Principles of Software Construction: Testing: One, Two, Three

Josh Bloch Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

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

Key concepts from Tuesday…

- Code must be clean and concise
  - Repetition is toxic
- Good coding habits matter
- Enums provide all `Object` methods & `compareTo`
- Zero is not an acceptable hash function!
- Not enough to be **merely** correct; code must be **clearly** correct – **nearly** correct is right out.

Outline

- Test suites and coverage
- Testing for complex environments
- Static Analysis

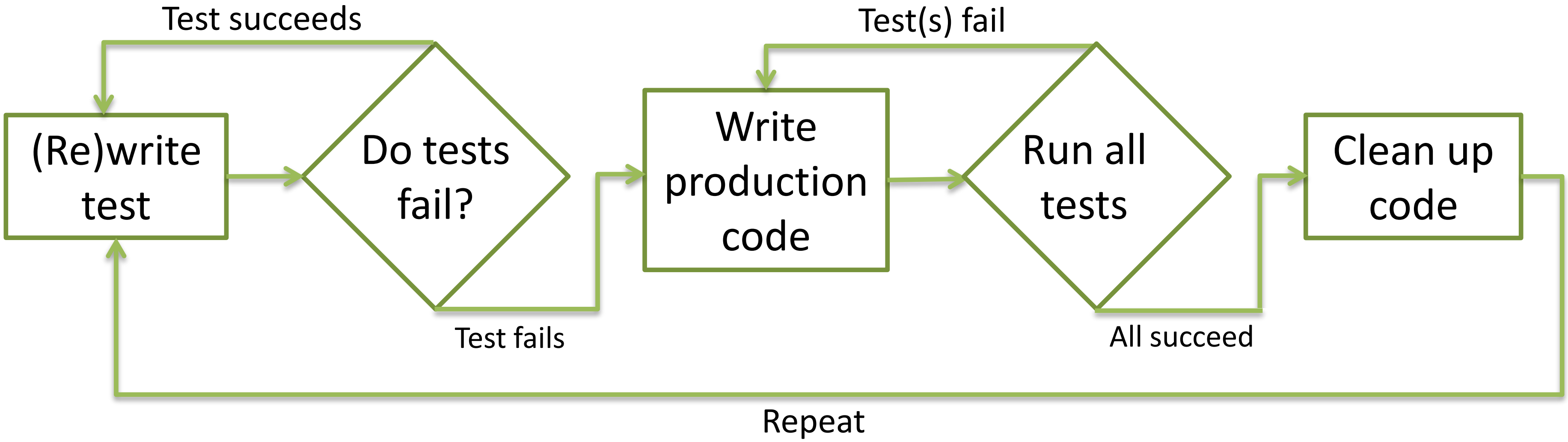
Correctness

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
- When to stop testing?

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 first
- Avoids writing unneeded code
- Higher product quality
  - Better code
  - Fewer defects
- Higher test suite quality
- Higher productivity
- More fun!

TDD in practice

- Empirical studies on TDD show
  - May require more effort
  - May improve quality and save time
- Selective use of TDD is best
- The only way to go 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 in test suites than in code
  - Realistically, test suites look worse than code
- Can be fun to write if approached in this spirit

Next best thing to exhaustive testing: *random inputs*

- Also know as *fuzz testing*, *bashing*
  - – Formerly known as *torture testing*
  - – Now known as *enhanced interrogation* ☺
- 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 (*cyclomatic complexity*) – too fine
    - Cost is high, value is low

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
  - Also 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

- Test suites and coverage
- Testing for complex environments
- Static Analysis

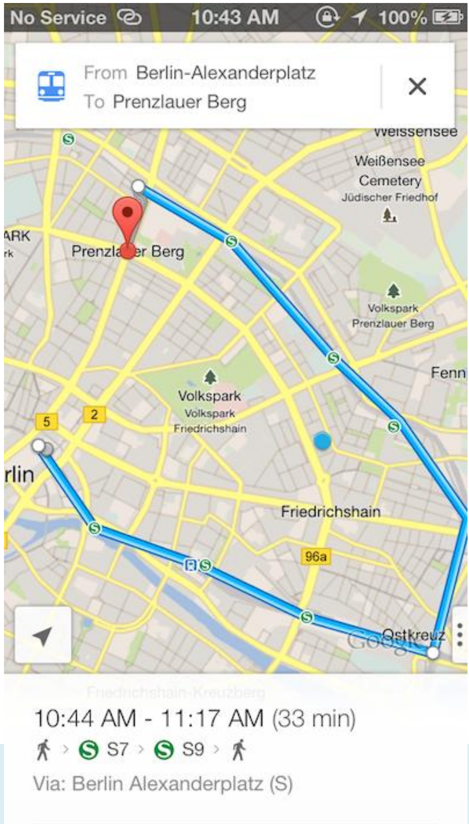
Problems when testing some apps

- User interfaces and user interactions
  - Users click buttons, interpret output
  - Waiting and timing issues
- Testing against big infrastructure
  - databases, web services, etc.
- Real world effects
  - Printing, mailing documents, etc.

- Collectively comprise *the test environment*

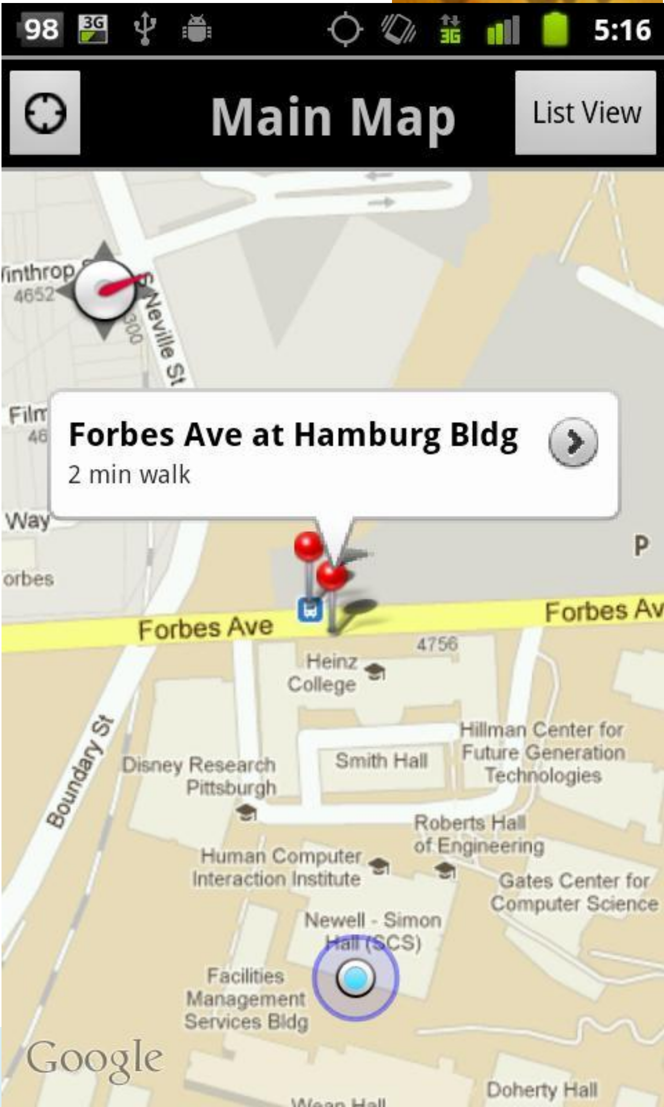
Example – Tiramisu app

- Mobile route planning app
- Android UI
- Back end uses live PAT data



From Berlin-Alexanderplatz
To Prenzlauer Berg

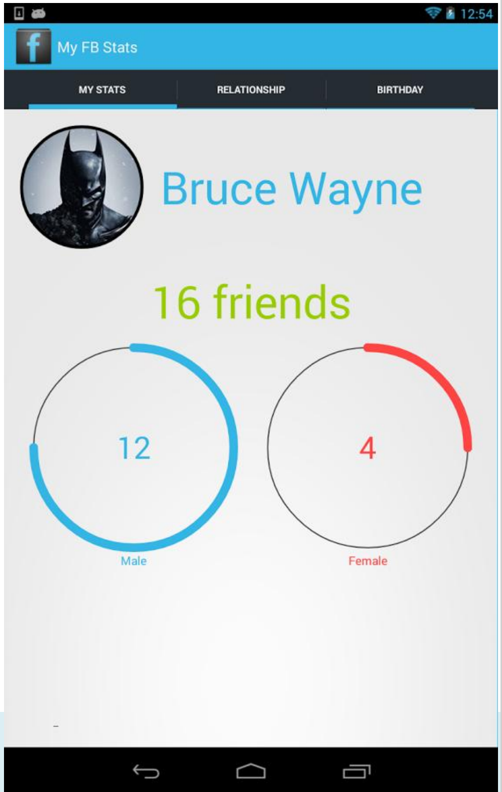
10:44 AM - 11:17 AM (33 min)
S7 › S9
Via: Berlin Alexanderplatz (S)

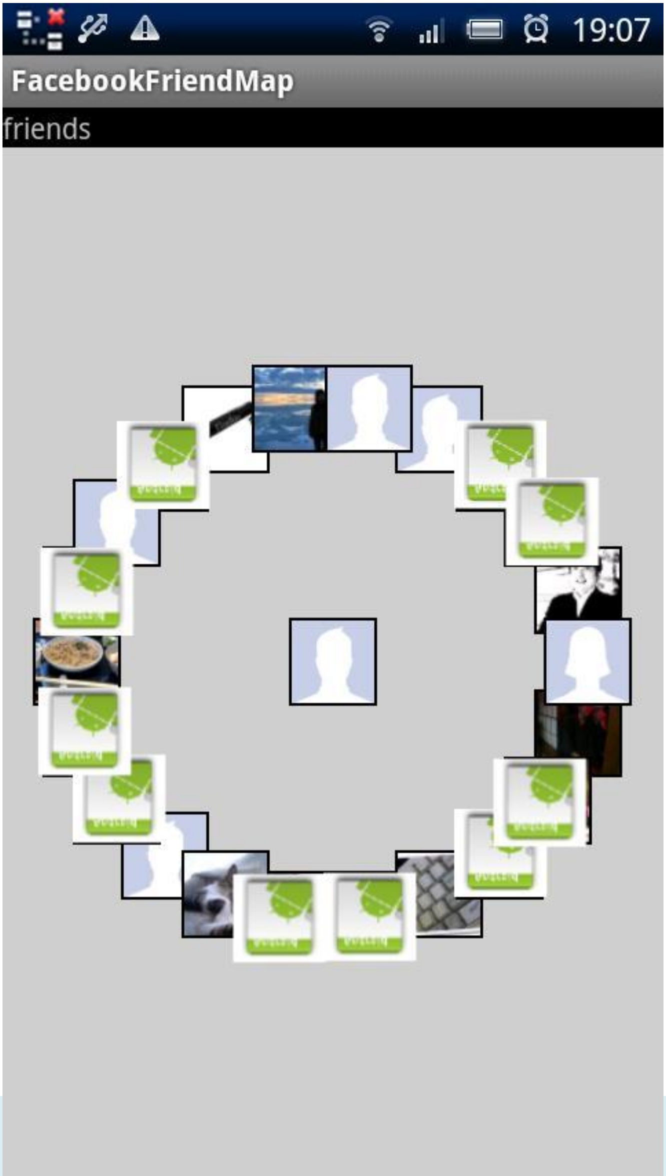

98
3G
5:16
Main Map
List View
Winthrop
4652
Neville St
Film
Way
orbes
P
Forbes Ave
Forbes Av
4756
Heinz College
Boundary St
Disney Research Pittsburgh
Smith Hall
Hillman Center for Future Generation Technologies
Roberts Hall of Engineering
Human Computer Interaction Institute
Gates Center for Computer Science
Newell - Simon Hall (SCS)
Facilities Management Services Bldg
Google
Doherty Hall
Wean Hall

Forbes Ave at Hamburg Bldg
2 min walk

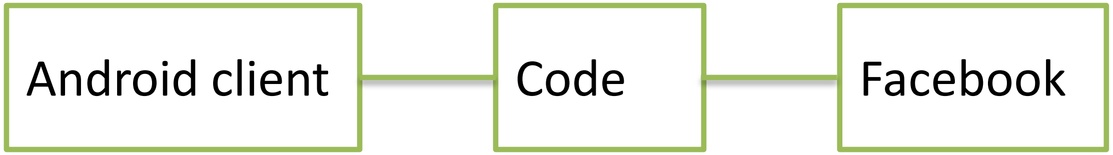
Another example

- 3rd party Facebook apps
- Android user interface
- Internal computations like HW1
- Backend uses Facebook data





Testing in real environments



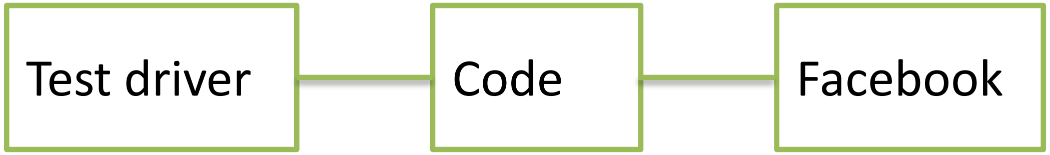
Android client

Code

Facebook

```
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

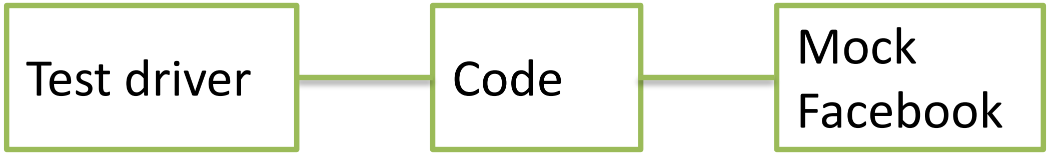


```
@Test void testGetFriends() {
   assert getFriends() == …;
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookAPI api = new FacebookAPI(c);
   List<Node> persons = api.getFriends("john");
   for (Node person1 : persons) {
       for (Node person2 : persons) {
      …
       }
   }
   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 driver

Code

Mock Facebook

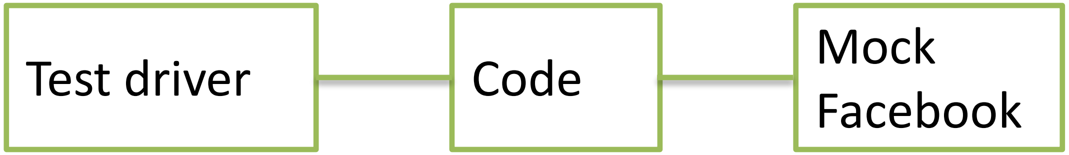
```
@Test void testGetFriends() {
   assert getFriends() == …;
}
List<Friend> getFriends() {

   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
- Use tools to facilitate this sort of testing
  - *Dependency injection* tools, e.g., Guice, Dagger
  - 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

- Write testable code
- When a mock may be appropriate, design for it
- Hide subsystem behind an interface
- Use factory, not constructor to instantiate
- Use appropriate tools
  - Dependency injection or mocking frameworks

More Testing in 313

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

Outline

- Test suites and coverage
- Testing for complex environments
- Static Analysis

Remember this bug?

```
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")));
    }
}
```

Here’s the problem

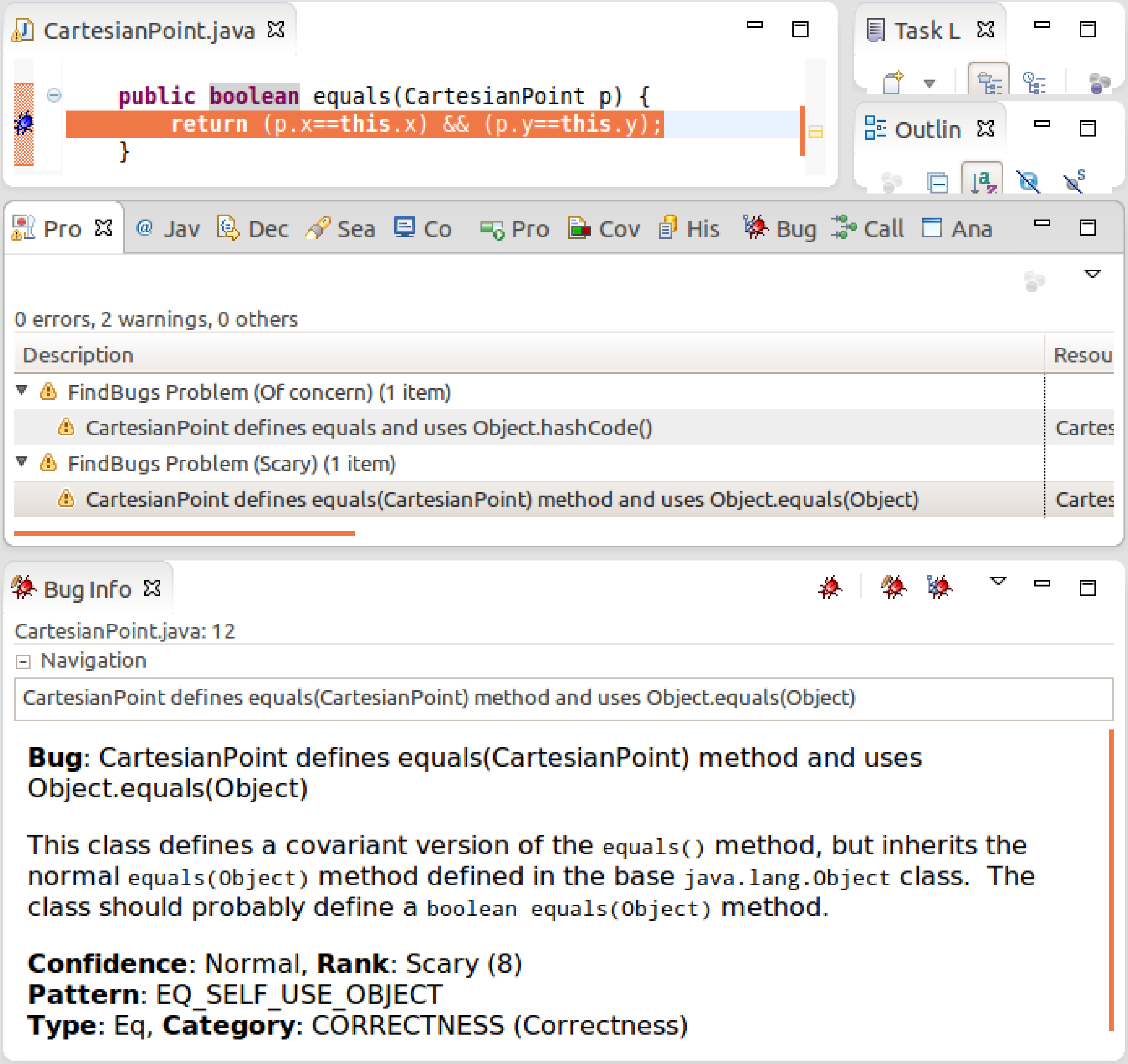
```
public class Name {
    private final String first, last;
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first; this.last = last;
    }
    public boolean equals(Name o) { // Accidental overloading
        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")));
    }
}
```

Here’s the solution

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);
}
```

FindBugs



```
    public boolean equals(CartesianPoint p) {
        return (p.x==this.x) && (p.y==this.y);
    }
```

0 errors, 2 warnings, 0 others

Description	Resou
▼ ⚠ FindBugs Problem (Of concern) (1 item)	
⚠ CartesianPoint defines equals and uses Object.hashCode()	Cartes
▼ ⚠ FindBugs Problem (Scary) (1 item)	
⚠ CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)	Cartes

CartesianPoint.java: 12

⊟ Navigation

CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)

Bug: CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)

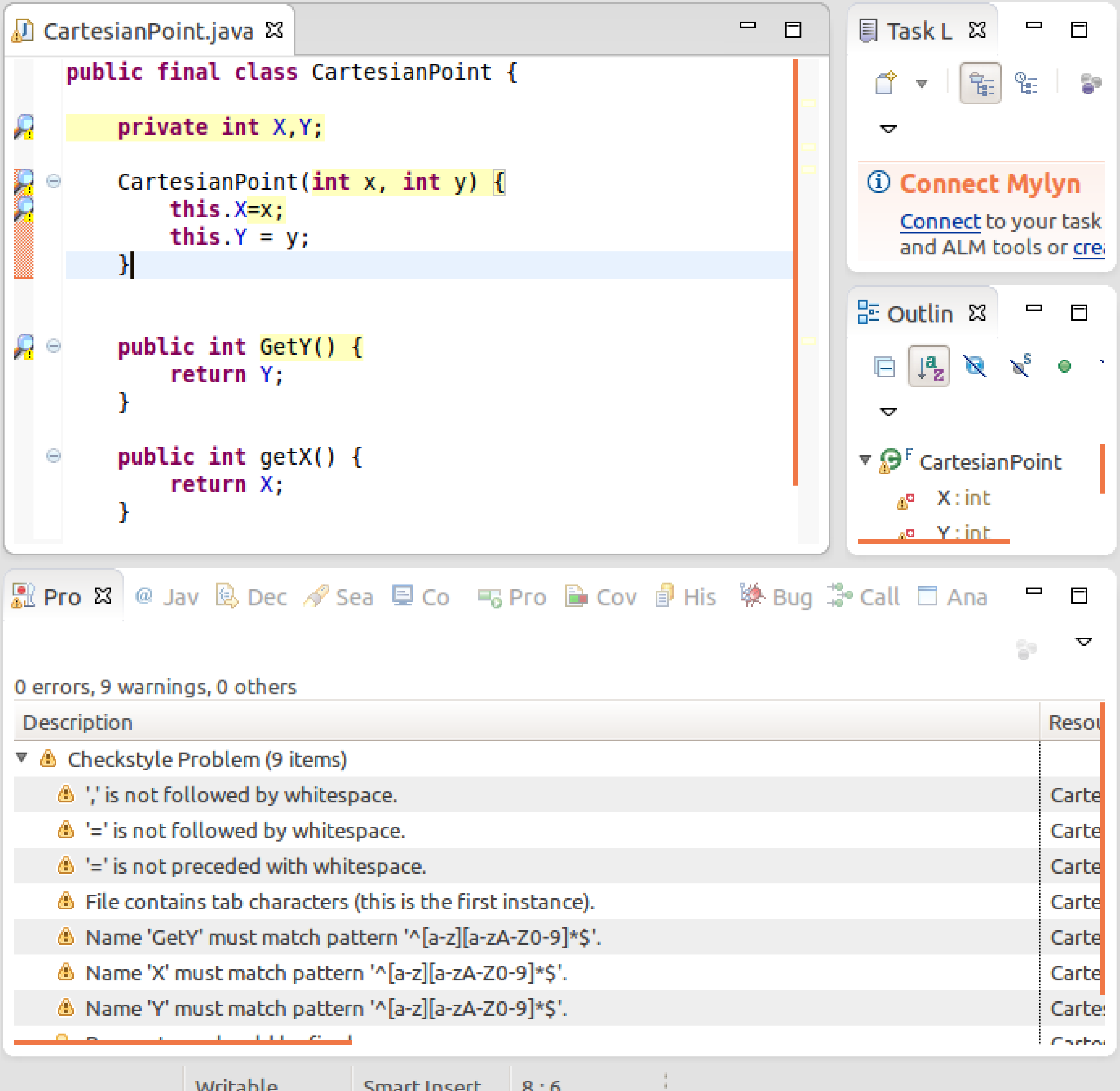
This class defines a covariant version of the `equals()` method, but inherits the normal `equals(Object)` method defined in the base `java.lang.Object` class. The class should probably define a `boolean equals(Object)` method.

Confidence: Normal, **Rank**: Scary (8)
Pattern: EQ_SELF_USE_OBJECT
Type: Eq, **Category**: CORRECTNESS (Correctness)

Improving bug-finding accuracy with annotations

- **`@NonNull`**
- **`@DefaultAnnotation(@NonNull)`**
- **`@CheckForNull`**
- **`@CheckReturnValue`**

CheckStyle



```
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;
    }
```

Connect Mylyn

Connect to your task and ALM tools or cre

0 errors, 9 warnings, 0 others

Description	Resou
Checkstyle Problem (9 items)	
',' is not followed by whitespace.	Carte
'=' is not followed by whitespace.	Carte
'=' is not preceded with whitespace.	Carte
File contains tab characters (this is the first instance).	Carte
Name 'GetY' must match pattern '^[a-z][a-zA-Z0-9]*$'.	Carte
Name 'X' must match pattern '^[a-z][a-zA-Z0-9]*$'.	Carte
Name 'Y' must match pattern '^[a-z][a-zA-Z0-9]*$'.	Carte

Static analysis

- Analyzing code without executing it
  - Also known as *automated inspection*
- Some tools looks for *bug patterns*
- Some formally verify specific aspects
- Typically integrated into IDE or build process
- Type checking by compiler is static analysis!

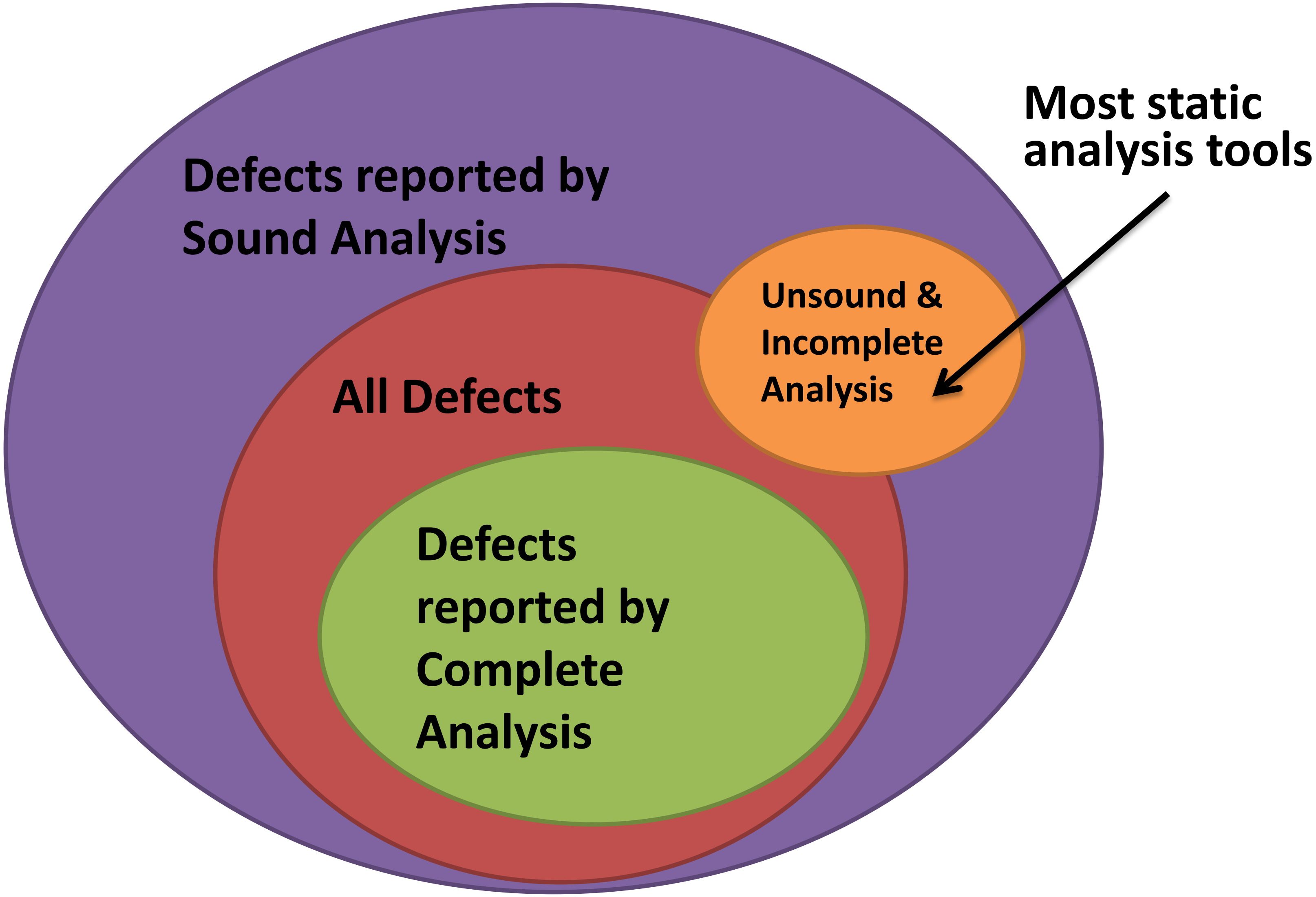
Static analysis: a formal treatment

- Static analysis is the systematic examination of an abstraction of a program’s state space
- By abstraction we mean
  - Don’t track everything!
  - Consider only an important attribute

	Error exists	**No error exists**
Error Reported	True positive (correct analysis result)	False positive (annoying noise)
No Error Reported	False negative (false confidence)	True negative (correct analysis result)

Results of static analysis can be classified as

- **Sound:**
    - Every reported defect is an actual defect
        - **No false positives**
    - Typically underestimated
- **Complete:**
    - Reports all defects
        - **No false negatives**
    - Typically overestimated



Defects reported by Sound Analysis

All Defects

Unsound & Incomplete Analysis

Defects reported by Complete Analysis

Most static analysis tools

The bad news: Rice's theorem

- There are limits to what static analysis can do
- Every static analysis is necessarily incomplete, unsound, or undecidable

> **"Any nontrivial property about the language recognized by a Turing machine is undecidable."**
>
> Henry Gordon Rice, 1953

Homework

- How would you test:
  - A numerical class that does arithmetic?
  - A sorting algorithm?
  - A shuffling algorithm?

Conclusion

- There are many forms of quality assurance
- **Testing is critical**
- Design your code to facilitate testing
- Coverage metrics can help approximate test suite quality
- Static analysis tools can detect certain bugs