Abram Hindle
Department of Computing Science
University of Alberta

Testing

Slides originally by Ken Wong

Goal

- **Does program P obey specification S?**
  - **what is P?**
  - **what is S?**

Approaches

- **Reasoning about the state model for P:**
  - **typically a huge number of states**
  - **every practical technique must be inaccurate**
  - **could *abstract* states**
  - **could *sample* states**
  - **or both**

Approaches

- **Abstraction:**
  - **often used in static software analysis techniques**
    - **e.g., model checking P for some specific S**
  - **techniques often pessimistically inaccurate**
    - **may report P is faulty when P is correct**

Approaches

- **Sampling:**
  - **often used in dynamic analysis techniques**
    - **e.g., testing, profiling**
  - **techniques often optimistically inaccurate**
    - **may report P is correct when P is faulty**
    - **testing drives P through a sampling of states, but the samples may not generalize to actual situations**

State-Based Testing

- **Steps:**
  - **set up software into a known state**
    - **e.g., initialize variables**
  - **trigger transitions to cause state changes**
    - **e.g., call methods to change variables**
  - **verify the actual arrived state is expected**
    - **e.g., set if actual values in variables meet expectations**

Software Defects

- **Some terms:**
  - **human *errors* can lead to *faults* in work products, which may cause *failures* when running the software**
  - **can try to find faults through *testing*, reviews, proof, model checking, code analysis, etc.**
  - **some avoid the term *bug*, since it implies something wandered into the code**

Failure

- **AT&T failure (1990):**
  - **114 switching nodes of their long distance system crashed**
  - **the outage lasted for 9 h,
    70 million calls went uncompleted**

- **Reason:**
  - **if a node crashes, it tells neighboring nodes to reroute traffic around it**
  - **a bug in handling this message caused the receiving node to also crash, etc.**

Fault in Code

- **Root cause:**

```
do {
    switch (…) {
    case …:
        if (…) {
            …
            break;
        } else {
            …
        }
    …
  }
} while (…);
```

after expensive testing phase, a small change was made without again retesting

Examples of Defects

- **Actual behavior differing from expected:**
  - **algorithmic**
    - **code logic does not produce the proper output**
  - **overload**
    - **data structure unexpectedly filled to capacity**
  - **performance**
    - **violates service level agreement**
  - **accuracy**
    - **calculated result not to the desired level of accuracy**
  - **timing**
    - **race condition in coordinating concurrent processes**

Why Test?

- **Goals:**
  - **verification**
    - **check that requirements are satisfied**
  - **not only to *confirm* normal behavior**
    - **find problems to *refute* that the program is correct**
  - **establish due diligence**
    - **evidence in case of product liability litigation**
  - **avoid regression**
    - **prevent previous problems from reoccurring**

Regression Testing

- **Goal:**
  - **to avoid breaking things that should work**
    - **collect, reuse, and re-run automated test cases**
  - **do regression test after a change or fix**
    - **re-run tests to check whether previously passing tests of the system now fail**
    - **e.g., old defect somehow became unfixed**

Limits of Testing

- **Issues:**
  - **a program cannot be tested completely**
    - **too many inputs and path combinations to cover**
  - **testing cannot find all defects**
    - **cannot show their absence, just their presence**
  - **challenging**
    - **testing may be expensive and frustrating**
    - **test code itself could add its own defects**

Black Box Testing

- **Example test cases:**
  - **be systematic about what to test,**
    not knowing the internal code

Addends		**Sum**	**Description (also check commutative)**
2	3	5	something simple
99	99	198	large positive pair
99	-14	85	large positive plus negative
99	16	115	large positive plus positive
-99	-99	-198	large negative pair
-99	-14	-113	large negative plus negative
-99	16	-83	large negative plus positive
-99	99	0	large positive plus large negative
9	9	18	largest single digit positive pair
			...

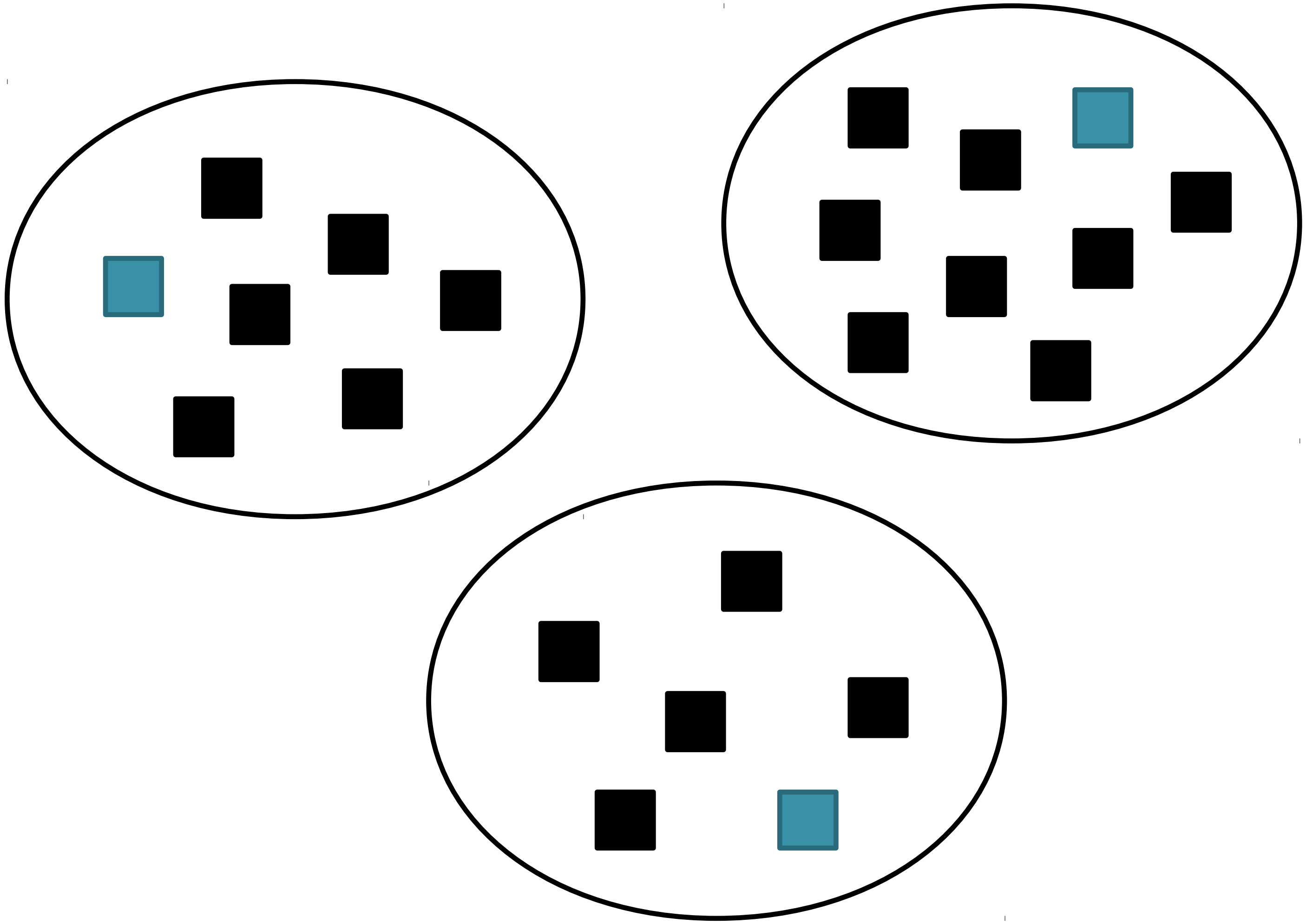
Black Box Testing

- **Tips:**
  - **avoid redundant tests**
    - **too easy to keep adding meaningless extra tests**
  - **determine *equivalence classes* of tests**

Black Box Testing

- **Equivalence classes:**
  - **each test inside an equivalence class checks the “same thing”**
  - **if a test inside the class will catch a defect, the other tests probably also will**
  - **if a test inside the class will not catch a defect, the other tests probably also will not**
  - **keep only a few tests in each class, as representatives**

partitioning of test cases



depiction of
equivalence classes

Black Box Testing

- **Example test cases:**
  - **guessing at internal algorithm or representation**

Addends		Sum	Description (also check commutative)
0	0	0	all zero special case
0	23	23	zero plus positive
-78	0	-78	negative plus zero
127	127	254	max signed bytes
-128	127	-1	min and max signed bytes
-128	-128	-256	min signed bytes
2147483647	2147483647		max signed integers
-2147483648	2147483647	-1	min and max signed integers
-2147483648	-2147483648		min signed integers
			...

Black Box Testing

- **Example test cases:**
  - **data input from fields in user interface**

Addends		Sum	Description (also check commutative)
4/3	2		expression
$2	$2		currency symbols
+5	3		plus sign
(9)	9		parentheses around negatives
l	1		lower case letter l
O	0		upper case letter O
<tab>	<tab>		no input
1.2	5		decimal
A	b		invalid characters

Black Box Testing

- **Example test cases:**
  - **and even more user interface explorations**
    - **editing with delete, backspace, cursor keys, etc.**
    - **using F1, escape, and control characters**
    - **vary timing of data entry**

Defect Tracking

- **Typically, for each reported defect:**
  - **identification**
    - **ID**
    - **program and version**
  - **classification**
    - **kind of defect (e.g., code or documentation)**
    - **severity (e.g., minor, major, critical)**
  - **description**
    - **issue**
    - **how to reproduce**
    - **suggested fix (optional)**

Defect Tracking

- **For each reported defect:**
  - **progress**
    - **status (open or closed)**
    - **resolution (e.g., pending, fixed, irreproducible, deferred, as designed, unfixable)**
  - **involved person**
    - **reported by and when**
    - **assigned to and when**
    - **resolved by and when**
    - **verified by and when**

Testing Strategies

- **Big-bang strategy:**
  - **test thoroughly only after the whole system is put together**
  - **pro?**
    - **"project almost finished, only testing left"**
  - **cons**
    - **hard to pinpoint the cause of a failure**

Testing Strategies

- **Top-down incremental strategy:**
  - **implement/test the highest-level modules first**
    - **provide stubs for lower-level functionality not yet implemented**
    - **higher-level modules are the test drivers**

- **Bottom-up incremental strategy:**
  - **implement/test the lowest-level modules first**
    - **need to write test drivers**

Testing Techniques

- **Creating good tests:**
  - **test every error message**
    - **error-handling code tends to be weaker**
  - **test under other configurations**
    - **programmers are biased to their own setup**

Design for Testing

Good Software Design

- **Want software to be flexible:**
  - **easy to change to respond to new needs**
  - **easy to understand**
  - **easy to extend, without exploding complexity**
- **Want software to be testable:**
  - **easy to construct the units**
  - **easy to set up units into desired state**
  - **easy to drive code and witness effects**

Example Bad Design 1

```
/**
 * Process photo album requests,
 * parse user preferences,
 * apply image transformations,
 * assemble images into albums,
 * deliver results to users
 */

public class PhotoAlbumServer {

    … // lots of code

}
```

Example Bad Design 1

- **Poor flexibility:**
  - **difficult to extract and reuse parts**
  - **complex to add new features**
  - **instance variables are "global"**

- **Poor testability:**
  - **only end-to-end testing possible**
  - **need golden results files for every combination of preference settings and image transformations**

Improved Design 1

- **Use separation of concerns:**
  - **RequestHandler class**
  - **UserPreferencesReader class**
  - **UserPreferencesParser cass**
  - **ImageEffect class**
  - **ImageTransformer class**
  - **…**

Improved Design 1

- **Better flexibility:**
  - **uses object-oriented design**
  - **easier to understand smaller, separate units**

- **Better testability:**
  - **more focused tests of each unit**
  - **test fixtures easier to provide for each unit**
  - **easier to check results**

Forming Dependencies

```
public class ExampleService {
    private DataSource theDataSource;
    …

    public ExampleService( … ) {
        theDataSource = new DataSource( … );
        …
    }

    public void doService() {
        …
        … = theDataSource.getInfo();
        …
    }
    …
}
```

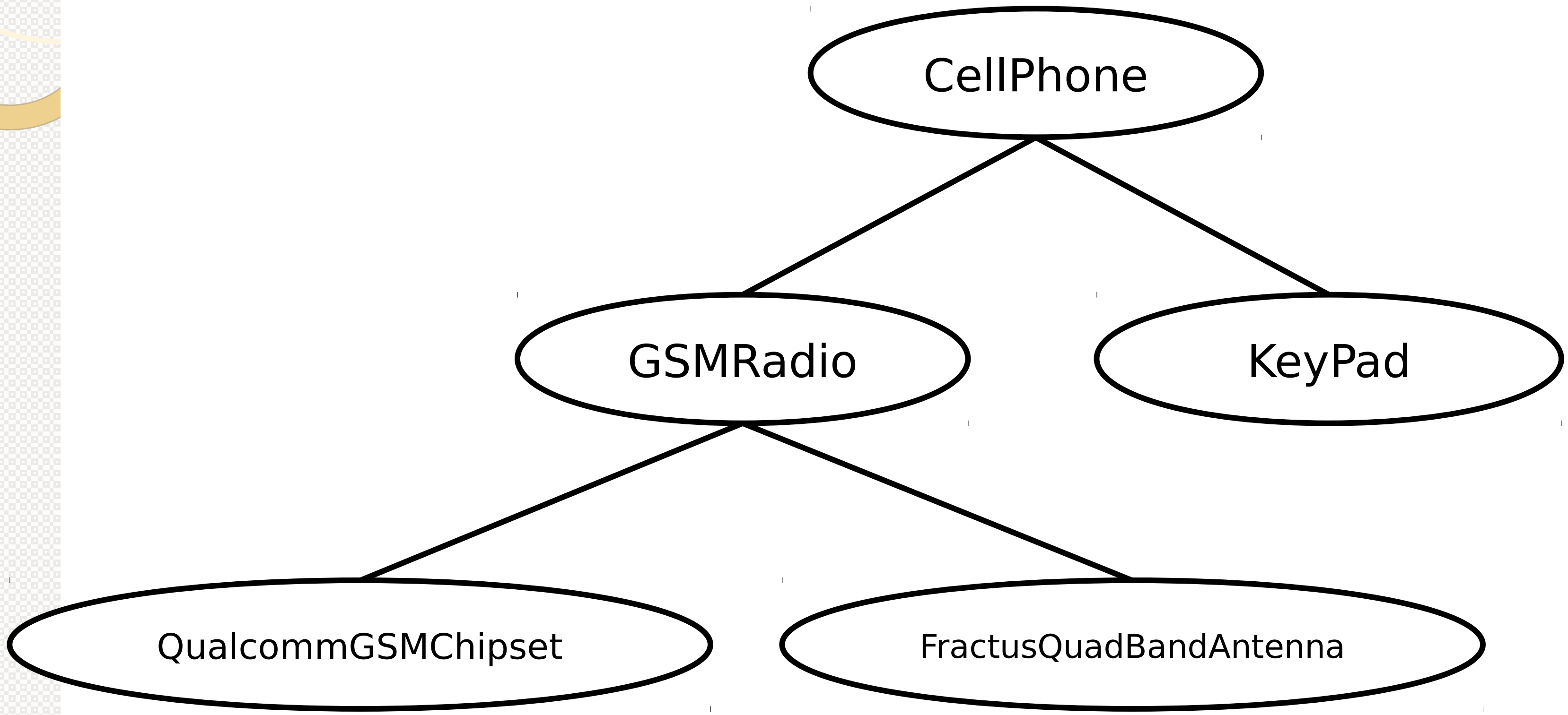
one approach is that the class makes what it depends on

"Dependency Injection"

```
public class ExampleService {
    private DataSource theDataSource;
    …

    public ExampleService(
        DataSource aDataSource ) {

        theDataSource = aDataSource;
        …
    }

    public void doService() {
        …
        … = theDataSource.getInfo();
        …
    }
    …
}
```

alternatively, construct what this class depends on outside the class

System Assembly



System Assembly without DI

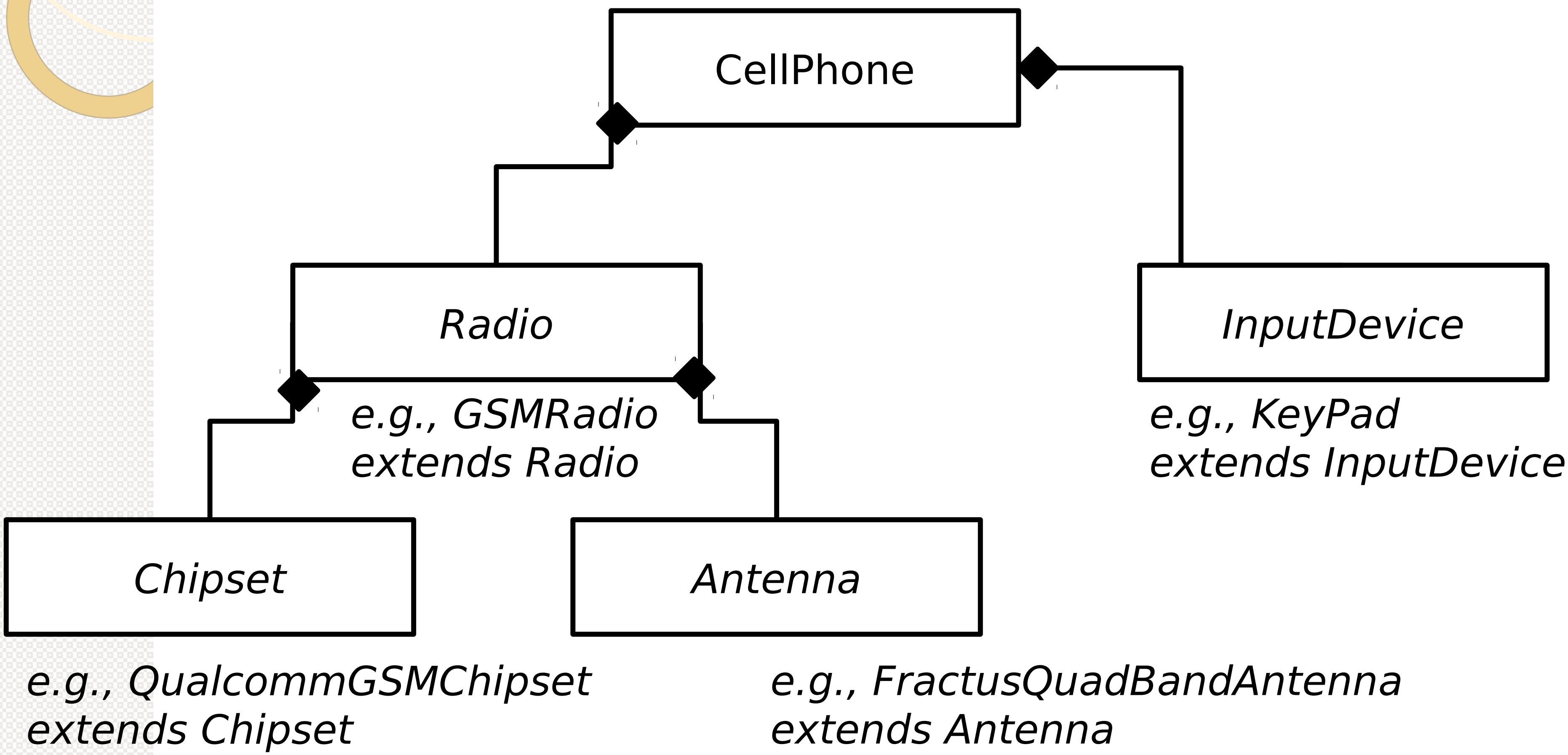
- ```
  public class CellPhone {
      …
      public CellPhone() {
          radio = new GSMRadio();
          inputDevice = new KeyPad();
          …
      }
  }
  ```
- ```
  public class GSMRadio {
      …
      public GSMRadio() {
          chipset = new QualcommGSMChipset();
          antenna = new FractusQuadBandAntenna();
      }
  }
  ```
- ```
  CellPhone phone = new CellPhone();
  // fully assembled
  ```
  ```

System Assembly without DI

- **Poor flexibility:**
  - **difficult to change and plug in parts**
    - **for different radio, different input device, etc.**

- **Poor testability:**
  - **can’t supply test versions of parts**
    - **stuck with given parts**
  - **entire aggregate is constructed**
    - **could be expensive**

System Assembly with DI

```
public class CellPhone {
    …
    public CellPhone( Radio radio,
        InputDevice inputDevice ) {

        this.radio = radio;
        this.inputDevice = inputDevice;
    }
    …
}
```

```
public class GSMRadio {
    …
    public GSMRadio( Chipset chipset,
        Antenna antenna ) {

        this.chipset = chipset;
        this.antenna = antenna;
    }
}
```

System Assembly with DI

- ```
  // in some high-level class

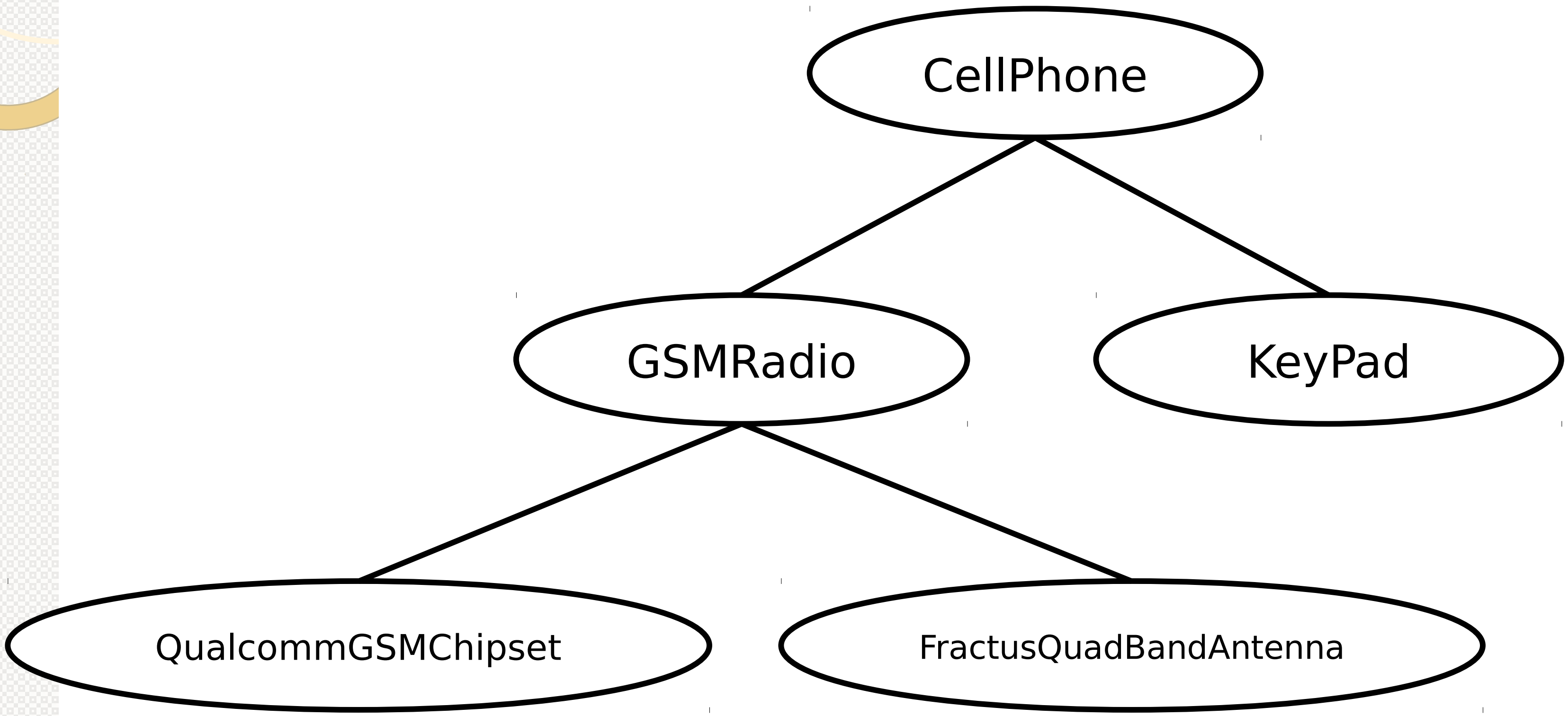
  CellPhone phone = new CellPhone(
      new GSMRadio(
          new QualcommGSMChipset(),
          new FractusQuadBandAntenna()
      ),
      new KeyPad()
  );
  ```

separates out "dependency resolution" from the constituent classes
  ```

System Assembly with DI



could have other subclasses beyond these examples

System Assembly with DI



the bottom-up assembly process instantiates the children and inserts them into the parents

Example Bad Design 2

```
public class User {
    private Preferences prefs;

    public User( File prefFile ) {
        prefs = parseFile( prefFile );
        …
    }
    public void doSomething() {
        … // use prefs
    }
    …
    private Preferences parseFile( File prefFile ) {
        …
        aPrefs = new Preferences( … );
        … // setup prefs
        return aPrefs;
    }
}
```

Example Bad Design 2

- **Poor flexibility:**
  - **changing preferences requires changing User**
    - **file format changes**
  - **difficult to reuse User**
    - **embedded preference file reading and parsing**

- **Poor testability:**
  - **tests that deal with files are slow**
  - **need test file for each preference combination**

Improved Design 2

```
class User {
    private Preferences prefs;

    public User( Preferences prefs ) {
        this.prefs = prefs;
        …
    }
    public void doSomething() {
        … // use prefs
    }
    …
}
```

dependency injection

Improved Design 2

- **Better flexibility:**
  - **no change to User if file format changes**
  - **preferences not limited to be made from files**

- **Better testability:**
  - **can run fast**
    - **pass in mock or fake Preferences object**

“Mock Object”

```
public class UserTest {
    …
    public void testdoSomething() {

        // MockPreferences extends Preferences,
        // but is overridden with canned settings
        // (no test preference file needed)

        MockPreferences mockPrefs =
            new MockPreferences();

        User aUser = new User( mockPrefs );

        aUser.doSomething();
        …

        mockPrefs.AssertNoChange();
    }
}
```

Example Bad Design 3

- **Situation:**
  - **many pieces of information are needed by classes throughout the system**
  - **but each class needs just one or a few items**
  - **how to provide this information to the consumers?**

Example Bad Design 3

- **Typical approaches:**
  - **consumers get the data they need …**
  - **make the data global,**
  - **pass around a context object, or**
  - **put the data in widely known and used classes**

Example Bad Design 3

```
public class Account {
    …
    public Account( User user ) {
        this.country =
user.getPreferences().getLocation().getCountry();
        …
    }
    …
}
```

Example Bad Design 3

- **Poor flexibility:**
  - **method parameters do not show what the method really needs**
  - **code “locks in” the structure it walks**

- **Poor testability:**
  - **test needs to recreate this structure …**

Example Bad Design 3

```
public void testSomethingForAccount() {
    // set up for test

    Country country = new Country( “Canada” );

    Location location = new Location();
    location.setCountry( country );

    Preferences prefs = new Preferences();
    prefs.setLocation( location );

    User user = new User( prefs );

    Account account = new Account( user );

    … // test Canadian account
}
```

test code should be simple (less likely to have defects)

Improved Design 3

-

```
public void testSomethingForAccount() {

    Country country = new Country( “Canada” );

    // redesigned constructor
    // (requires only what is needed)
    Account account = new Account( country );

    … // test Canadian account
}
```

Test-Driven Development

Automated Testing

- **Purpose:**
  - **write software to help test software**
    - **automation essential to test-driven development and refactoring**
- **Limitations:**
  - **manual testing still need to observe certain problems**
    - **e.g., strange noises from the speaker, flickering graphics**

Automated Testing

- **A good automated unit test:**
  - **is simple to write and understand**
    - **reduces the chance of defects in the test code**
  - **runs quickly**
    - **so it can be re-run frequently while developing**
  - **is isolated**
    - **could run multiple unit tests in parallel**
  - **shows exactly what went wrong if it fails**
    - **reduce time spent in a debugger**

Automated Testing

- **Quote:**
  - **"Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead."**

    — Martin Fowler

"Way of Testivus"

- **"Think of code and tests as one**
  - **When writing the code, think of the tests.**
    When writing the tests, think of the code.

    When you think of code and tests as one,
    testing is easy and the code is beautiful."

    — Alberto Savoia

"Way of Testivus"

- **"Best time to test is when the code is fresh**
  - **Your code is like clay.
When it's fresh, it's soft and malleable.
As it ages, it becomes hard and brittle.

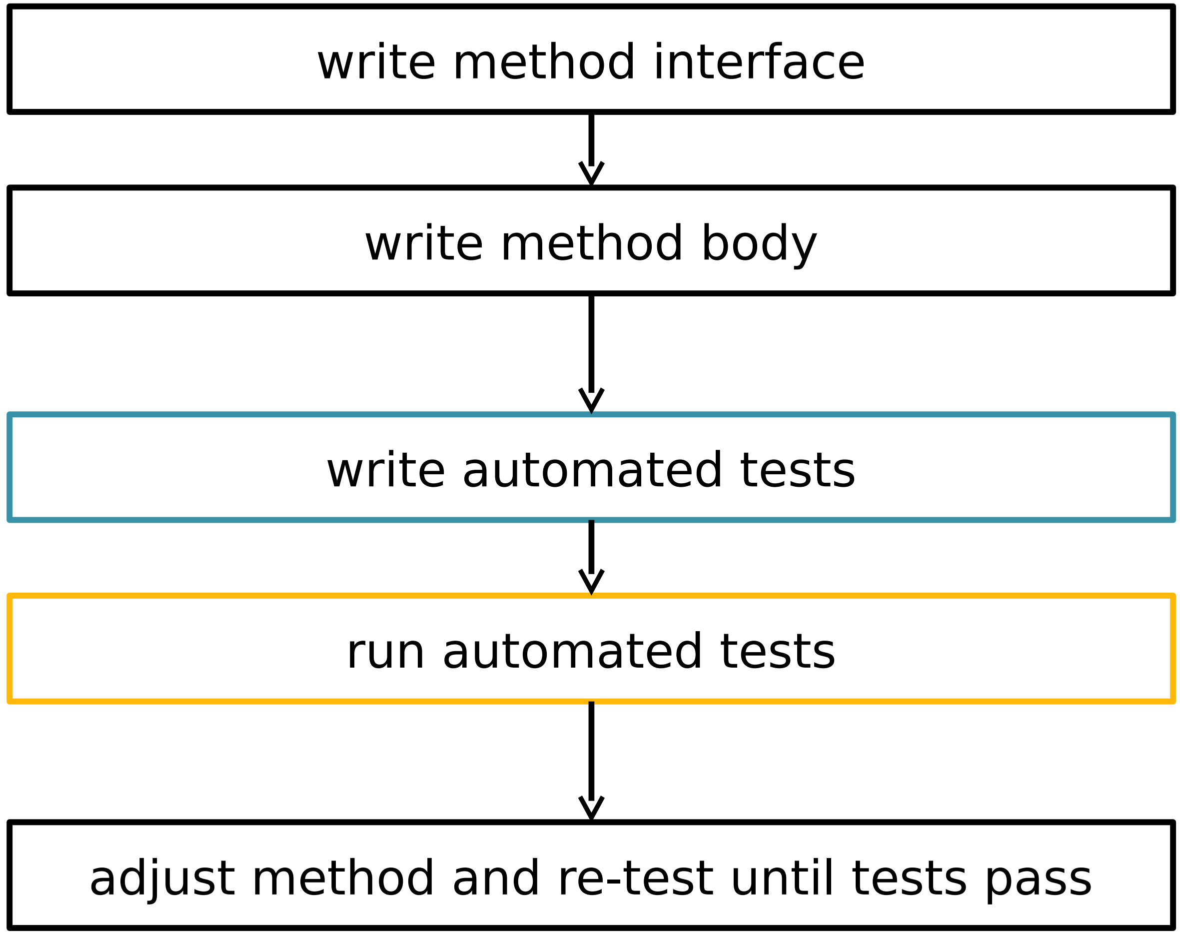
If you write tests when the code is fresh and easy to change, testing will be easy,
and both the code and the tests will be strong."

— Alberto Savoia**

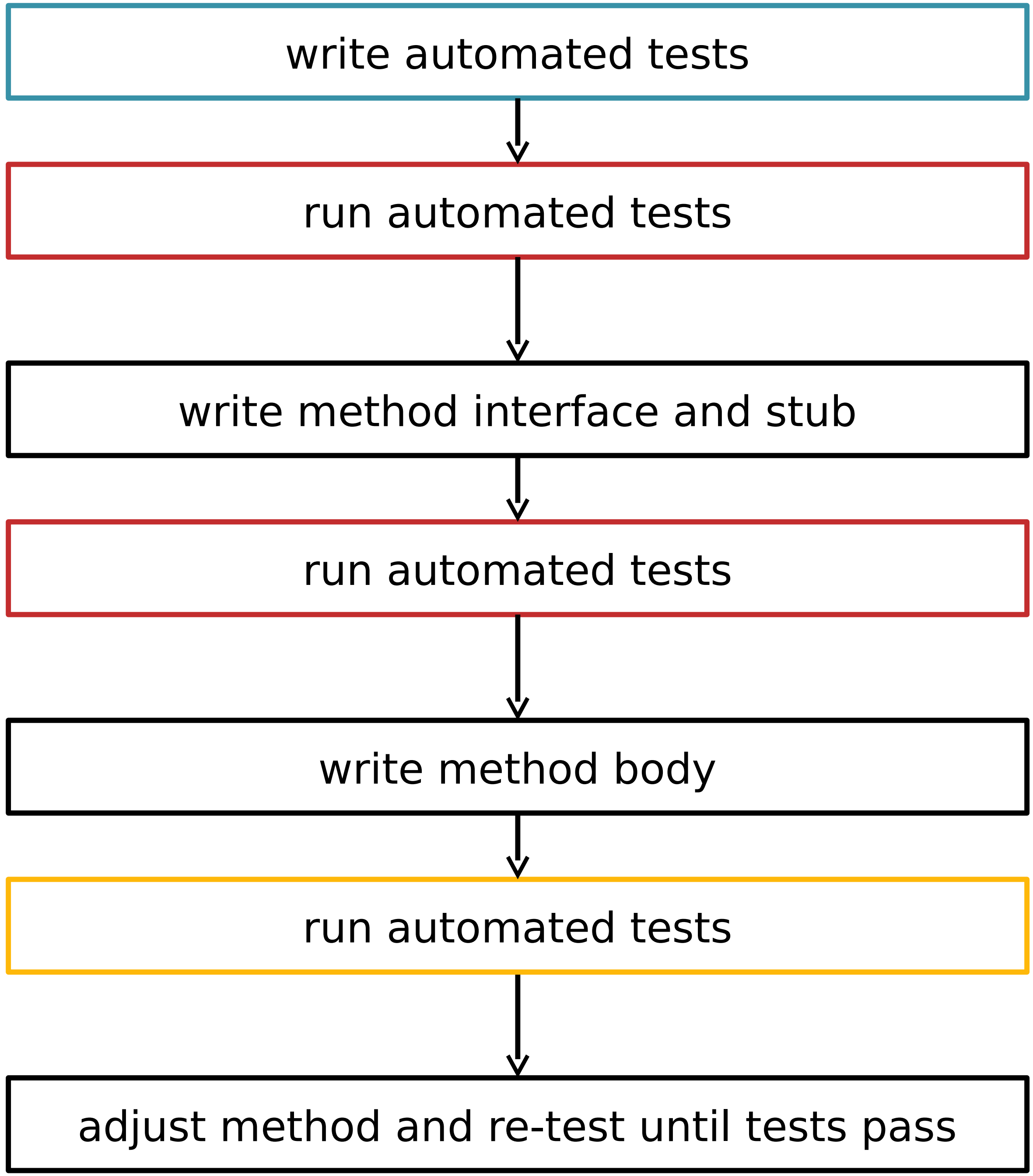
Test-Driven Development

- **Idea:**
  - **if testing is so useful, let's write the tests first**
  - **these automated tests capture *code-level requirements* to be satisfied**
  - **once code is written so that these tests pass, then these requirements are considered to be met**

traditional development

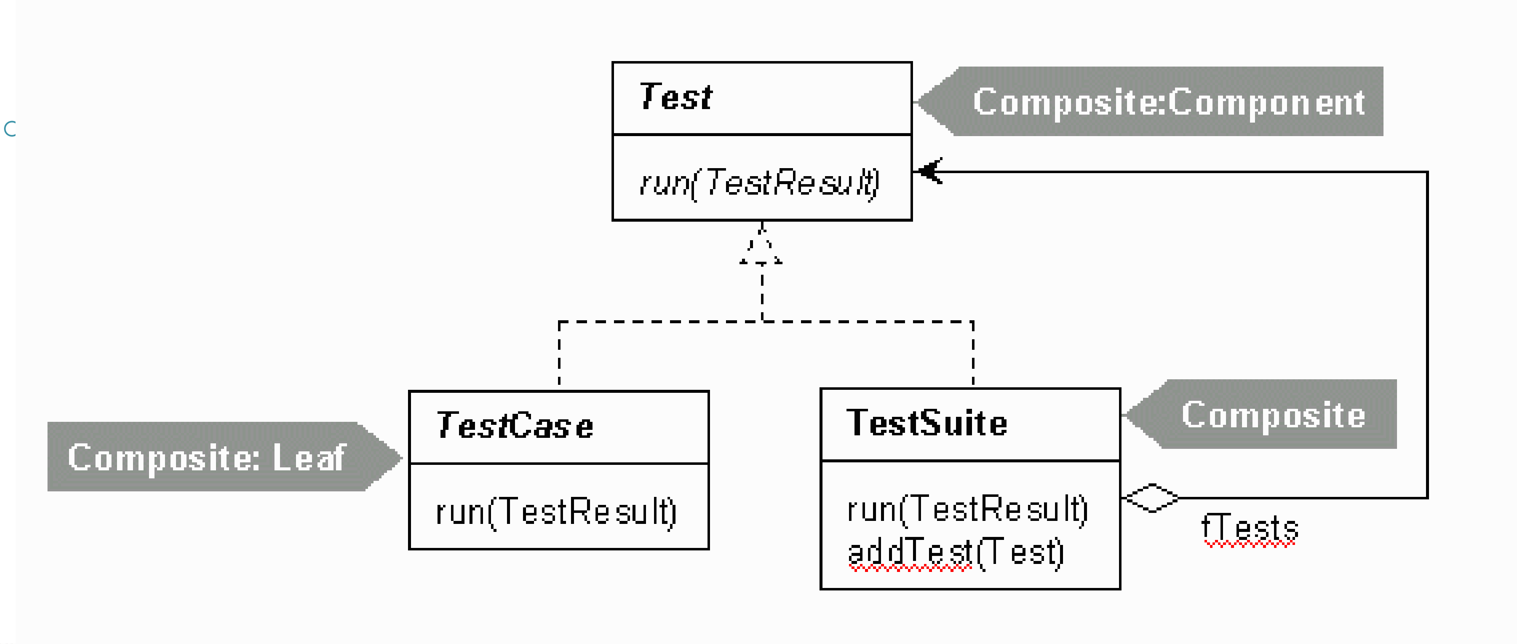


test-first or test-driven development



JUnit Framework

- **Usage:**
  - **for each class *Foo* to be tested, implement a subclass named *Foo*`Test` of `TestCase` in the same**



JUnit Framework

- ***Foo*`Test` class has:**
  - **test objects that may be used in the test methods**
  - **`setUp()` method to initialize the test objects (or *fixture*) before each test method is run**
  - **`tearDown()` method to clean up the fixture afterwards**

Each test method

is named `test`*`Something`*

may initialize more specific test objects

for the test objects, calls the method in *Foo* to be tested

checks the results against what is expected using assertion statements

JUnit Framework

- **Example test code:**

```
public class NumberTest extends TestCase {
    private Number aNumber;
    private Number anotherNumber;

    protected void setUp() {
        aNumber = new Number(2);
        anotherNumber = new Number(3);
    }

    // check that value-based equality works
    public void testEquals() {
        Assert.assertTrue(!aNumber.equals(null));
        Assert.assertEquals(aNumber, aNumber);
        Assert.assertEquals(aNumber, new Number(2));
        Assert.assertTrue(!aNumber.equals(anotherNumber));
    }
```

JUnit Framework

- **Example test code:**

```
    public void testAdd() {
        // more test data
        Number expected = new Number(5);
        // test Number.add method
        Number result = aNumber.add(anotherNumber);
        // check the result
        Assert.assertTrue(expected.equals(result));
    }

    …
}
```

- **Assert static methods:**
  - http://junit.sourceforge.net/javadoc/junit/framework/Assert.html

In the Application

- **Example functional code:**

```
public class Number {
    private int value;

    public boolean equals(Object anObject) {
        if (anObject instanceof Number) {
            Number aNumber = (Number)anObject;
            return aNumber.value == this.value;
        }
        return false;
    }
    …
}
```

In the Application

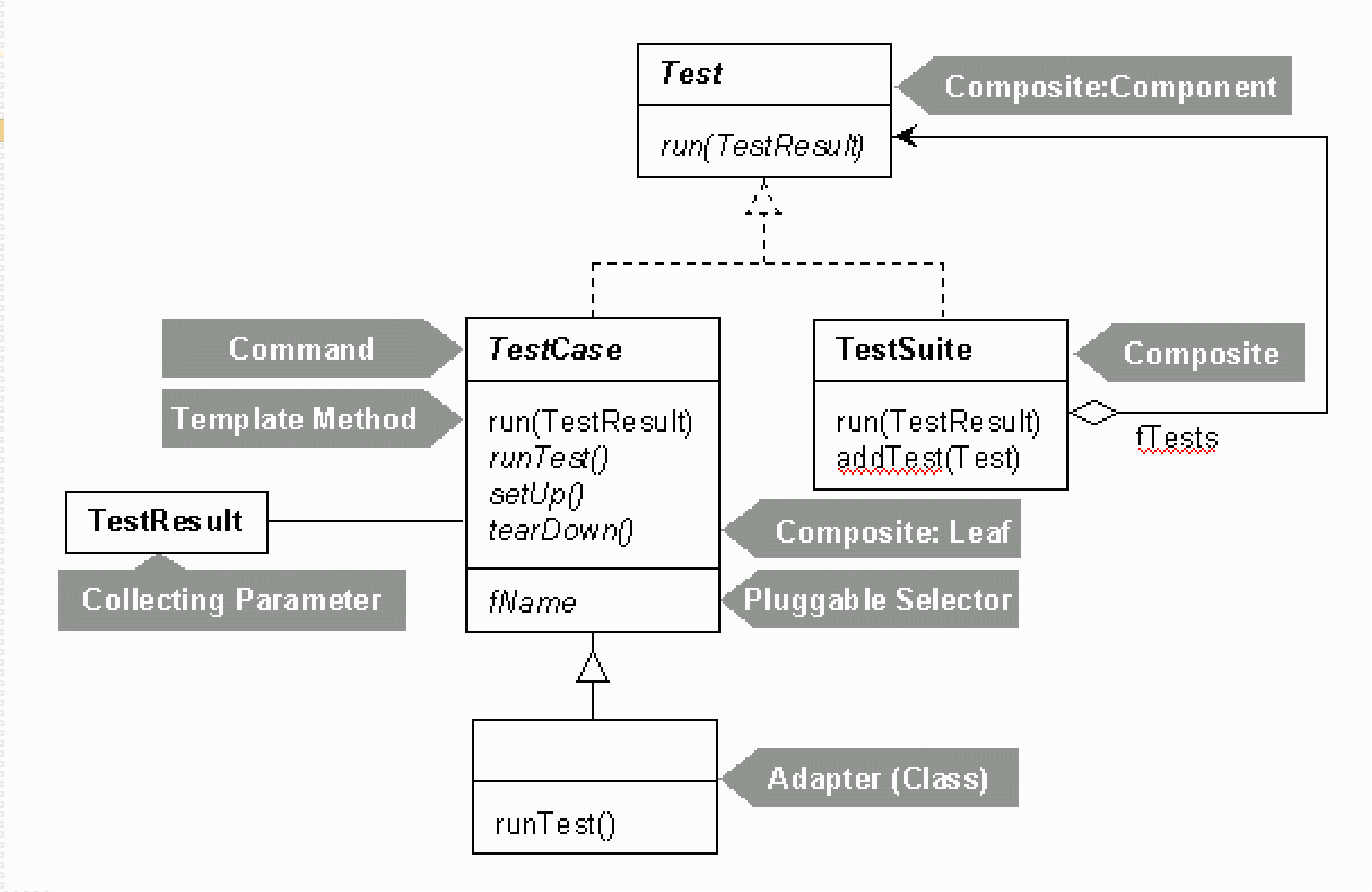
- **Issue:**
  - **What methods should be tested with JUnit?**

Approach:

write JUnit tests for methods of the application model that have side effects (i.e., not getter methods)

use assertions on the output of getter methods to check that constructors and setter methods worked properly

JUnit Framework



More Information

- **Books:**
  - **Test-Driven Development**
    - **K. Beck**
    - **Addison-Wesley, 2003**

More Information

- **Books:**
  - **Testing Computer Software**
    - **C. Kaner, J. Falk, H. Q. Nguyen**
    - **Wiley, 1999**
  - **Lessons Learned in Software Testing**
    - **C. Kaner, J. Bach, B. Pettichord**
    - **Wiley, 2002**

More Information

- **Links:**
  - **Cause of AT&T Network Failure**
    - **http://catless.ncl.ac.uk/Risks/9.62.html#subj2**
  - **History's Worst Software Bugs**
    - **http://www.wired.com/software/coolapps/news/2005/11/69355**

More Information

- **Links:**
  - **Flexible Design? Testable Design? You Don’t Have to Choose!**
    - **R. Rufer and T. Bialik**
  - **The Way of Testivus**
    - **http://www.agitar.com/downloads/TheWayOfTestivus.pdf**
  - **JUnit Resources for Test-Driven Development**
    - **http://www.junit.org/**

More Information

- **JUnit framework**
  - `http://www.junit.org/`
- **Test Infected**
  - `http://junit.sourceforge.net/doc/testinfected/testing.htm`