Module IV

Syllabus

- Concepts of Agile Development methodology; Scrum Framework.
- Software testing principles, Program inspections, Program walkthroughs, Program reviews; Blackbox testing: Equivalence class testing, Boundary value testing, Decision table testing, Pairwise testing, State transition testing, Use-case testing; White box testing: control flow testing, Data flow testing.
- Testing automation: Defect life cycle; Regression testing, Testing automation; Testing nonfunctional requirements.

Fundamentals of Agile Development

- Agile software engineering combines a philosophy and a set of development guidelines.
- The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity.
- The development guidelines stress delivery over analysis and design (although these activities are not discouraged), and active and continuous communication between developers and customers.
- It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more simplistic.
- It emphasizes rapid delivery of operational software and de-emphasizes the importance of intermediate work products (not always a good thing); it adopts the customer as a part of the development team.
- it recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.
- Agility can be applied to any software process.

Agile Process

- It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
- For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
- Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Agility Principles

- highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.

Agility Principles contd....

- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self– organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile development Models

Some of the Agile development models are:

- Scrum
- Extreme Programming
- Feature driven development
- Lean software Development

Scrum

- Scrum is an agile software development method.
- Scrum principles are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery.
- Within each framework activity, work tasks occur within a process pattern called a sprint.
- The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.

Scrum meetings

- **Scrum meetings** —are short (typically 15 minutes) meetings held daily by the Scrum team.
- Three key questions are asked and answered by all team members :
  1. What did you do since the last team meeting?
  2. What obstacles are you encountering?
  3. What do you plan to accomplish by the next team meeting?
- A team leader, called a Scrum master, leads the meeting and assesses the responses from each person.
- The Scrum meeting helps the team to uncover potential problems as early as possible.
- Also, these daily meetings lead to "knowledge socialization" and thereby promote a self-organizing team structure.

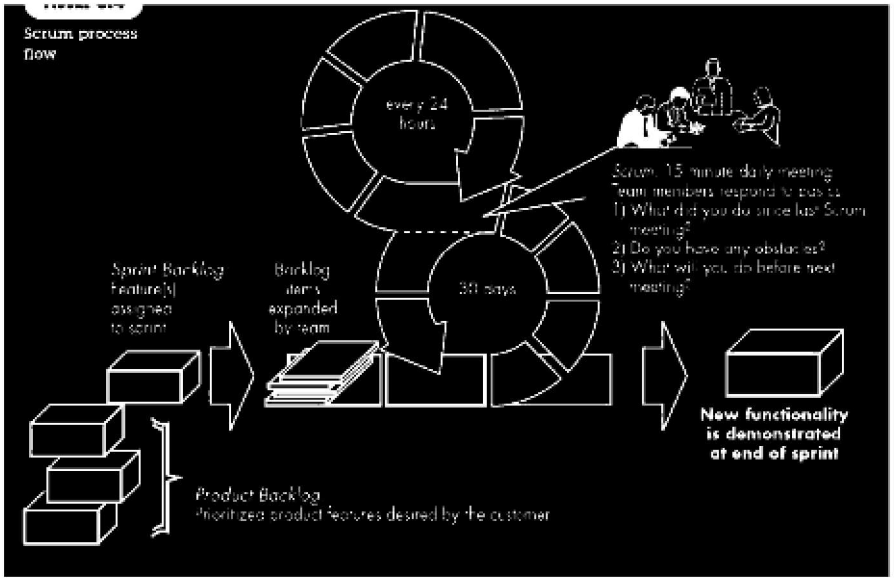
Scrum - Backlog and Sprint

- Backlog— a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time(this is how the changes are introduced).
- The product manager assesses the backlog and updates priorities as required.
- Sprints—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box.
- Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.

Scrum - Demos

- **Demos** — deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer.
- the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

Scrum Process Flow



Software Testing Principles (guidelines)

Principle Number	Principle
1	A necessary part of a test case is a definition of the expected output or result.
2	A programmer should avoid attempting to test his or her own program
3	A programming organization should not test its own programs.
4	Thoroughly inspect the results of each test
5	Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
6	Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
7	Avoid throwaway test cases unless the program is truly a throwaway program.
8	Do not plan a testing effort under the tacit assumption that no errors will be found
9	The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
10	Testing is an extremely creative and intellectually challenging task.

Software testing - Definition

- "Testing is the process of demonstrating that errors are not present. "
- "The purpose of testing is to show that a program performs its intended functions correctly."
- "Testing is the process of establishing confidence that a program does what it is supposed to do."
- Testing is the process of executing a program with the intent of finding errors.

Principle 1: A necessary part of a test case is a definition of the expected output or result.

a test case must consist of two components:

- 1. A description of the input data to the program.
- 2. A precise description of the correct output of the program for that set of input data.

Principle 2: A programmer should avoid attempting to test his or her own program.

- After a programmer has constructively designed and coded a program, it is extremely difficult to suddenly change perspective to look at the program with a destructive eye.
- programmer may subconsciously avoid finding errors for fear of retribution from peers or from a supervisor, a client, or the owner of the program or system being developed.
- The program may contain errors due to the programmer's misunderstanding of the problem statement or specification.

Principle 3: A programming organization should not test its own programs.

- it is difficult for a programming organization to be objective in testing its own programs, because the testing process, if approached with the proper definition, may be viewed as decreasing the probability of meeting the schedule and the cost objectives.
- it is more economical for testing to be performed by an objective, independent party.

Principle 4: Thoroughly inspect the results of each test.

- many subjects failed to detect certain errors, even when symptoms of those errors were clearly observable on the output listings.
- errors that are found on later tests are often missed in the results from earlier tests.

Principle 5: Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.

- test cases representing unexpected and invalid input conditions seem to have a higher error-detection yield than do test cases for valid input conditions.

Principle 6: Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.

- Programs must be examined for unwanted side effects.
- For instance, a payroll program that produces the correct paychecks is still an erroneous program if it also produces extra checks for nonexistent employees or if it over- writes the first record of the personnel file.

Principle 7: Avoid throwaway test cases unless the program is truly a throwaway program.

- Whenever the program has to be tested again (for example, after correcting an error or making an improvement), the test cases must be reinvented.
- More often than not, since this reinvention requires a considerable amount of work, people tend to avoid it.
- Saving test cases and running them again after changes to other components of the program is known as regression testing.

Principle 8: Do not plan a testing effort under the tacit assumption that no errors will be found.

- incorrect definition of testing— is, the assumption that testing is the process of showing that the program functions correctly.
- the definition of testing is the process of executing a program with the intent of finding errors.

Principle 9: The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.

- errors tend to come in clusters and that, in the typical program, some sections seem to be much more prone to errors than other sections, although nobody has supplied a good explanation of why this occurs.
- If a particular section of a program seems to be much more prone to errors than other sections, then this phenomenon tells us that, in terms of yield on our testing investment, additional testing efforts are best focused against this error-prone section.

Principle 10: Testing is an extremely creative and intellectually challenging task.

- the creativity required in testing a large program exceeds the creativity required in designing that program.

Conclusion - three important principles of testing:

• Testing is the process of executing a program with the intent of finding errors.

• A good test case is one that has a high probability of detecting an as yet undiscovered error.

• A successful test case is one that detects an as yet undiscovered error.

human testing methods

- The two primary human testing methods are code inspections and walkthroughs.

- participants must conduct some preparatory work.
- The climax is a "meeting of the minds," at a participant conference.
- The objective of the meeting is to find errors but not to find solutions to the errors. That is, to test, not debug.

Code Inspections

- A code inspection is a set of procedures and error-detection techniques for group code reading.

- An inspection team usually consists of four people. One of the four people plays the role of moderator.
- The moderator is expected to be a competent programmer, but he or she is not the author of the program and need not be acquainted with the details of the program.

An Error Checklist for Inspections

- An important part of the inspection process is the use of a checklist to examine the program for common errors.
- **Data Reference Errors**
- Data-Declaration Errors
- **Computation Errors - Are there any computations using variables having inconsistent (such as nonarithmetic) datatypes?**
- Comparison Errors - any comparisons between variables having different datatypes
- Control-Flow Errors
- Interface Errors - Does the number of parameters received by this module equal the number of arguments sent by each of the calling modules?
- Input/Output Errors

Walkthroughs

- is a set of procedures and error-detection techniques for group code reading.
- the walkthrough is an uninterrupted meeting of one to two hours in duration.
- The walkthrough team consists of three to five people.
- One of these people plays a role similar to that of the moderator in the inspection process, another person plays the role of a secretary (a person who records all errors found), and a third person plays the role of a tester.

Walkthroughs

- The initial procedure is identical to that of the inspection process:
- The participants are given the materials several days in advance to allow them to bone up on the program. However, the procedure in the meeting is different.
- Rather than simply reading the program or using error checklists, the participants “play computer.”
- The person designated as the tester comes to the meeting armed with a small set of paper test cases—representative sets of inputs (and expected outputs) for the program or module.
- During the meeting, each test case is mentally executed.
- That is, the test data are walked through the logic of the program.
- The state of the program (i.e., the values of the variables) is monitored on paper or whiteboard.

Testing strategies

- Black-Box Testing
- White-Box Testing

Black-Box Testing

- black-box, data-driven, or input/output- driven testing.
- view the program as a black box
- concentrate on finding circumstances in which the program does not behave according to its specifications.
- In this approach, test data are derived solely from the specifications(i.e., without taking advantage of knowledge of the internal structure of the program).
- to find all errors in the program, the criterion is exhaustive input testing, making use of every possible input condition as a test case.

White-Box Testing

- white-box or logic-driven testing, permits you to examine the internal structure of the program.
- This strategy derives test data from an examination of the program's logic.
- execute, via test cases, all possible paths of control flow through the program

Equivalence class testing

- Equivalence class testing is a technique used to reduce the number of test cases to a manageable level while still maintaining reasonable test coverage.
- First, identify the equivalence classes.
- Second, create a test case for each equivalence class. You could create additional test cases for each equivalence class if you have the time and money.
- Different types of input require different types of equivalence classes.
- An equivalence class consists of a set of data that is treated the same by the module or that should produce the same result. Any data value within a class is equivalent, in terms of testing, to any other value.

Equivalence class testing

A group of tests forms an equivalence class if you believe that:

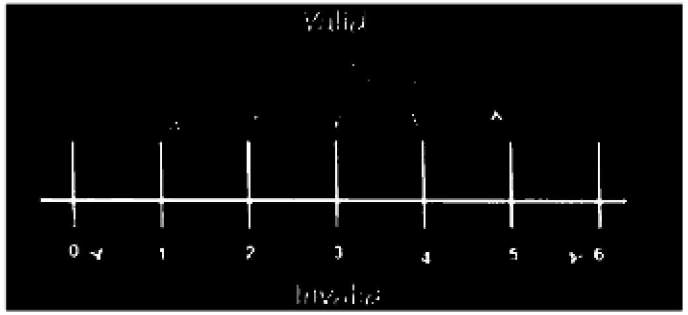
- They all test the same thing.
- If one test catches a bug, the others probably will too.
- If one test doesn't catch a bug, the others probably won't either.

- Equivalence class testing is a technique used to reduce the number of test cases to a manageable size while still maintaining reasonable coverage.
- An equivalence class consists of a set of data that is treated the same by the module or that should produce the same result. Any data value within a class is equivalent, in terms of testing, to any other value.

Equivalence class testing

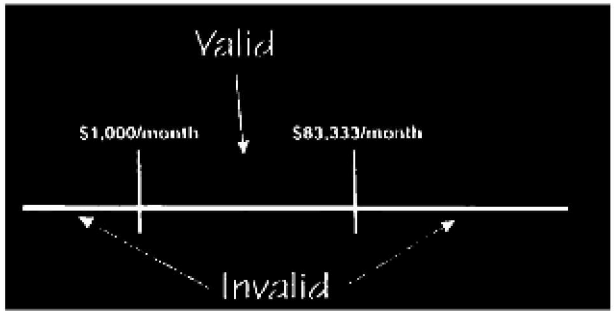
- It helps testers choose a small subset of possible test cases while maintaining reasonable coverage.

Discrete equivalence classes

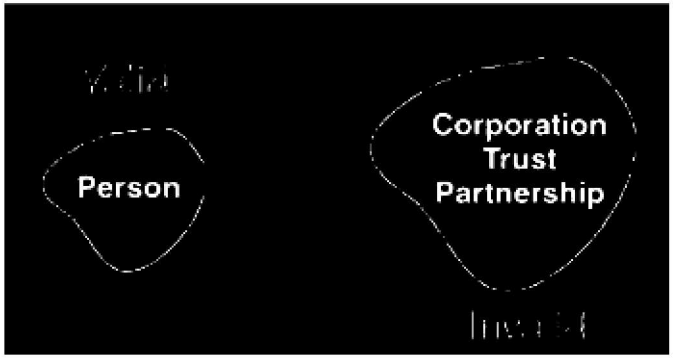


Continuous equivalence classes

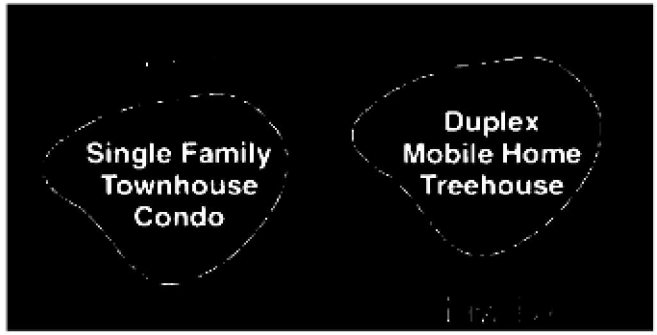
- If an input condition takes on discrete values within a range of permissible values, there are typically one valid and two invalid classes.



Single selection equivalence classes



Multiple selection equivalence class



Applicability and Limitations

- Equivalence class testing can significantly reduce the number of test cases that must be created and executed.
- It is most suited to systems in which much of the input data takes on values within ranges or within sets.
- Equivalence class testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs or outputs that can be partitioned based on the system's requirements.

Boundary Value Testing

- Boundary value testing focuses on the boundaries simply because that is where so many defects hide.
- identify the equivalence classes.
- Second, identify the boundaries of each equivalence class.
- Third, create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.
- "Below" and "above" are relative terms and depend on the data value's units.
- Create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.
- Boundary value testing is most appropriate where the input is a continuous range of values.

Applicability and Limitations

- Boundary value testing can significantly reduce the number of test cases that must be created and executed.
- It is most suited to systems in which much of the input data takes on values within ranges or within sets.
- Boundary value testing is equally applicable at the unit, integration, system, and acceptance test levels.
- All it requires are inputs that can be partitioned and boundaries that can be identified based on the system's requirements.

Decision Table Testing

- Decision tables are an excellent tool to capture certain kinds of system requirements and to document internal system design.
- They are used to record complex business rules that a system must implement. In addition, they can serve as a guide to creating test cases.

	Rule 1	Rule 2	...	Rule p
Conditions				
Condition 1				
Condition 2				
...				
Condition m				
Actions				
Action 1				
Action 2				
...				
Action n				

Decision Table Testing

- Decision tables are an excellent tool to capture certain kinds of system requirements and to document internal system design.
- They are used to record complex business rules that a system must implement.
- In addition, they can serve as a guide to creating test cases.
- Decision tables represent complex business rules based on a set of conditions.

- Conditions 1 through m represent various input conditions.
- Actions 1 through n are the actions that should be taken depending on the various combinations of input conditions.
- Each of the rules defines a unique combination of conditions that result in the execution ("firing") of the actions associated with that rule.
- the actions do not depend on the order in which the conditions are evaluated, but only on their values. (All values are assumed to be available simultaneously.)
- Also, actions depend only on the specified conditions, not on any previous input conditions or system state.

An auto insurance company gives discounts to drivers who are married and/or good students. Let's begin with the conditions. The following decision table has two conditions, each one of which takes on the values Yes or No.

	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
Married?	Yes	Yes	No	No
Good Student?	Yes	No	Yes	No

- the table contains all combinations of the conditions. Given two binary conditions (Yes or No), the possible combinations are {Yes, Yes}, {Yes, No}, {No, Yes}, and {No, No}.
- Each rule represents one of these combinations.
- As a tester we will verify that all combinations of the conditions are defined.
- Missing a combination may result in developing a system that may not process a particular set of inputs properly.
- Now for the actions. Each rule causes an action to "fire."
- Each rule may specify an action unique to that rule, or rules may share actions.

Adding a single action to a decision table.

	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
Married?	Yes	Yes	No	No
Good Student?	Yes	No	Yes	No
Actions				
Discount ($)	60	25	50	0

- Decision tables may specify more than one action for each rule. Again, these rules may be unique or may be shared.

A decision table with multiple actions.

n this situation, choosing test cases is simple—each rule (vertical column) becomes a test case. The Conditions specify the inputs and the Actions specify the expected results.

	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
Condition-1	Yes	Yes	No	No
Condition-2	Yes	No	Yes	No
Actions				
Action-1	Do X	Do Y	Do X	Do Z
Action 2	Do A	Do B	Do B	Do B

conditions can be more complex- A decision table with non-binary conditions.

	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
Condition-1	0–1	1–10	10–100	100–1000
Condition 2	<5	5	6 or 7	>7
Actions				
Action-1	Do X	Do Y	Do X	Do Z
Action-2	Do A	Do B	Do B	Do B

In this situation choosing test cases is slightly more complex— each rule (vertical column) becomes a test case but values satisfying the conditions must be chosen.

- Choosing appropriate values we create the following test cases:

Test Case ID	Condition-1	Condition-2	Expected Result
TC1	0	3	Do X / Do A
TC2	5	5	Do Y / Do B
TC3	50	7	Do X / Do B
TC4	500	10	Do Z / Do B

- If the system under test has complex business rules, and if your business analysts or designers have not documented these rules in this form, testers should gather this information and represent it in decision table form.
- The reason is simple. Given the system behavior represented in this complete and compact form, test cases can be created directly from the decision table.
- In testing, create at least one test case for each rule. If the rule's conditions are binary, a single test for each combination is probably sufficient.
- On the other hand, if a condition is a range of values, consider testing at both the low and high end of the range. In this way we merge the ideas of Boundary Value testing with Decision Table testing.

To create a test case table simply change the row and column headings:

A decision table converted to a test case table.

	Test Case 1	Test Case 2	Test Case 3	Test Case 4
Inputs				
Condition-1	Yes	Yes	No	No
Condition-2	Yes	No	Yes	No
Expected Results				
Action-1	Do X	Do Y	Do X	Do Z
Action 2	Do A	Do B	Do B	Do B

- Decision tables are used to document complex business rules that a system must implement. In addition, they serve as a guide to creating test cases.
- Conditions represent various input conditions. Actions are the processes that should be executed depending on the various combinations of input conditions. Each rule defines a unique combination of conditions that result in the execution ("firing") of the actions associated with that rule.
- Create at least one test case for each rule. If the rule's conditions are binary, a single test for each combination is probably sufficient. On the other hand, if a condition is a range of values, consider testing at both the low and high end of the range.

State-Transition Testing

- excellent tool to capture certain types of system requirements and to document internal system design.
- These diagrams document the events that come into and are processed by a system as well as the system's responses.
- they specify very little in terms of processing rules.
- When a system must remember something about what has happened before or when valid and invalid orders of operations exist, state-transition diagrams are excellent tools to record this information.
- state-transition diagram represents one specific entity (in this case a
- **Reservation**). It describes the states of a reservation, the events that affect the reservation, the transitions of the reservation from one state to another, and actions that are initiated by the reservation.

Defining the terms

- State (represented by a circle)—A state is a condition in which a system is waiting for one or more events. States "remember" inputs the system has received in the past and define how the system should respond to subsequent events when they occur. These events may cause state-transitions and/or initiate actions. The state is generally represented by the values of one or more variables within a system.
- Transition (represented by an arrow)—A transition represents a change from one state to another caused by an event.

Defining the terms contd…

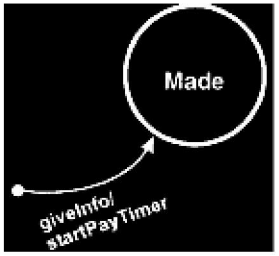
- Event (represented by a label on a transition)—An event is something that causes the system to change state. Generally, it is an event in the outside world that enters the system through its interface. Sometimes it is generated within the system such as **Timer expires** or **Quantity on Hand goes below Reorder Point.** Events are considered to be instantaneous. Events can be independent or causally related (event B cannot take place before event A). When an event occurs, the system can change state or remain in the same state and/or execute an action. Events may have parameters associated with them. For example, **Pay Money** may indicate **Cash, Check, Debit Card,** or **Credit Card.**

- Action (represented by a command following a "/")—An action is an operation initiated because of a state change. It could be **print a Ticket, display a Screen, turn on a Motor,** etc. Often these actions cause something to be created that are outputs of the system. Note that actions occur on transitions between states. The states themselves are passive.

- The entry point on the diagram is shown by a black dot while the exit point is shown by a bulls-eye symbol.

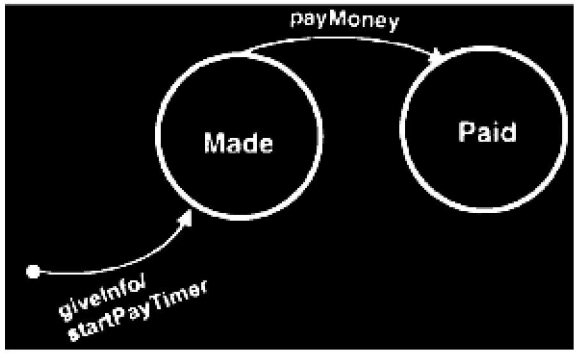
1. The Reservation is Made.

- The circle represents one state of the Reservation—in this case the Made state. The arrow
- shows the transition into the Made state. The description on the arrow, giveInfo, is an event
- that comes into the system from the outside world. The command after the "/" denotes an
- action of the system; in this case startPayTimer. The black dot indicates the starting point of the diagram.



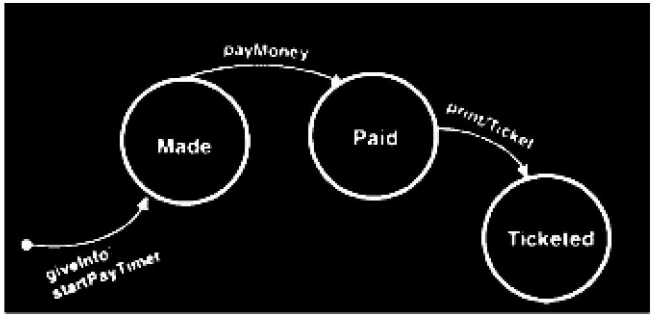
2. The Reservation transitions to the Paid state.

- Sometime after the Reservation is made, but (hopefully) before the PayTimer expires, the Reservation is paid for. This is represented by the arrow labeled PayMoney. When the Reservation is paid it transitions from the Made state to the Paid state.



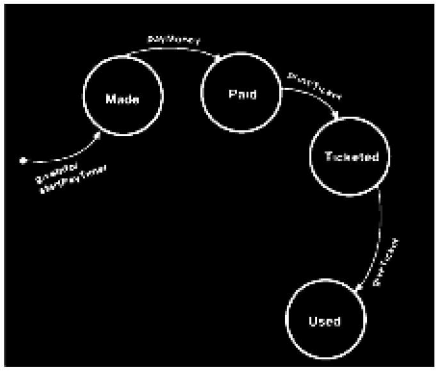
3. The Reservation transitions to the Ticketed state.

- From the Paid state the Reservation transitions to the Ticketed state when the print
- command (an event) is issued. Note that in addition to entering the Ticketed state, a Ticket is
- output by the system.



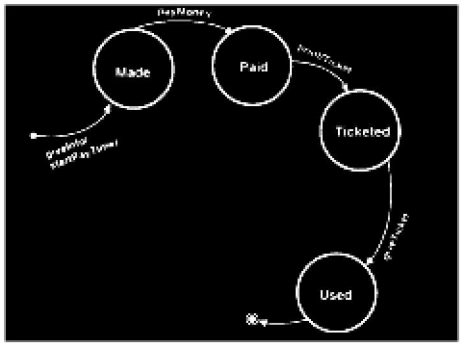
4. The Reservation transitions to the Used state.

- From the Ticketed state we giveTicket to the gate agent to board the plane.



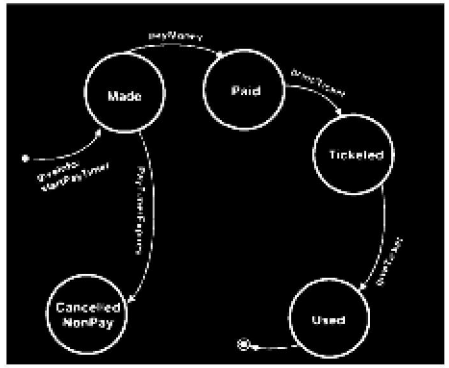
5. The path ends.

- After some other action or period of time, the state-transition
- path ends at the bulls-eye symbol.



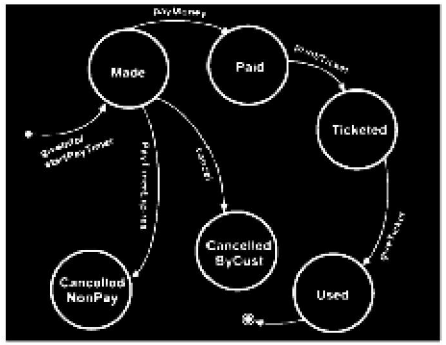
The PayTimer expires and the Reservation is cancelled for nonpayment.

- No. If the Reservation is not paid for in the time allotted (the PayTimer expires), it is cancelled for non-payment.



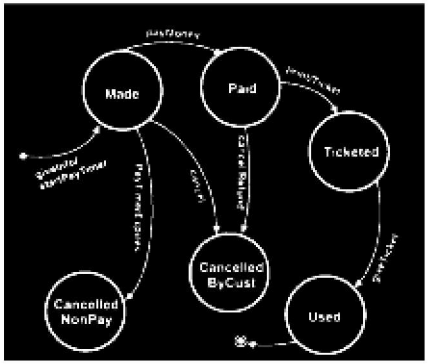
Cancel the Reservation from the Made state.

- Customers sometimes cancel their reservations. From the Made state the customer (through the reservation agent) asks to cancel the Reservation. A new state, Cancelled By Customer, is required.



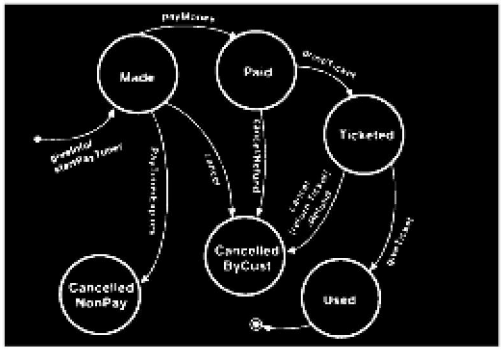
Cancellation from the Paid state.

- a Reservation can be cancelled from the Paid state. In this case a Refund should be generated and leave the system. The resulting state again is Cancelled By Customer.



Cancellation from the Ticketed state.

- From the Ticketed state the customer can cancel the Reservation. In that case a Refund should be generated and the next state should be Cancelled by Customer. But this is not sufficient. The airline will generate a refund but only when it receives the printed Ticket from the customer. This introduces one new notational element—square brackets [] that contain a conditional that can be evaluated either True or False. This conditional acts as a
- guard allowing the transition only if the condition is true.



state-transition diagrams

- state-transition diagrams express complex system rules and interactions in a very compact notation.

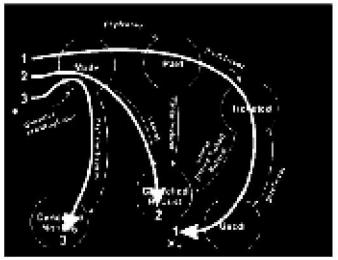
State-Transition Tables

- State-transition tables consist of four columns—Current State, Event, Action, and Next State.
- state-transition tables may be easier to use in a complete and systematic manner.
- a state-transition table is that it lists all possible state-transition combinations.
- creating a state-transition table often unearths combinations that were not identified, documented, or dealt with in the requirements. It is highly beneficial to discover these defects before coding begins.
- Using a state-transition table can help detect defects in implementation that enable invalid paths from one state to another.
- The disadvantage of such tables is that they become very large very quickly as the number of states and events increases. In addition, the tables are generally sparse; that is, most of the cells are empty.

State-Transition Table

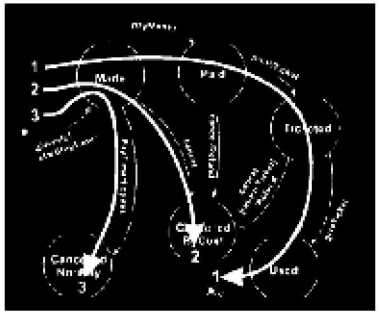
Current State	Event	Action	Next State
Null	guestIn	startPayTimer	Made
Null	payMoney		Null
Null	pin		Null
Null	guestBack		Null
Null	cancel		Null
Null	paymentExpires		Null
Made	guestIn		Made
Made	payMoney		Paid
Made	pin		Made
Made	guestBack		Made
Made	cancel		Can Cust
Made	paymentExpires		Can Cust
Can	guestIn		Null
Paid	payMoney		Paid
Paid	pin	Valid	Booked
Paid	guestBack		Paid
Paid	cancel	refund	Can Cust
Paid	paymentExpires		Paid
Booked	guestIn		Booked
Booked	payMoney		Booked
Booked	pin		Booked
Booked	guestBack		Used
Booked	cancel	refund	Can Cust
Booked	paymentExpires		Booked
Used	guestIn		Used
Used	payMoney		Used
Used	pin		Used
Used	guestBack		Used
Used	cancel		Used
Used	paymentExpires		Used

Can Null	guestIn		Can Null
Can Null	payMoney		Can Null
Can Null	pin		Can Null
Can Null	guestBack		Can Null
Can Null	cancel		Can Null
Can Null	paymentExpires		Can Null
Can Cust	guestIn		Can Cust
Can Cust	payMoney		Can Cust
Can Cust	pin		Can Cust
Can Cust	guestBack		Can Cust
Can Cust	cancel		Can Cust
Can Cust	paymentExpires		Can Cust



Creating Test Cases

- Information in the state-transition diagrams can easily be used to create test cases. Four different levels of coverage can be defined:
- 1. Create a set of test cases such that all states are "visited" at least once under test. The set of three test cases shown below meets this requirement. Generally this is a weak level of test coverage.



Create a set of test cases such that all paths are executed at least once under test. While this level is the most preferred because of its level of coverage, it may not be feasible. If the state-transition diagram has loops, then the number of possible paths may be infinite. For example, given a system with two states, A and B, where A transitions to B and B transitions to A. A few of the possible paths are: A→B

A→B→A

A→B→A→B→A→B

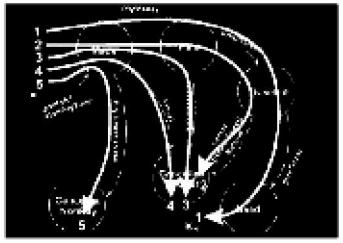
A→B→A→B→A→B→AA→B→A→B→A→B→A→B→A→B

...

and so on forever. Testing of loops such as this can be important if they may result in accumulating computational errors or resource loss (locks without corresponding releases, memory leaks, etc.).

A set of test cases that trigger all transitions at least once.

Create a set of test cases such that all transitions are exercised at least once under test. This level of testing provides a good level of coverage without generating large numbers of tests. This level is generally the one recommended.



Pairwise Testing

Consider these situations:

A Web site must operate correctly with different browsers—Internet Explorer 5.0, 5.5, and 6.0, Netscape 6.0, 6.1, and 7.0, Mozilla 1.1, and Opera 7;

using different plug-ins —RealPlayer, MediaPlayer, or none;

running on different client operating systems— Windows 95, 98, ME, NT, 2000, and XP;

receiving pages from different servers—IIS, Apache, and WebLogic;

running on different server operating systems—Windows NT, 2000, and Linux.

Web Combinations

8 browsers
3 plug-ins
6 client operating systems
3 servers
3 server OS
1,296 combinations.

Applicability and Limitations

- State-Transition diagrams are excellent tools to capture certain system requirements, namely those that describe states and their associated transitions. These diagrams then can be used to direct our testing efforts by identifying the states, events, and transitions that should be tested.

- State-Transition diagrams are not applicable when the system has no state or does not need to respond to real-time events from outside of the system. An example is a payroll program that reads an employee's time record, computes pay, subtracts deductions, saves the record, prints a paycheck, and repeats the process.

A bank has created a new data processing system that is ready for testing. This bank has different kinds of customers—consumers, very important consumers, businesses, and non-profits; different kinds of accounts—checking, savings, mortgages, consumer loans, and commercial loans; they operate in different states, each with different regulations—California, Nevada, Utah, Idaho, Arizona, and New Mexico.

Bank Combinations

4 customer types
5 account types
6 states
120 combinations.

- In an object-oriented system, an object of class A can pass a message containing a parameter P to an object of class X. Classes B, C, and D inherit from A so they too can send the message. Classes Q, R, S, and T inherit from P so they too can be passed as the parameter. Classes Y and Z inherit from X so they too can receive the message.

OO Combinations

4 senders
5 parameters
3 receivers
60 combinations.

significant reductions

- If a system had four different input parameters and each one could take on one of three different values, the number of combinations is 34 which is 81. It is possible to cover all the pairwise input combinations in only nine tests.
- If a system had thirteen different input parameters and each one could take on one of three different values, the number of combinations is 313 which is 1,594,323. It is possible to cover all the pairwise input combinations in only fifteen tests.
- If a system had twenty different input parameters and each one could take on one of ten different values, the number of combinations is 1020. It is possible to cover all the pairwise input combinations in only 180 tests.

- Each has a large number of combinations that should be tested. Each has a large number of combinations that may be risky if we do not test. Each has such a large number of combinations that we may not have the resources to construct and run all the tests, there are just too many. We must, somehow, select a reasonably sized subset that we could test given our resource constraints.
- **Random selection can be a very good approach to choosing a subset but most people have a difficult time choosing truly randomly.**

- Pairwise testing may not choose combinations which the developers and testers know are either frequently used or highly risky. If these combinations exist, use the pairwise tests, then add additional test cases to minimize the risk of missing an important combination.

techniques are used to identify all the pairs for creating test cases

- Orthogonal arrays
- Allpairs algorithm.

Orthogonal Arrays

- An orthogonal array is a two-dimensional array of numbers that has the property—choose any two columns in the array. All the pairwise combinations of its values will occur in every pair of columns.

	1	2	3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

The L4(23) array is orthogonal; that is, choose any two columns, all the pairwise combinations will occur in all the column pairs. L4 means an orthogonal array with four rows, (23) is not an exponent. It means that the array has three columns, each with either a 1 or a 2.

Orthogonal array notation

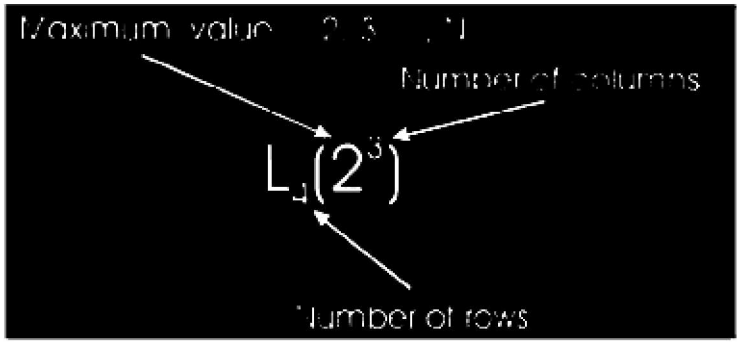


Table 6-2: $L_9(3^4)$ Orthogonal Array

	1	2	3	4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Examine columns 1 and 2—do the nine combinations of 1, 2, and 3 all appear in that column pair? Yes.
Examine columns 1 and 3—do the nine combinations of 1, 2, and 3 appear in that column pair? Yes, although in a different order.
Examine columns 1 and 4—do the nine combinations appear in that column pair also? Yes they do.
Continue on by examining other pairs of columns—2 and 3, 2 and 4, and finally 3 and 4.
The L9(34) array is orthogonal; that is, choose any two columns, all the combinations will occur in all of the column pairs.

Using Orthogonal Arrays

The process of using orthogonal arrays to select pairwise subsets for testing is:

1. Identify the variables.
2. Determine the number of choices for each variable.
3. Locate an orthogonal array which has a column for each variable and values within the columns that correspond to the choices for each variable.
4. Map the test problem onto the orthogonal array.
5. Construct the test cases.

Allpairs Algorithm

- an algorithm that generates the pairs directly without resorting to an "external" device

Use Case Testing

- test cases that exercise a system's functionalities from start to finish by testing each of its individual transactions.
- "use case" is a scenario that describes the use of a system by an actor to accomplish a specific goal.
- By "actor" we mean a user, playing a role with respect to the system, seeking to use the system to accomplish something worthwhile within a particular context.
- Actors are generally people although other systems may also be actors.
- A "scenario" is a sequence of steps that describe the interactions between the actor and the system.
- The set of use cases makes up the functional requirements of a system.

The Unified Modeling Language notion for use cases

- The stick figures represent the actors, the ellipses represent the use cases, and the arrows show which actors initiate which use cases.

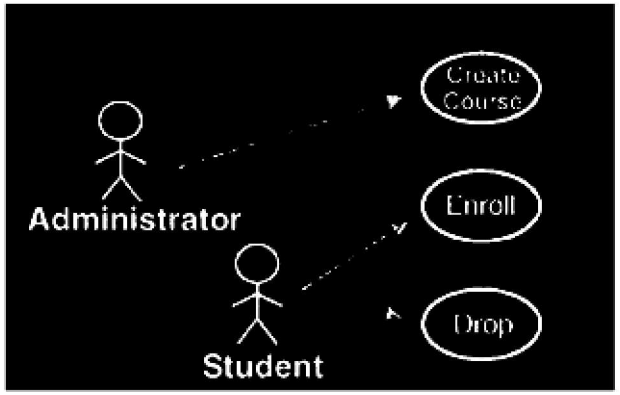


Table 9.1: Use case template.	
Use Case Component	Description
Use Case Number to identify	A unique number for the use case
Use Case Name	The name should be the goal stated as a short active verb phrase
Goal in Context	A more detailed statement of the goal if necessary
Scope	Corporate | System | Subsystem
Level	Summary | Primary task | Subfunction
Primary actor	Role name or description of the primary actor
Preconditions	The required state of the system before the use case is initiated
Success End Conditions	The state of the system upon successful completion of the use case
Failed End Conditions	The state of the system if the use case cannot execute or completes
Trigger	The action that initiates the execution of the use case
Main Success Scenario	Step | Action 1 2 ...
Extensions	Conditions under which the main success scenario will fail and a description of those variations
Sub-variations	Variations that do not affect the main flow but that must be considered
Priority	Criticality
Response Time	Time available to execute this use case
Frequency	How often the use case is executed
Channels to	Interactive | Batch | Database
Primary Actor	
Secondary Actors	Other actors needed to accomplish the use case
Channels to Secondary Actors	Interactive | File | Database | ...
Open Bugs	Sensitive information
Completeness Level	Use Case Identified (1.1) | Main scenario defined (1.5) | All scenarios defined (1.9) | A fully complete (2.0)
Open Issues	Unresolved issues awaiting decisions

use cases - uses

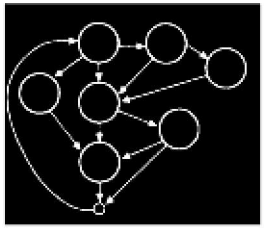
- Capture the system's functional requirements from the user's perspective; not from a technical perspective, and irrespective of the development paradigm to be used.
- Can be used to actively involve users in the requirements gathering and definition process.
- Provide the basis for identifying a system's key internal components, structures, databases, and relationships.
- Serve as the foundation for developing test cases at the system and acceptance level.

- create at least one test case for the main success scenario and at least one test case for each extension.
- Because use cases do not specify input data, the tester must select it.
- equivalence class and boundary value techniques can be used.
- Always remember to evaluate the risk of each use case and extension and create test cases accordingly.

- To create test cases, start with normal data for the most often used transactions. Then move to boundary values and invalid data. Next, choose transactions that, while not used often, are vital to the success of the system (i.e., **Shut Down The Nuclear Reactor).**
- Make sure you have at least one test case for every Extension in the use case.
- Try transactions in strange orders.
- Violate the preconditions (if that can happen in actual use).
- If a transaction has loops, don't just loop through once or twice—be diabolical.
- Look for the longest, most convoluted path through the transaction and try it.
- If transactions should be executed in some logical order, try a different order.
- Instead of entering data top-down, try bottom-up. Create "goofy" test cases.
- If you don't try strange things, you know the users will.

White box testing

- White box testing is a strategy in which testing is based on the internal paths, structure, and implementation of the software under test (SUT).
- white box testing generally requires detailed programming skills.
- White box testing can be applied at all levels of system development—unit, integration, and system.
- White box testing is more than code testing—it is **path** testing.
- Generally, the paths that are tested are within a module (unit testing). But we can apply the same techniques to test paths
- between modules within subsystems, between subsystems within systems, and even between entire systems.



general white box testing process

- The SUT's implementation is analyzed.
- Paths through the SUT are identified.
- Inputs are chosen to cause the SUT to execute selected paths. This is called path sensitization. Expected results for those inputs are determined.
- The tests are run.
- Actual outputs are compared with the expected outputs.
- A determination is made as to the proper functioning of the SUT.

Disadvantages

1. the number of execution paths may be so large than they cannot all be tested. Attempting to test all execution paths through white box testing is generally as infeasible as testing all input data combinations through black box testing.
2. the test cases chosen may not detect data sensitivity errors. For example:

   ```
   p=q/r;
   may execute correctly except when r=0.
   y=2*x // should read y=x2
   will pass for test cases x=0, y=0 and x=2, y=4
   ```

3. white box testing assumes the control flow is correct (or very close to correct). Since the tests are based on the existing paths, nonexistent paths cannot be discovered through white box testing.
4. the tester must have the programming skills to understand and evaluate the software under test. Unfortunately, many testers today do not have this background

Advantages

- When using white box testing, the tester can be sure that every path through the software under test has been identified and tested.

Control Flow Graphs

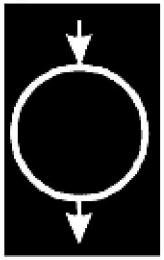
- Control flow graphs are the foundation of control flow testing.
- These graphs document the module's control structure.
- Modules of code are converted to graphs, the paths through the graphs are analyzed, and test cases are created from that analysis.
- Control flow graphs consist of a number of elements:

Control Flow Testing

- This testing approach identifies the execution paths through a module of program code and then creates and executes test cases to cover those paths.
- Path: A sequence of statement execution that begins at an entry and ends at an exit.

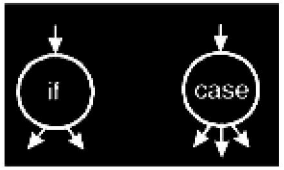
Process Blocks

- A process block is a sequence of program statements that execute sequentially from beginning to end.
- No entry into the block is permitted except at the beginning.
- No exit from the block is permitted except at the end.
- Once the block is initiated, every statement within it will be executed sequentially.
- Process blocks are represented in control flow graphs by a bubble with one entry and one exit.



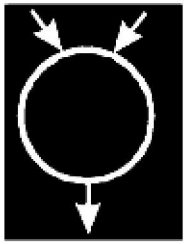
Decision Point

- A decision point is a point in the module at which the control flow can change.
- Most decision points are binary and are implemented by if-then-else statements.
- Multi-way decision points are implemented by case statements.
- They are represented by a bubble with one entry and multiple exits.



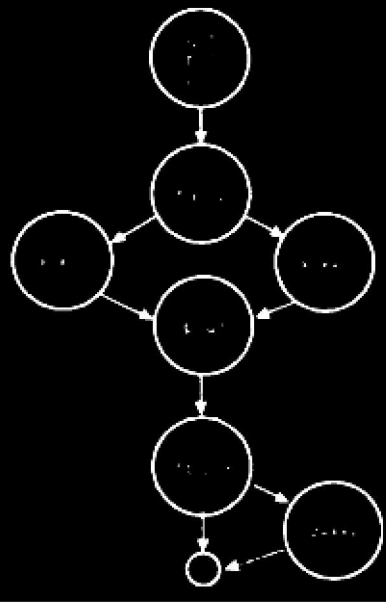
Junction Point

- A junction point is a point at which control flows join together.



Flow graph equivalent of program code.

```
q=1;
b=2;
c=3;
if (a==2) {x=x+2;}
else {x=x/2;}
p=q/r;
if (b/c>3) {z=x+y;}
```



Levels of Coverage

- "coverage" means the percentage of the code that has been tested vs. that which is there to test. In control flow testing we define coverage at a number of different levels.
- Level 1
- Level 0
- Level 2
- Level 3
- Level 4
- Level 5
- Level 7
- Level 6

Levels of Coverage

- **Level 1 -** The lowest coverage level is "100% statement coverage" (also is referred to as "statement coverage"). This means that every statement within the module is executed, under test, at least once.
- **Level 0 -** there is a level of coverage below "100% statement coverage." That level is defined as "test whatever you test; let the users test the rest."
- **Level 2 -** 100% decision coverage - also called "branch coverage." At this level enough test cases are written so that each decision that has a TRUE and FALSE outcome is evaluated at least once.
- **Level 3 -** 100% condition coverage." At this level enough test cases are written so that each condition that has a TRUE and FALSE outcome that makes up a decision is evaluated at least once. Condition coverage is usually better than decision coverage because every individual condition is tested at least once while decision coverage can be achieved without testing every condition.

Levels of Coverage

- Level 4 –

  ```
  if(x&&y) {conditionedStatement;}
  // note: && indicates logical AND
  ```

- We can achieve condition coverage with two test cases (x=TRUE, y=FALSE and x=FALSE, y=TRUE) but note that with these choices of data values the conditionedStatement will never be executed. Given the possible combination of conditions such as these, to be more complete "100% decision/condition" coverage can be selected. At this level test cases are created for every condition and every decision.
- Level 5 - 100% multiple condition coverage - Achieving 100% multiple condition coverage also achieves decision coverage, condition coverage, and decision/condition coverage. Note that multiple condition coverage does not guarantee path coverage.

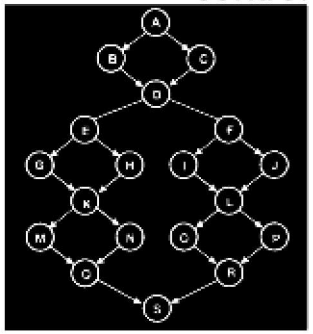
Levels of Coverage

- Level 7 - 100% path coverage - For modules with loops, the number of paths can be enormous and thus pose an intractable testing problem.
- Level 6 - When a module has loops in the code paths such that the number of paths is infinite, a significant but meaningful reduction can be made by limiting loop execution to a small number of cases. The first case is to execute the loop zero times; the second is to execute the loop one time, the third is to execute the loop *n* times where *n* is a small number representing a typical loop value; the fourth is to execute the loop its maximum number of times *m*. In addition you might try *m-1* and *m+1*.

Structured Testing / Basis Path Testing

- uses an analysis of the topology of the control flow graph to identify test cases.
- The structured testing process consists of the following steps:
  - Derive the control flow graph from the software module.
  - Compute the graph's Cyclomatic Complexity (C).
  - Select a set of C basis paths.
  - Create a test case for each basis path.
  - Execute these tests.

control flow graph - example



- Cyclomatic Complexity (C) of a graph as C = edges - nodes + 2
- Edges are the arrows, and nodes are the bubbles on the graph. The preceding graph has 24 edges and 19 nodes for a Cyclomatic Complexity of 24-19+2 = 7.
- In some cases this computation can be simplified. If all decisions in the graph are binary (they have exactly two edges flowing out), and there are p binary decisions, then C = p+1

- Cyclomatic Complexity is exactly the minimum number of independent, nonlooping paths (called basis paths) that can, in linear combination, generate all possible paths through the module.
- In terms of a flow graph, each basis path traverses at least one edge that no other path does.

- Creating and executing C test cases, based on the basis paths, guarantees both branch and statement coverage.

Applicability and Limitations

- Control flow testing is the cornerstone of unit testing. It should be used for all modules of code that cannot be tested sufficiently through reviews and inspections. Its limitations are that the tester must have sufficient programming skill to understand the code and its control flow. In addition, control flow testing can be very time consuming because of all the modules and basis paths that comprise a system.

Data Flow Testing

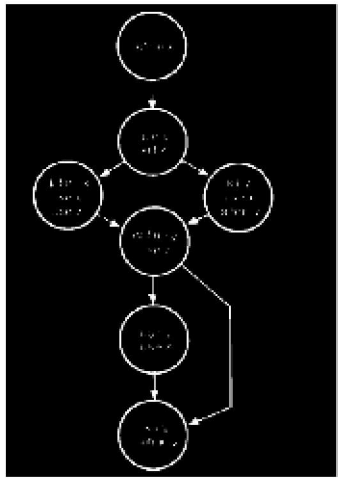
- Data flow testing is a powerful tool to detect improper use of data values due to coding errors.
- it shows the processing flow through a module.
- In addition, it details the definition, use, and destruction of each of the module's variables.
- construct these diagrams and verify that the define-use-kill patterns are appropriate.

- First, we will perform a static test of the diagram. By "static" we mean we examine the diagram (formally through inspections or informally through look-sees).
- Second, we perform dynamic tests on the module. By "dynamic" we mean we construct and execute test cases. Let's begin with the static testing.

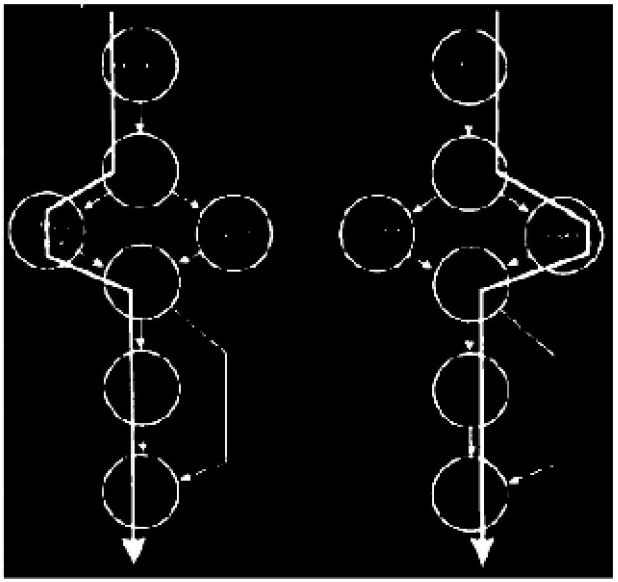
- Variables defined within a block are created when their definitions are executed and are automatically destroyed at the end of a block. This is called the "scope" of the variable.
- Variables can be used in computation (a=b+1). They can also be used in conditionals (if (a>42)). In both uses it is equally important that the variable has been assigned a value before it is used.
- Examine time-sequenced pairs of defined, used, and killed variable references.

Static Data Flow Testing

- The control flow diagram annotated with define-use-kill information for each of the module's variables.



The control flow diagram annotated with define-use-kill information for the x variable.



Dynamic Data Flow Testing

- The data flow testing process is to choose enough test cases so that:
  - Every "define" is traced to each of its "uses"
  - Every "use" is traced from its corresponding "define"

- enumerate the paths through the module.
- This is done using the same approach as in control flow testing: Begin at the module's entry point, take the leftmost path through the module to its exit.
- Return to the beginning and vary the first branching condition. Follow that path to the exit.
- Return to the beginning and vary the second branching condition, then the third, and so on until all the paths are listed.
- Then, for every variable, create at least one test case to cover every define-use pair.

Applicability and Limitations

- Data flow testing builds on and expands control flow testing techniques.
- As with control flow testing, it should be used for all modules of code that cannot be tested sufficiently through reviews and inspections.

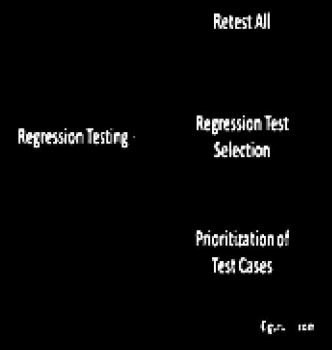
- Its limitations are that the tester must have sufficient programming skill to understand the code, its control flow, and its variables.
- Like control flow testing, data flow testing can be very time consuming because of all the modules, paths, and variables that comprise a system.

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, is done by code **testing** specialists.

What's the strategy?

Software maintenance is an activity which includes enhancements, error corrections, optimization and deletion of existing features. These modifications may cause the system to work incorrectly. Therefore , Regression Testing becomes necessary. Regression Testing can be carried out using following techniques:



When to do it

Regression Testing is required when there is a -

- Change in requirements and code is modified according to the requirement
- New feature is added to the software
- Defect fixing
- Performance issue fix

Retest All

- All the tests in the existing test bucket or suite should be re-executed.
- This is very expensive as it requires huge time and resources.

What's the strategy?

Continues..

Regression Test Selection

Instead of re-executing the entire test suite, it is better to select part of test suite to be run.

Test cases selected can be categorized as 1) Reusable Test Cases 2) Obsolete Test Cases.

Re-usable Test cases can be used in succeeding regression cycles. Obsolete Test Cases can't be used in succeeding cycles.

What's the strategy?

Continues..

Prioritization of Test Cases

Prioritize the test cases depending on business impact, critical & frequently used functionalities . Selection of test cases based on priority will greatly reduce the regression test suite.

What's the strategy?

Continues..

Area to focus during testing

- Test cases which have frequent defects.
- Functionalities which are more visible to the users.
- Test cases which verify core features of the product.
- Test cases of Functionalities which has undergone more and recent changes.
- All Integration Test Cases.
- All Complex Test Cases.
- Boundary value test cases.
- Sample of Successful test cases.
- Sample of Failure test cases

Pros and Cons

Challenges for regression testing

- With successive regression runs, test suites become fairly large. Due to time and budget constraints, the entire regression test suite cannot be executed.
- Minimizing test suite while achieving maximum test coverage remains a challenge.
- Determination of frequency of Regression Tests , i.e., after every modification or every build update or after a bunch of bug fixes, is a challenge.

Non Functional requirements **(NFRs)**

1. Capacity
2. Throughput and
3. Performance

- Performance is a measure of the time taken to process a single transaction, and can be measured either in isolation or under load.
- Throughput is the number of transactions a system can process in a given timespan. It is always limited by some bottleneck in the system.
- The maximum throughput a system can sustain, for a given workload, while maintaining an acceptable response time for each individual request, is its capacity.
- Nonfunctional requirements (NFRs) are important because they present a significant delivery risk to software projects.
- NFRs are referred to as “quality attributes”

Managing Nonfunctional Requirements

- At the beginning of the project, everybody involved in delivery—developers, operations personnel, testers, and the customer—need to think through the application’s NFRs and the impact they may have on the system architecture, project schedule, test strategy, and overall cost.

Analyzing Nonfunctional Requirements

- Create specific sets of stories or tasks for nonfunctional requirements as well, especially at the beginning of a project.
- Adding nonfunctional acceptance criteria to other requirements.
- Supply a reasonable level of detail when analyzing NFRs .
- Recognize the most common causes of capacity problems and work to avoid running into them.

strategy to address capacity problems

1. Decide upon an architecture for your application. Pay particular attention to process and network boundaries and I/O in general.
2. Understand and use patterns and avoid antipatterns that affect the stability and capacity of your system.
3. Keep the team working within the boundaries of the chosen architecture but, other than applying patterns where appropriate, ignore the lure to optimize for capacity. Encourage clarity and simplicity. Never compromise readability for capacity without an explicit test that demonstrates the value.
4. Pay attention to the data structures and algorithms chosen, making sure that their properties are suitable for your application
5. Be extremely careful about threading.
6. Establish automated tests that assert the desired level of capacity. When these tests fail, use them as a guide to fixing the problems.
7. Use tools as a focused attempt to fix problems identified by tests, not as a general “make it as fast as possible” strategy.
8. Wherever you can, use real-world capacity measures. Your production system is your only real source of measurement. Use it and understand what it is telling you. Pay particular attention to the number of users of the system, their patterns of behavior, and the size of the production data set.

Measuring capacity

- Measuring capacity involves investigating a broad spectrum of characteristics of an application. Here are some types of measurements that can be performed:

1. Scalability testing. How do the response time of an individual request and the number of possible simultaneous users change as we add more servers, services, or threads?
2. Longevity testing. This involves running the system for a long time to see if the performance changes over a protracted period of operation. This type of testing can catch memory leaks or stability problems.
3. Throughput testing. How many transactions, or messages, or page hits per second can the system handle?
4. Load testing. What happens to capacity when the load on the application increases to production-like proportions and beyond? This is perhaps the most common class of capacity testing.

Benchmark-style capacity tests

- An important aspect of capacity testing is the ability to simulate realistic use scenarios for a given application.
- The alternative to this approach is to benchmark specific technical interactions in the system: "how many transactions per second can the database store?", "How many messages per second can the message queue convey?"
- Benchmark-style capacity tests are extremely useful for guarding against specific problems in the code and optimizing code in a specific area. They can be useful by providing information to help with technology selection processes.

Automating Capacity Testing

- an automated capacity test suite should be created and run against every change to the system that passes the commit stage and (optionally) the acceptance test stage.

Capacity tests should,

- Test specific real-world scenarios, so we don't miss important bugs in real-world use through overly abstract testing
- Have a predefined threshold for success, so we can tell that they have passed
- Be of short duration, so that capacity testing can take place in a reasonable length of time
- Be robust in the face of change, to avoid constant rework to keep up with changes to the application
- Be composable into larger-scale complex scenarios, so that we can simulate real-world patterns of use
- Be repeatable, capable of running sequentially and in parallel, so that we can both build suites of tests to apply load and run longevity tests

Scenario-based testing

- Include scenario-based testing into our capacity testing strategy.
- We represent a specific scenario of use of the system as a test, and evaluate that against our business predictions of what it must achieve in the real world.
- Each of our scenario-based tests should be capable of running alongside other capacity tests involving other interactions.

Defining Success and Failure for Capacity Tests

- Success or failure is often determined by a human analysis of the collected measurements.
- An extremely useful property of any capacity test system if it is also able to generate measurements, providing insight into what happened, not just a binary report of failure or success.
- There are two strategies to adopt here.
- First, aim for stable, reproducible results. As far as practically possible, isolate capacity test environments from other influences and dedicate them to the task of measuring capacity. This minimizes the impact of other, non-test-related, tasks and so makes the results more consistent.
- Next, tune the pass threshold for each test by ratcheting it up once the test is passing at a minimum acceptable level.

The Capacity-Testing Environment

- Absolute measurements of the capacity of a system should ideally be carried out in an environment that replicates the production environment in which the system will ultimately run.
- If capacity or performance is a serious issue for your application, make the investment and create a clone of your production environment for the core parts of your system.
- In the real world, the ideal of capacity testing in an exact replica of the production environment isn't always possible.
- While capacity testing on lower-specification hardware will highlight any serious capacity problems, it won't be able to demonstrate that the application can fully meet its goals.
- Limit the test environment costs and to provide some sensibly accurate performance measures is available where the application is to be deployed into production on a farm of servers

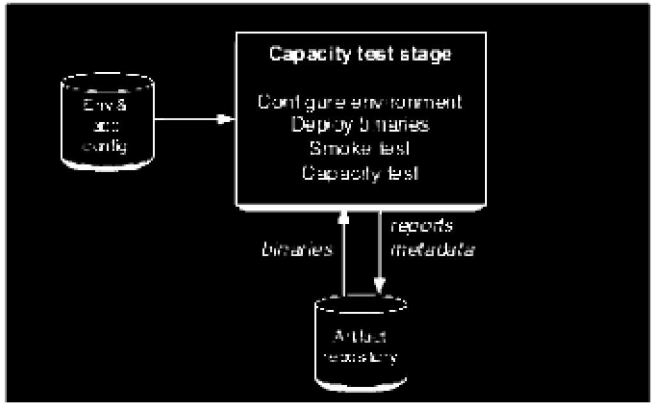
Automating Capacity Testing

- Add capacity testing as a stage to the deployment pipeline. an automated capacity test suite should be created and run against every change to the system that passes the commit stage and (optionally) the acceptance test stage.
- add automated capacity testing as a wholly separate stage in our deployment pipeline.

Models for capacity testing

- A fully automated deployment gate – unless the tests in the capacity test stage all pass, you can't deploy the application without a manual override.
- If there are real issues of throughput or latency, or information that is only relevant or accurate for specific windows of time, automated tests can act very effectively as executable specifications that can assert that the requirement is met.
- The acceptance test stage in the deployment pipeline is a template for all subsequent testing stages, including capacity testing. For capacity tests, as for others, the stage begins by preparing for deployment, deploying, then verifying that the environment and application are correctly configured and deployed. Only then are the capacity tests run.

The capacity test stage of the deployment pipeline



- Nonfunctional requirements are the software equivalent of a bridge builder making sure that the chosen beams are strong enough to cope with the expected traffic and weather. These requirements are real, they have to be considered, but they aren't what is in the mind of the business people paying for the bridge.
- technical people must work closely with customers and users to determine the sensitivity points of our application and define detailed nonfunctional requirements based upon real business value.
- Once this work has been done, the delivery team can decide upon the correct architecture to use for the application and create requirements and acceptance criteria capturing the nonfunctional requirements in the same way that functional requirements are captured. It thus becomes possible to estimate the effort involved in meeting nonfunctional requirements and prioritize them in the same way as functional requirements.

- Once this work is done, the delivery team needs to create and maintain automated tests to ensure that these requirements are met. These tests should be run as part of your deployment pipeline every time a change to your application, infrastructure, or configuration passes the commit test and acceptance test stages.
- Use your acceptance tests as a starting point for broader scenario-based testing of NFRs—this is a great strategy to get comprehensive, maintainable coverage of the characteristics of the system.