SWEN 6301 Software Construction

Module 8: Software Testing

Ahmed Tamrawi

Testing

- Some programmers use the terms “**testing**” and “**debugging**” interchangeably, but this is not right!
- **Testing** is a means of *detecting errors*.
- **Debugging** is a means of *diagnosing and correcting the root causes of errors* that have already been detected.

Testing can **only show the presence of errors**, not their absence!

Testing

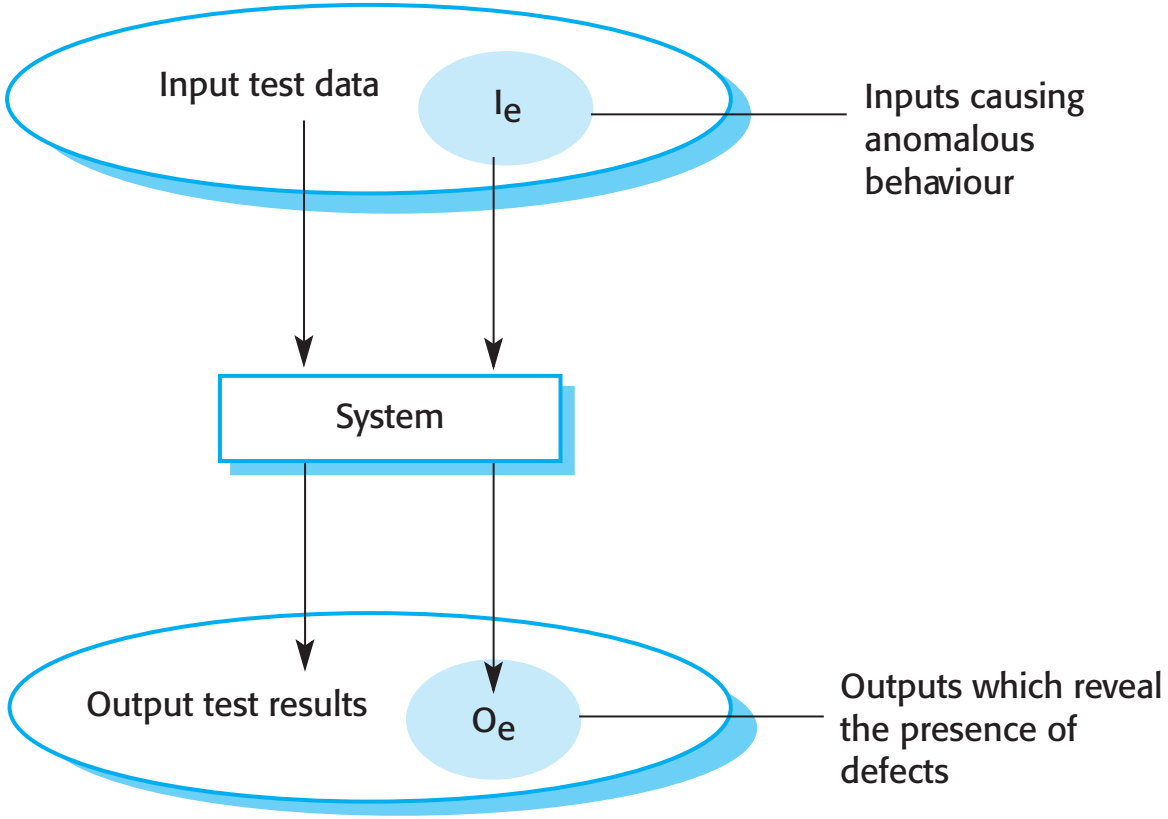
- **Testing** is intended to show that a **program does what it is intended to do** and to **discover program defects before it is put into use**.
- When you test software, **you execute a program using artificial (simulated) data**.
- You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
- Testing is part of a more **general verification and validation process**, which also includes **static validation techniques**.

Testing

The testing process has two distinct goals:

- The first goal leads to **validation testing**, where you expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
  - To demonstrate to the developer and the system customer that the software meets its requirements. A successful test shows that the system operates as intended.
- The second goal leads to **defect testing**, where the test cases are designed to expose defects.
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification. A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

An Input-Output Model of Software Testing



Verification vs Validation

- **Verification**:
  - "Are we building the product right".
  - *The software should conform to its specification*.
- **Validation**:
  - "Are we building the right product".
  - *The software should do what the user really requires*.
- Aim of V & V is to **establish confidence that the system is *fit for purpose*.**
- Depends on system's purpose, user expectations and marketing environment

Verification vs Validation

Software Purpose

- The more critical the software, the more important that it is reliable.
- For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a prototype that has been developed to demonstrate new product ideas.

Verification vs Validation

User Expectations

- Because of their experiences with buggy, unreliable software, many users have low expectations of software quality. They are not surprised when their software fails.
- When a new system is installed, users may tolerate failures because the benefits of use outweigh the costs of failure recovery.
- In these situations, you may not need to devote as much time to testing the software.
- However, as software matures, users expect it to become more reliable so more thorough testing of later versions may be required.

Verification vs Validation

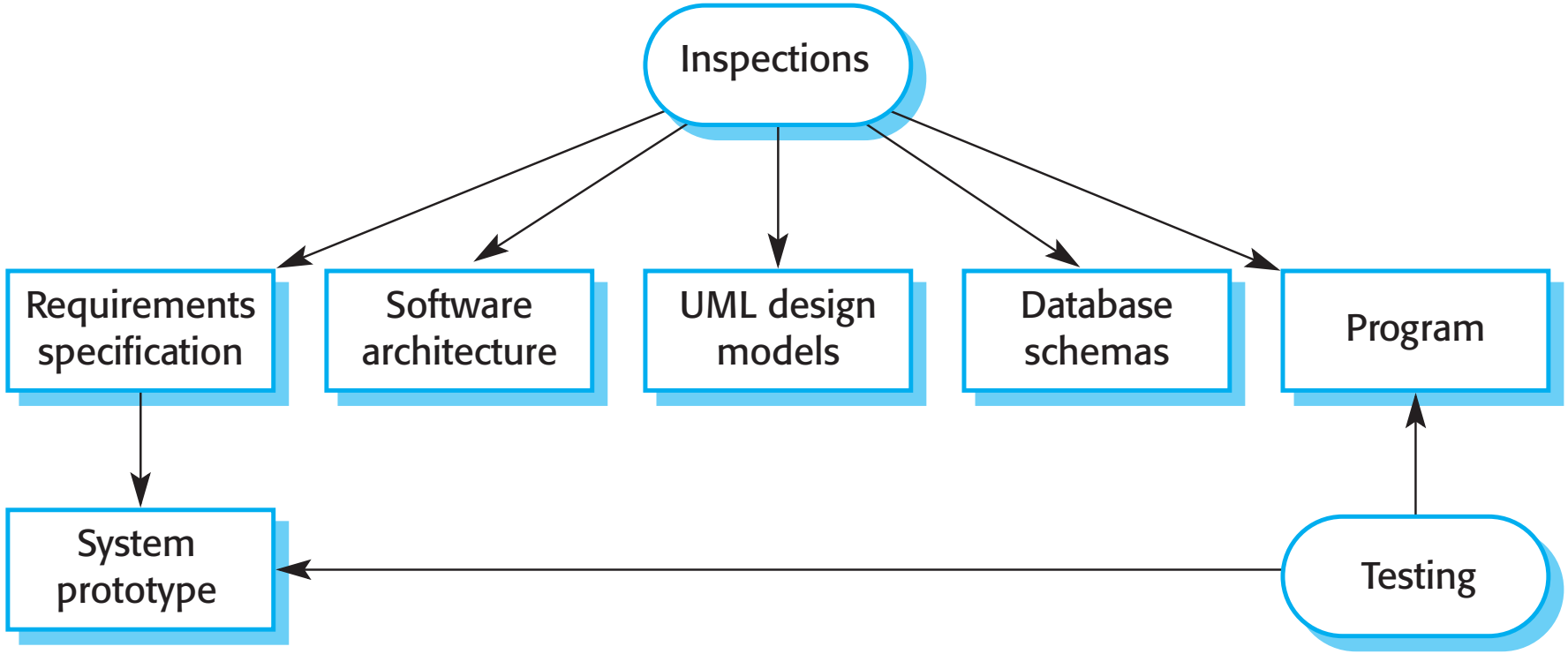
Marketing Environment

- When a system is marketed, the sellers of the system must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system.
- In a competitive environment, a software company may decide to release a program before it has been fully tested and debugged because they want to be the first into the market.
- *If a software product is very cheap, users may be willing to tolerate a lower level of reliability*.

Inspections and Testing

- **Software inspections** is concerned with analysis of the static system representation to discover problems (**static verification**)
  - May be supplement by tool-based document and code analysis.
- **Software testing** is concerned with exercising and observing product behaviour (**dynamic verification**)
  - The system is executed with test data (or ***simulation***) and its operational behaviour is observed.

Inspections and Testing



Software Inspections

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design,configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Advantages of Inspections

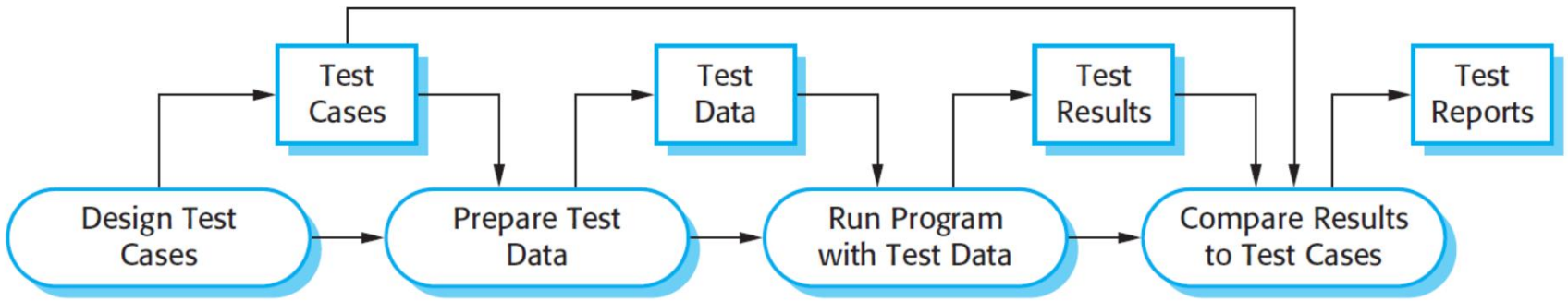
- During testing, errors can mask (hide) other errors. Because inspection is a static process, you don’t have to be concerned with interactions between errors.
- Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

Inspections and Testing

- **Inspections and testing are complementary and not opposing verification techniques**.
- Both should be used during the V & V process.
- Inspections can **check conformance with a specification but not conformance with the customer's real requirements**.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

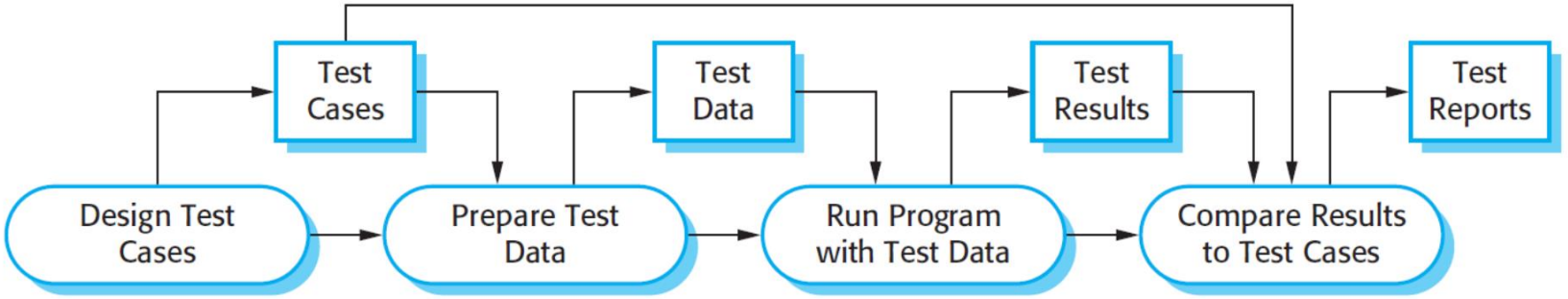
A Model of the Software Testing Process

- Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested.
- Test data are the inputs that have been devised to test a system.



Testing

- Test data can be generated automatically, but automatic test case generation is impossible?
- **What about mutation testing?** *is it feasible?*
- However, test execution can be automated.



Stages of Testing

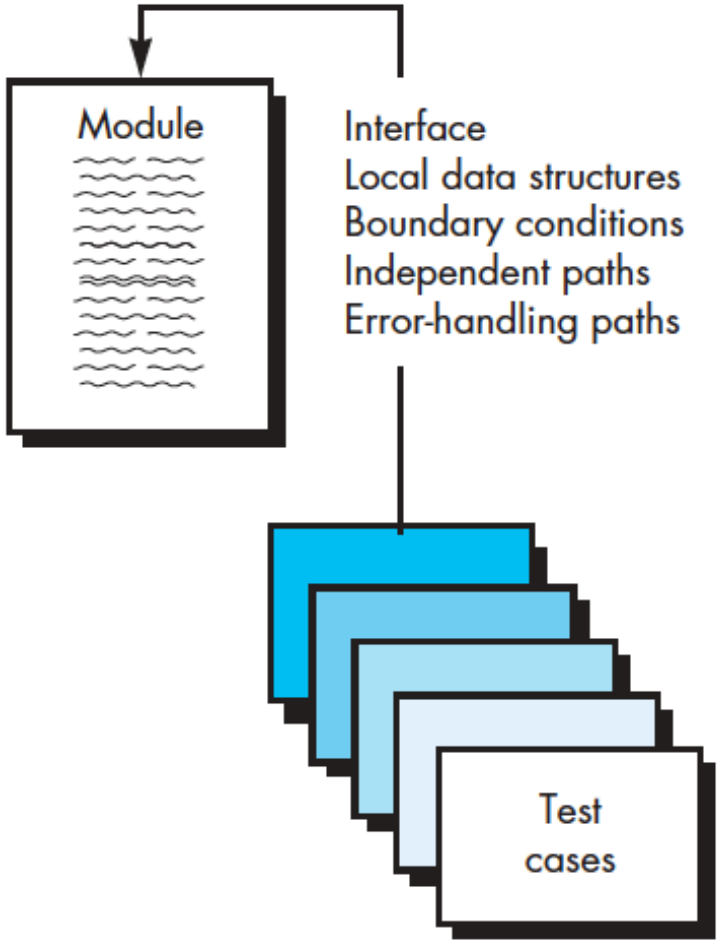
- **Development testing**, where the system is tested during development to discover bugs and defects.
- **Release testing**, where a separate testing team test a complete version of the system before it is released to users.
- **User testing**, where users or potential users of a system test the system in their own environment.

Development Testing

- **Development testing** includes all testing activities that are carried out by the team developing the system.
  - **Unit testing**, individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  - **Component testing**, several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
  - **Integration testing** is the combined execution of two or more classes, packages, components, or subsystems that have been created by multiple programmers or programming teams.
  - **System testing**, some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.
  - **Regression testing** is testing the system to check that changes have not ‘broken’ previously working code.

Unit Testing

- Unit testing **is the process of testing individual components in isolation**.
- It is a defect testing process.
- Units may be:
  - Individual **functions** or methods within an object
  - Object **classes** with several attributes and methods
  - Composite **components** with defined interfaces used to access their functionality.

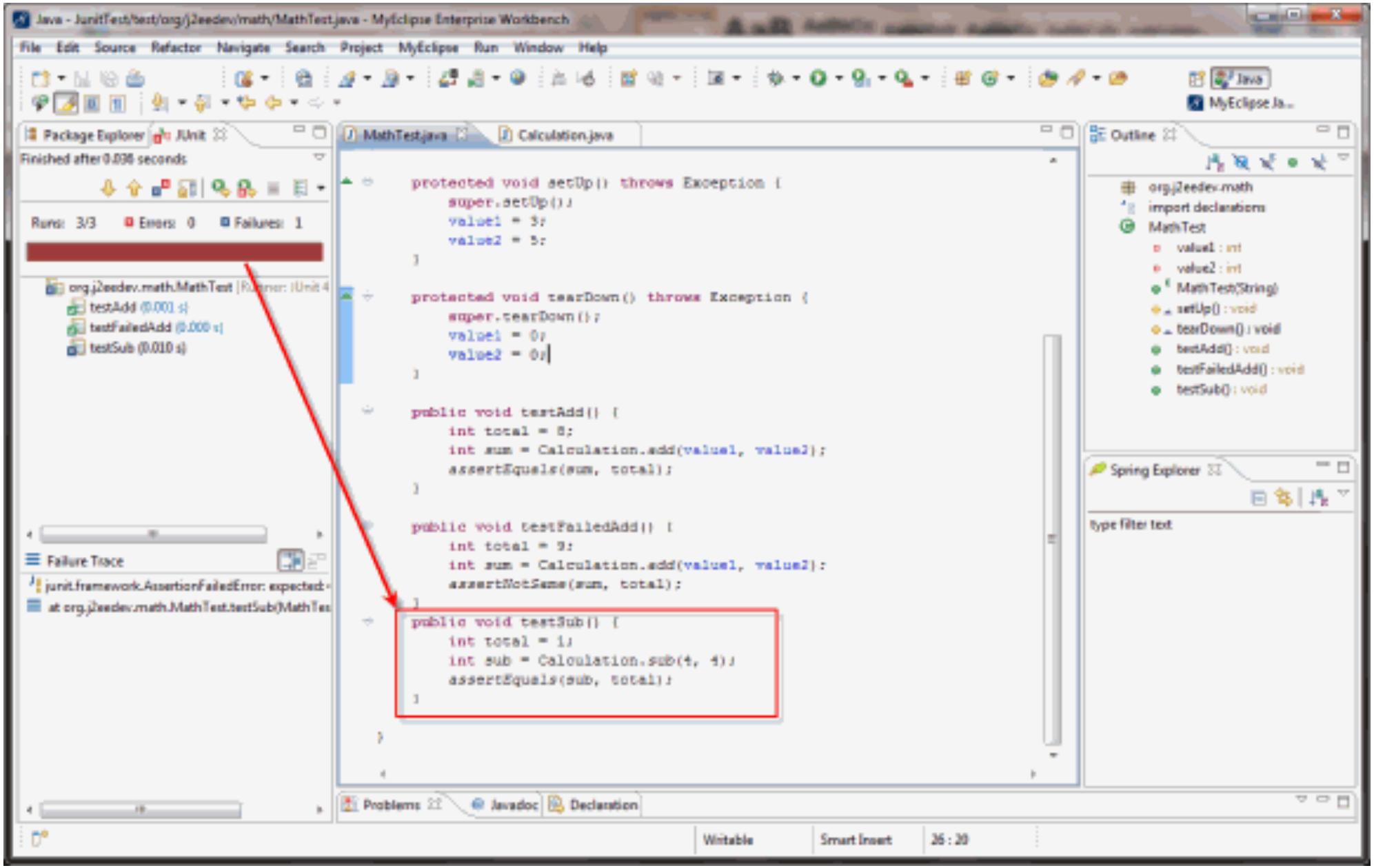


Unit Testing: *Object Class Testing*

- Complete test coverage of a class involves
  - **Testing all operations associated with an object**
  - **Setting and interrogating all object attributes**
  - **Exercising the object in all possible states**.
- *How many possible states are there?*
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

Unit Testing

- Unit testing is the execution of a complete class, routine, or small program that has been written by a single programmer or team of programmers, which is tested in isolation from the more complete system.
  - **Unit tests are basically written and executed by software developers** to make sure that code behaves as expected.

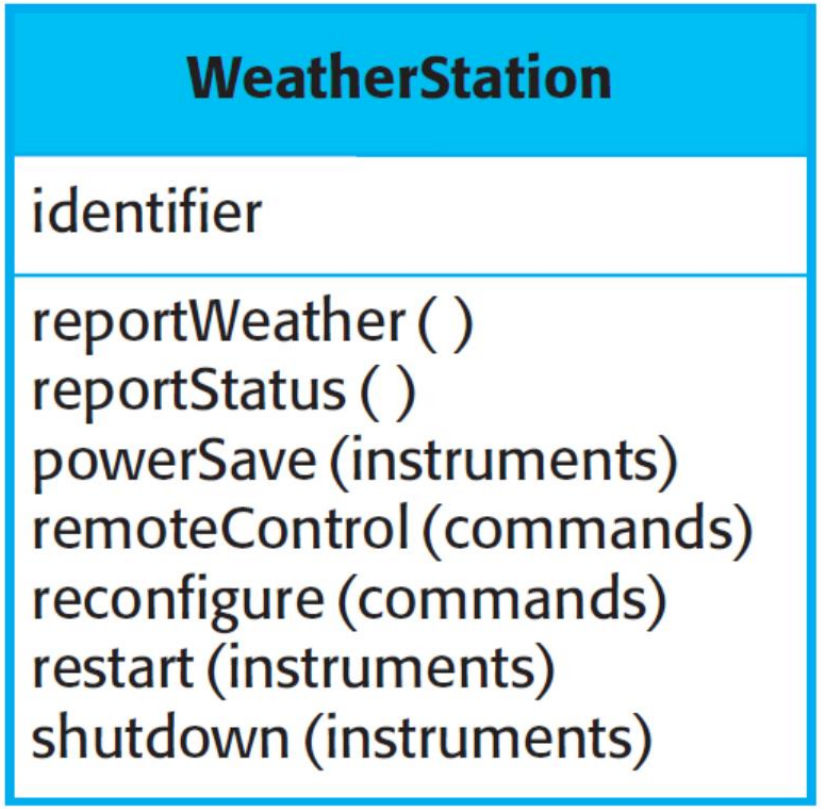

Java - JunitTest/test/org/j2eedev/math/MathTest.java - MyEclipse Enterprise Workbench
File Edit Source Refactor Navigate Search Project MyEclipse Run Window Help
Package Explorer
JUnit
Finished after 0.036 seconds
Runs: 3/3
Errors: 0
Failures: 1
org.j2eedev.math.MathTest
testAdd (0.001 s)
testFailedAdd (0.000 s)
testSub (0.010 s)
Failure Trace
junit.framework.AssertionFailedError: expected
at org.j2eedev.math.MathTest.testSub(MathTes
MathTest.java
Calculation.java
protected void setUp() throws Exception {
super.setUp();
value1 = 3;
value2 = 5;
}
protected void tearDown() throws Exception {
super.tearDown();
value1 = 0;
value2 = 0;
}
public void testAdd() {
int total = 8;
int sum = Calculation.add(value1, value2);
assertEquals(sum, total);
}
public void testFailedAdd() {
int total = 9;
int sum = Calculation.add(value1, value2);
assertNotSame(sum, total);
}
public void testSub() {
int total = 1;
int sub = Calculation.sub(4, 4);
assertEquals(sub, total);
}
}
Outline
org.j2eedev.math
import declarations
MathTest
value1 : int
value2 : int
MathTest(String)
setUp() : void
tearDown() : void
testAdd() : void
testFailedAdd() : void
testSub() : void
Spring Explorer
type filter text
Problems
Javadoc
Declaration
Writable
Smart Insert
26 : 20

Unit Testing

- When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. This means that you should:
  - test all operations associated with the object;
  - set and check the value of all attributes associated with the object;
  - put the object into all possible states. This means that you should simulate all events that cause a state change.

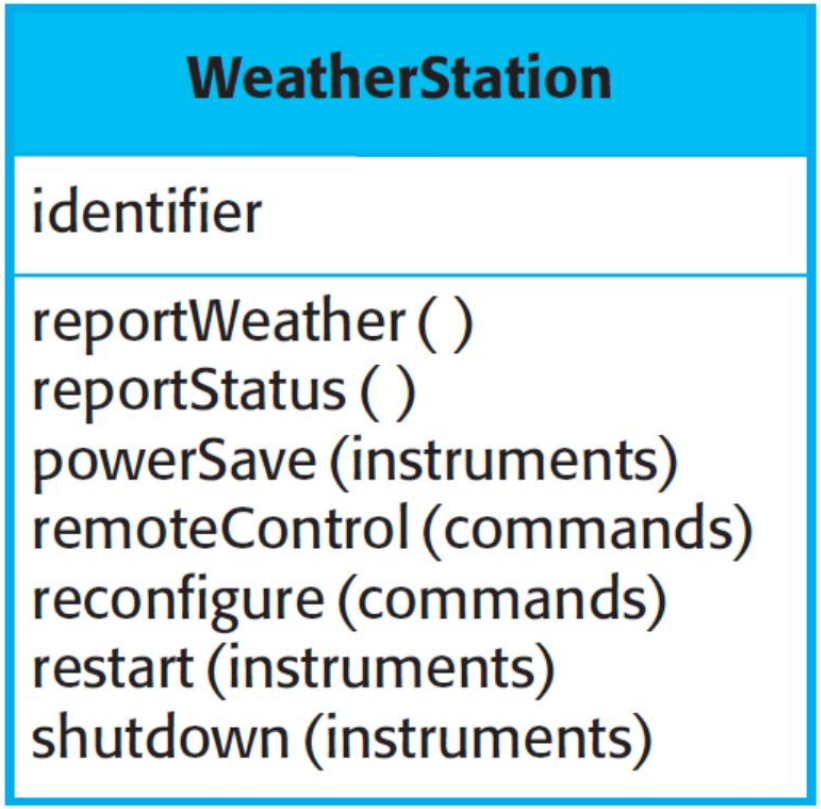
Unit Testing

- For example, **WeatherStation** object
- It has a single attribute, which is its **identifier**. This is a constant that is set when the weather station is installed.
- You therefore only need a test that checks if it has been properly set up.
- You need to define test cases for all of the methods associated with the object such as **reportWeather**, **reportStatus**, etc.



Unit Testing

- Ideally, you should test methods in isolation but, in some cases, some test sequences are necessary.
- For example, to test the method that shuts down the weather station instruments (**shutdown**), you need to have executed the **restart** method.

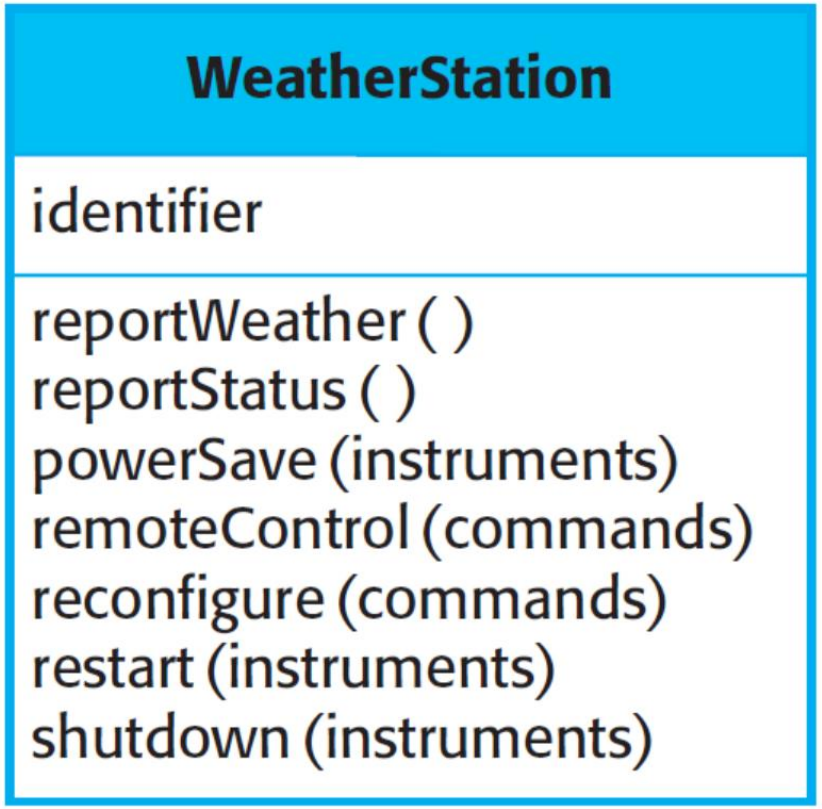


Unit Testing

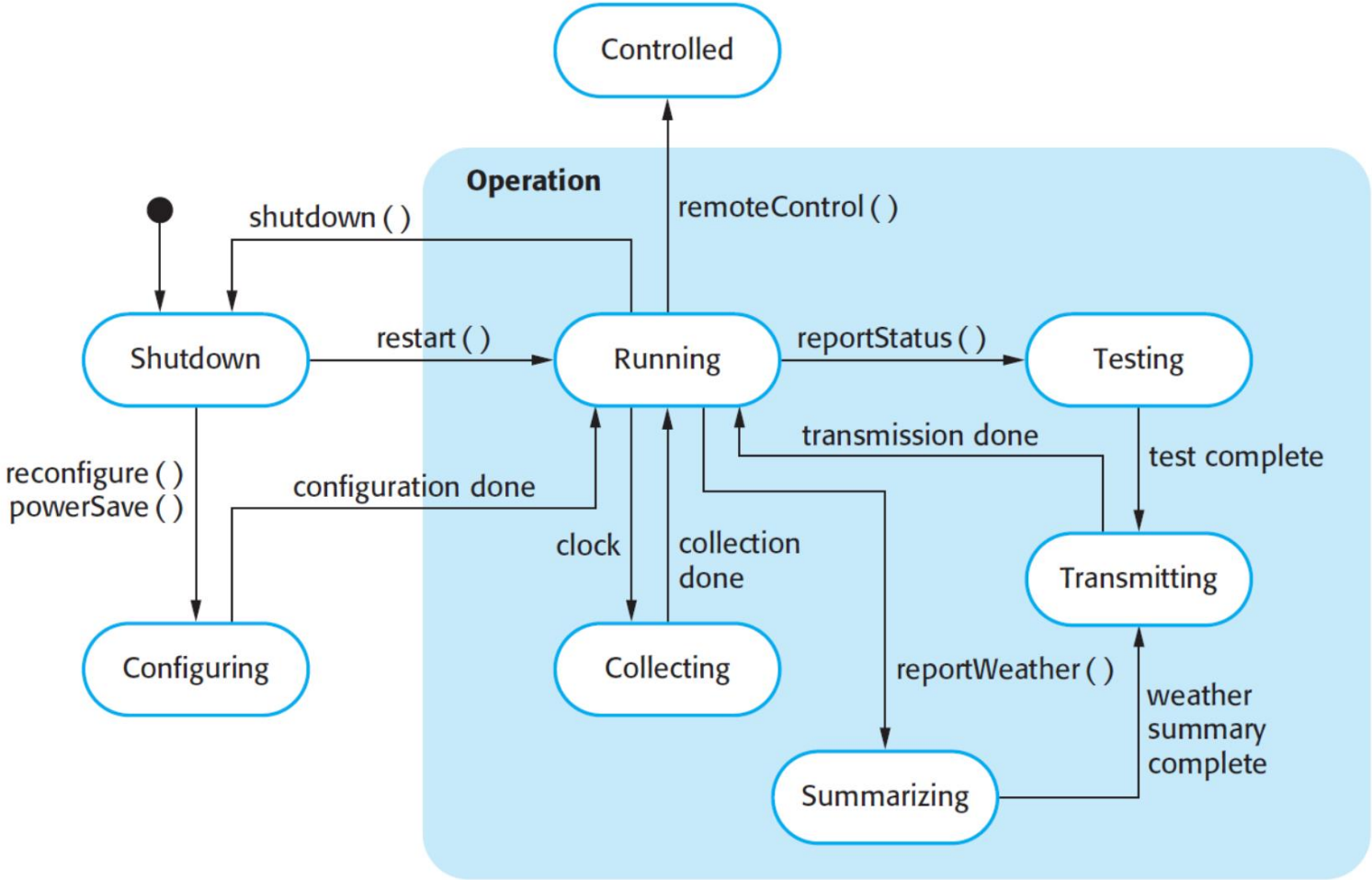
- ***Inheritance*** makes object class testing more complicated.
- You can’t simply test an operation in the class where it is defined and assume that it will work as expected in the subclasses that inherit the operation.
- The operation that is inherited may make assumptions about other operations and attributes.
- These may not be valid in some subclasses that inherit the operation. You therefore have to test the inherited operation in all of the contexts where it is used.

Unit Testing

- To test the states of the weather station, you use a state model (as on the next slide)
- Using this model, you can identify sequences of state transitions that have to be tested and define event sequences to force these transitions.
- In principle, you should test every possible state transition sequence, although in practice this may be too expensive.



Unit Testing



Unit Testing

- Whenever possible, you should automate unit testing.
- In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests.
- Unit testing frameworks provide generic test classes that you extend to create specific test cases.
- They can then run all of the tests that you have implemented and report, often through some GUI, on the success or failure of the tests.
- An entire test suite can often be run in a few seconds so it is possible to execute all the tests every time you make a change to the program.

Unit Testing

- An automated test has three parts:
  1. A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
  2. A call part, where you call the object or method to be tested.
  3. An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

```java
@RunWith(JUnit4.class)
public class BeforeAndAfterAnnotationsUnitTest {

    // ...

    private List<String> list;

    @Before
    public void init() {
        LOG.info("startup");
        list = new ArrayList<>(Arrays.asList("test1", "test2"));
    }

    @After
    public void finalize() {
        LOG.info("finalize");
        list.clear();
    }
}
```

```java
@Test
public void whenCheckingListSize_thenSizeEqualsToInit() {
    LOG.info("executing test");
    assertEquals(2, list.size());

    list.add("another test");
}

@Test
public void whenCheckingListSizeAgain_thenSizeEqualsToInit() {
    LOG.info("executing another test");
    assertEquals(2, list.size());

    list.add("yet another test");
}
```

Unit Testing

- Sometimes the object that you are testing has dependencies on other objects that may not have been written or which slow down the testing process if they are used.
- For example, if your object calls a database, this may involve a slow setup process before it can be used.
- In these cases, you may decide to use mock objects.

Unit Testing

- Mock objects are objects with the same interface as the external objects being used that simulate its functionality.
- Therefore, a mock object simulating a database may have only a few data items that are organized in an array.
- They can therefore be accessed quickly, without the overheads of calling a database and accessing disks.

REAL SYSTEM

Green = class in focus
Yellow = dependencies
Grey = other unrelated classes

CLASS IN UNIT TEST

Green = class in focus
Yellow = mocks for the unit test

COPY JAVA

```java
import static org.mockito.Mockito.*;

public class MockitoTest  {

    @Mock
    MyDatabase databaseMock; (1)

    @Rule public MockitoRule mockitoRule = MockitoJUnit.rule(); (2)

    @Test
    public void testQuery()  {
        ClassToTest t  = new ClassToTest(databaseMock); (3)
        boolean check = t.query("* from t"); (4)
        assertTrue(check); (5)
        verify(databaseMock).query("* from t"); (6)
    }
}
```

1. Tells Mockito to mock the databaseMock instance
2. Tells Mockito to create the mocks based on the @Mock annotation
3. Instantiates the class under test using the created mock
4. Executes some code of the class under test
5. Asserts that the method call returned true
6. Verify that the query method was called on the `MyDatabase` mock

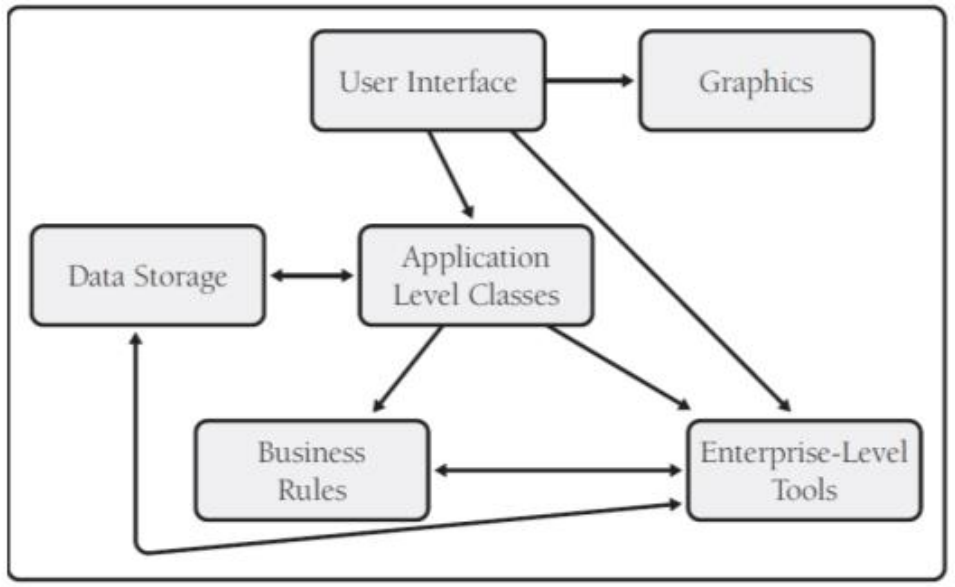
```java
@Test
public void test1()  {
        //  create mock
        MyClass test = mock(MyClass.class);

        // define return value for method getUniqueId()
        when(test.getUniqueId()).thenReturn(43);

        // use mock in test....
        assertEquals(test.getUniqueId(), 43);
}


// demonstrates the return of multiple values
@Test
public void testMoreThanOneReturnValue()  {
        Iterator<String> i= mock(Iterator.class);
        when(i.next()).thenReturn("Mockito").thenReturn("rocks");
        String result= i.next()+" "+i.next();
        //assert
        assertEquals("Mockito rocks", result);
}
```

Unit tests with Mockito

https://www.vogella.com/tutorials/Mockito/article.html

Component Testing

- Component testing is the execution of a class, package, small program, or other program element that involves the work of multiple programmers or programming teams, which is tested in isolation from the more complete system.



Component Testing

- Software components are often composite components that are made up of several interacting objects.
  - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
  - You can assume that unit tests on the individual objects within the component have been completed.

Interface Testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Interface types
  - **Parameter interfaces:** Data passed from one method or procedure to another.
  - **Shared memory interfaces:** Block of memory is shared between procedures or functions.
  - **Procedural interfaces:** Sub-system encapsulates a set of procedures to be called by other sub-systems.
  - **Message passing interfaces:** Sub-systems request services from other sub-systems

Integration Testing

- Integration testing is the combined execution of two or more classes, packages, components, or subsystems that have been created by multiple programmers or programming teams.
- This kind of testing typically starts as soon as there are **two classes** to test and continues until the entire system is complete.

System Testing

- System testing is the execution of the software in its final configuration, including integration with other software and hardware systems.
- It tests for security, performance, resource loss, timing problems, and other issues that can’t be tested at lower levels of integration.
- It may include tests based on
  - risks and/or requirement specifications, business process, use cases, or other high level descriptions of system behavior, interactions with the operating systems, and system resources.

System Testing

- System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- The focus in system testing is testing **the interactions between components**.
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- System testing tests the **emergent behavior** of a system.

System and Component Testing

- During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
  - In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

Regression Testing

- Regression testing is the repetition of previously executed test cases for the purpose of finding defects in modified software that previously passed the same set of tests.
  - Any new feature is added
  - Any enhancement is done
  - Any bug is fixed
  - Any performance related issue is fixed
- Regression testing is testing the system to check that changes have not ‘broken’ previously working code.
- In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program.
- Tests must run ‘successfully’ before the change is committed.

Testing

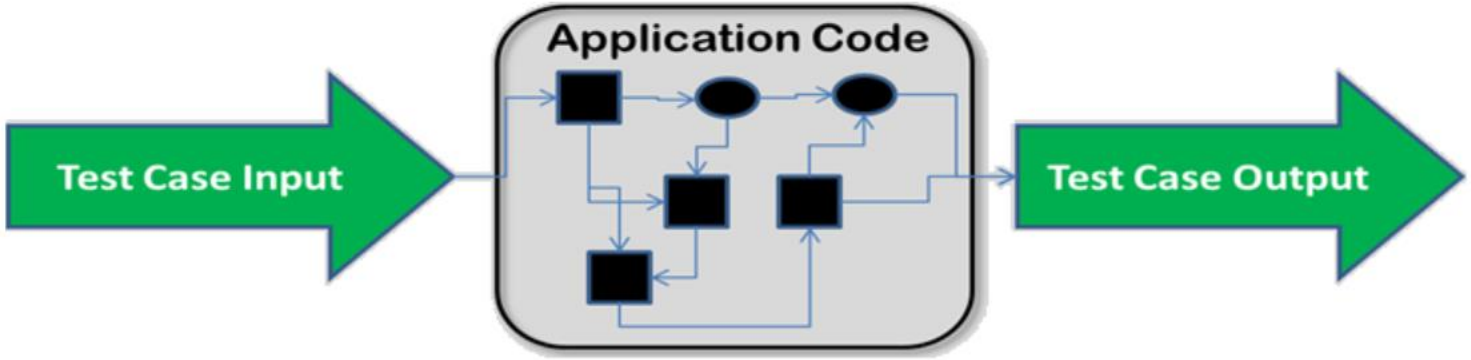
- Testing is usually broken into two broad categories:
  - black-box (or specification-based) testing
  - white-box (or glass-box) testing

Black-Box Testing

- Black-box testing refers to tests in which the tester cannot see the inner workings of the item being tested. This obviously does not apply when you test code that you have written!
  - a.k.a. Specification-based testing technique or input/output driven testing techniques because they view the software as a black-box with inputs and outputs.
  - Concentrating on what the software does, not how it does it.

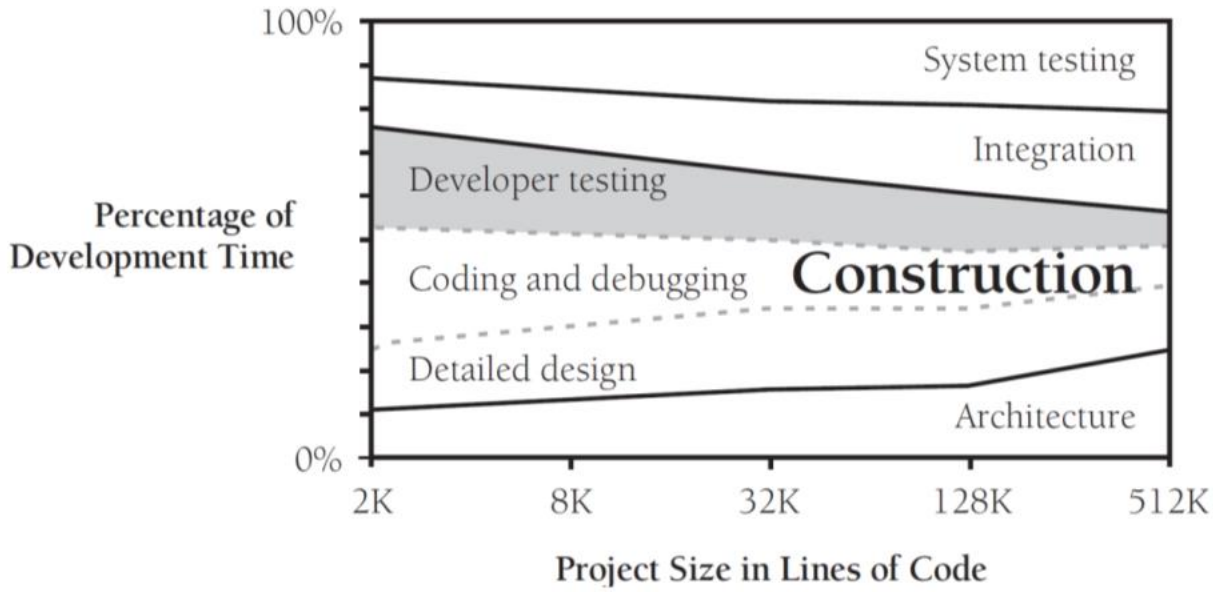
White-Box Testing

- White-box testing refers to tests in which the tester is aware of the inner workings of the item being tested. *This is the kind of testing that you as a developer use to test your own code*.
  - a.k.a. Structure-based testing technique is or 'glass-box' testing technique because here the testers require knowledge of how the software is implemented, how it works.



Development Testing

- Developer testing typically consists of unit tests, component tests, and integration tests
  - but can sometimes include regression tests and system tests.
- A key question is, How much time should be spent in developer testing on a typical project?



Development Testing

- What do you do with the results of developer testing?
  - To assess the reliability of the product under development. *Even if you never correct the defects that testing finds, testing describes how reliable the software is*.
  - To guide corrections to the software.
  - Finally, over time, the record of defects found through testing helps reveal the kinds of errors that are most common. You can use this information to select appropriate training classes, direct future technical review activities, and design future test cases.

Development Testing

- During construction, you generally write a routine or class, check it mentally, and then review it or test it.
- Regardless of your integration or system-testing strategy, you should test each unit thoroughly before you combine it with any others.

Development Testing

- *Test for each relevant requirement to make sure that the requirements have been implemented*.
- Plan the test cases for this step at the requirements stage or as early as possible—preferably before you begin writing the unit to be tested.
- Consider testing for common omissions in requirements.
- The level of security, storage, the installation procedure, and system reliability are all fair game for testing and are often missed at requirements time.

Development Testing

- *Test for each relevant design concern to make sure that the design has been implemented*.
- Plan the test cases for this step at the design stage or as early as possible-before you begin the detailed coding of the routine or class to be tested.

Development Testing

- *Use Basis Path Testing to add detailed test cases to those that test the requirements and the design*.
- Add data-flow tests, and then add the remaining test cases needed to thoroughly exercise the code.
- At a minimum, you should test every line of code.
- Basis Path Testing and Data-Flow Testing are described later...

Intractability of Testing

- To use testing to prove that a program works, you'd have to test every conceivable input values and every feasible paths.
- Number of paths **grows exponentially** with the number of non-nested if conditions (branch nodes in control flow graph).
- Number of input **grows exponentially** with the number of bits composing the value.
- Therefore:
  - **Input Coverage:** *Complete input value coverage is interactable in general.*
  - **Path Coverage:** *Complete path coverage is interactable in general.*

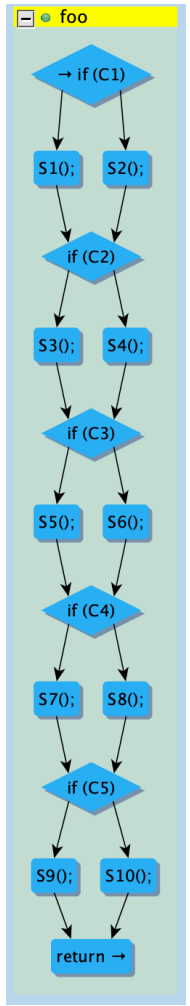
Intractability of Testing

- Suppose, for example, that you have a program that takes a name, an address, and a phone number and stores them in a file.
- Each of the names and addresses is 20 characters long and that there are 26 possible characters to be used in them. This would be the number of possible inputs:

Name	26^{20} (20 characters, each with 26 possible choices)
Address	26^{20} (20 characters, each with 26 possible choices)
Phone Number	10^{10} (10 digits, each with 10 possible choices)
Total Possibilities	$= 26^{20} * 26^{20} * 10^{10} \approx 10^{66}$

Intractability of Testing

```
1  public void foo() {
2     if(C1) {
3        S1();
4     } else {
5        S2();
6     }
7     if(C2) {
8        S3();
9     } else {
10       S4();
11    }
12    if(C3) {
13       S5();
14    } else {
15       S6();
16    }
17    if(C4) {
18       S7();
19    } else {
20       S8();
21    }
22    if(C5) {
23       S9();
24    } else {
25       S10();
26    }
27 }
```



5 non-nested if conditions (Branch Points)

$$2^5 = 32$$

Execution (Paths) Behaviors

Intractability of Testing

- Since exhaustive testing is impossible, practically speaking, the art of testing is that of picking the test cases most likely to find errors.
- You need to concentrate on picking a few that tell you different things rather than a set that tells you the same thing over and over.

Intractability of Testing

- Coverage Metrics include (but not limited to):
  - **Branch Coverage** (fair metric) assists how much branch nodes (conditions) have been covered by a test case.
  - **Statement Coverage** (fair metric) assists how much statements have been covered by a test case.
  - **Path Coverage** (better metric) assists how much paths have been covered by a test case.
  - **Input Coverage** (better metric) assists how much input space have been covered by a test case.
  - **Path + Input Coverage** (best metric) assists completeness of the test case.

Control Flow Testing : *Structured Basis Testing*

- **Basis path testing (structured testing)** is a white box method for designing test cases.
- The method analyzes the control flow graph of a program to find a **set of linearly independent paths of execution**.
- Basis path testing guarantees **complete branch coverage** (all edges of the control flow graph) and complete statement coverage, **but achieves that without covering all possible paths of the control flow graph—the latter is usually too costly.**
- Basis path testing has been widely used and studied

Control Flow Testing : *Structured Basis Testing*

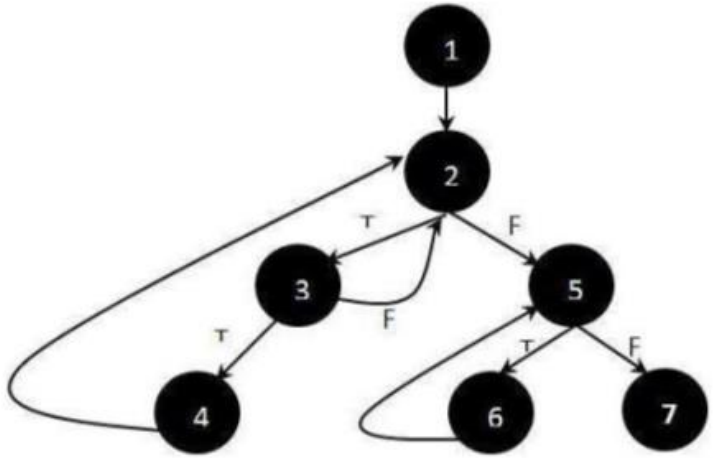
- The method normally uses **McCabe' cyclomatic complexity** to determine the number of linearly independent paths and then generates test cases for each path thus obtained.

V(G) = E - N + 2: Where, **E - Number of edges N - Number of Nodes**
V(G) = P + 1: Where **P = Number of predicate nodes (node that contains condition)**

- Basis path testing guarantees **complete branch coverage** (all edges of the control flow graph) and complete statement coverage, **but achieves that without covering all possible paths of the control flow graph—the latter is usually too costly.**

Control Flow Testing : *Structured Basis Testing*

- **Step 1 :** Draw the Flow Graph of the Function/Program under consideration as shown below:




```
Function fn_delete_element (int value, int array_size, int array[])
{
        1 int i;
        location = array_size + 1;

        2 for i = 1 to array_size
        3 if ( array[i] == value )
        4 location = i;
         end if;
         end for;

        5 for i = location to array_size
        6 array[i] = array[i+1];
        end for;
        7 array_size --;
}
```

Predicate (branching)

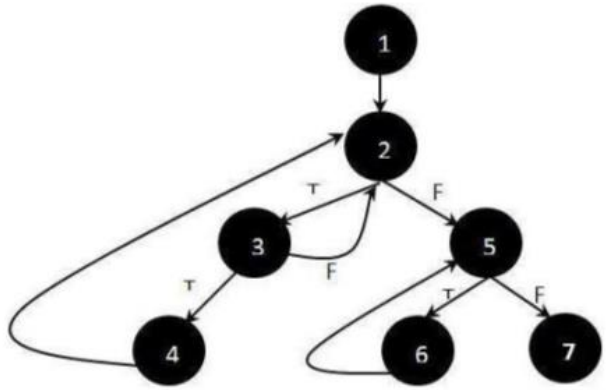
Predicate (branching)

Predicate (branching)

Control Flow Testing : *Structured Basis Testing*

- **Step 2 :** Determine the independent paths.

```
Path 1:  1 - 2 - 5 - 7
Path 2:  1 - 2 - 5 - 6 - 7
Path 3:  1 - 2 - 3 - 2 - 5 - 6 - 7
Path 4:  1 - 2 - 3 - 4 - 2 - 5 - 6 - 7
```




```
Function fn_delete_element (int value, int array_size, int array[])
{
        1 int i;
        location = array_size + 1;

        2 for i = 1 to array_size
        3 if ( array[i] == value )
        4 location = i;
         end if;
         end for;

        5 for i = location to array_size
        6 array[i] = array[i+1];
        end for;
        7 array_size --;
}
```

Predicate (branching)

Predicate (branching)

Predicate (branching)

Control Flow Testing : *Structured Basis Testing*

- You can compute the minimum number of cases needed for basis testing in this straightforward way:
  1. Start with 1 for the straight path through the routine.
  2. Add 1 for each of the following keywords, or their equivalents: **if**, **case**, **while**, **for**, **and**, and **or**, etc.

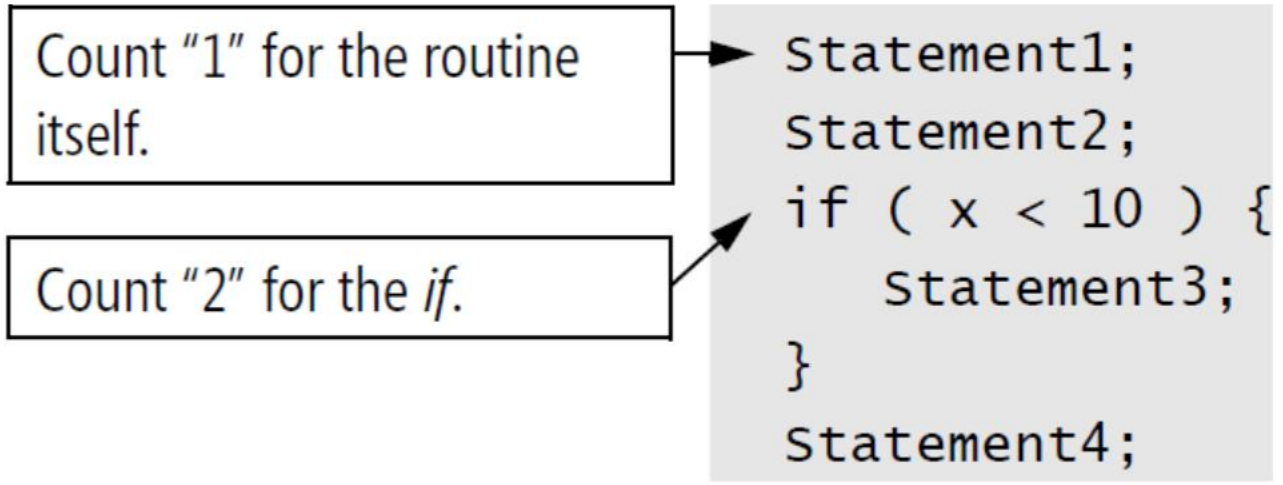
V(G) = E - N + 2: Where, **E - Number of edges N - Number of Nodes**
V(G) = P + 1: Where **P = Number of predicate nodes (node that contains condition)**

Control Flow Testing : *Structured Basis Testing*

- Start with one and count the if once to make a total of two. So, you need to have at least two test cases to cover all the paths through the program.
- In this example, you'd need to have the following test cases:

Statements controlled by *if* are executed (x < 10).

Statements controlled by *if* aren't executed (x >= 10)



Example of Computing the Number of Cases Needed for Basis Testing of a Java Program

```
1  // Compute Net Pay
2  totalWithholdings = 0;
3
4  for ( id = 0; id < numEmployees; id++ ) {
5
6     // compute social security withholding, if below the maximum
7     if ( m_employee[ id ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT ) {
8        governmentRetirement = ComputeGovernmentRetirement( m_employee[ id ] );
9     }
10
11    // set default to no retirement contribution
12    companyRetirement = 0;
13
14    // determine discretionary employee retirement contribution
15    if ( m_employee[ id ].WantsRetirement &&
16       EligibleForRetirement( m_employee[ id ] ) ) {
17       companyRetirement = GetRetirement( m_employee[ id ] );
18    }
19
20    grossPay = ComputeGrossPay ( m_employee[ id ] );
21
22    // determine IRA contribution
23    personalRetirement = 0;
24    if ( EligibleForPersonalRetirement( m_employee[ id ] ) ) {
25       personalRetirement = PersonalRetirementContribution( m_employee[ id ],
26          companyRetirement, grossPay );
27    }
28
29    // make weekly paycheck
30    withholding = ComputeWithholding( m_employee[ id ] );
31    netPay = grossPay - withholding - companyRetirement - governmentRetirement -
32       personalRetirement;
33    PayEmployee( m_employee[ id ], netPay );
34
35    // add this employee's paycheck to total for accounting
36    totalWithholdings = totalWithholdings + withholding;
37    totalGovernmentRetirement = totalGovernmentRetirement + governmentRetirement;
38    totalRetirement = totalRetirement + companyRetirement;
39 }
40
41 SavePayRecords( totalWithholdings, totalGovernmentRetirement, totalRetirement );
```

Example of Computing the Number of Cases Needed for Basis Testing of a Java Program

Count "1" for the routine itself.

Count "2" for the *for*.

Count "3" for the *if*.

Count "4" for the *if* and "5" for the *&&*.

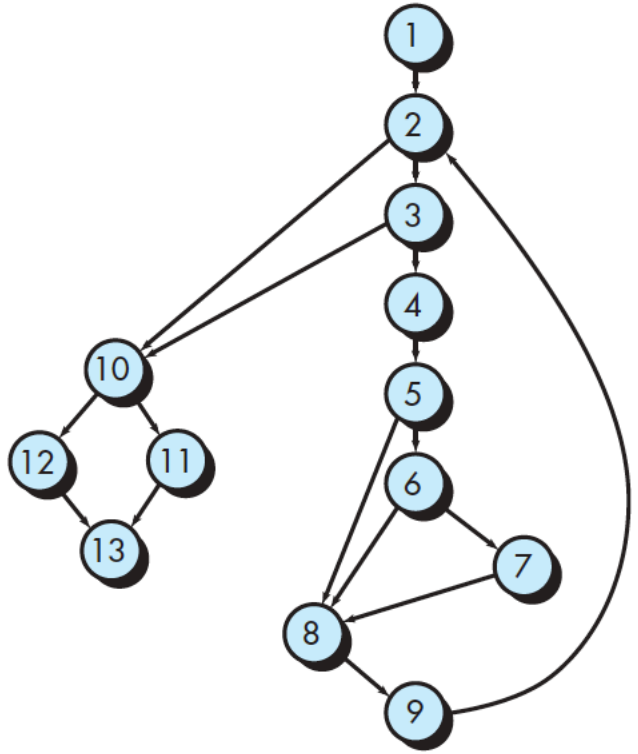
Count "6" for the *if*.

```
1  // Compute Net Pay
2  totalWithholdings = 0;
3
4  for ( id = 0; id < numEmployees; id++ ) {
5
6     // compute social security withholding, if below the maximum
7     if ( m_employee[ id ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT ) {
8        governmentRetirement = ComputeGovernmentRetirement( m_employee[ id ] );
9     }
10
11    // set default to no retirement contribution
12    companyRetirement = 0;
13
14    // determine discretionary employee retirement contribution
15    if ( m_employee[ id ].WantsRetirement &&
16       EligibleForRetirement( m_employee[ id ] ) ) {
17       companyRetirement = GetRetirement( m_employee[ id ] );
18    }
19
20    grossPay = ComputeGrossPay ( m_employee[ id ] );
21
22    // determine IRA contribution
23    personalRetirement = 0;
24    if ( EligibleForPersonalRetirement( m_employee[ id ] ) ) {
25       personalRetirement = PersonalRetirementContribution( m_employee[ id ],
26          companyRetirement, grossPay );
27    }
28
29    // make weekly paycheck
30    withholding = ComputeWithholding( m_employee[ id ] );
31    netPay = grossPay - withholding - companyRetirement - governmentRetirement -
32       personalRetirement;
33    PayEmployee( m_employee[ id ], netPay );
34
35    // add this employee's paycheck to total for accounting
36    totalWithholdings = totalWithholdings + withholding;
37    totalGovernmentRetirement = totalGovernmentRetirement + governmentRetirement;
38    totalRetirement = totalRetirement + companyRetirement;
39 }
40
41 SavePayRecords( totalWithholdings, totalGovernmentRetirement, totalRetirement );
```

Case	Test Description	Test Data
1	Nominal case	All boolean conditions are true
2	The initial *for* condition is false	*numEmployees < 1*
3	The first *if* is false	*m_employee[id].governmentRetirementWithheld >=MAX_GOVT_RETIREMENT*
4	The second *if* is false because the first part of the *and* is false	*not m_employee[id].WantsRetirement*
5	The second *if* is false because the second part of the *and* is false	*not EligibleForRetirement(m_employee[id])*
6	The third *if* is false	*not EligibleForPersonalRetirement(m_employee[id])*

Note: This table will be extended with additional test cases throughout the chapter.

```
PROCEDURE average;

    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
         minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
1   i = 1;
    total.input = total.valid = 0;
    sum = 0;
2   DO WHILE value[i] <> –999 AND total.input < 100
3       increment total.input by 1;
4       IF value[i] > = minimum AND value[i] < = maximum
5           THEN increment total.valid by 1;
6                sum = s sum + value[i]
7           ELSE skip
        ENDIF
8       increment i by 1;
9   ENDDO
    IF total.valid > 0
10      THEN average = sum / total.valid;
11      ELSE average = –999;
12  ENDIF
13  END average
```



$V(G)$ = 6 regions

$V(G)$ = 17 edges − 13 nodes + 2 = 6

$V(G)$ = 5 predicate nodes + 1 = 6

Path 1: 1-2-10-11-13

Path 2: 1-2-10-12-13

Path 3: 1-2-3-10-11-13

Path 4: 1-2-3-4-5-8-9-2-. . .

Path 5: 1-2-3-4-5-6-8-9-2-. . .

Path 6: 1-2-3-4-5-6-7-8-9-2-. . .

Control Flow Testing : *Structured Basis Testing*

- If the routine were much more complicated than this, the number of test cases you'd have to use just to cover all the paths would increase pretty quickly.
- Shorter routines tend to have fewer paths to test. Boolean expressions without a lot of ANDs and ORs have fewer variations to test. Ease of testing is another good reason to keep your routines short and your boolean expressions simple.
- Now that you've created six test cases for the routine and satisfied the demands of structured basis testing, can you consider the routine to be fully tested? *Probably not*.

Data Flow Testing

- **Data-flow testing** is a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects.
  - focuses on the points at which variables receive values and the points at which these values are used.
  - Data-flow testing is based on the idea that **data usage** is at least as error-prone as control flow.
- Data Flow testing helps us to pinpoint any of the following issues:
  - A variable that is declared but never used within the program.
  - A variable that is used but never declared.
  - A variable that is defined multiple times before it is used.
  - Deallocating a variable before it is used.

Data Flow Testing: *Equivalence Partitioning*

- A good test case covers a large part of the possible input data.
- *If two test cases flush out exactly the same errors, you need only one of them*.
- The concept of "equivalence partitioning" is a formalization of this idea and helps reduce the number of test cases required.
  - is a software testing technique that divides the input data of a software unit into **partitions** of **equivalent data** from which test cases can be derived.
  - In principle, test cases are designed to cover each partition **at least once**.

Data Flow Testing: *Equivalence Partitioning*

- ***Password***: must be a minimum 8 characters and maximum 12 characters.
- **What are the test cases?**

Invalid Partition	Valid Partition	Invalid Partition
Less than 8	8 - 12	More than 12

Data Flow Testing: *Equivalence Partitioning*

- ***Password***: must be a minimum 8 characters and maximum 12
  - **Test Case 1**: Consider password length less than 8.
  - **Test Case 2**: Consider password of length exactly 8.
  - **Test Case 3**: Consider password of length between 9 and 11.
  - **Test Case 4**: Consider password of length exactly 12.
  - **Test Case 5**: Consider password of length more than 12.

Data Flow Testing: *Equivalence Partitioning*

- Once you have identified a set of partitions, you choose test cases from each of these partitions.
- A good rule of thumb for test case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition.
- The reason for this is that designers and programmers tend to consider typical values of inputs when developing a system. You test these by choosing the midpoint of the partition.
- Boundary values are often **atypical** (e.g., zero may behave differently from other non-negative numbers) so are sometimes overlooked by developers.
- **Program failures often occur when processing these atypical values**.

Data Flow Testing: *Equivalence Partitioning*

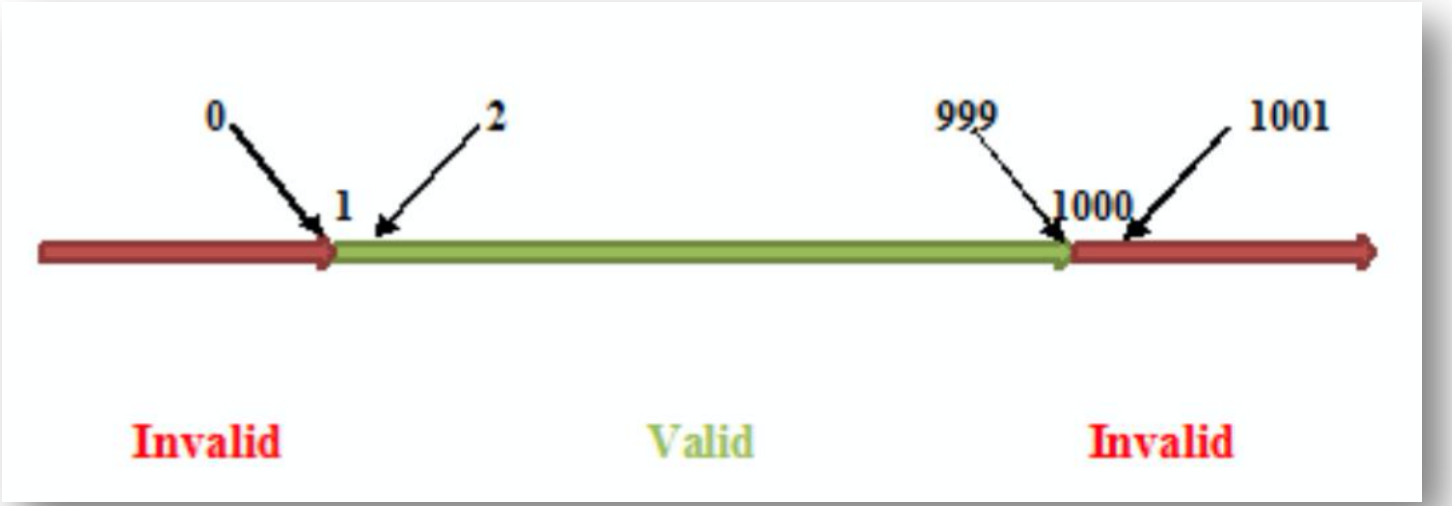
- Thinking about equivalence partitioning won't give you a lot of new insight into a program when you have already covered the program with basis and data-flow testing.
- It's especially helpful, however, when you're looking at a program from the outside (from a specification rather than the source code) or when the data is complicated and the complications aren't all reflected in the program's logic.

Data Flow Testing: *Error Guessing*

- In addition to the formal test techniques, good programmers use a variety of less formal, heuristic techniques to expose errors in their code.
- One heuristic is the technique of error guessing.
- It means creating test cases based upon guesses about where the program might have errors, although it implies a certain amount of sophistication in the guessing.

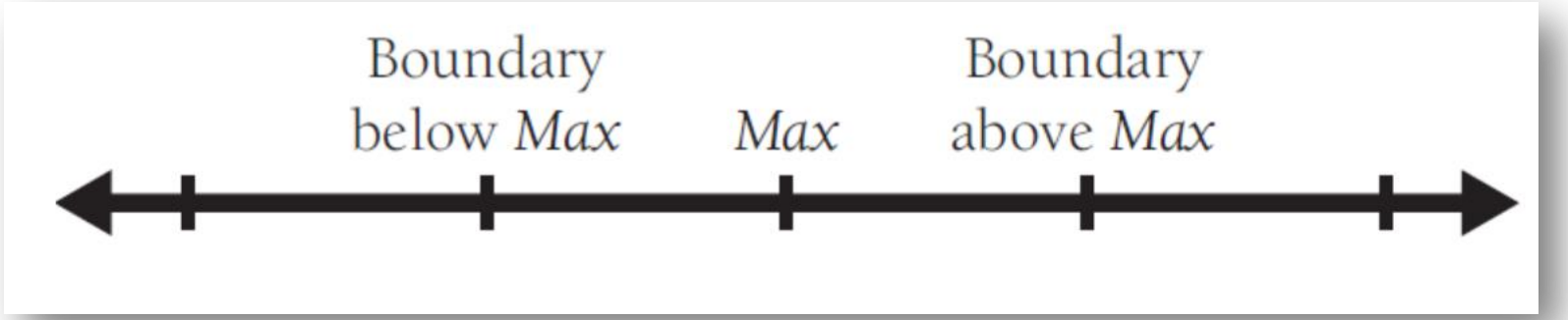
Data Flow Testing: *Boundary Analysis*

- One of the most fruitful areas for testing is boundary conditions—off-by-one errors.
- Saying num – 1 when you mean num and saying >= when you mean > are common mistakes.



Data Flow Testing: *Boundary Analysis*

- The idea of boundary analysis is to write test cases that exercise the boundary conditions.
- If you're testing for a range of values that are less than max, you have three possible conditions:



- As shown, there are three boundary cases: just less than max, max itself, and just greater than max. It takes three cases to ensure that none of the common mistakes has been made.

Data Flow Testing: *Classes of Bad Data*

- Aside from guessing that errors show up around boundary conditions, you can guess about and test for several other classes of bad data.
- Typical bad-data test cases include
  - Too little data (or no data)
  - Too much data
  - The wrong kind of data (invalid data)
  - The wrong size data
  - Uninitialized data

Data Flow Testing: *Classes of Good Data*

- When you try to find errors in a program, it's easy to miss the fact that the main case might contain an error.
- Following are other kinds of good data that are worth checking. Checking each of these kinds of data can reveal errors, depending on the item being tested:
  - Nominal cases—middle-of-the-road, expected values
  - Minimum normal configuration
  - Maximum normal configuration
  - Compatibility with old data

Use Test Cases That Make Hand-Checks Convenient

- Let's suppose you're writing a test case for a nominal salary; you need a nominal salary, and the way you get one is to type in whatever numbers your hands land on.
  - Testing $90,783.82
- Now, further suppose that the test case succeeds—that is, it finds an error. How do you know that it's found an error? Well, presumably, you know what the answer is and what it should be because you calculated the correct answer by hand.
- Better to use numbers that can be hand checked, e.g. $90,000

Development Testing: *Limitations*

Developer tests tend to be "clean tests"

- Developers tend to test for whether the code works (clean tests) rather than test for all the ways the code breaks (dirty tests).
- Immature testing organizations tend to have about five clean tests for every dirty test.
- Mature testing organizations tend to have five dirty tests for every clean test. This ratio is not reversed by reducing the clean tests; it's done by creating 25 times as many dirty tests (Boris Beizer in Johnson 1994).

Development Testing: *Limitations*

Developer testing tends to have an optimistic view of test coverage

- Average programmers believe they are achieving 95 percent test coverage, but they're typically achieving more like 80 percent test coverage in the best case, 30 percent in the worst case, and more like 50-60 percent in the average case (Boris Beizer in Johnson 1994).

Development Testing: *Limitations*

Developer testing tends to skip more sophisticated kinds of test

- Most developers view the kind of test coverage known as “100% statement coverage” as adequate.
- This is a good start, but it’s hardly sufficient.
- A better coverage standard is to meet what’s called “100% branch coverage,” with every predicate term being tested for at least one true and one false value.
- What is the ultimate coverage? Input + Path Coverage but its inteactable!

Which Classes Contain the Most Errors?

- It's natural to assume that defects are distributed evenly throughout your source code.
- If you have an average of 10 defects per 1000 lines of code, you might assume that you'll have one defect in a class that contains 100 lines of code.
- This is a natural assumption, but it's **wrong**.

Which Classes Contain the Most Errors?

- It was reported that a program at IBM identified 31 of 425 classes are error-prone.
- The 31 classes were repaired or completely redeveloped, and, in less than a year, customer-reported defects were reduced ten to one.
- Total maintenance costs were reduced by about 45 percent.
- *Customer satisfaction improved from "unacceptable" to "good"* (Jones 2000).

Which Classes Contain the Most Errors?

- Most errors tend to be concentrated in a few highly defective routines. Here is the general relationship between errors and code:
  - 80% of the errors are found in 20% of a project's classes or routines (Endres 1975, Gremillion 1984, Boehm 1987b, Shull et al 2002).
  - 50% of the errors are found in 5% of a project's classes (Jones 2000).

Which Classes Contain the Most Errors?

- *These relationships might not seem so important until you recognize a few conclusions*.
- **First**, 20% of a project's routines contribute 80% of the cost of development (Boehm 1987b). *That doesn't necessarily mean that the 20% that cost the most are the same as the 20% with the most defects, but it's pretty suggestive*.
- **Second**, regardless of the exact proportion of the cost contributed by highly defective routines, highly defective routines are extremely expensive.

Which Classes Contain the Most Errors?

- In a classic study in the 1960s, IBM performed an analysis of its OS/360 operating system and found that errors were not distributed evenly across all routines but were concentrated into a few.
- Those error-prone routines were found to be “the most expensive entities in programming” (Jones 1986a).
- They contained as many as 50 defects per 1000 lines of code, and fixing them often cost 10 times what it took to develop the whole system (The costs included customer support and in-the-field maintenance.)

Which Classes Contain the Most Errors?

- **Third**, the implication of expensive routines for development is clear.
- If you can cut close to 80% of the cost by avoiding troublesome routines, you can cut a substantial amount of the schedule as well.
- This is a clear illustration of the General Principle of Software Quality: *improving quality improves the development schedule and reduces development costs.*

Which Classes Contain the Most Errors?

- **Fourth**, the implication of avoiding troublesome routines for maintenance is equally clear.
- Maintenance activities should be focused on identifying, redesigning, and rewriting from the ground up those routines that have been identified as error-prone.
- In the IBM project mentioned earlier, productivity of the product releases improved about 15% after replacement of the error-prone classes (Jones 2000).

Release Testing

- Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
  - Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a black-box testing process where tests are only derived from the system specification.

Release Testing and System Testing

- Release testing is a form of system testing.
- Important differences:
  - A separate team that has not been involved in the system development, should be responsible for release testing.
  - System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

Performance Testing

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Tests should reflect the profile of use of the system.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- **Stress testing** is a form of performance testing where the system is deliberately overloaded to test its failure behavior.

User Testing

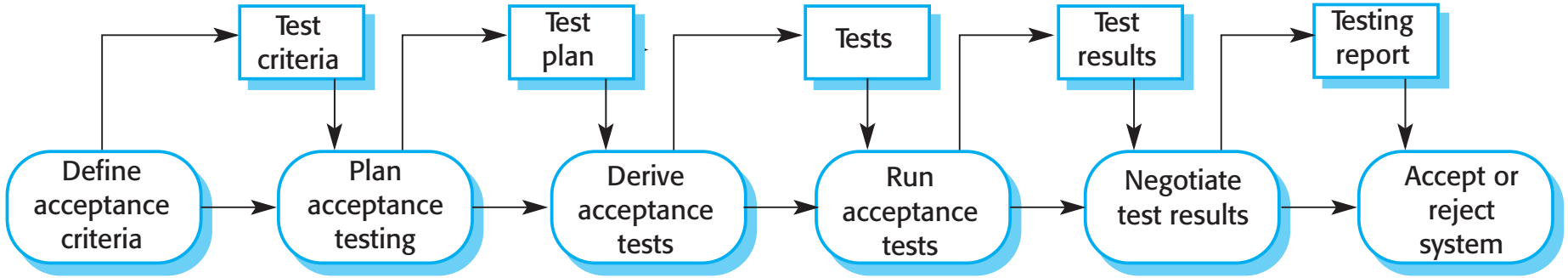
- User Testing *where users or potential users of a system test the system in their own environment*
- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

Types of User Testing

- Alpha testing
  - Users of the software work with the development team to test the software at the developer's site.
- Beta testing
  - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- Acceptance testing
  - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

Acceptance Testing

- Acceptance testing is the execution of the software after it is released, by the customer
  - After the system test has corrected all or most defects, the system will be delivered to the user or customer for acceptance testing.



Agile Methods and Acceptance Testing

- In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- There is no separate acceptance testing process.
- Main problem here is whether or not the embedded user is ‘typical’ and can represent the interests of all system stakeholders.

Smoke Testing (Build Verification Testing)

- A common approach for creating “daily builds” for product software
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a “build.”
    - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
    - The intent should be to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
    - The integration approach may be top down or bottom up.

Comparison Testing

- Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
  - Separate software engineering teams develop independent versions of an application using the same specification
  - Each version can be tested with the same test data to ensure that all provide identical output
  - Then all versions are executed in parallel with real-time comparison of results to ensure consistency

Model-Based Testing

- Analyze an existing behavioral model for the software or create one.
  - Recall that a behavioral model indicates how software will respond to external events or stimuli.
- Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.
  - The inputs will trigger events that will cause the transition to occur.
- Review the behavioral model and note the expected outputs as the software makes the transition from state to state.
- Execute the test cases.
- Compare actual and expected results and take corrective action as required.

MobileApp Testing

- User experience testing – ensuring app meets stakeholder usability and accessibility expectations
- Device compatibility testing – testing on multiple devices
- Performance testing – testing non-functional requirements
- Connectivity testing – testing ability of app to connect reliably
- Security testing – ensuring app meets stakeholder security expectations
- Testing-in-the-wild – testing app on user devices in actual user environments
- Certification testing – app meets the distribution standards

つづく