Software Testing

Lecture 19

Why does software fail?

Bugs !!!

A more complete answer includes the famous triplet:

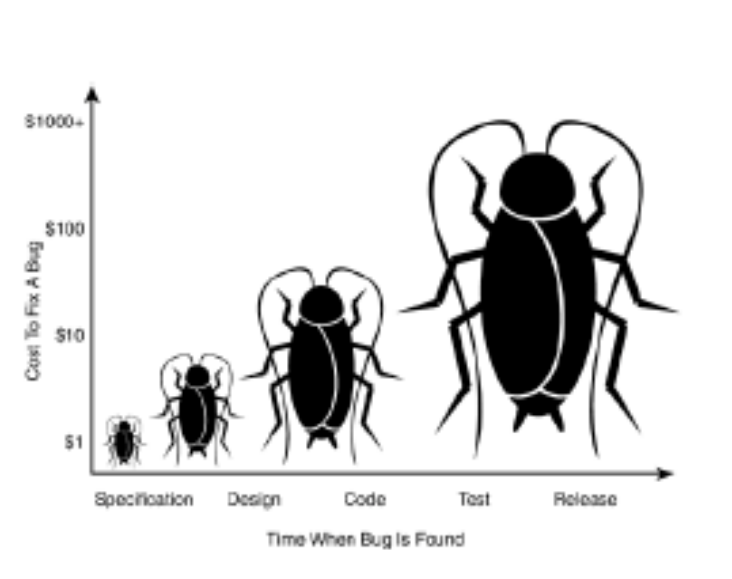
1. Errors
2. Faults
3. Failures

What do we mean by error?

Error

People make errors. A good synonym is mistake.

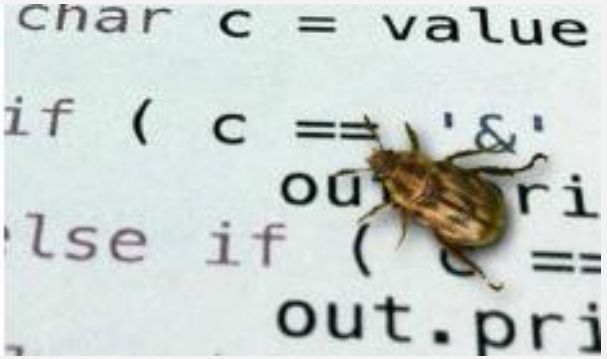
Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.



What do we mean by Fault?

Fault

A fault is the result of an error.



It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, UML diagrams, hierarchy charts, and source code.

Defect is a good synonym for fault, as is **bug**.

What do we mean by Failure?

Failure

A failure **occurs** when the code corresponding to a fault executes.

In general a failure is the manifestation of a fault.

What is software Testing?

Testing is the act of exercising software with **test cases**.

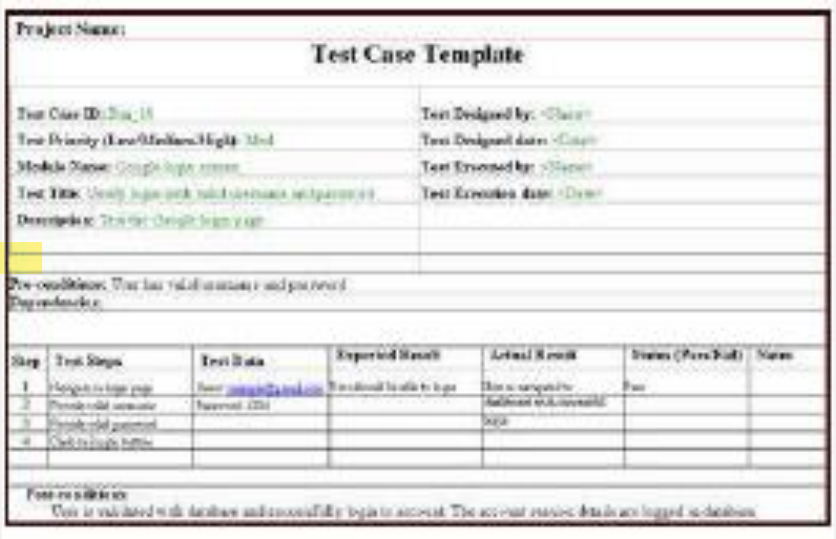
A test has two distinct goals:

1. To find failures (**verification** aspect).
2. To demonstrate correct execution (**validation** aspect).

What is software Test Case?

The essence of software testing is to determine
a set of test cases for the item to be tested.

A **test case** is (or should be) a recognized work product.


Project Name:

Test Case Template

Step	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1						
2						
3						
4						

A **complete test case** will contain

1. a test case identifier,
2. a brief statement of purpose,
3. a description of preconditions,
4. the actual test case inputs,
5. the expected outputs,
6. a description of expected post-conditions, and
7. an execution history.

Which are the targets of testing?

The target of the test can vary:

A single module, a group of such modules (related by purpose, use, behavior, or structure), or an entire system.

Three test stages can be distinguished:

1. unit,
2. integration, and
3. system.

Observations

It is impossible to completely test any nontrivial module or system

- Practical limitations: Complete testing is prohibitive in time and cost
- Theoretical limitations: e.g. Halting problem

"Testing can only show the presence of bugs, not their absence" (Dijkstra).

Testing takes creativity

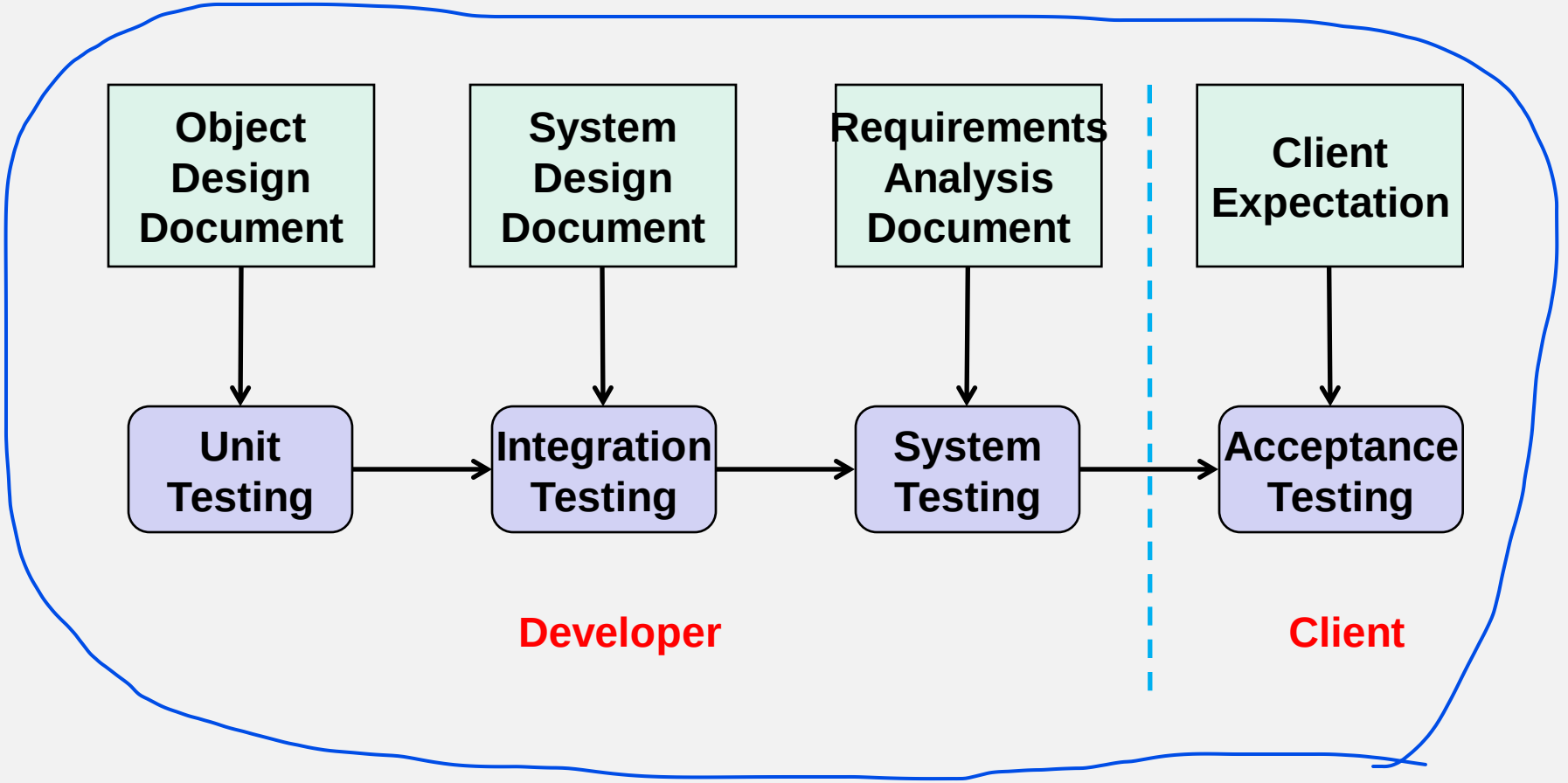
To develop an effective test, one must have:

- Detailed understanding of the system
- Application and solution domain knowledge
- Knowledge of the testing techniques
- Skill to apply these techniques

Testing is done best by independent testers

- We often develop a certain mental attitude that the program should in a certain way when in fact it does not
- Programmers often stick to the data set that makes the program work
- A program often does not work when tried by somebody else.

Testing Activities



Types of Testing

Unit Testing

- Individual component (class or subsystem)
- Carried out by developers
- Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality

Integration Testing

- Groups of subsystems (collection of subsystems) and eventually the entire system
- Carried out by developers
- Goal: Test the interfaces among the subsystems.

Types of Testing continued...

System Testing

- The entire system
- Carried out by developers
- Goal: Determine if the system meets the requirements (functional and nonfunctional)

Acceptance Testing

- Evaluates the system delivered by developers
- Carried out by the client. May involve executing typical transactions on site on a trial basis
- Goal: Demonstrate that the system meets the requirements and is ready to use.

What is unit testing?

Common to most conceptions of unit tests
is the idea that they are **tests in isolation**
of individual components of software.

What are components?

In unit testing, we are usually concerned with
the most atomic behavioral units of a system.

- In procedural code, the units are often functions.
- In object oriented code, the units are classes.

Why Testing in isolation?

Unit testing is important for:

Error localization

As tests get further from what they test, it is harder to determine what a test failure means. Often it takes considerable work to pinpoint the source of a test failure.

Execution time

Larger tests tend to take longer to execute. This tends to make test runs rather frustrating. Tests that take too long to run end up not being run.

What is a good unit Test?

Here are qualities of **good unit tests**:

- They run fast. If they don't run fast, they aren't unit tests.
- They help us localize problems.

A test is **not a unit test** if:

- It talks to a database.
- It communicates across a network.
- It touches the file system.
- You have to do special things to your environment (such as editing configuration files) to run it.....

When to do Unit Testing?

Static Testing (at compile time)

- Static Analysis
- Review
  - Walk-through (informal)
  - Code inspection (formal)

Dynamic Testing (at run time)

- Black-box testing
- White-box testing.

Static Analysis with Eclipse

Compiler Warnings and Errors

- *Possibly uninitialized Variable*
- *Undocumented empty block*
- *Assignment has no effect*

Checkstyle

- Check for code guideline violations

FindBugs

- Check for code anomalies

Metrics

- Check for structural anomalies

Black-box testing

Focus: I/O behavior

- If for any given input, we can predict the output, then the component passes the test
- Requires test oracle

Goal: Reduce number of test cases by equivalence partitioning:

Divide input conditions into equivalence classes

Choose test cases for each equivalence class.

Black-box testing: Test case selection

a) Input is valid across range of values

Developer selects test cases from 3 equivalence classes:

1. Below the range
2. Within the range
3. Above the range

b) Input is only valid, if it is a member of a discrete set

Developer selects test cases from 2 equivalence classes:

1. Valid discrete values
2. Invalid discrete values

No rules, only guidelines.

Black box testing: An example

```
public class MyCalendar {

   public int getNumDaysInMonth(int month, int year)
    throws InvalidMonthException
   { … }
}
```

Representation for month:
1: January, 2: February, …., 12: December

Representation for year:
1904, … 1999, 2000,…, 2006, …

How many test cases do we need for the black box testing of **`getNumDaysInMonth()`**?

White-box testing overview

Statement/Code coverage: Each statement executed at least once by some test case

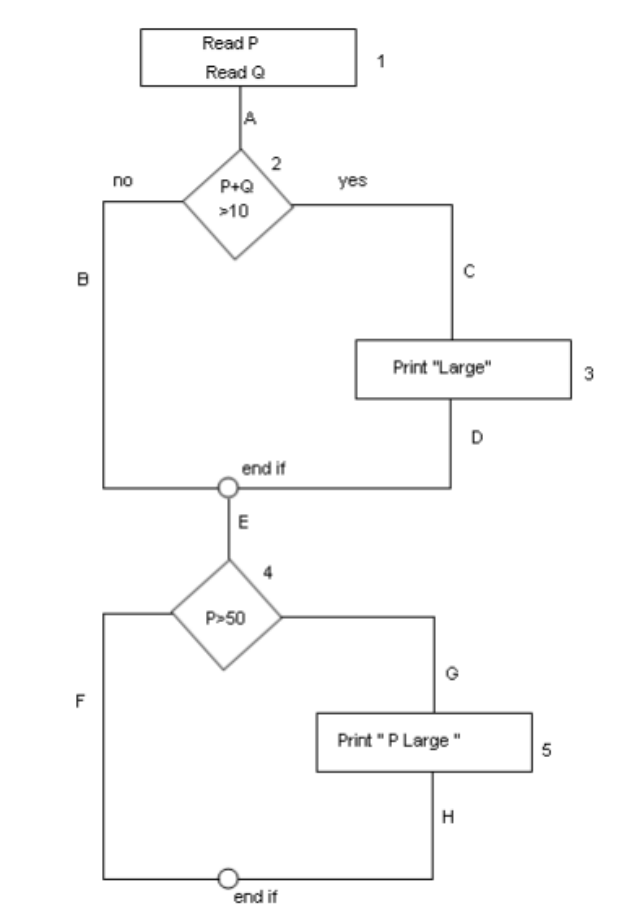
Branch coverage :Every edge (branch) of the control flow is traversed at least once by some test case

Condition coverage: Every condition takes TRUE and FALSE outcomes at least once in some test case

Path coverage: Finds the number of distinct paths through the program to be traversed at least once

White-box testing overview

Statement/Code coverage: Each statement executed at least once by some test case



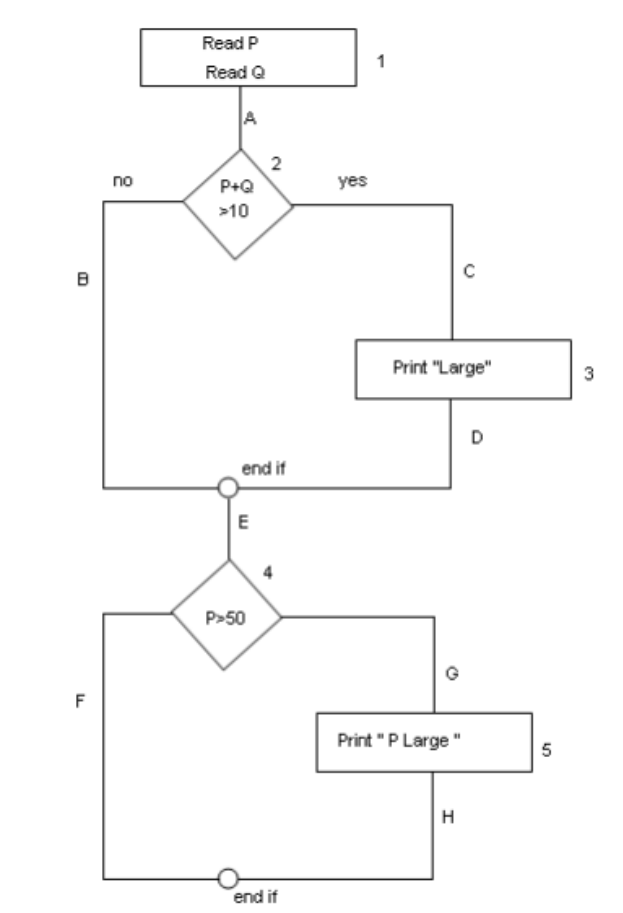
1A – 2C – 3D – E – 4G – 5H

All the nodes 12345 are covered

Statement coverage: 1

White-box testing overview

Constructing the control graph of a program for Branch Coverage/conditional coverage:

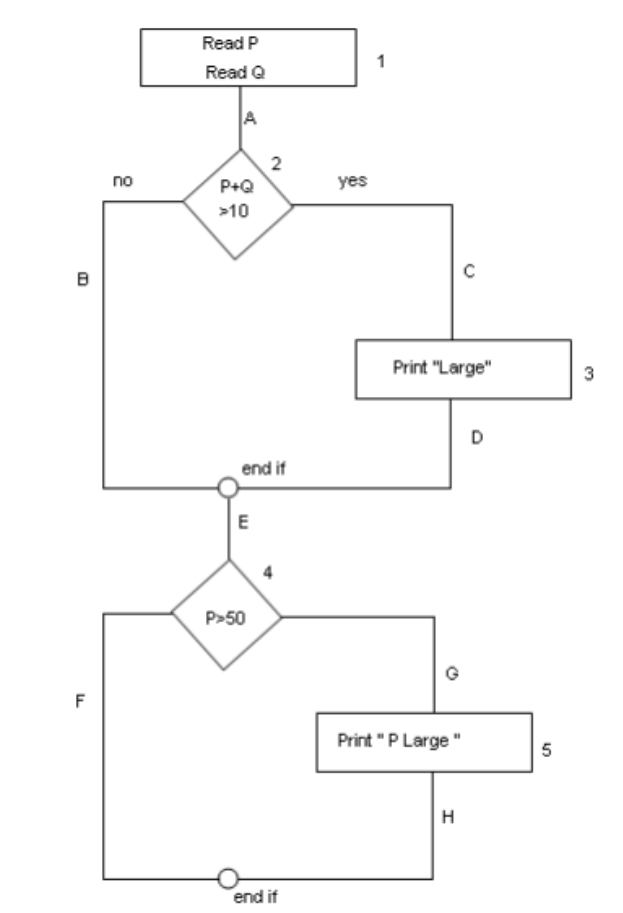


1A – 2C – 3D – E – 4G – 5H
1A-2B-E-4F

Branch Coverage: 2

White-box testing overview

Path coverage: Finds the number of distinct paths through the program to be traversed at least once



1A-2B-E-4F
1A-2B-E-4G-5H
1A – 2C – 3D – E – 4G – 5H
1A-2C-3D-E-4F

Path Coverage: 4

Unit Testing

To **put tests in place**, we often have to **change code** !!

Why ??

- **Dependency** is one of the most critical problems in software development. Often, classes depend directly on things that are hard to use in a test, they are hard to modify and hard to work with.
- Much legacy code work involves breaking dependencies so that change can be easier.

Unit Testing

Reasons for breaking dependencies

When we want to get tests in place, there are two reasons to break dependencies*:*

- **Sensing -** We break dependencies to *sense when we can't access values* our code computes.
- **Separation -** We break dependencies to *separate when we can't even get* a piece of code into a test harness to run.

- ***Test harness*** *is a generic term for the testing code that we* write to exercise some piece of software and the code that is needed to run it.

Faking Collaborator

Breaking dependencies on **other code** is hardly ever that simple.

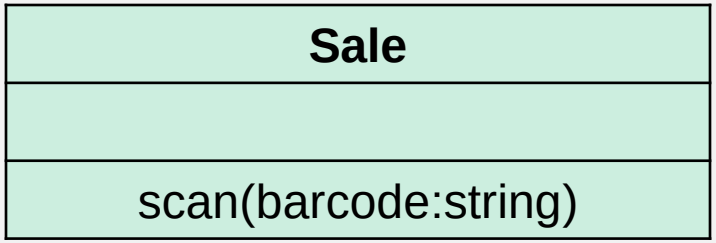
- Often that other code is the only place we can easily sense the effects of our actions.
- If we can put some other code in its place and test through it, we can write our tests.
- In object orientation, these other pieces of code are often called *fake objects*.

  - ***Stub*** *is another way to call these elements*

A fake object is an object that impersonates some collaborator of a class when it is being tested.

Faking Collaborator

In a point-of-sale system, we have a class called Sale. Whenever scan() is called, the Sale object needs to display the name of the item that was scanned, along with its price on a cash register display.

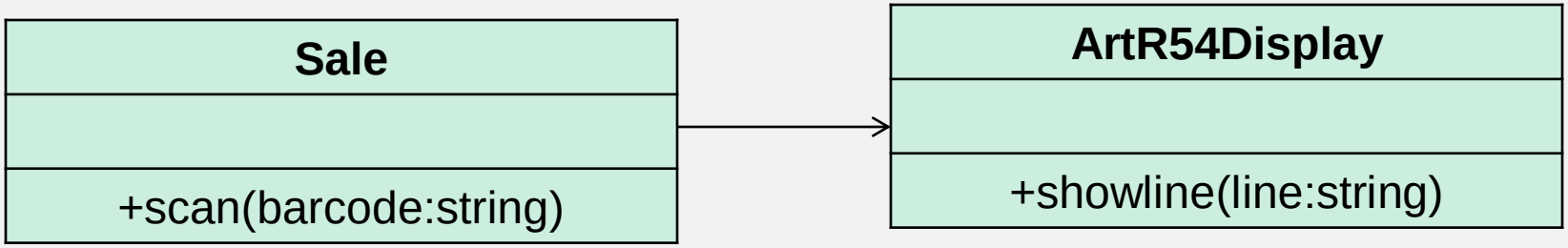


How can we test this to see if the right text shows up on the display?

If the calls to the cash register's **display API are buried** deep in the Sale class, it's going to be hard.

Faking Collaborator

We can move all of the display code from Sale over to ArtR56Display and have a system that does exactly the same thing that it did before.

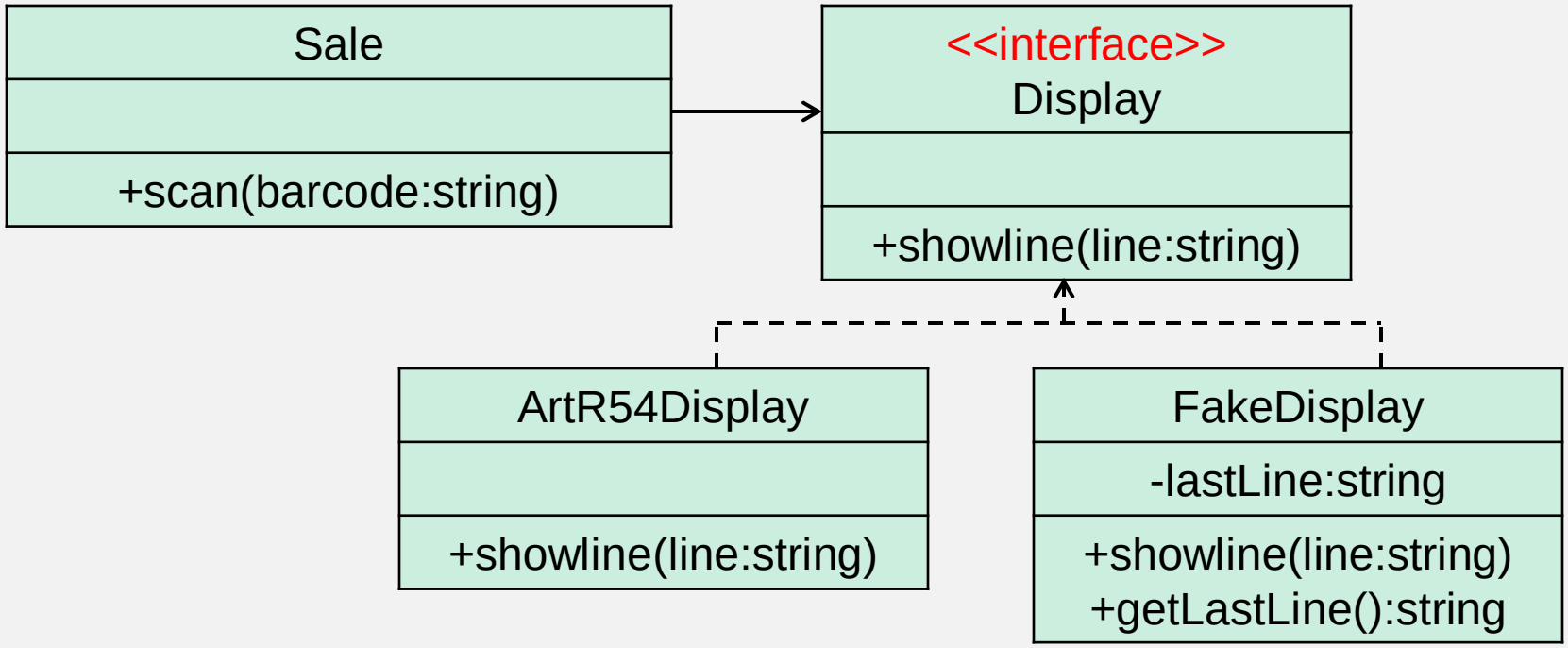


Does that get us anything?

Faking Collaborator

The Sale class can now hold on to either an ArtR56Display or something else, a FakeDisplay.

The nice thing **about having a fake display** is that we can write tests against it **to find out what the Sale does.**



Faking Collaborator

```
Public interface Display
{showLine (String line);}
```

Class to be tested

```
Public class Sale{
  private Display dispaly;
  public Sale(Display display){
     this.display=display;
   }
  public void scan(String barcode){
     ….
      String itemLine = item.name()
         +" "+item.price().asDisplayText();
       display.showLine(itemLine);
     …….
   }
}
```

Faking Collaborator

Fake Object

```
Public class FakeDisplay implements Display{
  private String lastLine="";
  public showLine(String line){
     lastLine=line;
   }
  public String getLastLine(){
     return lastLine;
   }
}
```

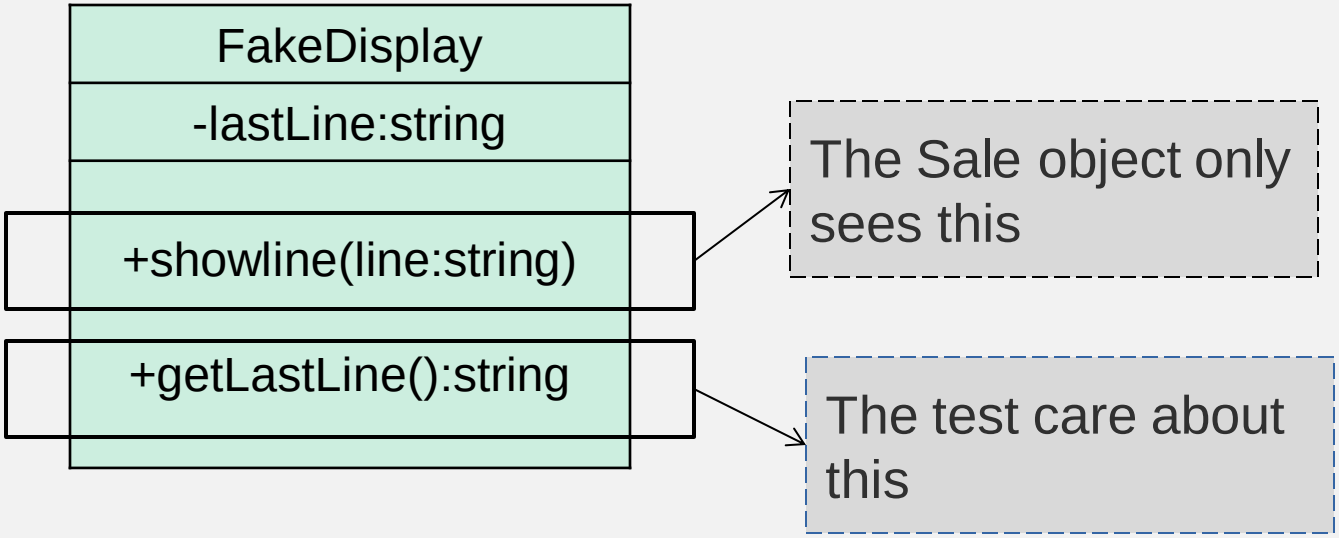
Test Driver

```
import junit.framework.*;
Public class SaleTest extends TestCase{
public void testDisplayItem(){
     FakeDisplay display = new FakeDisplay();
     Sale sale =  new Sale(display);
     sale.scan("1");
    assertEquals("Milk Taka 70", display.getLastLine());
   }}
```

Faking Collaborator

Fake objects can be confusing in a way...

One of the oddest things about them
is that they have **two sides**.



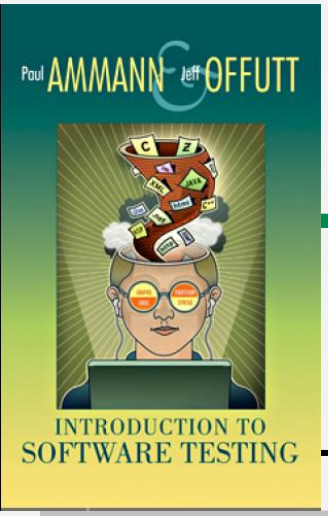
Faking Collaborator

A more advanced type of fake is called a ***mock object***.

Mock objects are fakes that perform assertions internally

i.e., we tell them what calls to expect, and then we tell them to check and see if they received those calls.

```
Public class SaleTest extends TestCase{
public void testDisplayItem(){
     MockDisplay display = new MockDisplay();
     display.setEcpectation("Showline", "Milk Taka 70");
     Sale sale =  new Sale(display);
     sale.scan("1");
    display.verify();
   }
}
```



Graph Coverage

Four Structures for Modeling Software

- Input Space
- Graphs
  - Applied to
    - Source
    - Specs
    - Design
    - Use cases
- Logic
  - Applied to
    - Source
    - FSMs
    - Specs
    - DNF
- Syntax
  - Applied to
    - Source
    - Models
    - Integ
    - Input

Thank You