COMP S350F / S356F CHAPTER 10 - TESTING

Oliver Au, The Open University of Hong Kong
(Slides updated on January 25, 2016)

Chapter Learning Objectives

After studying the chapter, you should be able to:

- Explain the importance and limitations of testing
- Perform code review to improve software quality
- Describe types of testing - unit testing, integration testing and so on
- Apply basic testing techniques - equivalence testing, boundary testing, path testing and state-based testing
- Write test cases
- Perform automated unit testing with JUnit
- Apply Test-Driven Development (TDD)
- **Apply advanced testing technique - pairwise testing**

Poor Software on Set Top Box

- I bought my first HDTV and set top box in 2008.
- I bought an LG TV (mid to high end) and TopCon set top box (low end).
- The LG TV shows great colour in high resolution.
- The TopCon set top box
  - often hung during fast forward when I skipped commercials (this problem has gone)
  - occasionally fail to record when I am watching another channel
  - has an inconsistent UI on going up and down the channels
  - only achieve high-resolution in interlaced mode
- As a consumer, I will avoid *TopCon* in the future at all costs.

Poor software on Medical Devices & CPUs

- Therac-25 uses radiation on patients to kill cancer cells.
- First made in 1983, a total of 11 machines were installed in 1987.
- Programs were written in assembly language on PDP-11.
- Multiple threads accessing shared variables were not synchronised correctly leading to data corruption. But the problem did not happen every time making it hard to trace.
- Patients might receive radiation many times higher than the intended dosage. (Problems of the machines were not discovered quickly because many cancer patients would die anyway.)
- Safety devices were later refitted to ensure unsafe dosage are forbidden.
- Intel processors made CPUs numeric model numbers 8086, 80286, 80386 and 80486 before Pentium.
- In 1986, Intel sold over 100,000 units of 80386 before it was discovered that the processor can make mistakes multiplying certain numbers.

Poor Software on Electricity Network

The worst power outage in history happened on August 14th, 2003.

- 55 million people in U.S. and Canada were without electricity for up to two days.
- They have to cook food using barbecue. Coals and gas replaced electricity. (Due to the extreme cold weather in North America, most homes were built relatively airtight. Gas cooking requires fresh air. Therefore home cooking more commonly use electricity.)
- Partly caused by a bug in General Electric (GE) energy management system http://en.wikipedia.org/wiki/Northeast_Blackout_of_2003
- Fortunately, the outage did not happen in Winter. Otherwise home heating could not work and some people would be frozen to death. (Even home heating furnaces often use gas, they still need electricity to circulate heated air in the house.)

Limitations of Testing

- The GE energy management system has been executed for a total of 340 computer-years, equivalent to 34 computers running the system for 10 years or 68 computers running the system for 5 years.
- How could the bug hide itself in the pre-production testing and the production use of 340 computer-years? (The word "production" refers to real usage of the software not testing.)
- In our examples of the set top box, Therac-25 and power system all have parallel programs running multiple threads or processes. They are much harder to test than sequential programs.
- Testing is necessary but not sufficient to achieve the best quality.

Why Software Testing is Difficult?

- For sequential systems, there would be too many different test input data to try exhaustively.
- For concurrent systems, the same input test data do not produce the same output every time. Timing adds an unpredictable dimension to testing. Nearly all computing devices today are concurrent systems including the smartphones.
- Testing shows the presence, not the absence of bugs. (Edsger W. Dijkstra, 1930 - 2002)
- Ensuring quality while building a product may be more effective than adding quality after the product has been built.

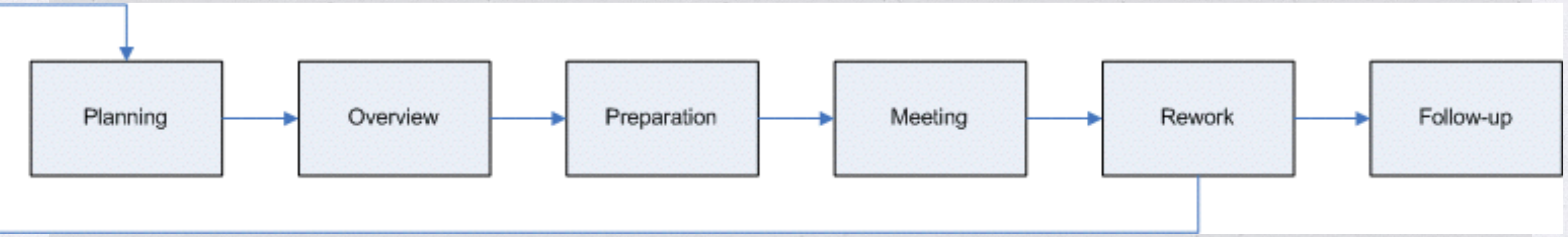
Code Review

- A few persons get together to study program code with the goal to improve its quality.
- This is called **code review**. It can improve code quality without testing.
- When **casually** done, reviews are called **walkthrough**.
- When **rigorously** done, reviews are called **inspection**.
- We can review anything created by the software development team: program code, documents, project plans, and other artefacts (work products).
- Some people are not convinced that code review is the most cost-effective means to improve software quality so it is not used in all software projects.

Fagan Inspection

There are three main roles in the Fagan Inspection.

- The programmer/author whose work is being inspected.
- The moderator coordinates the inspection, e.g. calls the meeting, books the meeting room and distributes inspection materials.
- Other participants inspect the work by applying their skills and experiences.



Steps in Fagan Inspection

1. **Planning** - The moderator notifies the participants the time and place of the inspection meeting. He or she assembles and distributes the inspection materials with the help from the programmer/author whose work is being inspected. (Participants check the code quality and see if it corresponds to the specification.)

2. **Overview** - The author presents his or her work to participants.

3. **Preparation** - Participants review the item being inspected by referring to the supporting materials whenever necessary.

4. **Inspection meeting** - Participants identify things that need improvements.

5. **Rework** - The programmer revise the work based on reviewers' feedback.

6. **Follow-up** - The moderator ensures that the revision has been done as required.

Types of Software Faults

- **Requirement faults**
  - Customers omitted requirements when communicating to developers.
  - Customers use ambiguous language in their communication with developers.
  - A requirement contradicts with other requirements or external constraints.
- **Procedural faults** - We have not adequately inform users how to use the application.
- **Hardware faults** - We want to able to recover fully or perform at a degraded level on hardware faults.
- **Coding faults** - Programmers are solely responsible for them.

Types of Testing

- Unit testing
- Integration testing
- Functional testing
- Nonfunctional testing
  - Performance testing
  - Load testing
  - Stress testing
- User acceptance testing (UAT)
- Installation testing
- Usability testing
- Regression testing

Unit Testing

- Unit testing tries to see if units (e.g. classes or functions) **work correctly in isolation**.
- Programmers are responsible for unit testing. They should be experts.
- Unit testing is often automated. The test program would call a unit and compare the returned result with the expected result.
- Renown unit testing frameworks include SUnit for Smalltalk, JUnit for Java, PHPUnit for PHP, PyUnit for Python, CppUnit for C++, CUnit for C and NUnit for .Net. Some languages may have more than one unit testing framework.
- Unit testing is the earliest kind of testing to do. Test-Driven Development (TDD) practitioners even write unit tests before they write the application programs.

Integration Testing

- Integration testing happens right after unit testing.
- Integration testing makes sure that units **work well together**.
- In **bottom-up integration testing**, small units (components) are combined one-by-one until the complete system is built.
- In **top-down integration testing**, you build a system with many test stubs. Once a unit is completed, we use it to replace a test stub. Eventually, the complete system is built when all test stubs are replaced by real code.
- Consider *getTotal*( ) function that calculates the total amount of items checked through a POS (Point-of-sale) terminal. In top-down integration testing, we may have a test stub for that function hard-coded to return a value of $300. After *getTotal*( ) function is written and unit tested, we replace the test stub with this real unit.
- We are ready to do integration testing with *getTotal*( ) function. The system will return amount based on the actual products being checked out (not $300 all the time as before).

Functional and Nonfunctional Testing

- **Functional testing** checks the conformance of an integrated system against the function specification. It happens after integration testing.
- Now that we know a system is functional, we also want to check if it performs fast enough. This is called **performance testing**. For example, can the system return a response within 2 seconds when 100 users are logged on?
- **Load testing** tries to put demand on a system, The purpose is to determine the maximum capacity of the system and identify its bottleneck.
- **Stress testing** puts the system beyond its normal operational capacity. The purpose is to determine the safe usage limit of the system. We will make sure that the limit agrees with the requirements specification.
- Load testing and stress testing belong to the family of performance testing which test the system according to its nonfunctional (quality) requirements. Therefore they are also called **nonfunctional testing**.

User Acceptance Testing (UAT) and Installation Testing

- In **user acceptance testing**, real users decide if the system is acceptable for production use.
- http://www.exforsys.com/tutorials/testing/what-is-user-acceptance-testing.html has more.
- A system can run in one of three environments:
  - Development environment for programmers
  - Test environment for testers
  - Production environment for intended users
- **Installation testing** goes through the installation procedure to install the system. It could be manually done, automated or a combination of the two.
- If the installation procedure does not work as planned, it will need to be modified.

Usability Testing

Usability testing ensures user friendliness in the 3 aspects below. (Items 2 and 3 have different meanings in other contexts.)

1. **Learnability** - How easy it is for the user to learn to use the system? It can be measured by the time for a typical user to learn to perform a task.

2. **Throughput** - How long does it take a user to complete a task? It measures user productivity using the application. The back button on some query webpage clearing out all my previous input is a poor design. It does not allow me to change the previous query slightly.

3. **Robustness** - How easy for a user error to be caught and corrected? A system poor in robustness may require me to reenter all my input data even only one field was incorrect.

Who, How and What of Usability Testing

- **Who** - Usability should be conducted by prospective users who have not been involved in the UI design.
- **How** - You can ask users to fill out questionnaires or observe how users actually use the system. Each approach has its strengths and weaknesses.
- **What** - Usability testing on the completed system is most accurate but improvements at that point will be expensive. Usability testing on prototypes is more economical and less risky.

Regression (復原) Testing

- After a program is modified to add a feature or fix a bug, we may inadvertently (不慎地) introduce new bugs.
- The tests that we repeat after program modifications are called **regression tests** which are good candidates for automation because their repeated use increase the chance of recovering the development costs.
- Automated testing allow you to economically rerun many tests after modifying the products. Unit tests are the easiest to automate.
- The testing code written specially to automate testing is called a test script (測試劇本). It will be followed exactly by the computer.

Oliver at IBM

- IBM worked at IBM from 1987 to 1992.
- When he left IBM, he was a development analyst at IBM Toronto Lab which specialised in 3 areas: AS/400, databases and compilers.
- Oliver worked in the C compiler front-end team. The front end team is responsible for generating intermediate code from C programs. The backend team is responsible for optimising the intermediate code and generating the executable code for the target platform.
- The size of testing team is only slightly smaller than the development team that includes front-end and back-end.
- When Oliver was a student, he would not consider any testing job because he thought it is not challenging enough and requires no programming skills. Of course, he was wrong (because he did not take any software engineering course in his undergraduate years).
- After this experience at IBM, Oliver has great respect for testers.
- The testers wrote hundreds or thousands of test cases to test the compiler.
- Some testers are very good programmers. They just have a different objective which is to break a system. They write many C programs to make sure that the compiler works properly. They don't just write programs randomly. They studied the C definition and looked for places that would likely cause problems to C compilers.
- After writing a fix to the compiler, we would start the automated test to come back next morning to see the result of the regression tests.

Types of Testing (1 of 2)

Types	Who	Objective
Unit test	Programmer	Components (classes in Java and files in C) work correctly by themselves in isolation
Integration test	Tester	Components work well after being integrated with other components
Functional test	Tester	A completely integrated system meets the functional requirements
Performance test	Tester	A completely integrated system meets the performance requirements that mainly concern **response time** and **throughput**. We may also deal with scalability and reliability in performance testing.

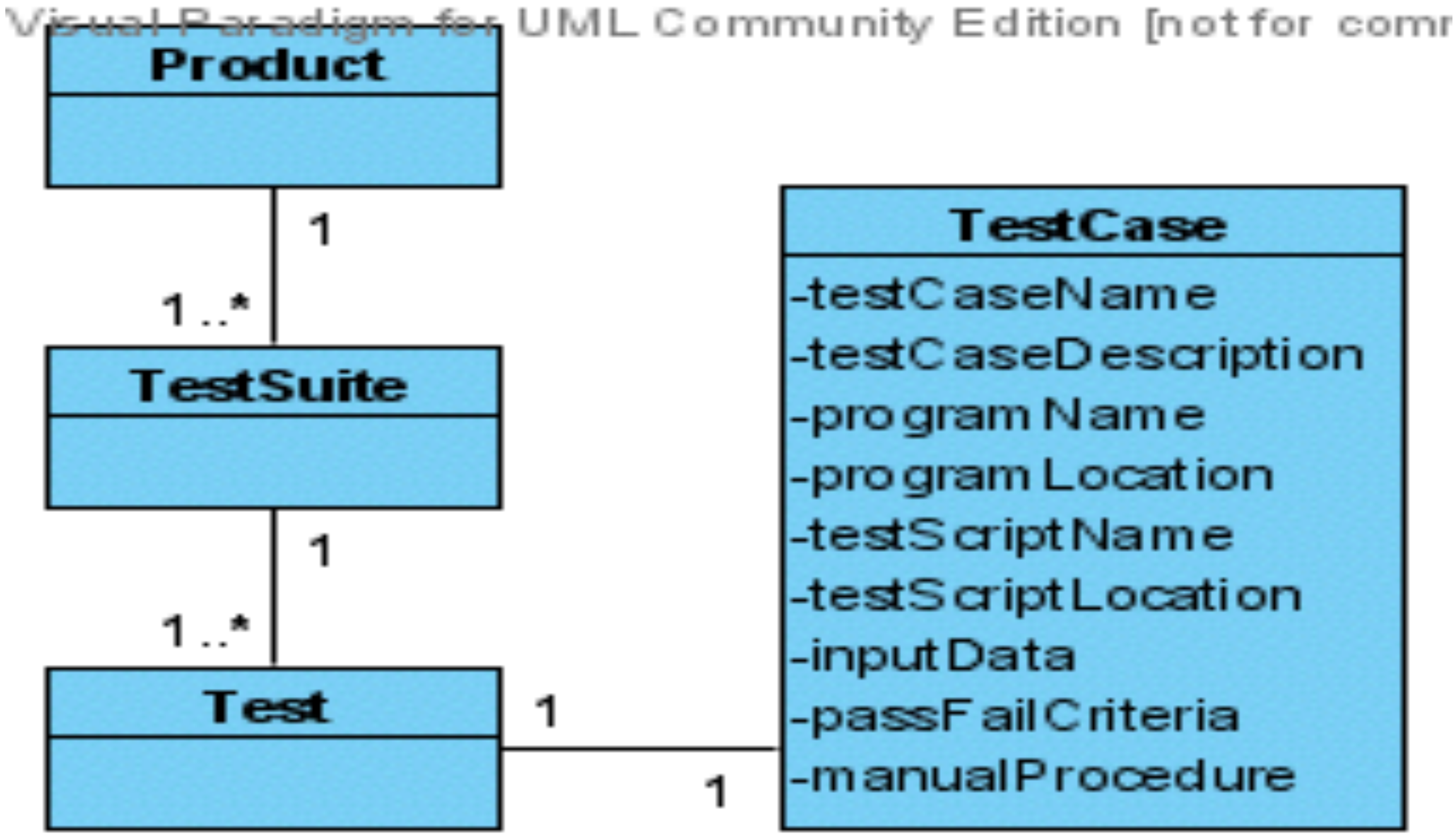
Types of Testing (2 of 2)

Types	Who	Objective
User acceptance test (UAT)	User	Real users determine if the system is acceptable to them.
Installation test	Multiple parties	The installation procedures work as expected.
Usability test	Prospective user	The system can be learned easily (**learnability**), be used productively (**throughput**) and catch/correct user errors effectively (**robustness**).
Regression test	Programmer / tester	Existing features continue to work after code modification. Regression tests are good candidates for automation.

Testing Terminology

- A **test suite** is a collection of test cases.
- A **test case** documents how the **system under test** (SUT) behaves in a particular situation.
- A **test script** contains procedural instructions to perform in a test. Depending on the context, automated instead of manual scripts may be implied.
- An **oracle** is a mechanism to determine if a test has passed or failed. It tells us what output to expect from particular input data.
    - The oracle of the C programming language is a document with several hundred pages. The drawback is that it is challenging to unambiguously interpret the document.
    - In the case of the Ruby programming language, its inventor Matz uses a **reference implementation** as the oracle. He wrote the Matz's Ruby Interpreter (MRI). Perhaps you want to create a Ruby compiler that run Ruby programs faster than MRI. After you have created the compiler, you can compare it to the MRI. If both your compiler and the MRI work the same for all Ruby programs, you can say that your compiler is a correct Ruby implementation. The MRI instead of a document is the oracle for Ruby.

Test Cases


Visual Paradigm for UML Community Edition [not for comm
Product
1
1..*
TestSuite
1
1..*
Test
1
1
TestCase
-testCaseName
-testCaseDescription
-programName
-programLocation
-testScriptName
-testScriptLocation
-inputData
-passFailCriteria
-manualProcedure

Two Categories of Testing

1. **Black box testing** is done based on the program behaviour only. It regards the system under test (SUT) as a black box. The testing focuses on what output is produced from what input.

2. **White box testing** is done based on the program structure of the SUT, e.g. data structures and algorithms. We still care about what input is producing what output but we design our input with the help of our knowledge of the program structure.

- When we explain testing techniques in later slides, see if you can tell which ones are black box and which ones are white box. When people hire software testers or engineers, they may ask similar questions.

Equivalence Testing

- Also called **equivalence partitioning** (分割) because it partitions (divides) test data into different categories.
- Partitioning aims to cover testing thoroughly by taking one input from each partition of test data.
- Test data should be at least divided into successful and unsuccessful but in practice we should have more partitions.
- Consider the example of a password update function where passwords of 6 to 12 characters are allowed.
    - First we divide them into two big categories: valid passwords and invalid passwords.
    - From the two main categories, we have 10 partitions in total as shown next.

The Partitions

The **valid passwords** can be divided into seven categories according to their lengths:

- 6-char passwords
- 7-char passwords
- 8-char passwords
- 9-char passwords
- 10-char passwords
- 11-char passwords
- 12-char passwords

The **invalid passwords** can be divided into more categories:

- Too short – passwords with 0, 1, 2, 3, 4 and 5 characters
- Too long – passwords with 13, 14, 15 characters and so on
- Contains invalid characters, for example spaces

Concluding Equivalence Testing

- For valid passwords, we can have 7 partitions.
- For passwords that are too short, we can have up to 6 partitions.
- For passwords that are too long, we can have infinitely many test cases. Or we have can 8 partitions by focusing on oversized passwords with 13 to 20 characters.
- To thoroughly test our password checking functions, we can use 7 + 6 + 8 = 21 partitions and test cases.
- If we treat all the passwords that are too short as one partition and all the passwords that are too long as one partition, we may have 7 + 1 + 1 = 9 partitions and test cases.
- The same equivalence testing techniques can give rise to different number of partitions.

Boundary Testing

- In boundary testing, we do not just take any value from a partition to test. We **choose a value at or near the boundary of a partition**.
- Suppose we have just divided our password input data into 3 main equivalence partitions each with smaller partitions.
  - too short - 0, 1, 2, 3, 4 or 5 characters (one partition instead of 6)
  - valid length - 6, 7, 8, 9, 10, 11 or 12 characters (one partition instead of 7)
  - too long - 13, 14, 15, 16, 17, 18, 19 or 20 characters (one partition instead of 8)
- If we use equivalence testing alone, we would only have 3 test cases.
- Programming errors are more likely to happen near boundaries. For example, when writing loops, programmers may mistaken <= for < or 0 for 1 in a condition.
  - while i <= 0 do …
  - while i < 0 do …
- By combining **boundary testing with equivalence testing**, we would select passwords with:
  - 0 and 5 characters at the boundaries of the “too short” partition
  - 6 and 12 characters at the boundaries of the “valid length” partition
  - 13 characters at the boundary of the “too long” partition
- With just 5 test cases, we are quite thorough except for the “invalid characters” partition.

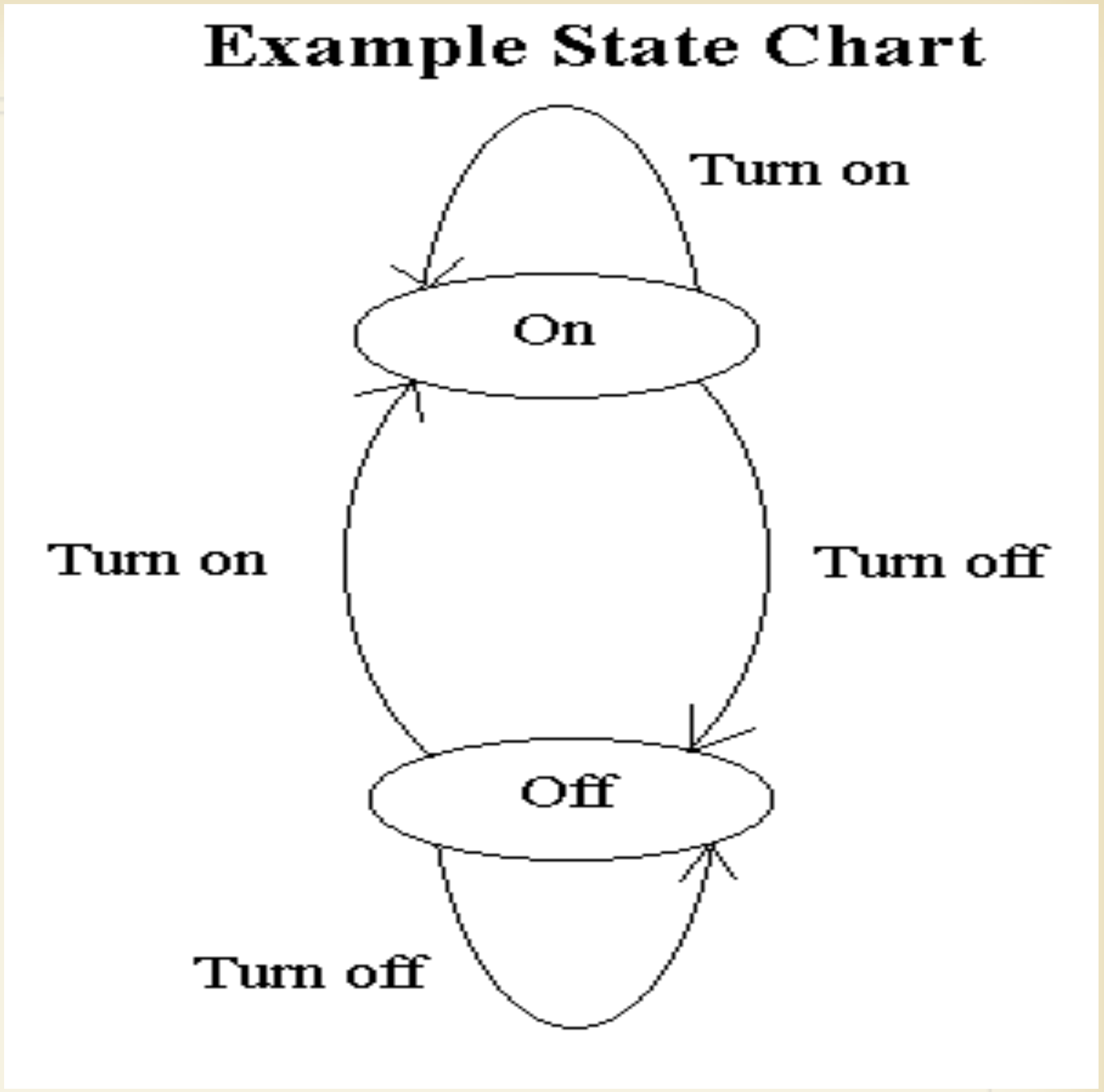
Path Testing

```
if (x > 0)
    A1
else
    A2


if (y > 0)
    B1
else
    B2
```

- Path testing ensures all possible paths in a program are covered. It uses knowledge of program structure so it is a whitebox testing technique.
- The execution path of the program can take one of the followings based on the values of X and Y.

```
1. A1; B1 (x= 1 and y= 1)
2. A1; B2 (x= 1 and y= -1)
3. A2; B1 (x= -1 and y= 1)
4. A2; B2 (x= -1 and y= -1)
```

State-based Testing



- State-based testing ensures that all possible states of an object are covered. It is a whitebox testing technique.
- We demonstrate this testing with the help of a state chart. It is not drawn according to the standard UML state diagram notation because it uses ovals instead of round-cornered rectangles to represent states.

Deriving Test Cases

- A scenario for cash withdrawal on an ATM contains a flow of events with actual data values, e.g. the customer is “Peter Chan” and the withdrawn amount is “$300”.
- A use case is a generalisation of several related scenarios. A scenario uses different values for *customer* and *withdrawn amount* to create test cases.
- We can code an executable test script for each test case derived from a scenario.

RegisterCourses Use Case

Use Case Name: RegisterCourses (main flow)

Actor: Student

Flow:

1. Student logs on with student id and password
2. System displays a list of functions
3. Student selects the "register courses" function
4. System displays a course catalogue *
5. Student selects a course to add/change/drop *
6. System accepts the addition/change/drop *

Pre-condition: The student account is valid

Post-condition: Course registration upodated for the student

Asterisks denote repeatable events.

Test Case Description

Test Case: RC1

Description: Student adds a few courses

Flow of Events (input iata are underlined):

1. Student logs on with id <u>s12345678</u> and password <u>mysecret</u>.
3. Student selects <u>“register courses”</u> from the main menu.
5. Student selects to <u>add</u> <u>COMP S260F</u>
5. Student selects to <u>add</u> <u>COMP S311F</u>
5. Student selects to <u>add</u> <u>COMP S356F</u>
5. Student selects to <u>add</u> <u>COMP S358F</u>

Pass Criteria:

The four courses appear in the designated screen area.

The four courses are added for s12345678 in the database.

Another Test Case Example

Test Case: RC2

Description: Student changes two courses

Flow / (Input Data Underlined):

1. Student logs on with id s68686868 and password nola.
3. Student selects "register courses" from the main menu.
5. Student changes COMPS411F to COMP S311F
5. Student changes COMPS451F to COMP S356F

Pass Criteria:

The two old courses are changed to two new courses in the designated screen area.

The two courses are updated for s68686868 in the registration database.

Alternatives in Steps 1 & 5

- The main step 1 is logon successfully.
Alternatives are:

(a) System not available

(b) Id does not exist

(c) Password incorrect

- The main step 5 is to add a course. Alternatives
are:

(a) Change a course

(b) Drop a course

Test Matrix

Step / TestId	1	1 a	1 b	1 c	2	2 a	2 b	3	4	4 a	5	5 a	5 b	6	6 a	6 b	6 c	Description (He = Student)
RC1	X				X			X	X		X			X				He registers 4 courses
RC2	X				X			X	X			X		X				He changes 2 courses
RC3				X														He forgot password
RC4	X					X												He tries to register a day after the registration period has closed.
RC5	X			X	X			X		X								System crashes after he has chosen “register courses” function.
RC6	X				X			X			X			X	X			The second course he tries to add is full already. The attempt fails..

Traceability Matrix / Test Log

Requirements / Test Case	REQ 1.1	REQ 1.2	REQ 1.3	REQ 2.1	Outcome
RC1	x				2014-01-08 succeeded
RC2		X			
RC3			X		
PF1				X	

The UserSystem Class (1 of 2)

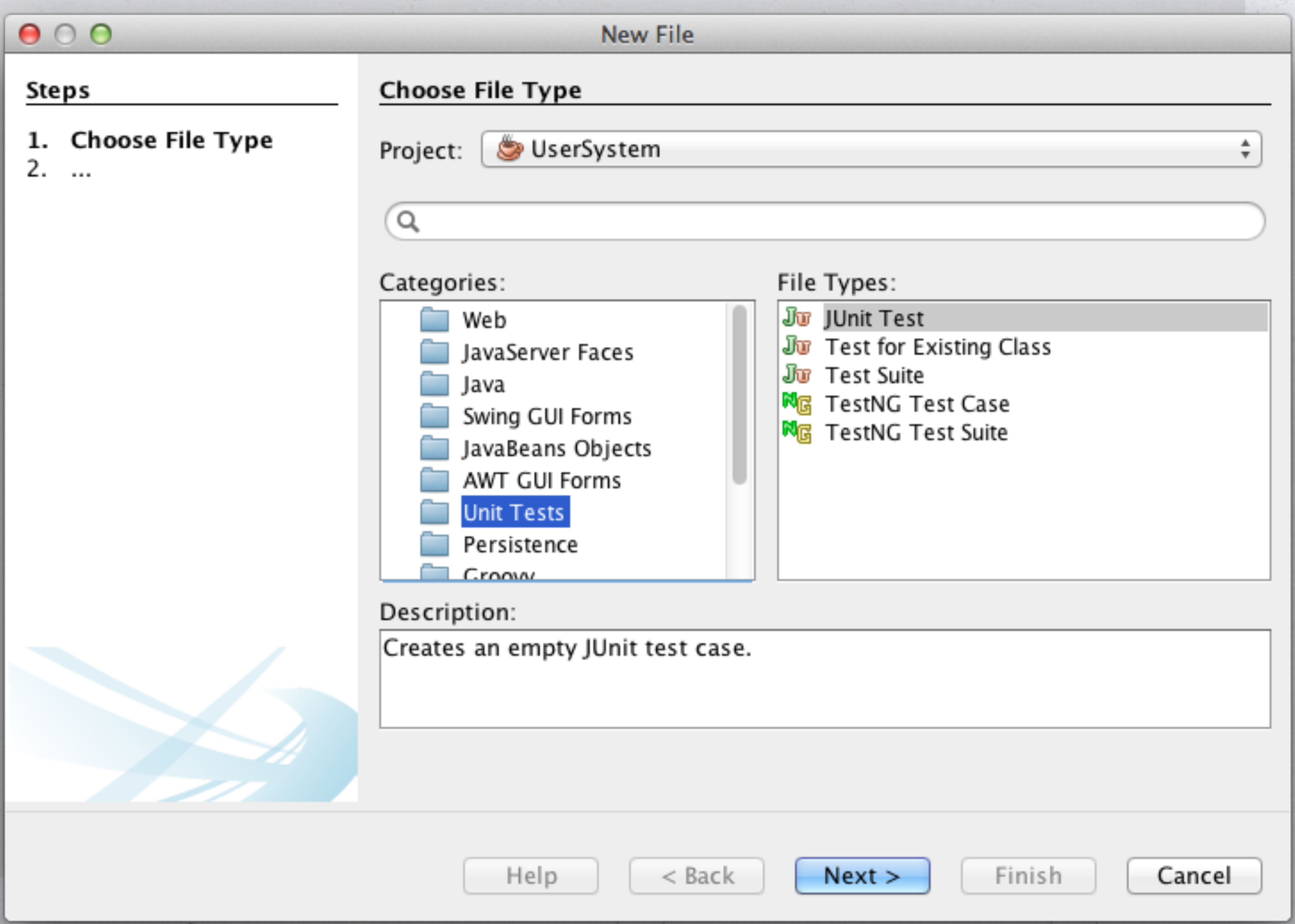
```
package usersystem;

import java.util.* ;
public class UserSystem {
    public HashMap acTable = new HashMap();
    public boolean add( String id, String pswd) {
        String password = (String) acTable.get(id);
        // Before adding new id, make sure it does not exist
        if (password == null) {
            acTable.put( id, pswd);
            return true;
        }
        else
            return false;
    }
```

The UserSystem Class (2 of 2)

```
    public boolean logon( String id, String pswd) {
        String password = (String) acTable.get(id);
        if (password == null)
            return false; // id not found
        else
            // see if entered pswd equals retrieved pswd
            return password.equals(pswd);
    }

    public boolean setPassword( String id, String pswd) {
        String password = (String) acTable.get(id);
        if (password == null)
            // Cannot set password if id does not exist
            return false;
       else {
            acTable.put( id, pswd);
            return true;
        }
    }
}
```

Create JUnit Test File

- Install JUnit plugin on your favourite IDE, NetBeans or Eclipse.
- Create a test file for JUnit. A skeleton will be provided.



Creates an empty JUnit test case.

JUnit - Import

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
import usersystem.UserSystem;

public class LogonTest {

    UserSystem mySystem;

    public LogonTest() {
        mySystem = new UserSystem();
    }
```

JUnit - Test Initialisation & Cleanup

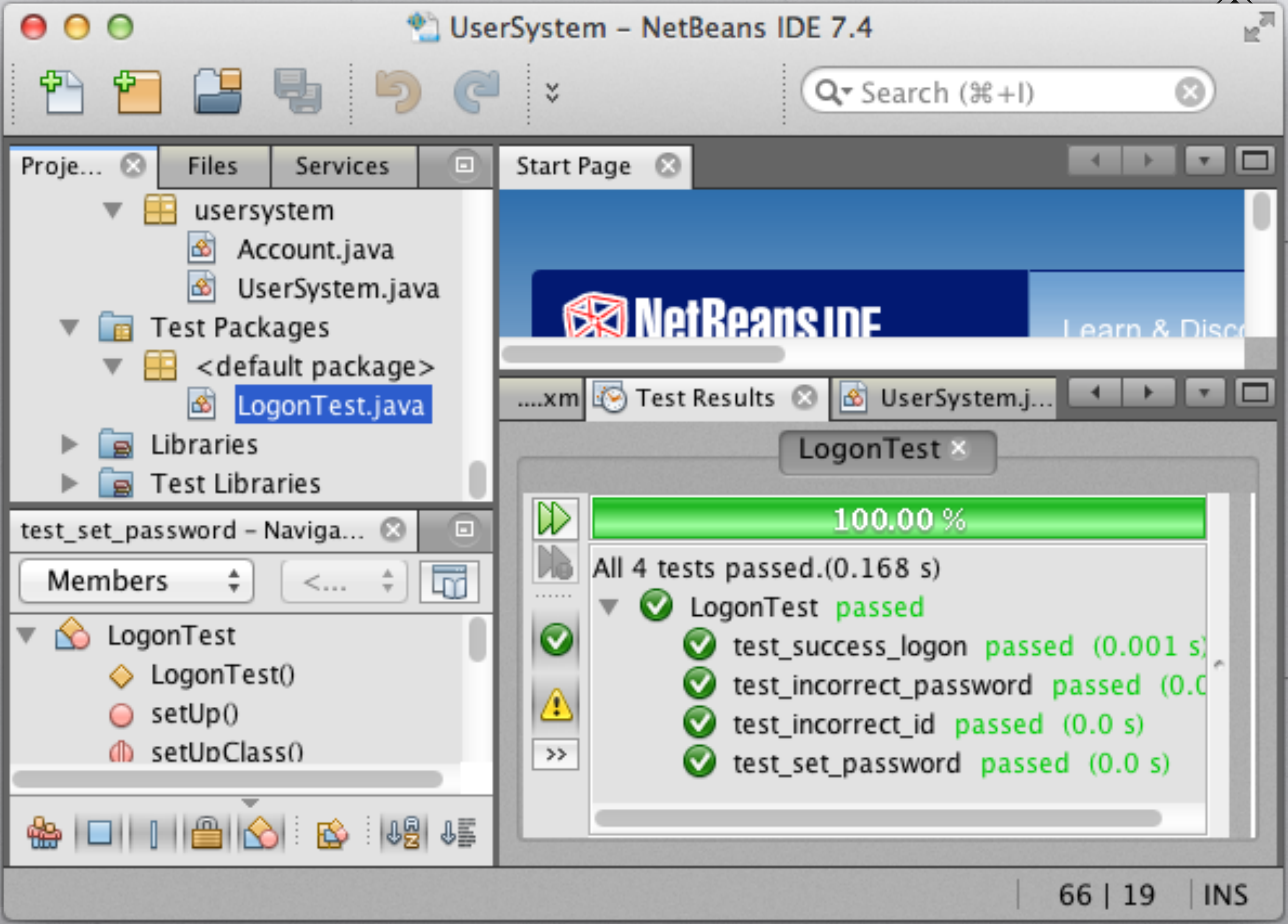
```
@BeforeClass
public static void setUpClass() {
}
@AfterClass
public static void tearDownClass() {
}
@Before
public void setUp() {
    mySystem.add( "cudie", "7there59");
    mySystem.add( "francis", "hello246");
    mySystem.add( "meng", "family");
}
@After
public void tearDown() {
}
```

JUnit Tests

```
@Test
public void test_success_logon() {
    assertTrue(mySystem.logon("francis", "hello246"));
}
@Test
public void test_incorrect_password() {
   assertFalse(mySystem.logon("francis", "goodbye"));
}
@Test
public void test_incorrect_id() {
   assertFalse(mySystem.logon("jasper", "alberta"));
}
@Test
public void test_set_password() {
    assertTrue(mySystem.logon("francis", "hello246"));
    assertTrue(mySystem.setPassword("francis", “new13579"));
    assertFalse(mySystem.logon("francis", "hello246"));
    assertTrue(mySystem.logon("francis", "new13579"));
}
```

Run JUnit Test File

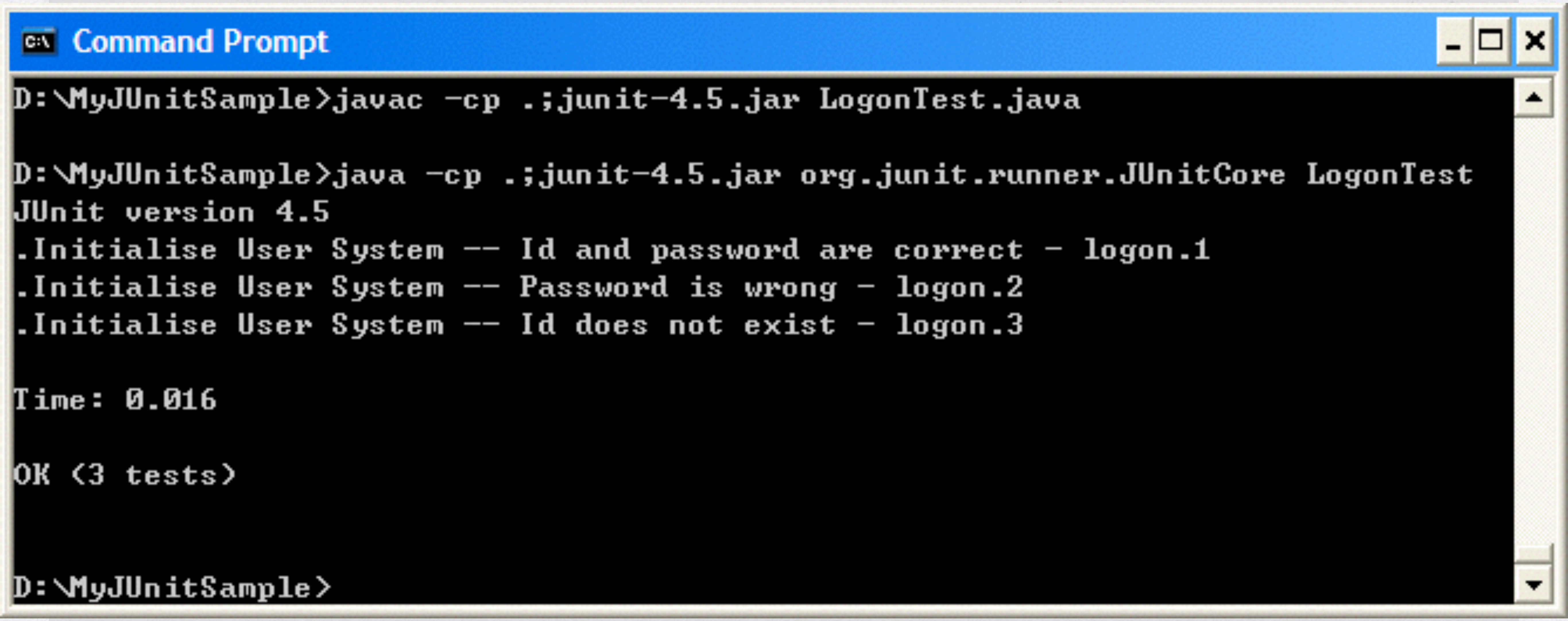
- “Right-click on your test file.
- From dropdown menu, select “Run File”.
- Click ▶ to expand on all tests in “LogonTest”.



```
100.00 %
All 4 tests passed.(0.168 s)
LogonTest passed
    test_success_logon passed (0.001 s)
    test_incorrect_password passed (0.0
    test_incorrect_id passed (0.0 s)
    test_set_password passed (0.0 s)
```

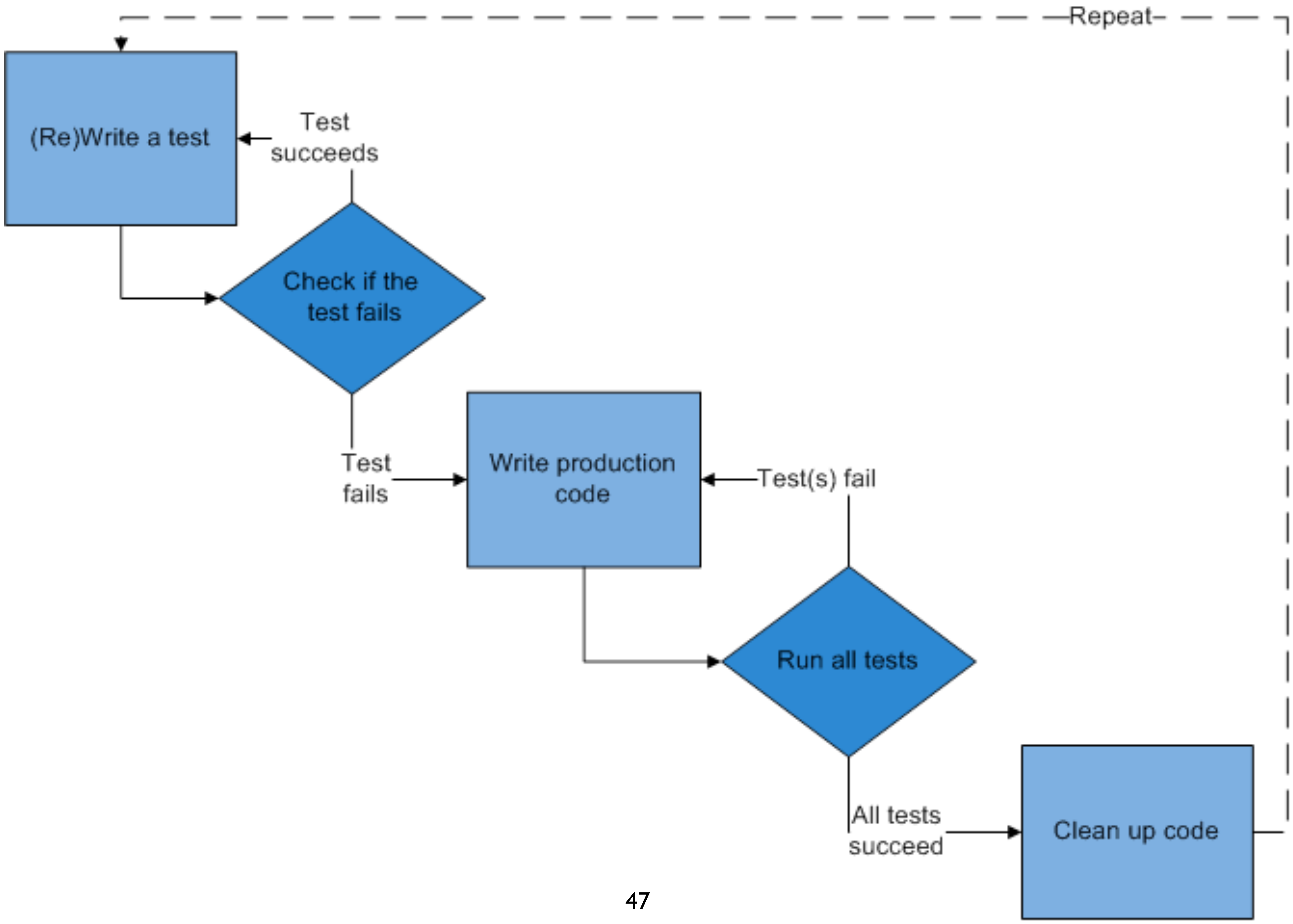
Run JUnit from Command Prompts

* Shown are Windows commands. Mac OS X commands are different.



```
D:\MyJUnitSample>javac -cp .;junit-4.5.jar LogonTest.java

D:\MyJUnitSample>java -cp .;junit-4.5.jar org.junit.runner.JUnitCore LogonTest
JUnit version 4.5
.Initialise User System -- Id and password are correct - logon.1
.Initialise User System -- Password is wrong - logon.2
.Initialise User System -- Id does not exist - logon.3

Time: 0.016

OK (3 tests)


D:\MyJUnitSample>
```

Test-Driven Development (TDD)



TDD Step 1
Write Test for New Feature

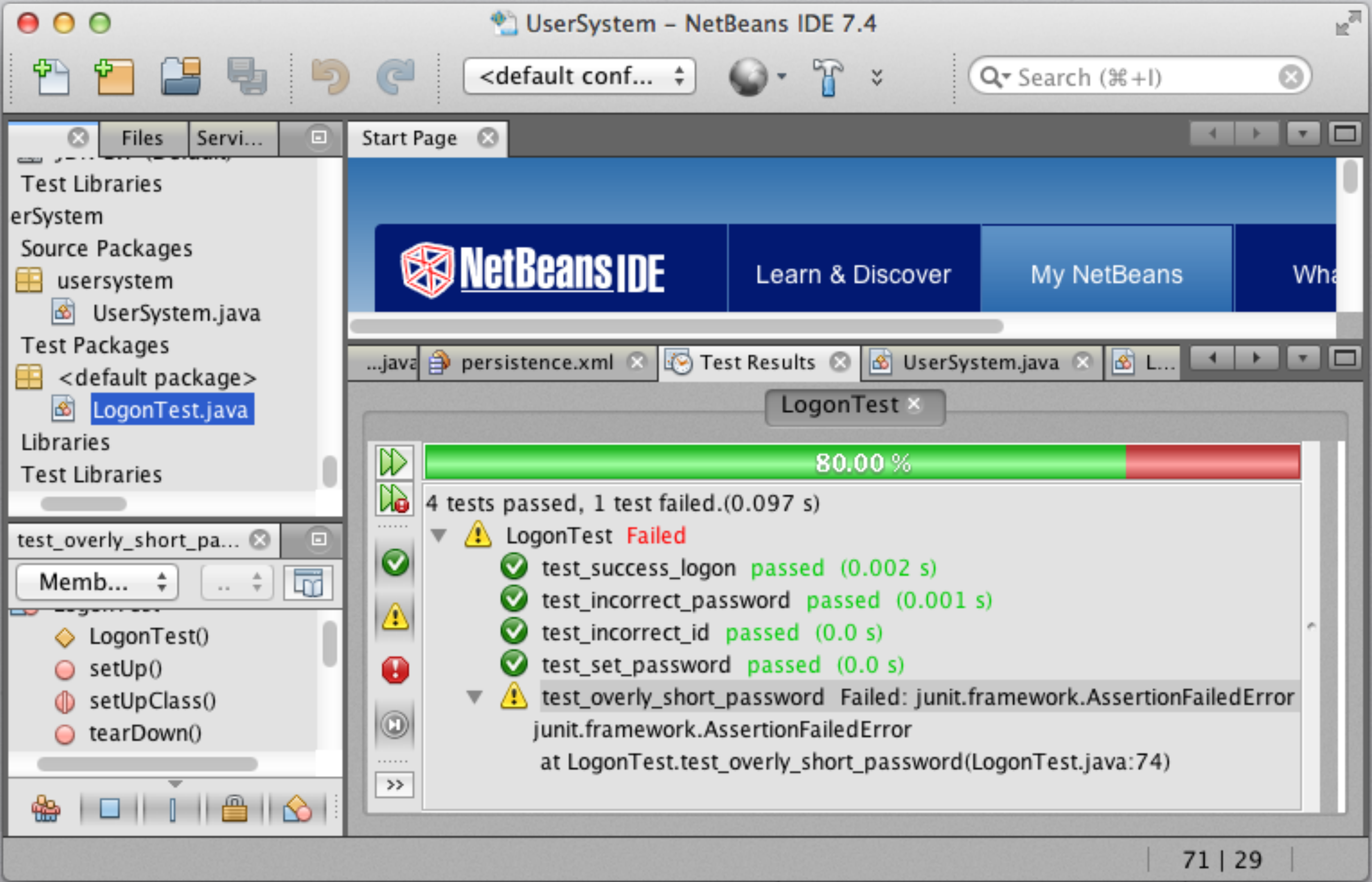
- Kent Beck advocates writing tests before SUT because it involves users to clarify requirements early.

- Suppose we want to prevent users from having passwords less than 6 characters long.

```
@Test
  public void test_overly_short_password() {
    // Enforce minimum password length to be 6
    assertTrue(mySystem.logon("francis", "hello246"));
    assertFalse(mySystem.setPassword("francis", "12345"));
    assertTrue(mySystem.logon("francis", "hello246"));
    assertFalse(mySystem.logon("francis", "12345"));
}
```

TDD Step 2
Run Test to See It Failed

* Method name and line number where the test failed are shown.

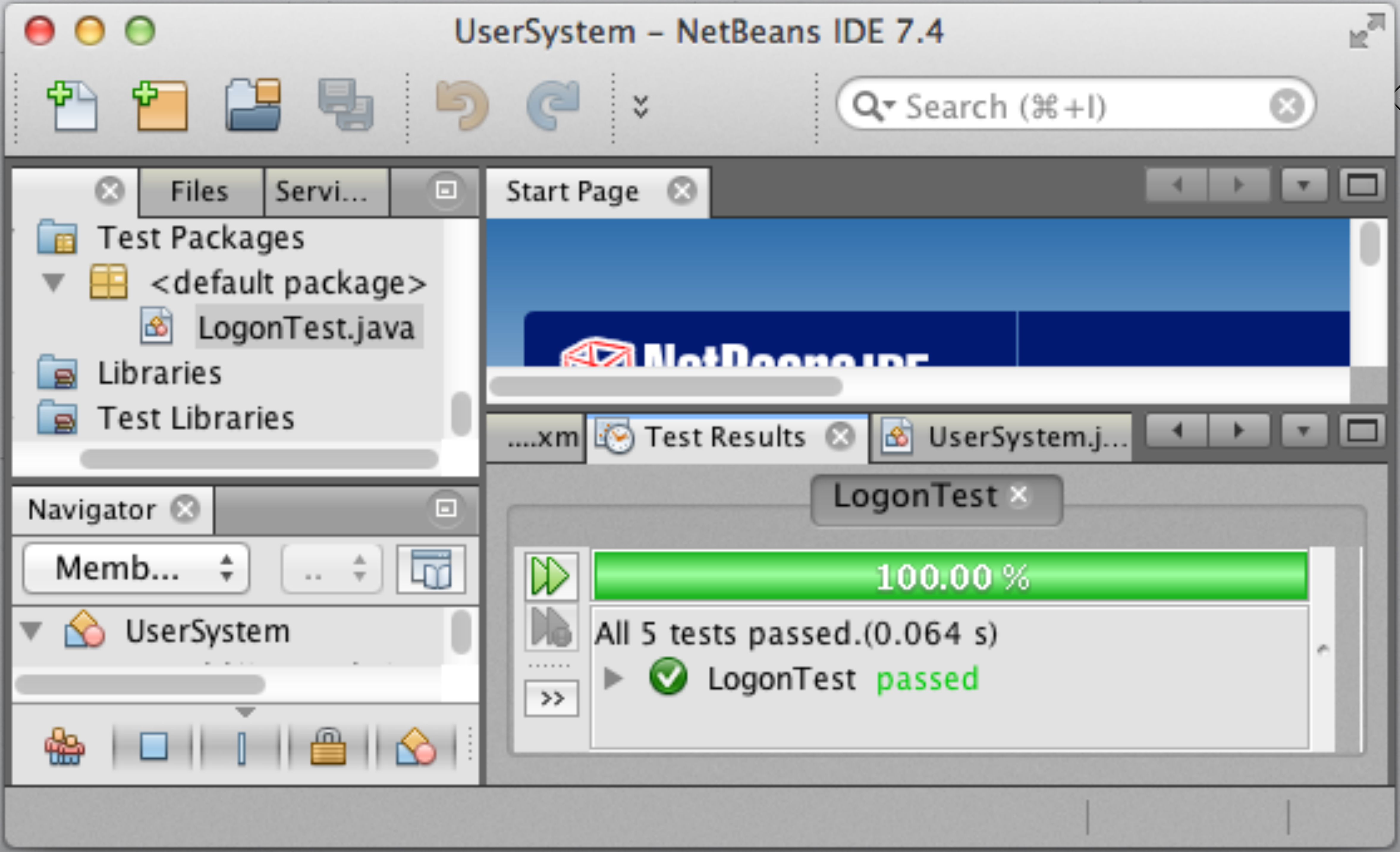


```
80.00 %
4 tests passed, 1 test failed.(0.097 s)
LogonTest Failed
    test_success_logon passed (0.002 s)
    test_incorrect_password passed (0.001 s)
    test_incorrect_id passed (0.0 s)
    test_set_password passed (0.0 s)
    test_overly_short_password Failed: junit.framework.AssertionFailedError
   junit.framework.AssertionFailedError
    at LogonTest.test_overly_short_password(LogonTest.java:74)
```

TDD Step 3
Revise SUT for New Feature

```
boolean passwordTooShort( String pswd {
   return pswd.length() < 6;
}
public boolean setPassword( String id, String pswd) {
   String password = (String) acTable.get(id);
   if (password == null)
      return false;
   else if (passwordTooShort(pswd))
      return false;
   else {
      acTable.put( id, pswd);
      return true;
   }
}
```

TDD Step 4
Run Test Again Until It Works



100.00 %
All 5 tests passed.(0.064 s)
LogonTest passed

✻ We are not showing Step 5 of code refactoring which will be taught in a later chapter.

Motivation of Pairwise Testing

- Consider a running event registration application. Athletes will be registered into different categories according to their
  - Distance - 10km, half-marathon, full marathon (3 values)
  - Gender - male, female (2 values)
  - Age - 14 to 100 (87 values)
- There are 3 x 3 x 87 = 522 test cases for assigning the athlete to a correct category.
- Other values and situations can give rise to many more test cases, too many to test exhaustively.
- We can apply equivalence testing on the age to reduce the number of test cases. For example, consider 3 partitions:
  - Open - Age 35 or under
  - Master 1 - Age 36 to 50
  - Master 2 - Age 51 or above
- Pairwise testing can help us further reduce the number of test cases.

Pairwise Testing

Distance	Gender	Age
10k	M	Open
10k	F	Master 1
Half	M	Master 2
Half	F	Open
Full	M	Master 1
Full	F	Master 2
10k	M	Master 2
Half	F	Master 1
Full	M	Open

- It does not try to exhaust all combinations. In this case, 3 x 2 x 3 = 18 test cases would be required.
- Pairwise testing only tries to exhaust pairwise combinations.
  - Distance & Gender - Rows 1 to 6
  - Gender & Age - Rows 1 to 6
  - Distance & Age - Rows 1 to 9