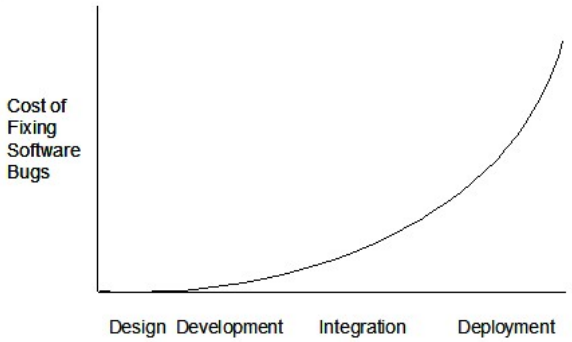
[Handout for L5P2]
Never Too Early to Test: an Introduction to Early Developer Testing

Developer testing

When we test the system as a whole, we call it *system testing*. Intuitively, it seems like the logical thing to do is to wait till the system is built and then test it. But what if a test case fails during such testing? First, we need to locate the cause of the failure. However, the search space could be huge. In a large system, the search space could be millions of lines of code, written by hundreds of developers. Furthermore, the failure may be due to multiple interconnected bugs. Next, we need to fix the bug. This could result in major rework, especially if the bug was in the design or in the requirements. Furthermore, this process is also costly and unpredictable. The whole team needs to work together to locate and fix bugs. One bug might 'hide' other bugs, which could emerge only after the first bug is fixed. Too many bugs found during system testing can lead to delivery delays. As illustrated by the graph below, the earlier we find a bug, the easier and cheaper to fix it. That is why developers need to start testing early, while the system is still in development. Such early testing done by developers is called developer testing.



One such form of early testing is called *Unit testing*. Unit testing is for testing of individual units (methods, classes, subsystems, …) and finding out whether each piece work correctly in isolation.

Another form of developer testing is called *Integration testing*. Integration testing allows verification of the correctness of interactions between subsystems, i.e. verifies whether the pieces work with each other correctly.

Automated API testing

Unit testing is in fact API testing (i.e. testing a programmable interface), as opposed to system testing, which can be done using UI. That means such testing may require *test drivers* and *stubs*. A test driver is a module written specifically for testing whose job is to invoke the SUT (Software Under Test) with test inputs. We need test drivers for unit testing because most of the units do not have a UI. In the code example given next, PayrollTestDriver is a test driver for the Payroll class.

```
public class PayrollTestDriver {
   public static void main(String[] args) {
      //test setup
      Payroll p = new Payroll();
      //test case 1
      p.setEmployees(new String[]{"E001", "E002"});
      print("Test 1 output " + p.totalSalary());
      //test case 2
      p.setEmployees(new String[]{"E001"});
      print("Test 2 output " + p.totalSalary());
      //more tests
      System.out.println("Testing completed");
   }

}
```

A stub is a dummy module that receives outgoing messages from the SUT. During unit and integration testing, we use stubs to isolate the SUT from other collaborating objects. That is, we replace the collaborating objects with stubs so that possible bugs in the collaborating objects do not complicate our test effort. A stub essentially has the same interface as the collaborator it replaces, but its implementation is meant to be so simple that it cannot have any bugs. A stub does not perform any real computations or manipulate any real data. Typically, a stub could do the following tasks:

- **Do nothing** – A stub could simply receive method calls without doing anything at all. When a method is required to return something, it will return a default value.
- **Keep records** – A stub could dutifully record information (e.g. by writing to a log file) about the messages it receives. This record can later be used to verify whether the SUT sent the correct messages to collaborating objects.
- **Return hard-coded responses** – A stub could be written to mimic the responses of the collaborating object, but only for the inputs used for testing. That is, it does not know how to respond to any other inputs. These mimicked responses are hard-coded in the stub rather than computed or retrieved from elsewhere, e.g. from a database.

When we are testing a programmatically accessible component, we can write a test driver to automatically test it. The code given next is an example of such **a**utomated **t**est **d**river (ATD), in this case, written for the Payroll class.

```
public class Payroll_ATD {
   public static void main(String[] args) throws Exception {
      //test setup
      Payroll p = new Payroll();
      p.setSalaryManager(new SalaryManagerStub());
      //test case 1
      p.setEmployees(new String[]{"E001", "E002"});
      // automatically verify the response
      if(p.totalSalary()!= 6400){
           throw new Exception("case 1 failed ");
      }
```

```
        //test case 2
        p.setEmployees(new String[]{"E001"});
        if(p.totalSalary()!= 2300){
            throw new Exception("case 2 failed ");
        }

        //more tests...
        System.out.println("All tests passed");
    }
}
```

JUnit is an automated general-purpose testing tool that provides an automated testing framework for Java programs. Similar tools are available for other languages, e.g. nUnit for C#, and cppUnit for C++. Recent editions of VisualStudio have a built-in *Test Manager* that you can use to create automated tests.

Given next is the code of an ATD for the Payroll class, written using JUnit libraries.

```
public class JUnitPayrollATD {

    @Test
    public void testTotalSalary(){
        Payroll p = new Payroll();
        p.setSalaryManager(new SalaryManagerStub());
        //test case 1
        p.setEmployees(new String[]{"E001", "E002"});
        assertEquals(p.totalSalary(), 6400);
        //test case 2
        p.setEmployees(new String[]{"E001"});
        assertEquals(p.totalSalary(), 2300);
        //more tests
        System.out.println("All tests passed");
    }
}
```

Most IDEs nowadays come with integrated support for testing tools. Figure 11 shows the JUnit output when the JUnitPayrollATD is run using the NetBeans IDE. While JUnit is meant for automating unit testing, it can easily automate integration and system testing (if the system can be accessed programmatically).

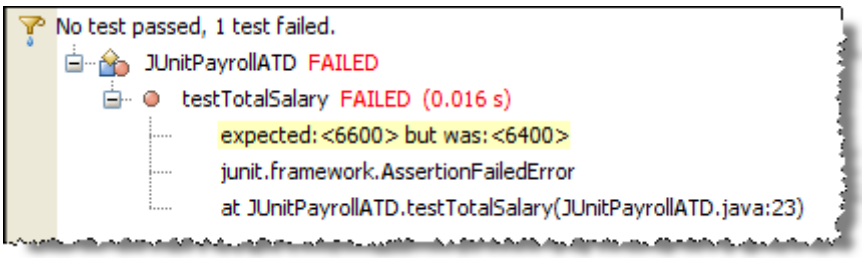


Figure 11. JUnit output on the NetBeans IDE

We can further improve an ATD by making it read test cases from a file and write test results to another file i.e. the ATD will read test cases from a test case file, exercise the SUT as specified by the test cases, capture the system behavior, compare the actual

behavior with the expected behavior specified by the test case, and report the test results.

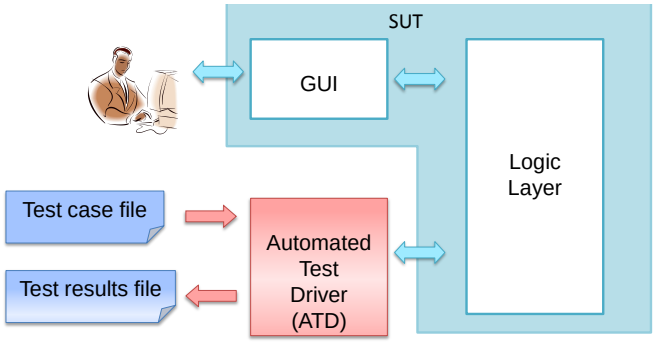


Figure 12. Automated test driver

Next, let us look at how to write an ATD for a simple application called the CAP system. As it is not easy to automate actions performed on the GUI, ATDs often bypass the GUI and access the application layer directly, as shown in Figure 12. Let us assume the CAP system application layer has two operations:

Operation: computeCAP(double[] marks): String
Description: calculates the CAP for the given marks

Operation: comment(double CAP): String
Description: displays the following comments based on CAP

- CAP > 4.0 → "Excellent"
- else CAP > 3.0 "Good"
- else CAP > 2.0 "Work hard"
- otherwise, "Work extremely hard"

The test case file contains a set of test cases (a set of related test cases is sometimes called a test *suite*). The format of the test case file should be easy to handle both by the ATD and a human user. The following sample test case file follows the format:

case id : operation to test : input comma separated parameters : expected output : optional description

Here is an extract from a sample test case file.

```
Case1 : comment : 4.2 : Excellent
Case1b : comment : 1 : Weak
Case2 : comment : 3.5 : Good
Case3 : comment : 3.0 : Average  :  checks the boundary value
Case6 : comment : 1.5 : Weak
...
Case8 : computeCAP : 2,1,3,4,5 : 3.0
Case8b : computeCAP : 2 : 2.0 :  checks with only one value
...
```

The result file allows for easy human inspection of the test outcome. The following sample result file documents the result of each test case in the format result:[test case]actual result (if the case failed), followed by a summary of the test run.

pass[Case1 : comment : 4.2 : Excellent]
pass[Case1b : comment : 1 : Weak]
pass[Case2 : comment : 3.5 : Good]
fail[Case3 : comment : 3.0 : Average : checks the boundary value] Good
pass[Case6 : comment : 1.5 : Weak]
…
pass[Case8 : computeCAP : 2,1,3,4,5 : 3.0]
pass[Case8b : computeCAP : 2 : 2.0 : checks with only one value]
…
pass = 48
fail = 2
total = 50
Passing rate (pass/total) = 96%

Sample code snippets from the Java implementation of the CAP system ATD is given next (to save space, some sections of the code cover more than one level of abstraction).

```
public static void main(String[] args) throws Exception {
    String workingDir = "D:\\myCourses\\CS2103\\Jul2007\\C
    String inputFile = workingDir+"testcases.txt";
    String outputFile = workingDir+"testresults.txt";;
    CAP_TD cap_td = new CAP_TD(inputFile, outputFile);
    cap_td.readTestCases();
    cap_td.executeTests();
    cap_td.writeResult();
}
```

Figure 13. CAP_TD main operation

```
private void readTestCases() throws Exception{
    FileReader myFr = new FileReader(inputFile);
    BufferedReader myBr = new BufferedReader(myFr);
    String line;
    while ((line = myBr.readLine()) != null) {
        if(!line.trim().equals(""))
            cases.add(new TestCase(line));
    }
    myBr.close(); myFr.close();
}
```

Figure 14. Code for reading test cases

```
class TestCase {
    String line,id,method,parameters,expected,actual;
    String comment = "";
    boolean passed;

    public TestCase(String line) throws Exception{
        String[] caseInfo = line.split(":");
        if(caseInfo.length<4) throw new Exception("wrong test c
        this.line = line;
        this.id = caseInfo[0].trim();
        this.method =caseInfo[1].trim();
        this.parameters = caseInfo[2].trim();
        this.expected = caseInfo[3].trim();
        if(caseInfo.length>4)this.comment = caseInfo[4];
        this.actual = "";
    }

    public String toString(){
        String result = passed? "pass" : "failed";
        return result + "[" + line + "]" + (passed?"":actual);
    }

}//end TestCase
```

Figure 15. Code for the TestCase class

Testability

Testability is an indication of how easy it is to test an SUT. Testability depends on controllability, observability, availability (of executables, information), simplicity, stability, and coupling between components. We should try to increase the testability of our software when we design and implement them.

Test-Driven Development (TDD)

Generally, we write tests after we write the SUT. TDD advocates writing the tests *before* writing the SUT. That is, we define the precise behavior of the SUT using test cases. Then we write the SUT to match the specified behavior. While TDD has its share of detractors, it is considered a good way to reduce defects. Note that TDD does not mean we write all the test cases first and then start writing functional code. Rather, we proceed in small steps:

i. Decide what behavior to implement
ii. write test cases to test that behavior
iii. if required, write a stub so that the test can compile
iv. run those test cases and watch them fail
v. implement the behavior
vi. run the test case
vii. keep modifying the code and rerunning test cases until they all pass
viii. refactor code to improve quality
ix. repeat the cycle for each small unit of behavior you want to implement

Worked examples

[Q1]

Consider the minimal Minesweeper (Text UI) version we discussed in previous handouts. Can we change the MSLogic API to increase its testability, in particular, to enable automated testing of the MSLogic API?

[A1]
When the MS is operating normally, mines will appear at random locations. However, such randomness interferes with testing because the tester (or the automated test driver) should be able to predict exact output. Therefore, we can add more operations to the API specifically for testing, to have more control over where the mines appear. For example, we can add the following operation to the MSLogic API:

setMinefield(int W, int H, String mineInfo) : void

Note that mineInfo is a string of a specific format that specifies where to put the mines e.g. "[(1,2)(2,2)(2,3)]"

We can also add a UI command to activate it (for manual testing),
For example,
set 3 4 [(1,2)(2,2)(2,3)] creates a 3x4 minefield with mines at cells (1,2),(2,2), (2,3).

[Q2]
Discuss advantages and disadvantages of developers testing their own code.

[A2]
Advantages:

- Can be done early (the earlier we find a bug, the cheaper it is to fix).
- Can be done at lower levels, for examples, at operation and class level (testers usually test the system at UI level).
- It is possible to do more thorough testing since developers know the expected external behavior as well as the internal structure of the component.
- It forces developers to take responsibility for their own work (they cannot claim that "testing is the job of the testers").

Disadvantages:

- Developer may unconsciously test only situations that he knows to work (i.e. test it "gently").
- Developer may be blind to his own mistakes (if he did not consider a certain combination of input while writing code, he is likely to miss it again during testing).
- Developer may have misunderstood what the SUT is supposed to do in the first place.
- Developer may lack the testing expertise.

---End of Document---