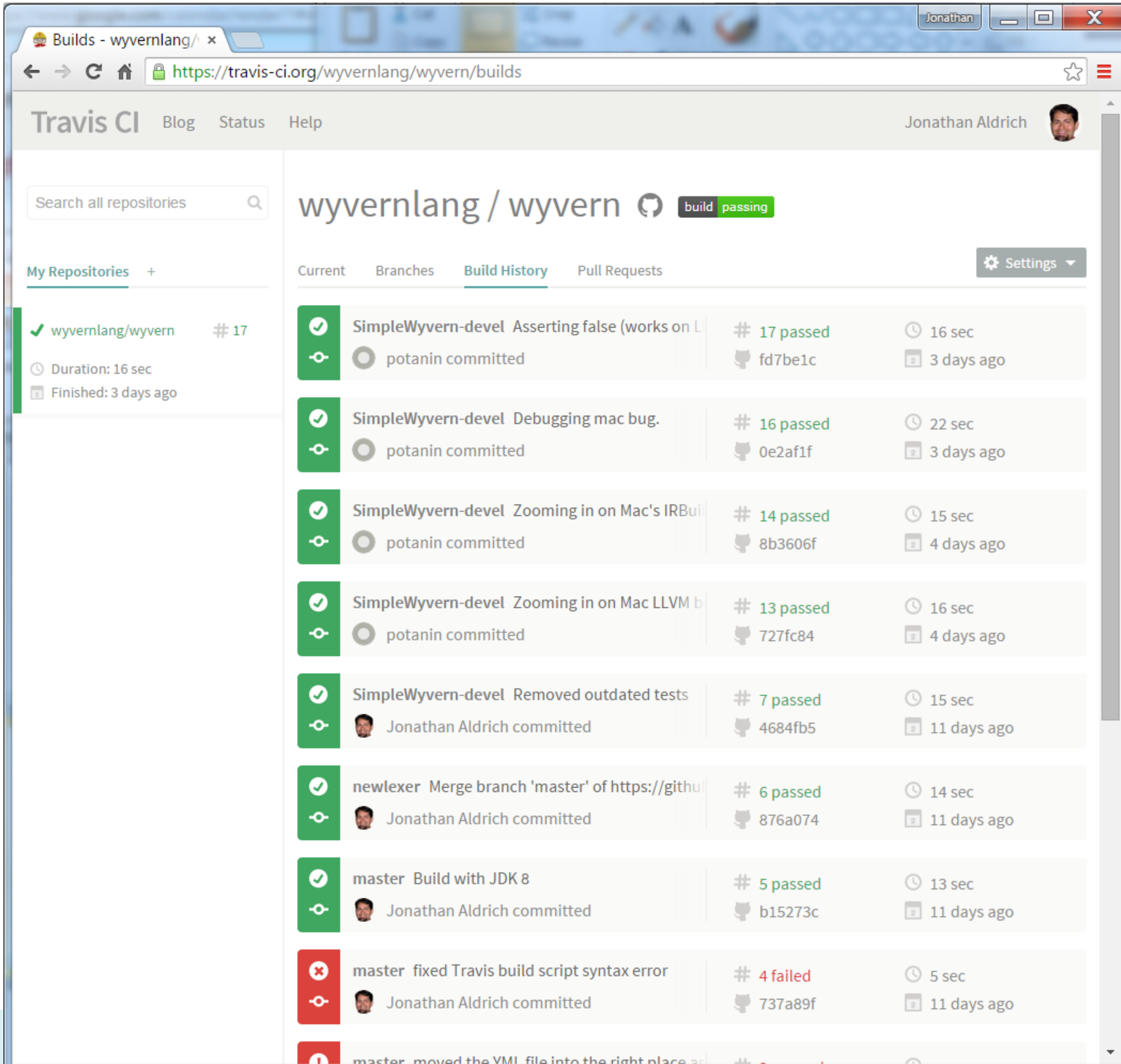
Run tests frequently

- Run tests before every commit
  - Do not commit code that fails a test
- If entire test suite becomes too large and slow:
  - Run local package-level tests ("smoke tests“) frequently
  - Run all tests nightly
  - Medium sized projects easily have 1000s of test cases
- Continuous integration servers scale testing

Continuous integration: Travis CI



Continuous integration: Travis CI build history



When should you stop writing tests?

When should you stop writing tests?

- When you run out of money…
- When your homework is due…
- When you can't think of any new test cases...
- The *coverage* of a test suite
  - Trying to test all parts of the implementation
  - Statement coverage
    - Execute every statement, ideally
    - Compare to: method coverage, branch coverage, path coverage

Summary

- Please complete the course reading assignments
- Java has a bipartite type system: primitives and objects
- Power of OO programming comes from dynamic dispatch
- Collections framework is powerful and easy to use
- Test early, test often!