Module: 6

Software Testing

lecture 1

1. Motivation:

- Object design which comes before implementation, if not well understood, and if not well done leads to system degradation.
- The average software product released on the market is not error free

2. Objective:

To get knowledge about the activities in the different testing strategies.

3. Learning Objective:

This module explains the meaning of Software Testing, with the help of concepts that define the Testing process.
Students will be able to understand the importance of Software Testing during the testing of the software.

1. Students will be able to understand Software Testing Concepts and the various Software standards.
2. Students will be able to understand the Testing Process and Basic concept and terminology, Verification & validation, White Box Testing Path Testing, Control Structures Testing, DEFUSE testing.
3. Students will be able to understand Black Box Testing, OO testing methods.
4. Students will be able to understand Software Maintenance and Reverse Engineering.

4. Prerequisite:

Activities involved in requirement, analysis and design phases of software life cycle

5. Learning Outcomes:

- Student will be able to test a software with the help of various techniques like Black Box Testing, OO testing methods, White Box Testing Path Testing, Control Structures Testing, DEFUSE testing.

- Students will be able to Maintain a software and also able to do Reverse Engineering.

6. Syllabus:

Module	Content	Duration	Self Study Time
7.1	Basic concept and terminology, Verification & validation, White Box Testing Path Testing, Control Structures Testing , DEFUSE testing,	3 lectures	3 hours
7.2	Black Box Testing – BVA Integration, Validation and system testing.	3 lectures	3 hours
7.3	OO testing methodsClass Testing, Interclass testing, testing architecture, Behavioral testing.	3 lectures	3 hours
7.4	Software Maintenance – Reverse Engineering.	3 lectures	3 hours

5. Learning

- Various testing strategies
- Types of White box and black box testing
- Object oriented testing
- Maintenance types
- Maintenance log and defect reports
- Software reengineering

6. Weightage: Marks

7. Abbreviations:

(1) BVA – Boundary value Analysis

8. Key Definitions:

1. **Testing:** Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation.
2. **Black box testing:** It is a software testing technique whereby the internal workings of the item being tested are not known by the tester. For example, in a black box test on a software design, the tester only knows the inputs and what the expected outcomes should be and not how the program arrives at those outputs. The tester does not ever examine the programming code and does not need any further knowledge of the program other than its specifications.
3. **White box testing:** It is a software testing technique whereby explicit knowledge of the internal workings of the item being tested is used to select the test data. Unlike black box testing, white box testing uses specific knowledge of programming code to examine outputs. The test is accurate only if the tester knows what the program is supposed to do. He or she can then see if the program diverges from its intended goal. White box testing does not account for errors caused by omission, and all visible code must also be readable.
4. **Reverse engineering:** Software reverse engineering is done to retrieve the source code of a program because the source code was lost, to study how the program performs certain operations, to improve the performance of a program, to fix a bug (correct an error in the program when the source code is not available), to identify malicious content in a program such as a virus or to adapt a program written for use with one microprocessor for use with another.
5. **Forward engineering:** Forward engineering is defined as the normal execution of the software life cycle, i.e. in the forward direction. Therefore it is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.
6. **Defect report:** It is a document reporting on any flaw in a component or system that can cause the component or system to fail to perform its required function.
7. **Maintenance log:** It means a log in which unserviceabilities, rectifications and daily inspections are recorded.
8. **Unit testing:** Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Unit testing is often automated but it can also be done manually. This testing mode is a component of Extreme Programming (XP), a pragmatic method of software development that takes a meticulous approach to building a product by means of continual testing and revision.

9. **Integration testing:** Integration testing, also known as integration and testing (I&T), is a software development process which program units are combined and tested as groups in multiple ways. In this context, a unit is defined as the smallest testable part of an application. Integration testing can expose problems with the interfaces among program components before trouble occurs in real-world program execution. Integration testing is also a component of Extreme Programming (XP).
10. **Regression testing:** Regression testing is the process of testing changes to computer programs to make sure that the older programming still works with the new changes. Regression testing is a normal part of the program development process and, in larger companies, is done by code testing specialists.
11. **Acceptance testing:** Acceptance testing is a final stage of testing that is performed on a system prior to the system being delivered to a live environment. Systems subjected to acceptance testing might include such deliverables as a software system or a mechanical hardware system. Acceptance tests are generally performed as "black box" tests.

9. Theory

9.1 Testing

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation.

The following sections explain the different testing strategies.

9.1.1 FTR (Formal Technical Review)

The purpose of FTR is to find defects (errors) before they are passed on to another software engineering activity or released to the customer. It is an effective means for improving the software quality. Walkthroughs and inspection are two types of reviews. The fundamental difference between them is that walkthroughs have fewer steps and are less formal than inspections.

Walkthroughs:

A walkthrough team should consist of four to six individuals. The team should include at least one representative drawing up the specifications (in case of specification walkthrough), the manager responsible for the specifications, a client representative, a representative of the team that will perform the next phase of the

development (for specification walkthrough it is a representative from design team), and a representative of the Software quality assurance group. The walkthrough should be chaired by the SQA representative. The members of the walkthrough team, as far as possible, should be experienced senior technical staff members because they tend to find the important faults. The material for the walkthrough must be distributed to the participants well in advance to allow for careful preparation. Each reviewer should study the material and develop two lists: a list of items the reviewer does not understand and a list of items the reviewer believes are incorrect.

The person leading the walkthrough guides the other members of the walkthrough team through the document to uncover any faults. It is not the task of the team to correct faults, merely to record them for later correction. There are two ways of conducting a walkthrough. The first is *participant driven*. Participants present their lists of unclear items and items they think are incorrect. The representative of the specifications team must respond to each query, clarifying what is unclear to the reviewer and either agreeing that indeed there is a fault or explaining why the reviewer is mistaken. The second way of conducting a review is *document driven*. A person responsible for the document, either individually or as part of a team, walks the participants through that document, with the reviewers interrupting either with their prepared comments or comments triggered. The second is likely to be more thorough leading to detection of more faults.

The primary role of the walkthrough leader is to elicit questions and facilitate discussion. A walkthrough is an iterative process; it is not supposed to be one-sided instruction by the presenter. It also is essential that the walkthrough not be used as a means of evaluating the participants, because the walkthrough degenerates into a point-scoring session and does not detect faults. Walkthrough includes all kinds like: specification walkthrough, design walkthrough, plan walkthrough and code walkthrough.

Inspections:

Inspections were first proposed by Fagan for testing designs and code. An inspection goes far beyond a walkthrough and has five formal steps. First, an *overview* of the document to be inspected (specification, design, code, or plan) is given by one of the individuals responsible for producing that document. At the end of the overview session, the document is distributed to the participants. In the

second step, *preparation*, the participants try to understand the document in detail. Lists of fault types found in recent inspections, with the fault types ranked by frequency, are excellent aids. The third step is the *inspection*. To begin, one participant walks through the document with the inspection team, ensuring that every item is covered and that every branch is taken at least once. Then fault finding commences. Within one day the leader of the inspection team (the *moderator*) must produce a written report of the inspection to ensure meticulous follow-through. The fourth stage is the *rework*, in which the individual responsible for that document resolves all faults and problems noted in the written report. The final stage is the *follow-up*. The moderator must ensure that every single issue raised has been resolved satisfactorily, by either fixing the document or clarifying items incorrectly flagged as faults. All fixes must be checked to ensure that no new faults have been introduced. If more than 5 percent of the material inspected has been reworked, then the team must reconvene for a 100 percent reinspection.

The inspection should be conducted by a team of four. For example, in the case of a design inspection, the team will consist of a moderator, designer, implementer, and tester. The moderator is both manager and leader of the inspection team. There must be a representative of the team responsible for the current phase as well as a representative of the team responsible for the next phase. The tester can be any programmer responsible for setting up test cases; preferably the tester should be a member of the SQA group. The IEEE standard recommends a team of between three and six participants. Special roles are played by the *moderator*; the *reader*, who leads the team through the design (or code etc.); and the *recorder*, who is responsible for producing a written report of the detected faults.

An essential component of an inspection is the checklist of potential faults. For example, the checklist for a design inspection should include items such as these; Is each item of the specification document adequately and correctly addressed? For each interface, do the actual and formal arguments correspond? etc. An important component of the inspection procedure is the record of fault statistics. Faults must be recorded by severity (major or minor) and fault type (e.g., interface faults, logic faults).

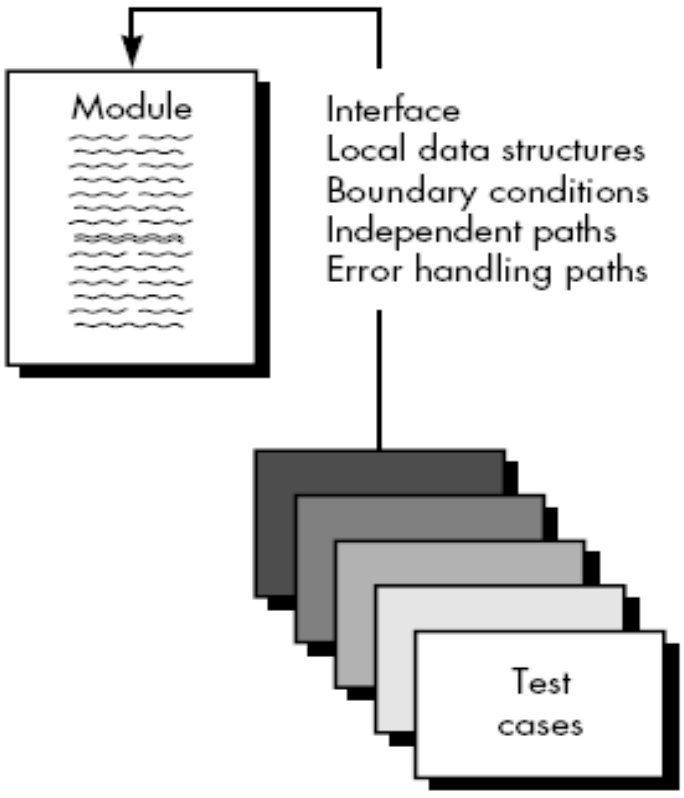
9.1.2 Unit Testing, Integration, System and Regression Testing

Unit Testing:

Unit testing focuses on the building blocks of the software system, that is, objects and subsystems. The specific candidates for unit testing are chosen from the object model and the system decomposition. In principle, all the objects developed

during the development process should be tested, which is often not feasible because of time and budget constraints. The minimal set of objects to be tested should be the participating objects in the use cases. Subsystems should be tested after each of the objects and classes within that subsystem have been tested individually. Unit testing focuses verification effort on the smallest unit of software design—the software component or module.The unit test is white-box oriented. . In Unit testing the following are tested,

1. The module interface is tested to ensure that information properly flows into and out of the program unit under test.
2. The local data structure is examined to ensure that data stored temporarily maintains its integrity.
3. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
4. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
5. And finally, all error handling paths are tested



The most important unit testing techniques are discussed below:

i. Equivalence testing

This black box testing technique minimizes the number of test cases. The possible inputs are partitioned into equivalence classes, and a test case is selected for each class. The assumption of equivalence testing is that systems usually behave in similar ways for all members of a class. Only one member of an equivalence class needs to be tested in order to test the behavior associated with that class. Equivalence class consists of two steps: identification of the equivalence classes and selection of the test inputs. The following criteria are used in determining the equivalence classes.

- *Coverage* (Every possible input belongs to one of the equivalence classes)
- *Disjointedness* (No input belongs to more than one equivalence class)
- *Representation* (If the execution demonstrates an erroneous state when a particular member of a equivalence class is used as input, then the same erroneous state can be detected by using any other member of the class as input)

For each equivalence class, at least two pieces of data are selected: a typical input, which exercises the common case, and an invalid input, which exercises the exception handling capabilities of the component. After all equivalence classes have been identified, a test input for each class has to be identified that covers the equivalence class. If not all the elements of the equivalence class are covered by the test input, the equivalence class must be split into smaller equivalence classes, and test inputs must be identified for each of the new classes.

A method that returns the number of days in a month, given the month and year is considered as an example here. The month and year are specified as integers. By convention 1 represents the month of January, 2 the month of February, and so on. The range of valid inputs for the year is 0 to maxInt.

```
class MyGregorianCalendar {
...
    public static int getNumDaysInMonth(int month, int year) {…}
...
}
```

Figure 4.16: Interface for a method computing the number of days in a given month.

From the code it is understood that 'getNumDaysInMonth()' is that particular method that takes two parameters, a month and a year, both specified as integers.

Three equivalence classes can be found out for the month parameter: months with 31 days (i.e., 1, 3, 5, 7, 8, 10, 12), months with 30 days (i.e., 4, 6, 9, 11), and February, which can have 28 or 29 days. Nonpositive integers and integers larger than 12 are invalid values for the month parameters. Also negative integers are invalid values for the year. At the first, one valid value for each parameter and equivalence class (e.g., February, June, July, 1901, and 1904) are selected. Given that the return value of the getNumDaysInMOnth() method depends on both parameters, these values are combined to test interaction. This results in the six equivalence classes displayed by the following table.

Equivalence class	**Value for month input**	**Value for year input**
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Months with 30 days, leap years	6 (June)	1904
Month with 28 or 29 days, non-leap year	2 (February)	1901
Month with 28 or 29 days, leap year	2 (February)	1904

ii. Boundary testing

This special case of equivalence testing focuses on the conditions at the boundary of the equivalence classes. Rather than selecting any element in the equivalence class, boundary testing requires that the elements be selected from the "edges" of the equivalence class. In the example discussed in equivalence testing, the month of February presents several boundary cases. In general, years that are multiples of 4 are leap years. Years that are multiples of 100, however, are not leap years, unless they are also multiple of 400. For example, 2000 was a leap year, whereas 1900 was not. Both 1900 and 2000 are good boundary cases that should be tested. Other boundary cases include the months 0 and 13. A disadvantage of equivalence class and boundary testing is that these techniques do not explore combinations of test input data. This problem is addressed by Cause-Effect testing that establishes logical relationships between input and outputs or inputs and

transformations. The inputs are called causes, the outputs or transformations are effects.

iii. Path testing

This whitebox testing technique identifies faults in the implementation of the component. The assumption behind path testing is that, by exercising all possible paths through the code at least once, most faults will trigger failures.

The starting point for path testing is the flow graph. A flow graph consists of nodes representing executable blocks and associations representing flow of control. A flow graph is constructed from the code of a component by mapping decision statements (e.g., if statements, while loops) to nodes lines. Statements between each decision (e.g., then block, elser block) are mapped to other nodes. The following figures depict the example faulty implementation of the getNumDaysInMonth() method and the equivalent flow graph as a UML activity diagram. In the activity diagram, decisions are modeled with UML branches, blocks with UML action states, and control flow with UML transitions.

```
public class MonthOutOfBounds extends Exception {…};
public class YearOutOfBounds extends Exception {…};

class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%4) == 0){
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
    public static int getNumDaysInMonth(int month, int year)
            throws MonthOutOfBounds, YearOutOfBounds {
        int numDays;
        if (year < 1) {
            throw new YearOutOfBounds(year);
        }
        if (month == 1 || month == 3 || month == 5 || month == 7 ||
                month == 10 || month == 12) {
            numDays = 32;
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            numDays = 30;
        } else if (month == 2) {
            if (isLeapYear(year)) {
                numDays = 29;
            } else {
                numDays = 28;
            }
        } else {
            throw new MonthOutOfBounds(month);
        }
        return numDays;
    }
}
```

Figure 4.17: An example of a (faulty) implementation of the getNumDaysInMonth() method (Java)

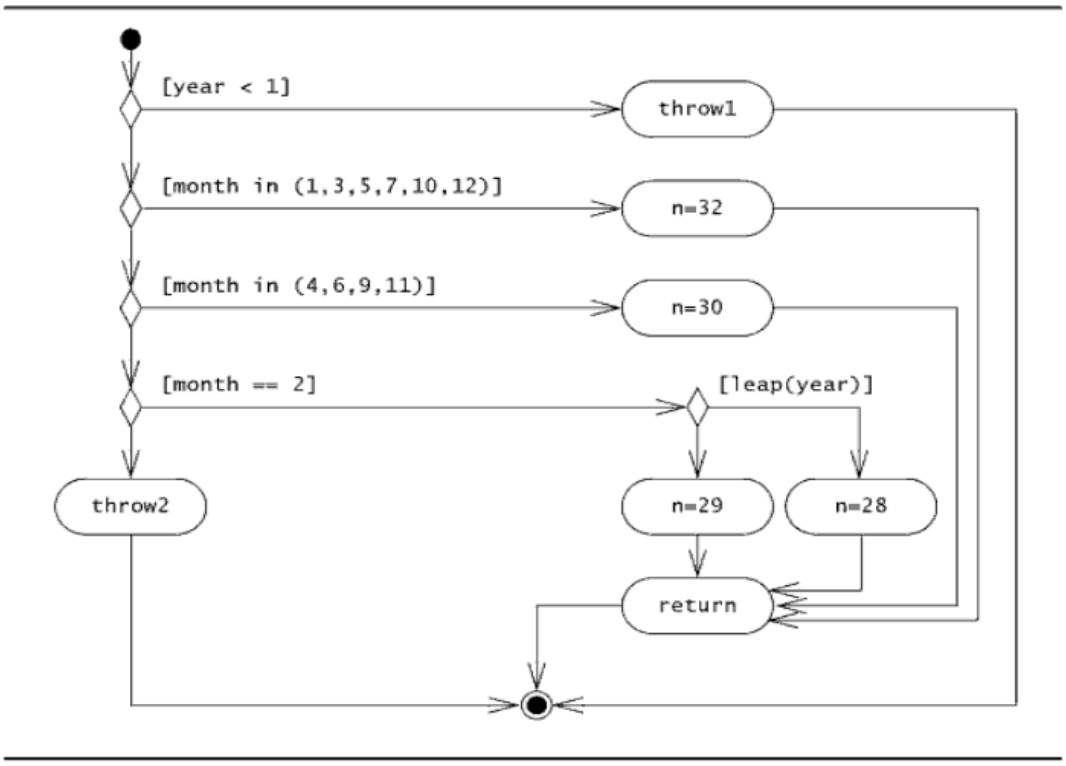


Figure 4.18: Equivalent flow graph for the (faulty) implementation of the getNumDaysInMonth() method of the above implementation (UML activity diagram)

Complete path testing is done by examining the condition associated with each branch point and selecting an input for the true branch and another input for the false branch. For example, examining the first branch point in the UML activity diagram, two inputs are selected: 'year=0' (such that year<1 is true) and 'year=1901 (such that year<1 is false). The process is then repeated for the second branch and the inputs such as 'month=1' and 'month=2' are selected. The input (year=0, month=1) produces the path {throw1}. The input (year=1901, month=1) produces a second complete path {n=32 return}, which uncovers one of the faults in the getNumDaysInMonth() method. The following table depicts the test cases and equivalent paths generated by repeating the above process for each node.

Test case	**Path**

(year=0, month=1)	{throw1}
(year=1901, month=1)	{n=32 return}
(year=1901, month=2)	{n=28 return}
(year=1904, month=2)	{n=29 return}
(year=1901, month=4)	{n=30 return}
(year=1901, month=0)	{throw2}

Using graph theory, it can be shown that the minimum number of tests necessary to cover all edges is equal to the number of independent paths through the flow graph. This is defined as the cyclomatic complexity CC of the flow graph, which is

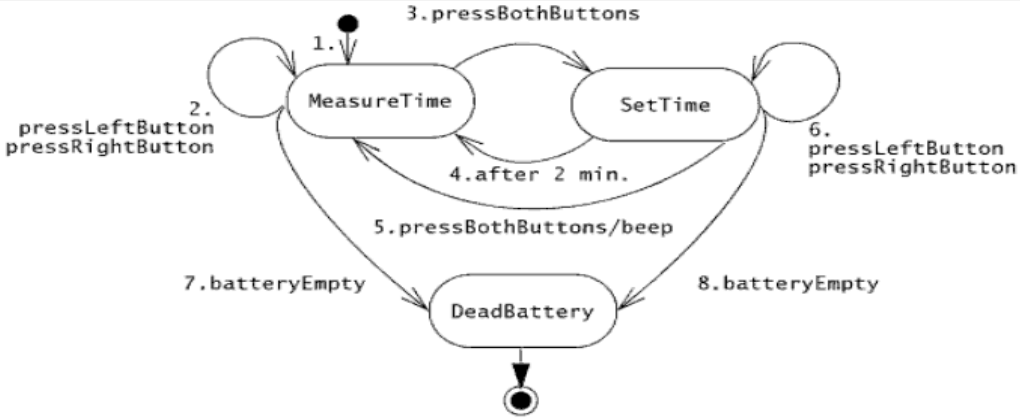
CC= number of edges-number of nodes+2

where the number of nodes is the number of branches and action states, and the number of edges is the number of transitions in the activity diagram. The cyclomatic complexity of the getNumDaysInMonth() method is 6, which is also the number of test cases found (shown in the above table). Path testing and whitebox methods can detect only faults resulting from exercising a path in the program, such as the faulty numDays=32 statement. Whitebox testing methods cannot detect omissions, such as the failure to handle the non-leap year 1900. Also path testing is heavily based on the control structure of the program. However, no testing method short of exhaustive testing can guarantee the discovery of all faults.

iv. State-based testing

This testing technique was recently developed for object-oriented systems. State-based testing compares the resulting state of the system with the expected state. In the context of a class, state-based testing consists of deriving test cases from the UML statechart diagram for the class. For each state, a representative set of stimuli is derived for each transition (similar to equivalence testing). The attributes of the class are then instrumented and tested after each stimulus has been applied to ensure that the class has reached the specified state.

This testing can be illustrated using an application called '2Bwatch' (explanation about this application is given inside the box). The statechart diagram



Stimuli	Transition tested	Predicted resulting state
Empty set	1. *Initial transition*	`MeasureTime`
Press left button	2.	`MeasureTime`
Press both buttons simultaneously	3.	`SetTime`
Wait 2 minutes	4. *Timeout*	`MeasureTime`
Press both buttons simultaneously	3. *Put the system into the SetTime state to test the next transition.*	`SetTime`
Press both buttons simultaneously	5.	`SetTime->MeasureTime`
Press both buttons simultaneously	3. *Put the system into the SetTime state to test the next transition.*	`SetTime`
Press left button	6. Loop back onto `MeasureTime`	`MeasureTime`

of this application is given below:

Figure 4.19: UML statechart diagram and resulting tests for 2Bwatch SetTime function (only the first eight stimuli are shown)

2Bwatch is a watch with two buttons (so is the name). Setting the time on 2Bwatch requires the actor 2BwatchOwner to first press both buttons simultaneously, after which 2Bwatch enters the set time mode. In the set time mode, 2Bwatch blinks the number being changed (e.g., the hours, minutes, seconds, day, month, or year). Initially, when the 2BWatchOwner enters the set time mode, the hours blink. If the actor presses the first button, the next number blinks (e.g., if the hours are blinking and the actor presses the first button, the hurs stop blinking and the minutes start blinking). If the actor presses the second button, the blinking number is incremented by one unit. If the blinking number reaches the end of its range, it is reset to the beginning of its range (e.g., assume the minutes are blinking and its current value is 59, its new value is set to 0 if the actor presses the second button). The actor exits the set time mode by pressing both buttons simultaneously

such that each transition is traversed at least once. After each input, instrumentation code checks if the watch is in the predicted state and reports a failure otherwise. Some transitions (e.g., transition 3) are traversed several times, as it is necessary to put the watch back into the SetTime state (e.g., to test transitions 4, 5, and 6). The test inputs for the DeadBattery state were not generated (only the first eight stimuli are displayed).

State-based testing presents several difficulties. Because the state of a class is encapsulated, test sequences must include sequences for putting classes in the desired state before given transitions can be tested. It can become an effective testing technique for object-oriented systems if proper automation is provided.

v. Polymorphism testing

Polymorphism introduces a new challenge in testing because it enables messages to be bound to different methods based on the class of the target. Although this enables developers to reuse code across a larger number of classes, it also introduces more cases to test.

A NetworkInterface Strategy design pattern is considered as example here. The strategy pattern is applied for encapsulating multiple implementations of a NetworkInterface. The LocationManager implementing a specific policy configures NetworkConnection with a concrete NetworkInterface (i.e., the mechanism) based on the current location. The Application uses the NetworkConnection independently of concrete NetworkInterfaces. The strategy design pattern uses polymorphism to shield the context (i.e., the NetworkConnection class) from the concrete strategy (i.e., the Ethernet, WaveLAN, and UMTS classes). For example, the NetworkConnection.send() method calls he NetworkInterface.send() method to send bytes across the current NetworkInterface, regardless of the actual concrete strategy. This means that, at run time, the NetworkInterface.send() mehod invocation can be bound to one of three methods, Ethernet.send(), WaveLAN.send(), UMTS.send().

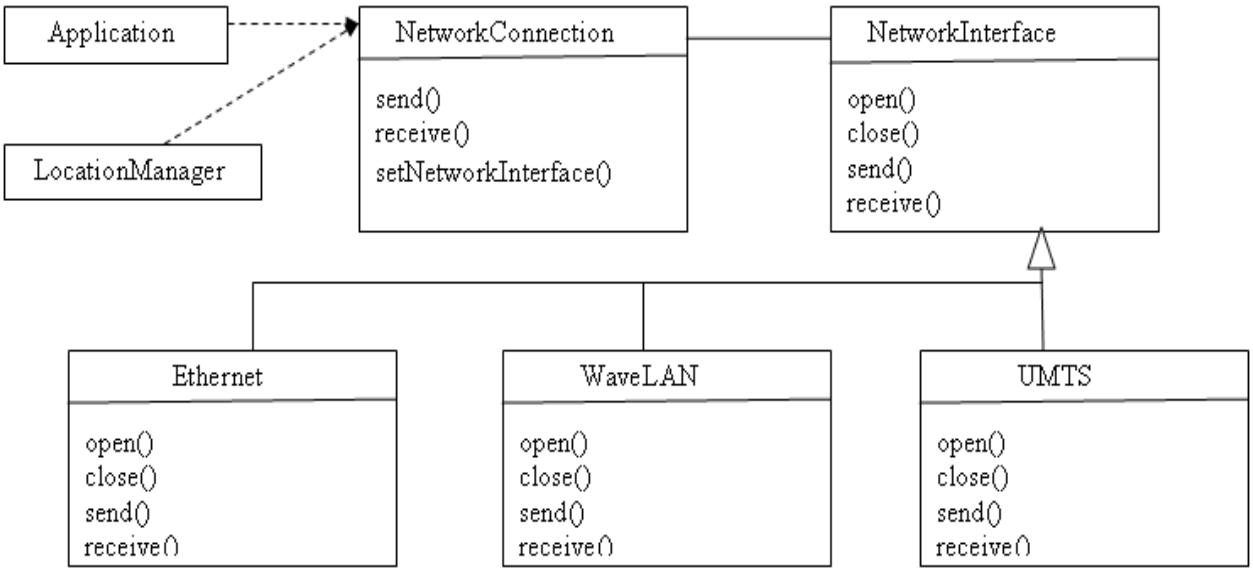


Figure 4.20: A strategy design pattern for encapsulating multiple implementation of a NetworkInterface (UML class diagram)

```
public class NetworkConnection
//.....
private NetworkInterface nif,
void send(byte msg]);
if (nif.isReady()) {
    nif.send (queue);
    queue.setLength (0);
    }
}
}
```

```
public class NetworkConnection
//....
private NetworkInterface nif,
void send (byte msg ]) {
queue.concat (msg);
boolean ready=false;
if (nif instanceof Ethernet) {
   Ethernet eNif=(Ethernet)nif,
    ready= eNif.isReady();
} else if (nif instanceof WaveLAN) {
   WaveLAN wNif=(WaveLAN) nif,
    ready= wNif.isReady();
} else if (nif instanceof UMTS) {
   UMTS uNif=(UMTS) nif,
    ready= uNif.isReady(); }
if (ready) {
   if (nif instanceof Ethernet) {
       Ethernet eNif=(Ethernet) nif,
       eNif.send(queue);
} else if (nif instanceof WaveLAN) {
       WaveLAN wNif=(WaveLAN) nif,
       wNif.send(queue);
} else if (nif instanceof UMTS) {
       UMTS uNif=(UMTS) nif,
      uNif.send(queue);)
      queue.setLength(0);
     }
    }
    }
```

Figure 4.21: Java source code for the NetworkConnection.send() message (left) and equivalent Java source code without polymorphism (right).

When applying the path testing technique to an operation that uses polymorphism, all dynamic bindings should be considered, one for each message that could be sent. In NetworkConnect.send() (found in the top part of java source code), the NetworkInterface.send() operation is invoked, which can be bound to either Ethernet.send(), WaveLAN.send(), or the UMTS.send() methods, depending on the class of the nif object. To deal with this situation explicitly, the original source code is expanded by replacing each invocation of NetworkInterface.send() with a nested 'if else' statement that tests for all subclasses of NetworkInterface. Once the source code is expanded, the flow graph is extracted and test cases are generated covering all paths. When many interfaces and abstract classes are involved, generating the flow graph for a method of medium complexity can result in an explosion of paths. The equivalent flow graph is shown below:

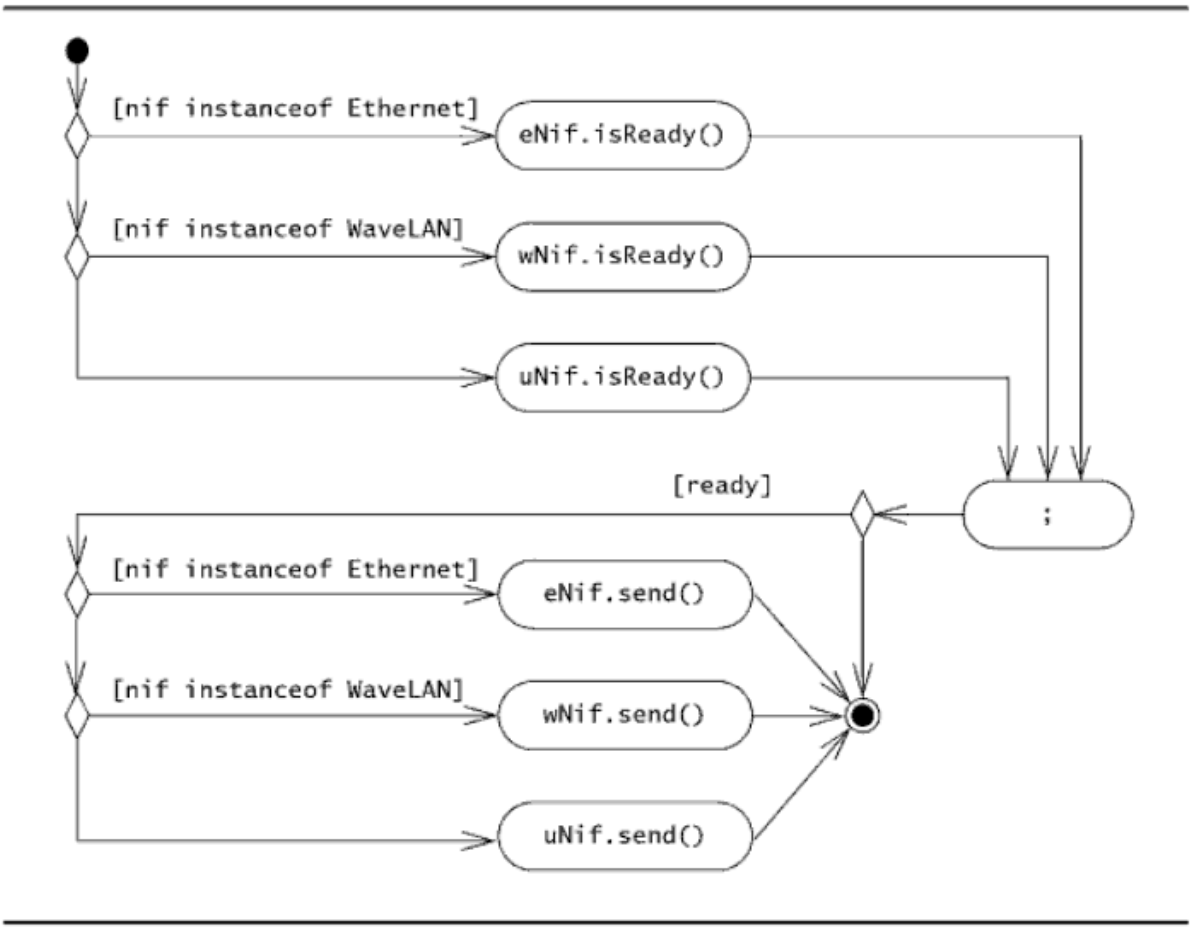


Figure 4.22: Equivalent flow graph for the expanded source code of the NetworkConnection.send() method of the Java implementation (UML activity diagram)

Integration Testing:

Integration testing detects faults that have not been detected during unit testing by focusing on small groups of components. Two or more components are integrated and tested, and when no new faults are revealed, additional components are added to the group. Since developing test stubs and drivers is time consuming, Extreme Programming is used. The order in which components are tested, influences integration testing. A careful ordering of components can reduce the resources needed for the overall integration test.

Integration testing strategies:

Several approaches have been devised to implement an integration testing strategy: Big-bang testing, bottom-up testing, top-down testing and sandwich testing.

The **big-bang testing** strategy assumes that all components are first tested individually and then tested together as a single system. The advantage is that no additional test stubs or drivers are needed. Although this sounds simple it is expensive. If attest uncovers a failure, it is impossible to distinguish failures in the interface from failures within a component. Moreover, it is difficult to pinpoint a specific component.

The **bottom-up testing** first tests each component of the bottom layer individually, and then integrates them with components of the next layer up. If two components are tested together, it is called a double test. Similarly, three components means triple test and four means quadruple test. This is repeated until all components from all layers are combined. Test drivers are used to simulate the components of higher layers that have not yet been integrated. Test stubs are not necessary during this testing. The advantage of bottom-up testing is that interface faults can be more easily found. The disadvantage is that it tests the most important subsystem, namely the components of the user interface, last.

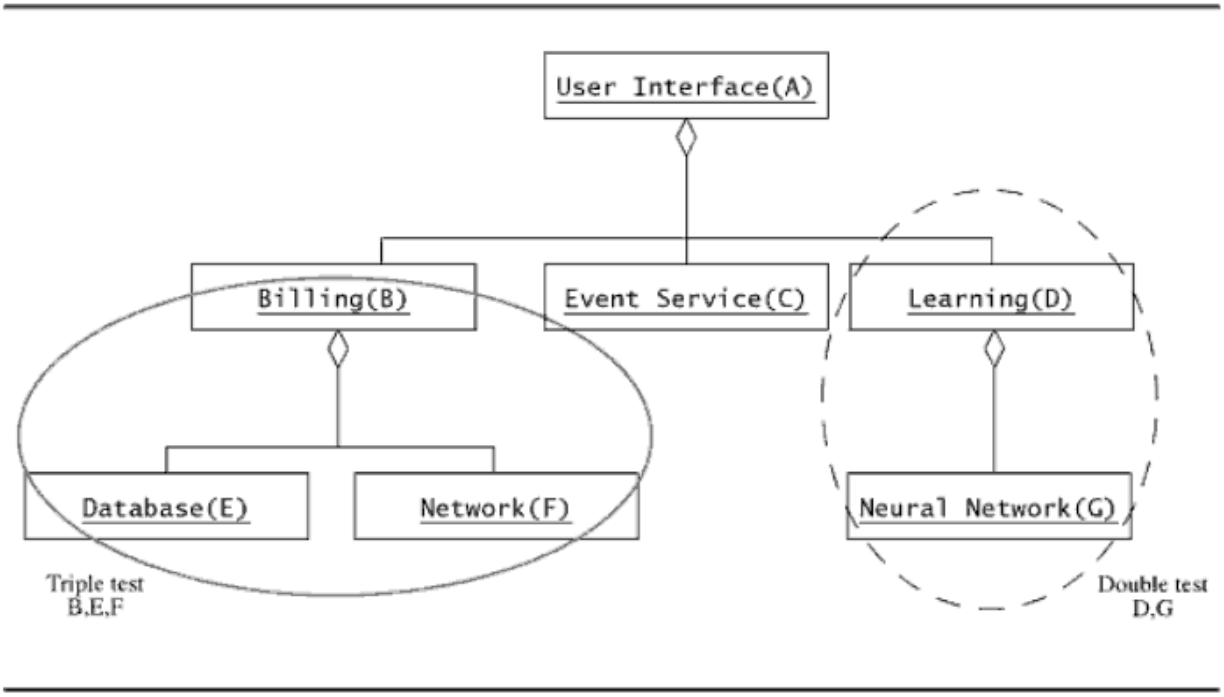


Figure 4.23: Bottom-up testing strategy

The above figure illustrates bottom-up testing. Here the subsystems E, F, and G are unit tested first, and then the triple test B-E-F and the double test D-G are executed and so on.

The **top-down testing** strategy unit tests the components of the top layer first, and then integrates the components of the next layer down. When all components of the new layer have been tested together, the next layer is selected. Again, the tests incrementally add one component at a time. This is repeated until all layers are combined and involved in the test. Test stubs are used to simulate the components of lower layers that have not yet been integrated. Test drivers are not required during this testing process. The advantage of top-down testing is that it starts with user interface components. The same set of tests, derived from the requirements, can be used in testing the increasingly more complex set of subsystems. The disadvantage is that the development of test stubs is time consuming and prone to error.

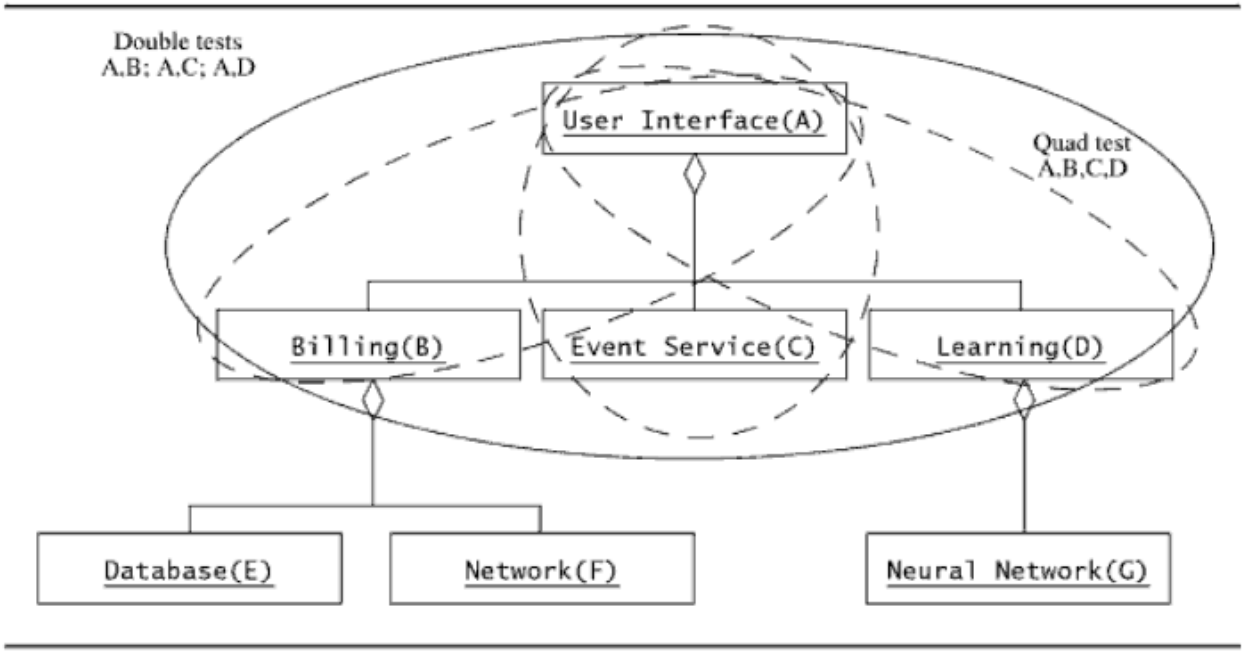


Figure 4.24: Top-down testing strategy

The above figure illustrates top-down testing strategy. Here, the subsystem A is unit tested, then double tests A-B, A-C, and A-D are executed, then the quad test A-B-C-D is executed, and so on.

The **sandwich testing** strategy combines the top-down and bottom-up strategies to make use of the best of both. During this testing, the tester must be able to map the subsystem decomposition into three layers, a target layer ("the meat"), a

layer above the target layer ("the top slice of bread"), and a layer below the target layer ("the bottom slice of bread"). Top-down integration testing is done by testing the top layer incrementally with the components of the target layer, and bottom-up testing is used for testing the bottom layer incrementally with the components of the target layer. As a result, test stubs and drivers need not be written for the top and bottom layers, because they use the actual components from the target layer. The problem with this testing is that it does not thoroughly test the individual components of the target layer before integration. For example, the sandwich test shown below does not unit test component C of the target layer.

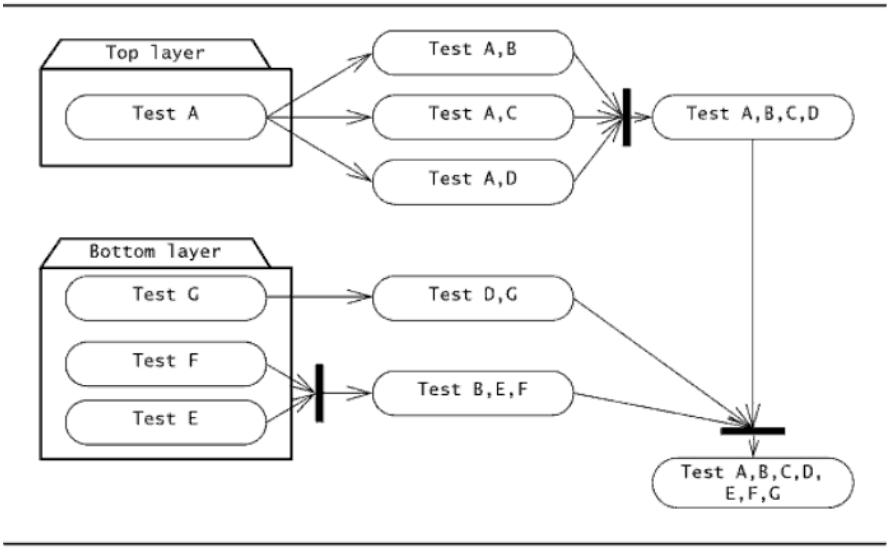


Figure 4.25: Sandwich testing strategy (UML activity diagram)

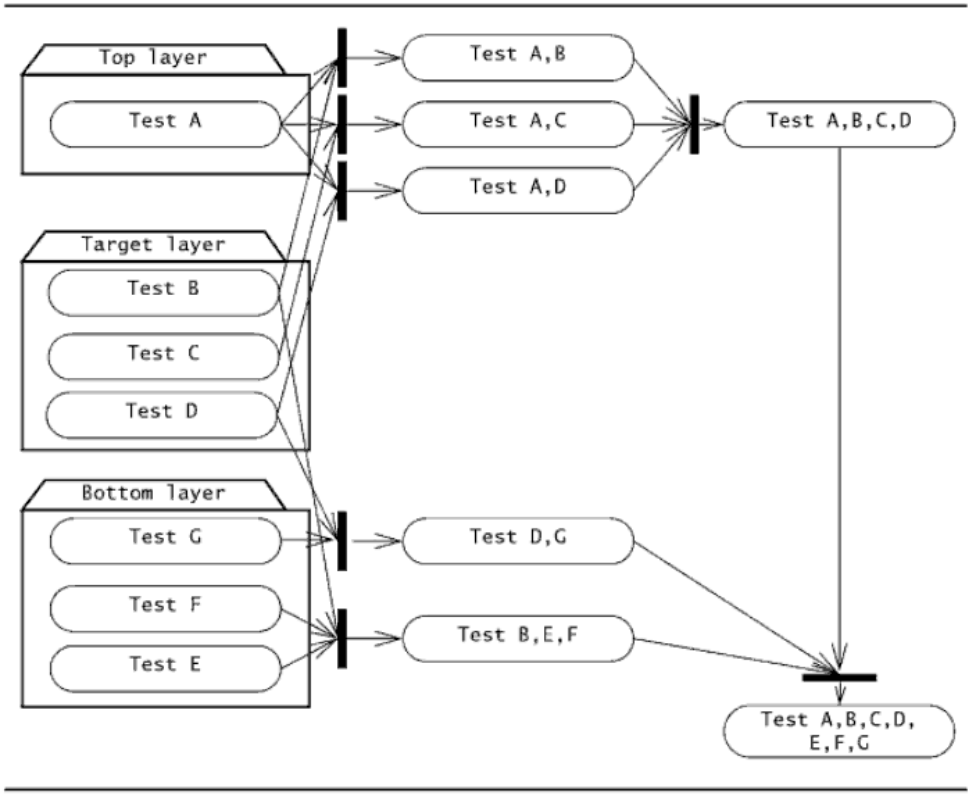


Figure 4.26: Modified sandwich testing strategy (UML activity diagram)

The *modified sandwich testing* strategy tests the three layers individually before combining them in incremental tests with one another. The individual layer tests consist of a group of three tests:

- a top layer test with stubs for the target layer
- a target layer test with drivers and stubs replacing the top and bottom layers
- a bottom layer test with a driver for the target layer.

The combined layer tests consist of two tests:

- The top layer the target layer. This test can reuse the target layer tests from the individual layer tests, replacing the drivers with components from the top layer.
- The bottom is accessed by the target layer. This test can reuse the target layer tests from the individual layer tests, replacing the stub with components from the bottom layer.

The advantage of modified sandwich testing is that many testing activities can be performed in parallel, as indicated by the UML activity diagrams. The disadvantage is the need for additional test stubs and drivers. But, modified testing leads to a significantly shorter overall testing time than top-down or bottom-up testing.

System testing:

Once components have been integrated, system testing ensures that the complete system complies with the functional and nonfunctional requirements. System testing is a blackbox technique: test cases are derived from the use case model. Several system testing activities are performed which are listed below:

- Functional testing
- Performance testing
- Pilot testing
- Acceptance testing
- Installation testing

Functional testing:

Functional testing, also called requirements testing, finds differences between the functional requirements and the system. The goal of the tester is to select those tests that are relevant to the user and have a high probability of uncovering a failure. Functional testing is different from usability testing. Functional testing finds differences between the use case model and the observed system behavior, whereas usability testing finds difference between the use case model and user's expectation of the system.

To identify functional tests, the use case model is inspected and use case instances that are likely to cause failures are identified. Test cases identified should exercise both common and exceptional use cases. For example, the use case model for a 'subway ticket distributor' (shown by the UML use case diagram) is considered. Here the common functionality is 'PurchaseTicket' use case, describing the steps necessary for a passenger to successfully purchase a ticket and the various exceptional conditions are 'TimeOut', Cancel, OutOfOrder, and NoChange use cases. The exceptional conditions result from the state of the distributor or actions by the Passenger.

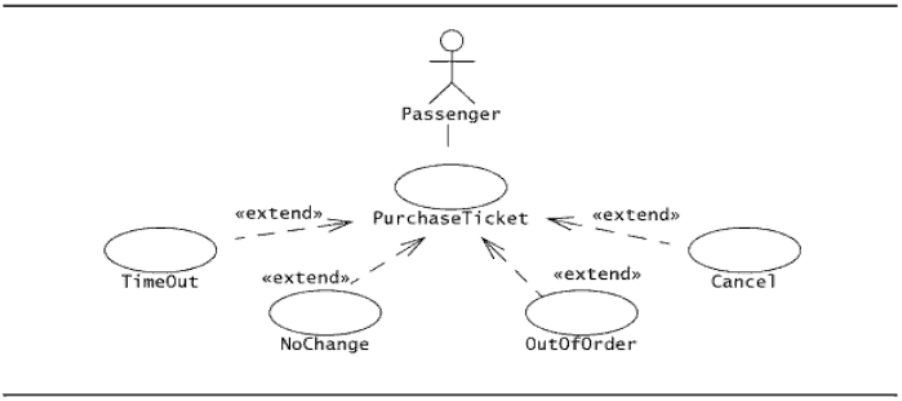


Figure 4.27: An example of use case model for a subway ticket distributor (UML use case diagram)

The following is the PurchaseTicket use case which describes the normal interaction between the Passenger actor and the Distributor.

Use case name	PurchaseTicket
Entry condition	The Passenger is standing in front of ticket Distributor The Passenger has sufficient money to purchase ticket
Flow of events	1. The Passenger selects the number of zones to be traveled. If the Passenger presses multiple zone buttons, only the last button pressed is considered by the Distributor. 2. The Distributor displays the amount due. 3. The Passenger inserts money. 4. If the Passenger selects a new zone before inserting sufficient money, the Distributor returns all the coins and bills inserted by the Passenger. 5. If the Passenger inserted more money than the amount due, the Distributor returns excess change. 6. The Distributor issues tickets. 7. The Passenger picks up the change and the ticket.
Exit condition	The Passenger has the selected ticket.

Figure 4.28: An example of use case from the ticket distributor use case model PurchaseTicket.

The following are the three features of the Distributor that are likely to fail and should be tested:

- The Passenger may press multiple zone buttons before inserting money, in which case the Distributor should display the amount of the last zone.
- The Passenger may select another zone button after beginning to insert money, in which case the Distributor should return all money inserted by the Passenger.
- The Passenger may insert more money than needed, in which case the Distributor should return the correct change.

The following is the test case PurchaseTicket CommonCase, which exercises the above three features. The flow of events describes both the input to the system

(stimuli that the Passenger sends to the Distributor) and desired outputs (correct responses from the Distributor).

Test case name	PurchaseTicket_CommonCase
Entry condition	The Passenger standing in front of ticket Distributor The Passenger has two $5 bills and three dimes.

Flow of events	1. The Passenger presses in succession the zone buttons 2, 4, 1, and 2. 2. The Distributor should display in succession $i.25, $2.25, $0.75, and $1.25. 3. The Passenger inserts a $5 bill. 4. The Distributor returns three $1 bills and three quarters and issues a 2- zone ticket. 5. The Passenger repeats steps 1-4 using his second $5 bill. 6. The Passenger repeats steps 1-3 using four quarters and three dimes. The Distributor issues a 2-zone ticket and returns a nickel. 7. The Passenger selects zone 1 and inserts a dollar bill. The Distributor issues a 1-zone ticket and returns a quarter. 8. The Passenger selects zone 4 and inserts two $1 bills and a quarter. The Distributor issues a 4-zone ticket. 9. The Passenger selects zone 4. The Distributor displays $2.25. The Passenger inserts a $1 bill and a nickel, and selects zone 2. The Distributor returns the $1 bill and the nickel and displays $1.25.
Exit condition	The Passenger has three 2-zone tickets, one 1-zone ticket, and one 4-zone ticket.

Figure 4.29: An example of test case derived from the PurchaseTicket use case.

Similar test cases can also be derived for the exceptional use cases NoChange, OutOfOrder, TimeOut, and Cancel. Test cases such as PurchaseTicket_CommonCase, are derived for all use cases, including use cases representing exceptional behavior. Test cases are associated with the use cases from which they are derived, making it easier to update the test cases when use cases are modified.

Performance testing:

Performance testing finds differences between the design goals selected during system design and the system. Because the design goals are derived from the nonfunctional requirements, the test cases can be derived from the SDD (System

Design Document) or from the RAD (Rapid Application Development). The following tests are performed during performance testing:

- *Stress testing* checks if the system can respond to many simultaneous requests.
- *Volume testing* attempts to find faults associated with large amounts of data, such as static limits imposed by the data structure, or high-complexity algorithms, or high disk fragmentation.
- *Security testing* attempts to find security faults in the system. Usually this test is accomplished by "tiger teams" who attempt to break into the system, using their experience and knowledge of typical security flaws.
- *Timing testing* attempts to find behaviors that violate timing constraints described by the nonfunctional requirements.
- *Recovery tests* evaluate the ability of the system to recover from errorneous states, such as the unavailability of resources, a hardware failure, or a network failure.

After all the functional and performance tests have been performed, and no failures have been detected during these tests, the system is said to be validated.

Pilot testing:

During the pilot test, also called the field test, the system is installed and used by a selected set of users. Users exercise the system as if it had been permanently installed. No explicit guidelines or test scenarios are given to the users. A group of people is invited to use the system for a limited time and to give their feedback to the developers. This test is useful for systems without a specific set of requirements.

An *alpha test* is a pilot test with users exercising the system in the development environment. In a *beta test*, the acceptance test is performed by a limited number of end users in the target environment. The Internet has made the distribution of software very easy. As a result, beta tests are more and more common.

Acceptance testing:

There are three ways the client evaluates a system during acceptance testing. In a *benchmark test*, the client prepares a set of test cases that represent typical conditions under which the system should operate. Benchmark tests can be performed with actual users or by a special test team exercising the system functions.

Another kind of system acceptance testing is used in reengineering projects, when the new system replaces an existing system. In *competitor testing*, the new system is tested against an existing system or competitor product. In *shadow testing*, a

form of comparison testing, the new and the legacy systems are run in parallel and their outputs are compared.

After acceptance testing, the client reports to the project manager which requirements are not satisfied. If requirements must be changed, the changes should be reported in the minutes to the client acceptance review and should form the basis for another iteration of the software life-cycle process. If the customer is satisfied, the system is accepted, possibly contingent on a list of changes recorded in the minutes of the acceptance test.

Installation testing:

After the system is accepted, it is installed in the target environment. The desired outcome of the installation test is that the installed system correctly addresses all requirements. In most cases, the installation test repeats the test cases executed during function and performance testing in the target environment. Once the customer is satisfied with the results of the installation test, system testing is complete, and the system is formally delivered and ready for operation.

Regression Testing:

Object-oriented development is an iterative process. When modifying a component, developers design new unit tests exercising the new feature under consideration. They may also retest the component by updating and rerunning previous unit tests. The modification can introduce side effects or reveal previously hidden faults in other components. A system that fails after the modification of a component is said to regress. Hence, integration tests that are rerun on the system to produce such failures are called **regression tests**.

The most robust and straightforward technique for regression testing is to accumulate all integration tests and rerun them whenever new components are integrated into the system. This requires developers to keep all tests up-to-date, to evolve them as the subsystem interfaces change, and to add new integration tests as new services or new subsystems are added. As regression testing can become time consuming, different techniques have been developed for selecting specific regression tests. Such techniques include:

- *Reset dependent components*- Testing the components that are dependent on the modified component, because they are the most likely to fail.
- *Retest risky use cases*- Focusing first on use cases that present the highest risk. By this, the developers can minimize the likelihood of catastrophic failures.
- *Retest frequent use cases*- Focusing on the use cases that are most often used by the users.

In all cases, regression testing leads to running many tests many times. Hence, regression testing is feasible only when an automated testing infrastructure is in place.

User Acceptance Testing:

Refer to acceptance testing in the system testing section.

Product testing:

Product testing is a type of usability testing. Usability testing is a technique for ensuring that the intended users of a system can carry out the intended tasks efficiently, effectively and satisfactorily. During product testing the end users are presented with a functional version of the system. This test can be conducted after most of the system is developed. It also requires that the system be easily modifiable such that the results of the usability test can be taken into account. The basic elements of this test include:

- development of test objective
- a representative sample of end users
- the actual or simulated work environment
- controlled, extensive interrogation, and probing of the users by the person performing the test
- collection and analysis of quantitative and qualitative results
- recommendations on how to improve the system

Typical objectives in a usability test address the comparison of two user interaction styles, the main stumbling blocks, the identification of useful features for novice and expert users, when help is needed, and what type of training information is required.

Maintenance

Once the product has its acceptance test, it is handed over to the client. The product is installed and used for the purpose for which it was constructed. Any useful product, however, is almost certain to undergo maintenance during the maintenance phase, either to fix faults (corrective maintenance) or extend the functionality of the product (enhancement).

6.1.1 Types of Maintenance

The following are the types of maintenance:

i. Corrective maintenance: This involves correcting faults, whether specification faults, design faults, coding faults, documentation faults, or any other types of faults. A study of 69 organizations showed that maintenance programmers spend only 17.5 percent of their time on corrective maintenance.

ii. Perfective maintenance: Here, changes are made to the code to improve the effectiveness of the product. For instance, the client may wish additional functionality or request that the product be modified so that it runs faster. Improving the maintainability of a product is another example of perfective maintenance. The study showed that 60.5 percent time was spent on this type of maintenance.

iii. Adaptive maintenance: Here, changes are made to the product to react to changes in the environment in which the product operates. For example, a product almost certainly has to be modified if it is ported to a new compiler, operating system, or hardware. Adaptive maintenance is not requested by a client; instead, it is externally imposed on the client. The study showed that 18 percent of software maintenance was adaptive in nature.

The remaining 4 percent of maintenance time was devoted to other types of maintenance that did not fall into the above mentioned three categories. *Preventive maintenance* is another type which concerns activities aimed at increasing the system's maintainability, such as updating documentation, adding comments, and improving the modular structure of the system.

6.1.2 Maintenance Log and Defect reports

Maintenance Log keeps track of the cost of operations performed, scheduled maintenance, and unscheduled repairs. In short, it is the storage of activities and daily updates involved in the maintenance phase.

The first thing needed when maintaining a product is a mechanism for changing the product. With regard to corrective maintenance, that is, removing residual faults, if the product appears to be functioning incorrectly, then a fault/defect report should be filed by the user. This must include enough information to enable the maintenance programmer to recreate the problem, which usually will be sort of software failure.

Ideally, every fault reported by a user should be fixed immediately. In practice, programming organizations usually are understaffed, with a backlog of work, both development and maintenance. If the fault is critical, such as if a payroll product crashes the day before payday or overpays employees, immediate corrective action must be taken. Otherwise, each fault report must at least receive an immediate preliminary investigation.

The maintenance programmer should first consult the fault report file. This contains all reported faults that have not yet been fixed, together with suggestions for working around them, that is, ways for the user to bypass the portion of the product that apparently is responsible for the failure, until such time as the fault can be fixed. If the fault has been reported previously, any information in the fault report file should be given to the user. But if what the user reports appear to be a new fault, then the maintenance programmer should study the problem and attempt to find the cause and a way to fix it. In addition, an attempt should be made to find a way to work around the problem, because it may take 6 or 9 months before someone can be assigned to make the necessary changes to the software. In the light of the serious shortage of programmers, and in particular programmers good enough to perform maintenance, suggesting a way to live with the fault until it can be solved often is the only way to deal with fault reports that are not true emergencies.

The maintenance programmer's conclusions then should be added to the fault report file, together with any supporting documentation, such as listings, designs, and manuals used to arrive at those conclusions. The manager in charge of maintenance should consult the file regularly, setting priorities for the various fixes. The file also should contain the client's requests for perfective and adaptive maintenance. The next modification made to the product then will be the one with the highest priority.

When copies of a product have been distributed to a variety of sites, copies of fault reports must be circulated to all users of the product, together with an estimate of when each fault can be fixed. Then, if the same failure occurs at another site, the user can consult the relevant fault report to determine if it is possible to work around the fault and when it will be fixed. It would be preferable to fix every fault immediately and then distribute a new version of the product to all sites, of course. Also organizations prefer to accumulate noncritical maintenance tasks then implement the changes as a group.

6.1.3 Reverse and Reengineering

An application has served the business needs of a company for 10 or 15 years. During that time it has been corrected, adapted, and enhanced many times. People approached this work with the best intentions, but good software engineering practices were always shunted to the side (the press of other matters). Now the application is unstable. It still works, but every time a change is attempted, unexpected and serious side effects occur. Yet the application must continue to evolve. The solution is software reengineering that has been spawned by software maintenance.

Reengineering is concerned about adapting existing systems to changes in their external environment and making enhancements requested by users. Only about 20 percent of all maintenance work is spent "fixing mistakes", the remaining 80 percent is spent in reengineering for future use.

A Software Reengineering Process model:

Reengineering takes time; it costs significant amounts of money; and it absorbs resources that might be otherwise occupied on immediate concerns. For all of these reasons, reengineering is not accomplished in a few months or even a few years. Reengineering of information systems is an activity that will absorb information technology resources for many years. That's why every organization needs a pragmatic strategy for software reengineering. A software reengineering model that defines six activities is shown by the following figure:

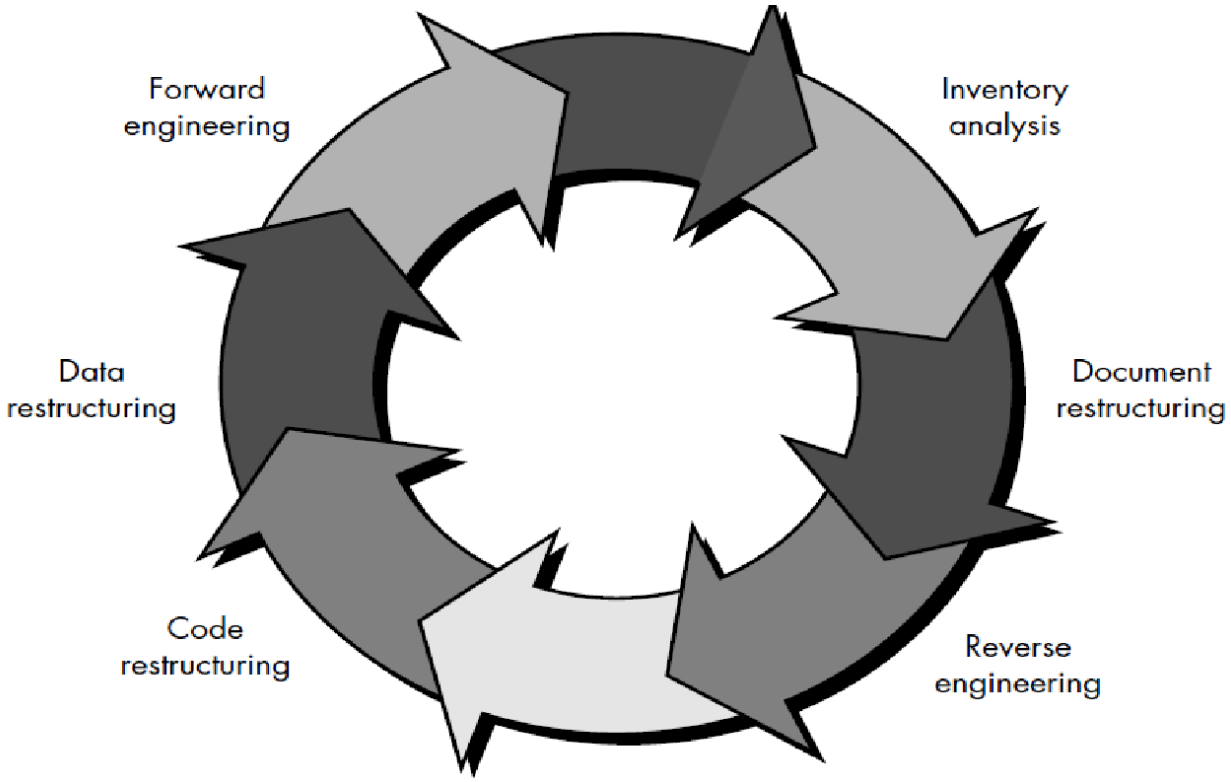


Figure 6.1: A software reengineering model

The reengineering paradigm shown in the figure is a cyclical model. This means that each of the activities presented as a part of the paradigm may be revisited. For any particular cycle, the process can terminate after any one of these activities. Each activity is described down.

Inventory analysis:

Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity, current maintainability, and other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.

Document restructuring:

Weak documentation is the trademark of many legacy systems. The following options can be adopted:

- Creating documentation is far too time consuming. If the system works, we'll live with what we have.
- Documentation must be updated, but we have limited resources. We'll use a "document when touched" approach. (i.e., documenting only changed portions of the system)
- The system is business critical and must be fully documented. (Even in this case, an intelligent approach is to pare documentation to an essential minimum)

Each of these options is viable. A software organization must choose the one that is most appropriate for each case.

Reverse Engineering:

The term *reverse engineering* has its origins in the hardware world. A company disassembles a competitive hardware product in an effort to understand its competitor's design and manufacturing "secrets." These secrets could be easily understood if the competitor's design and manufacturing specifications were obtained. But these documents are proprietary and unavailable to the company doing the reverse engineering. In essence, successful reverse engineering derives one

or more design and manufacturing specifications for a product by examining actual specimens of the product.

Reverse engineering for software is quite similar. In most cases, however, the program to be reverse engineered is not a competitor's. Rather, it is the company's own work (often done many years earlier). The "secrets" to be understood are obscure because no specification was ever developed. Therefore, reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is a process of design recovery. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

Code restructuring:

The most common type of reengineering is code restructuring. Some legacy systems have relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted and code is then restructured (this can be done automatically). The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

Data restructuring:

A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, data architecture has more to do with the long-term viability of a program that the source code itself.

Unlike code restructuring, which occurs at a relatively low level of abstraction, data structuring is a full-scale reengineering activity. In most cases, data restructuring begins with a reverse engineering activity. Current data architecture is dissected and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.

When data structure is weak, the data are reengineered. Because data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

Forward engineering:

In an ideal world, applications would be rebuilt using an automated “reengineering engine.” The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality. In the short term, it is unlikely that such an “engine” will appear,

but CASE vendors have introduced tools that provide a limited subset of these capabilities that addresses specific application domains (e.g., applications that are implemented using a specific database system). More importantly, these reengineering tools are becoming increasingly more sophisticated.

Forward engineering, also called *renovation* or *reclamation*, not only recovers design information from existing software, but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software reimplements the function of the existing system and also adds new functions and/or improves overall performance.

10. Objective Questions:

1. The following is a maintenance activity
    a. fixing faults
    b. generating code
    c. extending the functionality of the system
    d. both a and c
2. Which of the following is a maintenance type?
    a. corrective
    b. perfective
    c. adaptive
    d. all of the above
3. Maintenance Log keeps track of which of the following:
    a. cost of operations performed
    b. scheduled maintenance
    c. unscheduled repairs
    d. all of the above
4. Business process reengineering has no start or end—it is an evolutionary process.
    a. True
    b. False
5. How much software maintenance work involves fixing errors?
    a. 20 percent
    b. 40 percent
    c. 60 percent
    d. 80 percent
6. Which of the following activities is not part of the software reengineering process model?
    a. forward engineering
    b. inventory analysis
    c. prototyping
    d. reverse engineering

7. The software reengineering process model includes restructuring activities for which
of the following work items?
a. code
b. documentation
c. data
d. all of the above

8. Reverse engineering of data focuses on
a. database structures
b. internal data structures
c. both a and b
d. none of the above

9. Reverse engineering should precede the reengineering of any user interface.
a. True
b. False

10. Forward engineering is not necessary if an existing software product is producing
the correct output.
a. True
b. False

11. The cost benefits derived from reengineering are realized largely due to decreased
maintenance and support costs for the new software product.
a. True
b. False

Answers: 1. d 2. d 3. d 4. a 5. a 6. c 7. d 8. c 9. a 10. b 11. a

11. Subjective Questions:

1. Explain in detail the different types of maintenance?

Ans: Refer page no.207-208

2. Write notes on reengineering?

Ans: Refer page no.209-213

3. Explain in detail about reverse engineering and forward engineering?

Ans: Refer page no.211-213

12. Questions:

1. Write a short note on Re-engineering. (May 2010) (10 marks)

Ans:

It is concerned about adapting existing systems to changes in their external environment and making enhancements requested by users.

Software re-engineering model:

- Forward engineering
- Data restructuring
- Code restructuring
- Inventory analysis
- Document restructuring
- Reverse engineering

2. What are different types of maintenance and also explain the different steps involved
in creating a maintenance log? (Nov 2010) (10 marks)

Ans:

Types of maintenance:

- Corrective
- Perfective
- Adaptive
- Preventive

Maintenance log:

Maintenance Log keeps track of the cost of operations performed, scheduled maintenance, and unscheduled repairs. In short, it is the storage of activities and daily updates involved in the maintenance phase.

3. Write short notes on Reverse and Re-engineering. (Nov 2010) (10 marks)

Ans:

Reengineering is concerned about adapting existing systems to changes in their external environment and making enhancements requested by users. Only about 20 percent of all maintenance work is spent "fixing mistakes", the remaining 80 percent is spent in reengineering for future use.

A software reengineering model that defines six activities as follows:

- Forward engineering
- Inventory analysis
- Data restructuring
- Document restructuring
- Code restructuring
- Reverse engineering

4. Explain various software testing strategies. (May 2010, Nov 2010) (10 marks)

Ans:

- Unit testing
- Regression testing (additional test cases for software functions)

- Integration testing (bottom-up, top down, sandwich)
- Validation testing
  - Alpha- customer tests
  - Beta-developer tests
- Acceptance testing
  - Benchmark
  - Pilot
  - Parallel
- System testing
  - Recovery
  - Security
  - Stress
  - Performance

5. Explain objectives for testing. Also explain the following terms: (Nov 2010) (10 marks)

(i) System testing
(ii) Scalability
(iii) Regression
(iv) Black box testing.

Ans:
Objectives for testing:

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation.

(i) System testing:
Once components have been integrated, system testing ensures that the complete system complies with the functional and nonfunctional requirements. System testing is a blackbox technique: test cases are derived from the use case model. Several system testing activities are performed which are listed below:

- Functional testing
- Performance testing
- Pilot testing
- Acceptance testing
- Installation testing

(ii) Scalability:

Scalability is the ability of a system, or process, to handle growing amounts of work in a graceful manner or its ability to be enlarged to accommodate that growth.

(iii) Regression testing:

A system that fails after the modification of a component is said to regress. Hence, integration tests that are rerun on the system to produce such failures are called **regression tests**.

Different techniques for selecting specific regression tests:

- Reset dependent components
- Retest risky use cases
- Retest frequent use case

(iv) Black box testing

Black-box testing is a method of software testing that tests the functionality of an application as opposed to its internal structures or workings

Typical black-box test design techniques include:

- Decision table testing
- All-pairs testing
- State transition tables
- Equivalence partitioning
- Boundary value analysis

Object Oriented Testing methods

Fault-based testing

- The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.

Class Testing and the Class Hierarchy

- Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.

Scenario-Based Test Design

- Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests

13. Learning Resources:

1. Bernd Bruegge "Object oriented software engineering", Second Edition, Pearson Education.

2. Roger Pressman, "Software Engineering", sixth edition, Tata McGraw Hill.

Name of Student		
Class		
Roll No.		
Subject		
Module No.		
S.No		Tick Your choice
1.	Do you understand the Testing Process and Basic concept of software testing?	○ es ○ No
2.	Do you understand terminology, Verification & validation, White Box Testing Path Testing, Control Structures Testing, DEFUSE testing?	○ es ○

		o
3.	Do you recall the Software Maintenance and Reverse Engineering?	○ es ○ o