[L8P2]

Quality Assurance: Testing and Beyond

Quality assurance (QA) is the process of ensuring that the software being built has the required levels of quality.

Quality Assurance = Validation + Verification

QA involves checking two aspects:

a) Building the right system, i.e. are the requirements correct? This is called *validation*.
b) Building the system right, i.e. are the requirements implemented correctly? This is called *verification*.

Whether something belongs under validation or verification is not that important. What is more important is both are done, instead of limiting to verification (i.e., remember that the requirements can be wrong too).

In this handout, the testing of the *product as a whole* is discussed. This is in contrast to developer testing that is performed on a partial system.

Testing is the most common way of assuring quality, however there are other complementary techniques such as formal verification. The second part of the handout gives a brief introduction to such QA techniques.

Unit testing

Unit testing is a form of early developer testing. Unit testing involves testing individual units (methods, classes, subsystems, …) and finding out whether each piece works correctly in isolation. A proper unit test require the unit we are testing to be isolated from other code. If the unit depends on other code, we may have to use stubs to isolate the unit from its dependencies.

Stubs/Mocks

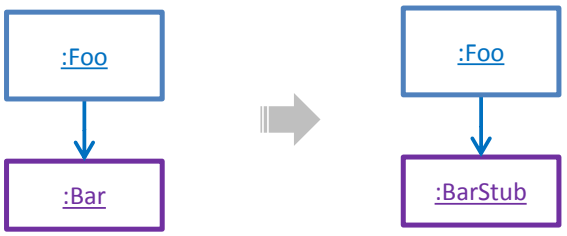
A *stub* or a *mock*[7] is a dummy component that receives outgoing messages from the SUT. During unit testing, stubs are used in place of collaborating objects in order to isolate the SUT from these objects. Doing this prevents bugs within the collaborating objects from interfering with the test. A stub has essentially 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.

[7] Although this handout uses *stub* and *mock* interchangeably, some define them slightly differently. However, such subtle differences are beyond the scope of this handout.

Dependency injection

Dependency injection is the process of 'injecting' objects to replace current dependencies with a different object. This is often used to inject stubs to isolate the SUT from other collaborating objects so that it can be tested independently. In the example below, a Foo object normally depends on a Bar object, but we have injected a BarStub object so that the Foo object no longer depends on a Bar object. Now we can test the Foo object in isolation from the Bar object.



Normal dependency After dependency injection

Given next is a sample testing scenario that tests the totalSalary of the Payroll class. The production version of the totalSalary method collaborates with the SalaryManager object to calculate the return value. During testing, the SalaryManager object is substituted with a SalaryManagerStub object which responds with hard-coded return values.

```
public class PayrollTestDriver {
    public static void main(String[] args) {
        //test setup
        Payroll p = new Payroll();
        p.setSalaryManager(new SalaryManagerStub()); //dependency injection
        //test case 1
        p.setEmployees(new String[]{"E001", "E002"});
        assertEquals(2500.0, p.totalSalary());
        //test case 2
        p.setEmployees(new String[]{"E001"});
        assertEquals(1000.0, p.totalSalary());
        //more tests
        System.out.println("Testing completed");
    }
}

//----------------------------
class Payroll{
  private SalaryManager manager = new SalaryManager();
  private String[] employees;

  void setEmployees(String[] employees) {
      this.employees = employees;
  }

  /*the operation below is used to substitute the actual SalaryManager
          with a stub used for testing */
  void setSalaryManager(SalaryManager sm) {
     this. manager = sm;
  }
```

```
  double totalSalary(){
      double total = 0;
      for(int i=0;i<employees.length; i++){
          total += manager.getSalaryForEmployee(employees[i]);
      }
      return total;
  }
}

//----------------------------
class SalaryManager{
  double getSalaryForEmployee(String empID){
      //code to access employee's salary history
      //code to calculate total salary paid and return it
  }
}

//----------------------------
class SalaryManagerStub extends SalaryManager{
  /* this method returns hard coded values used for testing */
 double getSalaryForEmployee(String empID){
      if(empID.equals("E001")) {
       return 1000.0;
      }else if(empID.equals("E002")){
       return 1500.0;
     }else {
       throw new Error("unknown id");
     }
  }
}
```

Integration testing

In *Integration testing* we test whether different parts of the software 'work together' (i.e. integrates) as expected. Here, we assume the individual parts have been unit tested already. Therefore, integration tests aim to discover bugs in the 'glue code' that are often the result of misunderstanding of what the parts are supposed to do vs what the parts are actually doing. For example let us assume a class Car users classes Engine and Wheel.

- First, we should unit test Engine and Wheel.
- Next, we should unit test Car in isolation of Engine and Wheel, using stubs for Engine and Wheel.
- After that, we can do an integration test for Car using it together with the Engine and Wheel classes to ensure the Car integrates properly with the Engine and the Wheel.

In the example above, if the Car class assumed a Wheel can support 200 mph speed but the Wheel can only support 150 mph, it is the integration test that is supposed to uncover this discrepancy.

System testing

Taking the whole system, instead of a part of the system, and testing it against the system specification is called *system testing*. System testing is typically done by a testing team (also called a QA team). System test cases are based exclusively on the specified external behavior of the system. Sometimes, system tests go beyond the bounds defined in the specification. This is useful when testing that the system fails 'gracefully' having pushed beyond its limits. Take the example of an SUT (software under test) that is a browser capable of handling web pages containing up to 5000 characters. A test case can involve loading a web page containing more

than 5000 characters. The expected 'graceful' behavior would be to 'abort the loading of the page and show a meaningful error message'. This test case would fail if the browser attempted to load the large file anyway and crashed.

Note that system testing includes testing against non-functional requirements too. Here are some examples.

- *Performance testing* – to ensure the system responds quickly.
- *Load testing* (also called *stress testing* or *scalability testing*) – to ensure the system can work under heavy load.
- *Security testing* – to test how secure the system is.
- *Compatibility testing*, *interoperability testing* – to check whether the system can work with other systems.
- *Usability testing* – to test how easy it is to use the system.
- *Portability testing* – to test whether the system works on different platforms.

Acceptance testing

Acceptance testing, also called User Acceptance Testing (UAT), is a type of validation test carried out to show that the delivered system meets the requirements of the customer. Similar to system testing, acceptance testing involves testing the whole system against the requirements specification (rather than the system specification). Note the two specifications need not be the same. For example, requirements specification could be limited to how the system behaves in normal working conditions while the system specification can also include details on how it will fail gracefully when pushed beyond limits, how to recover, additional APIs not available for users (for the use of developers/testers), etc.

Acceptance testing comes after system testing. It is usually done by a team that represents the customer, and it is usually done on the deployment site or on a close simulation of the deployment site. UAT test cases are often defined at the beginning of the project, usually based on the use case specification. Successful completion of UAT is often a prerequisite to the project signoff.

Acceptance tests gives an assurance to the customer that the system does what it is intended to do. Besides, acceptance testing is important because a system could work perfectly on the development environment, but fail in the deployment environment due to subtle differences between the two.

Alpha and Beta testing

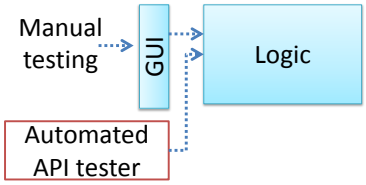
Alpha testing is performed by the users, under controlled conditions set by the software development team. *Beta testing* is performed by a selected subset of target users of the system in their natural work setting. An *open beta release* is the release of not-yet-production-quality-but-almost-there software to the general population. For example, Google's Gmail was in 'beta' for years before the label was finally removed.

GUI testing

If a software product has a GUI component, all product-level testing (i.e. the types of testing mentioned above) need to be done using the GUI. However, testing the GUI is much harder than testing the CLI (command line interface) or API, for the following reasons:

- Most GUIs contain a large number of different operations, many of which can be performed in any arbitrary order.

- GUI operations are more difficult to automate than API testing. Reliably automating GUI operations and automatically verifying whether the GUI behaves as expected is harder than calling an operation and comparing its return value with an expected value. Therefore, automated regression testing of GUIs is rather difficult. However, there are testing tools that can automate GUI testing. For example, TestFx and support automated testing of JavaFX GUIs and Selenium (http://seleniumhq.org/) can be used to automate testing of Web application UIs. VisualStudio supports 'record replay' type of GUI test automation.
- The appearance of a GUI (and sometimes even behavior) can be different across platforms and even environments. For example, a GUI can behave differently based on whether it is minimized or maximized, in focus or out of focus, and in a high resolution display or a low resolution display.



One approach to overcome the challenges of testing GUIs is to minimize logic aspects in the GUI. Then, bypass the GUI to test the rest of the system using automated API testing. While this still requires the GUI to be tested manually, the number of such manual test cases can be reduced as most of the system has been tested using automated API testing.

Test coverage

In the context of testing, *coverage* is a metric used to measure the extent to which testing exercises the code. Here are some examples of different coverage criteria:

- **Function/method coverage** measures the coverage in terms of functions executed e.g. testing executed 90 out of 100 functions.
- **Statement coverage** measures coverage in terms of the number of line of code executed e.g. testing executed 23k out of 25k LOC.
- **Decision/branch coverage** measures coverage in terms of decision points e.g. an if statement evaluated to both true and false with separate test cases during testing.
- **Condition coverage** measures coverage in terms of boolean sub-expressions, each evaluated to both true and false with different test cases. Condition coverage is not the same as the decision coverage; e.g. if(x>2 && x<44) is considered one decision point but two conditions. For 100% branch or decision coverage, two test cases are required:

  (x>2 && x<44) == true : [e.g. x = 4]

  (x>2 && x<44) == false : [e.g. x = 100]

  For 100% condition coverage, three test cases are required

  (x>2) == true , (x<44) == true : [e.g. x = 4]

  (x<44) == false : [e.g. x = 100]

  (x>2) == false : [e.g. x = 0]
- **Path coverage** measures coverage in terms of possible paths through a given part of the code executed. 100% path coverage means all possible paths have been executed. A commonly used notation for path analysis is called the *Control Flow Graph (CFG)*. For an introduction to CFGs, refer to the side-note below.
- **Entry/exit coverage** measures coverage in terms of possible *calls to* and *exits from* the operations in the SUT.

Measuring coverage is often done using *coverage analysis tools*. Coverage measurements are used to improve testing E&E (Effectiveness and Efficiency). For example, if a set of test cases does not achieve 100% branch coverage, more test cases are added to cover missed branches.

[Side-Note] Control Flow Graphs (CFG)

This topic is not examinable

CFG is a graphical representation of the execution paths of a code fragment. A CFG consists of:

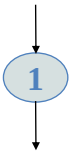
Nodes: Each node represents one or more sequential statements with no branches

Directed Edges: Each edge represents a branch, a possible execution path

Given below is the CFG notation :

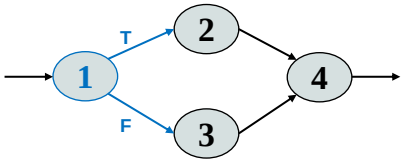
A set of sequential statements (without any branches) is represented as a single node. E.g.

```
x=2;      //node 1
y=3;      //node 1
z=x+y;    //node 1
print (z); //node 1
```



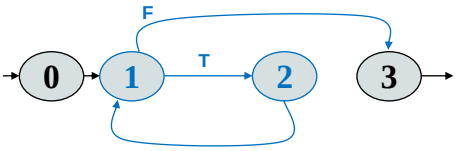
Conditional statements: E.g.

```
if (x < 10) then     //node 1
    z = x + y;       //node 2
else  z = x – y;     //node 3
z = z + 2;           //node 4
```



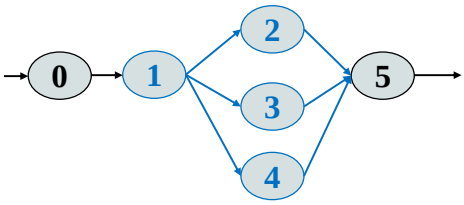
Loops: E.g.

```
x++; //node 0
while (x < 10) { //node 1
   z = x+ y; //node 2
   x++;     //node 2
}
resetX(); //node 3
```



Multi-way branching: E.g.

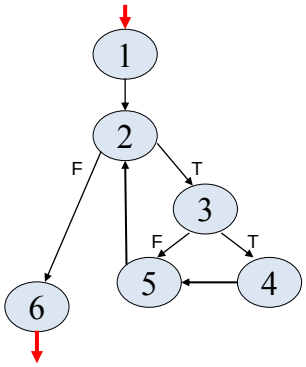
```
x++;                 //node 0
switch (x){          //node 1
  case 0:
    z = x; break;    //node 2
  case 1:
  case 2:
    z = y; break;    //node 3
  default:
    z = x-y;         //node 4
}
z = x+y ;            //node 5
```



Note how the same edge represents both case 1 and case 2.

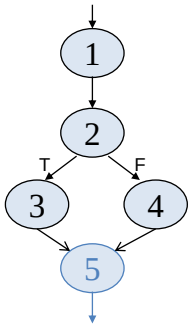
The figure below shows the complete CFG for the `min` function given below.

```
void min(int[] A){
  int min = A[0];          //node 1
  int i = 1;               //node 1
  int n = A.length;        //node 1
  while (i < n){           //node 2
   if (A[i] < min)         //node 3
      min = A[i];          //node 4
      i++;                 //node 5
   }
  print(min);              //node 6
}
```



It is recommended to have exactly one entry edge and exactly one exit edge for each CFG. Sometimes a *logical node* (i.e. a node that does not represent an actual program statement) is added to enforce the "exactly one exit edge" rule. Node 5 in the figure below is a logical node.

```
void foo(){
   int min = A[0];         //node 1
   if (A[i] < min)         //node 2
      min = A[i];          //node 3
   else
      i++;                 //node 4
   }
}
```



A *path* is a series of nodes that can be traversed from the entry edge to the exit edge in the direction of the edges that link them. For example, 1-2-4-5 in the above CFG is a path.

Other QA techniques

There are many QA techniques that do not involve executing the SUT. Given next are a number of such techniques that can complement the testing techniques discussed so far.

Inspections & reviews

Inspections involve a group of people systematically examining a project artefact to discover defects. Members of the inspection team play various roles during the process, such as the *author* - the creator of the artefact, the *moderator* - the planner and executer of the inspection meeting, the *secretary* - the recorder of the findings of the inspection, and the *inspector/reviewer* - the one who inspects/reviews the artefact. All participants are expected to have adequate prior knowledge of the artefact inspected. An inspection often requires more than one meeting. For example, the first meeting is called to brief participants about the artefact to be inspected. The second meeting is called once the participants have studied the artefact. This is when the actual inspection is carried out. A third meeting could be called to re-inspect the artefact after the defects discovered as an outcome of the inspection are fixed. An advantage of inspections is

that it can detect functionality defects as well as other problems such as coding standard violations. Furthermore, inspections can verify non-code artefacts and incomplete code, and do not require test drivers or stubs. A disadvantage is that an inspection is a manual process and therefore, error prone.

Formal verification
Formal verification uses mathematical techniques to prove the correctness of a program. An advantage of this technique over testing is that it can prove the absence of errors. However, one of the disadvantages is that it only proves the compliance with the specification, but not the actual utility of the software. Another disadvantage is that it requires highly specialized notations and knowledge which makes it an expensive technique to administer. Therefore, formal verifications are more commonly used in safety-critical software such as flight control systems.

Static analyzers
These are tools that automatically analyze the code to find anomalies such as unused variables and unhandled exceptions. Detection of anomalies helps in improving the code quality. Most modern IDEs come with some inbuilt static analysis capabilities. For example, an IDE will highlight unused variables as you type the code into the editor. Higher-end static analyzers can check for more complex (and sometimes user-defined) anomalies, such as overwriting a variable before its current value is used.