Unit Testing

219343: Software Testing

Some materials are from Alberto Savoia's slides on unit testing,
George Necula's software engineering course,
and Hunt and Thomas, "Pragmatic Unit Testing," 2003.

Quiz

- Consider class `SortedIntArray`, where
  - Member `array` is initialized to be of length `maxsize`.
  - It always stores `eltCount` elements in `array[0]`, ..., `array[eltCount-1]`.
  - All elements are in an ascending order.
- Write one test method for method `insert`, which inserts `x` into `array` (all invariants must hold afterwards).

```
public class SortedIntArray {

    protected int maxsize;
    protected int [] array;
    protected int eltCount;

    // ...

    public void insert(int x) {
        // ...
    }
}
```

A classic example

- John
  - John works hard. He codes everyday. The project deadline is tomorrow. He types in about two hundred new lines per hour, and thinks that after 6 hours and roughly a thousand new lines added the program would work flawlessly.
- Betty
  - Betty works hard. She codes everyday. The project deadline is tomorrow. She types in about one hundred new lines per hour, and keeps testing each method she adds. She does not proceed to write new codes unless all previously written pieces work correctly.
- Guess who will go to bed earlier?

Some rules from eXtreme Programming

- Coding:
  - Code the unit test first
  - All production code is pair programmed
  - Integrate often
- Testing:
  - All code must have unit test
  - When a bug is found, tests are created
  - Acceptance tests are run often

from: http://www.extremeprogramming.org/rules.html

Developer Testing Revolution

- Developer testing is a key component in a hot paradigm: Agile/eXtreme Programming
- The Developer Testing Trinity
  - Test
  - Test early and often
  - Test well

Good reasons for developer testing

- Reduces unit-level bugs
- Forces you to slow down and think
- Improves design
- Makes development faster
- Tests are good documentation
- Tests constrain features
- Tests allows safe refactoring and reduce the cost of change
- Tests defend against other programmers
- Tests reduce fear

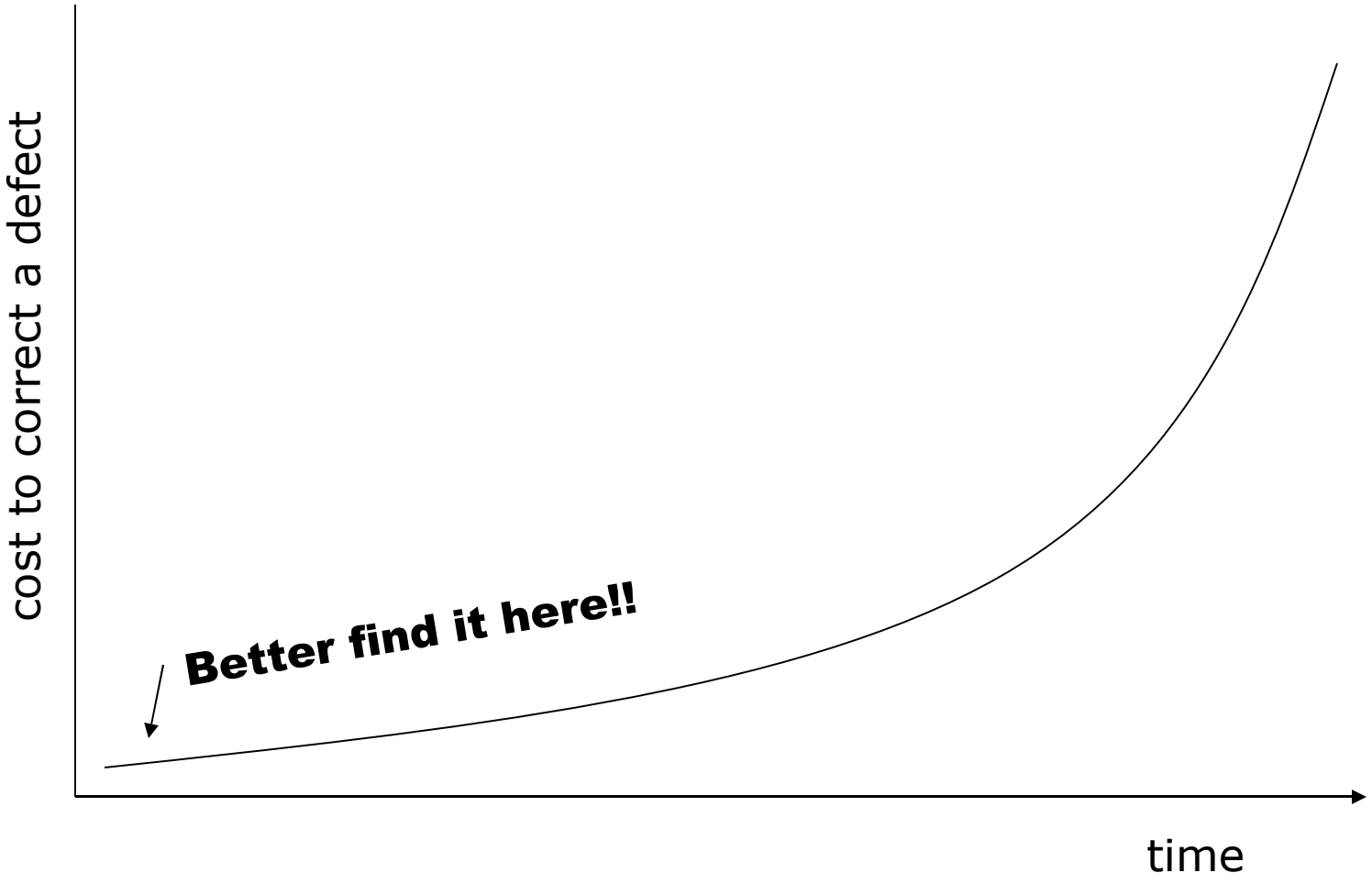
Goals

- Does the code do what I want?
- Does the code do what I want all the time?
- Can I depend on it?
- Also: get a document for the code.
  - Always correct documentation for your intention.

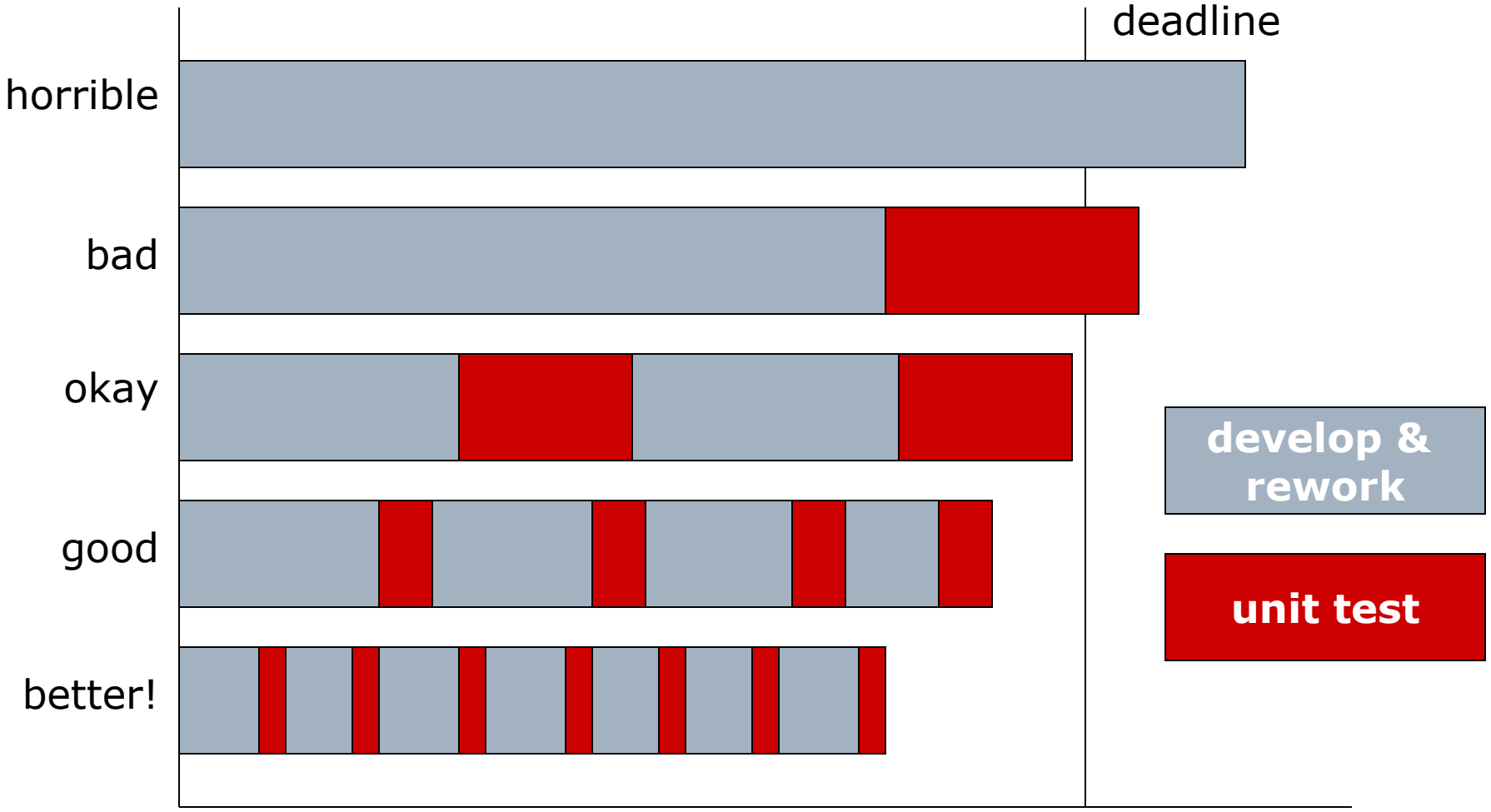
Test your code

- It is your code, and your responsibility
  - Do it for your current colleagues
  - Do it for future generation of colleagues
  - Do it for yourself

Test early and often



Test early and often



Heaven!

- Every class has unit tests
- The tests are executed many times each day
- The tests are thorough, up to date, and easy to maintain and analyze

- **In this class, we might not aim for that.**

What's a unit

- What's a unit
  - A single method/function/procedure
  - A collection of related methods/functions/procedues
- Ideal world
  - independent, self-sufficient, standalone
- Real world
  - lots of dependence

Basic structure

- Setup
  - Create initial states
  - Initialize method parameters
  - Store pre-execute values
- Execute code
- Compare results

Partial correctness assertions

- In the context of formal verification
  - *Notation*: {P} S {Q}
  - *Meaning*: if *precondition* P is met, after S terminates, *postcondition* Q holds
- Think of unit testing as a way to test partial correctness
  - setup P
  - run methods S
  - check Q

JUnit

JUnit

- JUnit is a unit test framework for Java developed by Kent Beck and Erich Gamma
- The current version is 4.1
  - Most documents on the Internet consider older versions (3.8.x)
  - Also in some tools (e.g., Netbeans 5.5)

- http://www.junit.org/

Class Median: a simple example

- Median.java

```
public class Median {

    public static int median(int a, int b, int c) {
        if(((b<a) && (a<c)) || ((c<a) && (a<b)))
            return a;
        else if(((c<b) && (b<a)) || ((a<b) && (b<c)))
            return b;
        else
            return c;
    }

}
```

TestMedian (for JUnit 4)

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

public class TestMedian {

    @Test public void testMedian1() {
        assertEquals(2, Median.median(1,2,3));
        assertEquals(2, Median.median(3,1,2));
        assertEquals(2, Median.median(3,2,1));
    }

    @Test public void testMedianDup() {
        assertEquals(2, Median.median(3,2,2));
        assertEquals(1, Median.median(1,2,1));
    }

}
```

Using Junit 4

- Import:
  - `import org.junit.Test;`
  - `import static org.junit.Assert.*;`
- Annotate test methods with @Test
  - Example:
    - `@Test public void TestMedian1() {…}`
- In test methods, verify results using assertXXXX

TestMedian (for JUnit 3)

```
import junit.framework.TestCase;

public class MedianTest extends TestCase {

    public void testMedian1() {
        assertEquals(2, Median.median(1,2,3));
        assertEquals(2, Median.median(3,1,2));
        assertEquals(2, Median.median(3,2,1));
    }

    public void testMedianDup() {
     assertEquals(2, Median.median(3,2,2));
     assertEquals(1, Median.median(1,2,1));
    }
}
```

Using Junit 3

- Import:
  - `import junit.framework.TestCase;`
- Class extends `TestCase`

- Begin test method names with "`test`"
  - Example:
    - `public void testMedian1() {…}`

- In test methods, verify results using assertXXXX

org.junit.Assert.assertXXXX

- Methods in class Assert
  - assertEquals([message,] obj1, obj2)
  - assertNull([msg,] obj)
  - assertNotNull([msg,] obj)
  - assertSame([msg,] obj1, obj2)
  - assertNotSame([msg,] obj)
  - assertTrue([msg,] cond)
  - assertFalse([msg,] cond)

  - fail([msg])

Fixtures

- To avoid writings duplicate codes.

- Initialization/cleaning-up for each testcase:
  - Add members holding required objects
  - Annotate initialization method with **@Before**
  - Annotate deinitialization method with **@After**
  - For one-time set up and tear down use: **@BeforeClass** and **@AfterClass**
  - *Don't forget to import org.junit.Before, org.junit.After, org.junit.BeforeClass, org.junit.AfterClass*
- For Junit 3, use: setUp() and tearDown()

Practice

- SortedIntArray.insert

Expected Exceptions

- With try and catch

```
@Test
public void testOverInsert2() {
    SortedIntArray s = new SortedIntArray(1);

    try {
        s.insert(10);
        s.insert(100);
        fail("Expected exception here");
    }catch(SortedIntArray.TooManyEltsException tml){
    }
}
```

Expected Exceptions with Junit 4

- With try and catch

```
@Test(expected =
   SortedIntArray.TooManyEltsException.class)
public void testOverInsert() {
   SortedIntArray s = new SortedIntArray(1);

   s.insert(10);
   s.insert(100);
}
```

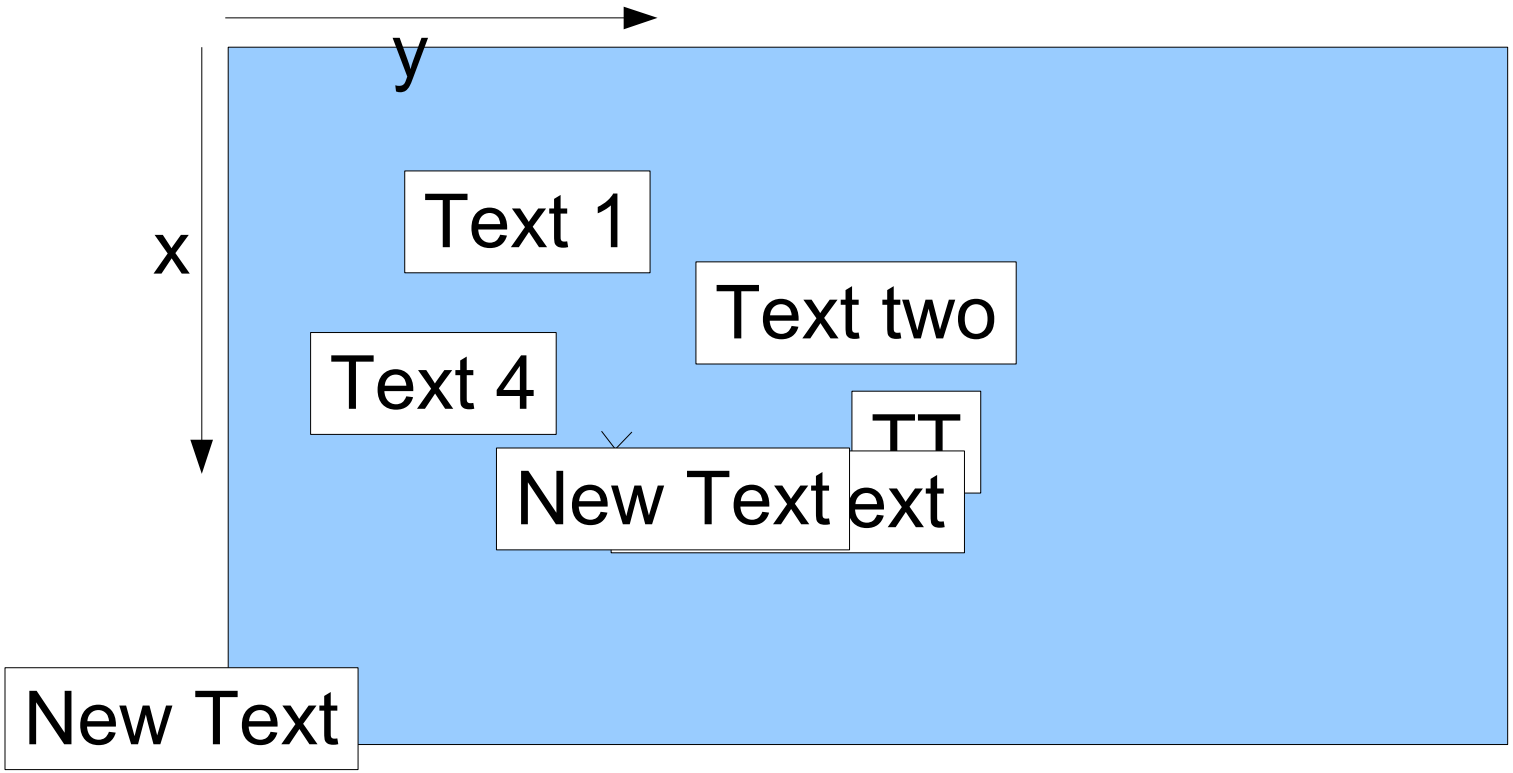
JUnit

- Don't forget that you can look at what lies inside the classes. *It is not just a blind test!!*

Example for Stack class

```
@Test public void testStress() {
    for(int i=0; i<100; i++)
        stack.push(""+i);

    Stack.StackNode node = stack.stackTop;

    for(int i=0; i<100; i++) {
        assertTrue(node != null);
        assertEquals(node.item,""+(99-i));
        node = node.next;
    }
    assertTrue(node == null);
}
```

Map Application: Locating box



- Want to find the location to display text.

Method isFree

```
class TextLocator {
   public void add(int x, int y,
                   int tw, int th) {...}

   // method under test
   public boolean isFreeFor(int x, int y,
                            int tw, int th) {
      ...
   }
}
```

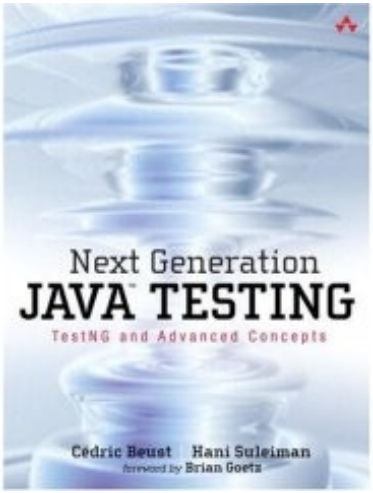
Practice

- We have class `TopK` which is a container class that allows you to add new integers and keeps K highest integers.
- Its interface is:

```
public class TopK {

    public TopK(int k) {...} // constructor:
                             // k = # of items it keeps

    public void add(int c) {...}   // add new integer.

    public int getNumKept() {...} // get # of integer it keeps

    // get the i-th highest integer in the list.
    public int get(int i) {...}
}
```

Another testing framework

- TestNG
  - More flexible, e.g., test cases can have parameters
  - Covers more area of testint, e.g., functional, integration
  - http://testng.org/doc/index.html
  - Read about the difference from JUnit 4 at http://www.ibm.com/developerworks/java/library/j-cq08296/



What to test

- **Right** – are the result *right*?
- **B** – are all the *boundary conditions* CORRECT?
- **I** – can you check *inverse* relationship?
- **C** – can you *cross-check* results using other means?
- **E** – can you force *error condition* to happen?
- **P** – are *performance* characteristics within bounds?

From Hunt and Thomas, “Pragmatic Unit Testing,” 2003

Are the results RIGHT?

- If the code runs correctly, how would I know?
  - Can be done even when the requirement is not completely known. --- Testing also helps clarify the requirement.

Boundary conditions

- Most bugs live at the edge.
- Cases to consider:
  - Random input
  - Badly formatted data
  - Empty data
  - Values out of normal ranges
  - Duplications
  - Ordered lists that aren't, and vise-versa
  - Things our of order

Boundary conditions: practice

- Practice with TopK

Inverse relationship (1)

- Use inverse operation to check

```
public void testSquareRoot() {

  double x = mySquareRoot(4.0)
  assertEquals(4.0, x * x, 0.00001);
}
```

Inverse relationship (2)

- Use inverse operation to check

```
public void testInsert() {
  list.insert(100);                    // we insert item

  List.ListNode node = list.head;   // to check, we look for it
  int i;

  for(i=0; i<list.length; i++)
    if(node.item == 100)
      break;
  assertTrue(i!=list.length);
}
```

Cross-check

- Check you method with other means
  - Use another (slower) method to check result, e.g., for sorting algorithms.
  - Check that the aggregate characteristic is correct.

Forcing error conditions

- From other parts of program
  - Practice: TopK
- From environment
  - Out of memory
  - Out of disks
  - Clock?
  - Network errors
  - System load
  - Limited color palette
  - Video resolutions

Performance characteristics

- Fast enough?
- Use `Timer`.

- In JUnit, can add
  - @Test (timeout=10) public void xxxx

Test cases

- Specific / general ?
  - specific: acctNum = 1234
  - general: acctNum >= 0
- Weak assertion / strong assertion ?
  - Weak: getBalance(acctNum) >= MIN_BALANCE
  - Strong: getBalance(acctNum) = 12.50

Weak assertion

- An assertion is weak if it can evaluate to true even if the aspect of the implementation that it's testing is incorrect
- WA == false -> bug
- bug !-> WA == false

Strong assertion

- An assertion is strong if it will evaluate to true *if and only if* the aspect of the implementation that it's testing is correct
- SA == false -> bug
- bug -> SA == false

Strong!=correct

- Since each strong assertion checks only one aspect of the implementation.

```
{
     Bank b
     bank.totalDeposits() == 10000000
     bank.getBalance(1234) == 500
}
     bank.deposit(1234,500)
{
     bank.totalDeposite() == 10000500
     bank.getBalance(1234) == 1000
}
```

  - There is no guarantee that there will be no side effects

Class invariants

- A class invariant is a property that is true of all objects of a given class before and after each public method calls
  - SortedIntList
    - array is sorted
    - eltCount <= maxsize
  - Employee class
    - hourlySalary >= MIN_WAGE
    - getManager() != null
- Class invariants are a *cheap* and *powerful* testing tools, but rarely used in manual unit testing.

Bottom lines

- Unit testing is not easy
- Testing effort
  - 3-4 lines of test code for every one line of code to get 90-100% coverage
- Usually, consider only normal execution path.
- Automation?

Testing dependent classes

- Each unit usually interacts with other units
- Techniques
  - using stub
  - using mock objects

Using stub

- If your code calls `System.currentTimeMillis()`, and this return value is crucial to your testing:
  - Encapsulate this call.

```
public long getTime() {
    return System.currentTimeMillis();
}
```

  - Add stub

```
public long getTime() {
   if(debug)
      retirm debug_cur_time;
   else
      return System.currentTimeMillis();
}
```

  - Quite messy!

Using mock objects

- Use an interface to describe the object
- Implement the interface for production code
- Implement the interface in a mock object for testing

- With mock objects, you can do interaction-based testing

Mock objects: example

- Interface

```
public interface Environmental {
    public long getTime();
}
```

- Real implementation

```
public class SystemEnv implements Environmental {
    public long getTime() {
        return System.getTimeMillis();
    }
}
```

- Mock implementation

```
public class MockSystemEnv implements Environmental {
    public long getTime() {
        return current_time;
    }
    public void setTime(long t) {
        current_time = t;
    }
    private long current_time;
}
```

State-Based Testing & Interaction-Based Testing

- What we have done so far could be called "**state-based testing**."
  - We inject inputs into the objects, and see if their states change accordingly.
  - If there is no state change in the objects, it is difficult to use state-based testing.

- **Interaction-based testing** looks at how the objects interact.

Further reading: Martin Fowler's article "*Mocks Aren't Stubs*," and Nat Pryce's article "*State vs. Interaction Based Testing*". Google it.

Mock Libraries

- EasyMock
  - Create mock objects by “record-and-playback”
  - Easy to use
- jMock
  - Create mock by specifying how it interacts

Easy Mock

- **Easy Mock** is a tool that let you create a mock object and specify how it interacts using a record-and-replay approach.

- Eliminate the need to write a concrete class.

Easy Mock: Steps

- Record:
  - Create a mock object
  - Record the interaction, specify the return values
  - Press “replay”.
- Replay:
  - Run the test
  - The mock object would act as recorded.
  - In every step, it would verify all the interactions, i.e., all the calls.

More on Easy Mock

- See the demo.

- http://www.easymock.org/
- Document: http://www.easymock.org/EasyMock2_2_Documentation.html

Note for JUnit

- Testing in a project
  - Declare members as *protected* so that testcases in the same package can see it.
  - If we want to place the testcases in another directory, we can duplicates the program package directory structure so that the testcases are still in the same package.

Design for test

- Testing force you to reorganize your design
- (More on this later)

Conclusion

- Unit testing is important
  - Mainly a partial correctness assertion
    - Weak assertion / strong assertion
- Good test:
  - RIGHT-BICEP
- Unit testing dependent systems
  - Use stub and mock object