软件质量保证与测试
PPT打印版

Software Testing and Quality Assurance

Joe Timoney

1

The nature of Software

- It is abstract and intangible
- There is a lack of physical constraints
- Software is not thought to have natural limits unlike real-world materials
- It easily becomes extremely complex
- It is effort intensive (need to organize carefully)

2

Many ordinary software failures

- Does not solve user's problem
- There is some schedule slippage and it is not ready in time
- There can be cost over-runs and in extreme cases it becomes too costly to complete
- It has poor quality and poor maintainability

3

总结一下各种软件模型

waterfall

incremental

scrum

devops

4

Software Testing and Debugging

- Software testing is concerned with confirming the presence of errors
- Debugging is concerned with locating and repairing these errors

5

Static Testing

- Static Verification (or Static Analysis) can be as straightforward as having someone of training and experience reading through the code to search for faults.
- It could also take a mathematical approach consisting of symbolic execution of the program

6

Static Testing

- Can use modelling such as UML
- Can apply formal methods
- Tools such as Spec# exist

7

Spec#

- Spec# is a formal language for API contracts (influenced by JML, AsmL, and Eiffel), which extends C# with constructs for non-null types, preconditions, postconditions, and object invariants.
- Spec# comes with a sound programming methodology that permits specification and reasoning about object invariants even in the presence of callbacks and multi-threading.

8

Dynamic Testing

- Dynamic Verification (or Software Testing) confirms the operation of a program by executing it.
- Test Cases are created that guide the selection of suitable Test Data (consisting of Input values and Expected Output values).
- The Input values are provided as inputs to the program during execution
- The Actual Outputs are collected from the program, and then they are compared with the Expected Outputs.

9

Black and White box Testing

- Black Box testing is based entirely on the program specification and aims to verify that the program meets the specified requirements
- White box testing uses the implementation of the software to derive the tests. The tests are designed to exercise some aspect of the program code

10

Software Testing

Introduction

11

What is Software?

- It is not just about computer programs
- It is also associated with documentation and configuration data that is needed to make these programs operate correctly

12

Define a Software System

- A software system usually consists of a number of:
  - instructions within separate programs that when executed give some desired function
  - data structures that enable the programs to adequately manipulate information
  - configuration files which are used to set up these programs
  - system documentation which describes the structure of the system
  - user documentation which explains how to use the system and web sites for users to download recent product information

13

Challenges of Software Systems

- Major challenges in building a software system:
  - Effort intensive (organize carefully)
  - High cost (if there is failure then the investment is lost)
  - Long development time
  - Changing needs/requirements for users
  - High risk of failure
    - user acceptance – may reject it
    - Performance
    - maintainability…

14

Quality and Software

- There are risks associated with Software Development
- Modern programs are complex and have thousands of lines of code
- The customer's requirements can be vague, lacking in exactness
- Deadlines and budgets put pressure on the development team

15

Quality and Software

- The combination of these factors can lead to a lack of emphasis being placed on the final quality of the software product
- Poor quality can result in software failure resulting in high maintenance costs and long delays before the final deployment
- The impact on the business can be loss of reputation, legal claims, decrease in market share

16

Quality and Software

- The International Standard ISO 91261 Software Engineering – Product Quality is structured around six main attributes
- These can be measured using a mix of objective and subjective metrics

17

ISO 91261 Attributes

Functionality	Suitability, accurateness, interoperability, compliance, security
Reliability	Maturity, fault tolerance, recoverability
Usability	Understandability, learnability, operability
Efficiency	Time behavior, resource behavior
Maintainability	Analysability, changeability, stability, testability
Portability	Adaptability, installability, conformance, replaceability

18

Software Engineering

- The actual term ‘software engineering’ was first proposed as far back as 1968 at a conference held to discuss what was then called the ‘software crisis’.
- It was becoming clear by then that individual approaches to program development did not scale up to large and complex software systems. These were unreliable, cost more than expected, and were delivered late.

19

Software Engineering

- To overcome these problems throughout 1970s and 1980s, a variety of new software engineering techniques and methods were developed
- These include structured programming, information hiding and object-oriented development.
- Tools and standard notations that were developed at that time are now extensively used

20

Errors, Faults and Failures

1. Errors: these are mistakes made by software developers. They exist in the mind, and can result in one or more faults in the software.
2. Faults: these consist of incorrect material in the source code, and can be the product of one or more errors. Faults can lead to failures during program execution.
3. Failures: these are symptoms of a fault, and consist of incorrect, or out-of specification behaviour by the software. Faults may remain hidden until a certain set of conditions are met which reveal them as a failure in the software execution.

21

Software Faults - Categories

- Algorithmic faults
  - Algorithmic faults are the ones that occurs when a unit of the software does not produce an output corresponding to the given input under the designated algorithm
- Syntax Faults
  - These occur when code is not in conformance to the programming language specification, (i.e. source code compiled a few years back with older versions of compilers may have syntax that does not conform to present syntax checking by compilers (because of standards conformity).

22

Software Faults - Categories

- Documentation faults
  - Incomplete or incorrect documentation will lead to Documentation faults
- Stress or overload faults
  - Stress or Overload faults happens when data structures are filled past their specific capacity where as the system characteristics are designed to handle no more than a maximum load planned under the requirements
- Capacity and boundary faults
  - Capacity or Boundary faults occur when the system produces an unacceptable performance because the system activity may reach its specified limit due to overload
- Computation and precision faults
  - Computation and Precision faults occur when the calculated result using the chosen formula does not confirm to the expected accuracy or precision

23

Software Faults - Categories

- Throughput or performance faults
  - This is when the developed system does not perform at the speed specified under the stipulated requirements
- Recovery faults
  - This happens when the system does not recover to the expected performance even after a fault is detected and corrected
- Timing or coordination faults
  - These are typical of real time systems when the programming and coding are not commensurate to meet the co-ordination of several concurrent processes or when the processes have to be executed in a carefully defined sequence

24

Software Faults - Categories

- Standards and Procedure Faults
  - Standards and Procedure faults occur when a team member does not follow the standards deployed by the organization which will lead to the problem of other members having to understand the logic employed or to find the data description needed for solving a problem.

25

Software Failures

1. Failure causes a system crash and the recovery time is extensive; or failure causes a loss of function and data and there is no workaround
2. Failure causes a loss of function or data but there is manual workaround to temporarily accomplish the tasks
3. Failure causes a partial loss of function or data where user can accomplish most of the tasks with a small amount of workaround
4. Failure causes cosmetic and minor inconveniences where all the user tasks can still be accomplished

26

Comparing Software with Hardware

- To gain an understanding of SW and how to approach testing it, it is important to examine the characteristics of software that make it different from other things that human beings built.
- When hardware is built, the human creative process (analysis, design, construction, testing) is ultimately translated into a physical form.
- Software is a logical rather than a physical system element.
- Therefore, software has characteristics that differ considerably from those of hardware

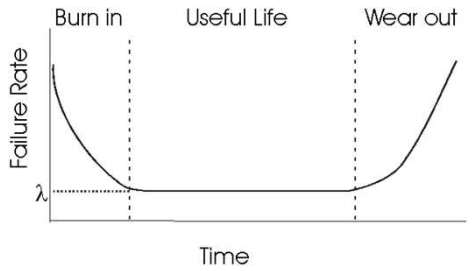
27

Difference between Software and Hardware

- Software is developed or engineered, it is not manufactured in the classical sense.
- Software does not wear out
- See failure curves for hardware and software
- Most software is custom-built, rather than being assembled from existing components

28

Failure Curve for Hardware



Burn-in: Procedure used in spotting weak parts or circuits of an electronic device (such as a computer) by running it at full power in a hot and humid environment for extended periods (up to 30 days for critical systems). This practice is based on the experience that semiconductor devices often show their defects in the first few days or weeks of operation.

29

Curve for Failure Rate over Product lifecycle

- Unlike hardware software does not physically wear out.
- It is subject to ongoing changes after release and is subject to changes in the external environment (such as an OS upgrade).
- These changes can introduce new faults or expose latent faults.

30

Curve for Failure Rate over Product lifecycle

- Initially, the first version has a high failure rate. With debugging and testing, as these are found and corrected the failure rate decreases until it reaches an acceptable level.
- Upgrades introduce new faults that are then fixed via maintenance releases
- Once the software is no longer supported the failure rate levels off until the product becomes obsolete.

31

Difficulties with Software vs Hardware

1. It is difficult for a customer to specify requirements completely.
2. It is difficult for the supplier to understand fully the customer needs.
3. In defining and understanding requirements, especially changing requirements, large quantities of information need to be communicated and assimilated continuously.
4. Software is easy to change.
5. Software is primarily intangible; much of the process of creating software is also intangible, involving experience, thought and imagination.
6. It is difficult to test software exhaustively

32

Forward Engineering

- This is an approach to designing correct systems. It starts with the user and ends with the correct implementation. The end product matches the specification.

33

Specifications

- Specifications play a key role. Detailed specifications provide the correct behaviour of the software.
- They must describe normal and error behaviour.

34

Testing in the development process

- Software has three key characteristics
  - User requirements that state the user's needs
  - A functional specification stating what the software must do
  - A number of modules that are integrated to form the final system
- These must be verified using the following four test activities

35

Unit Testing

- An individual unit of software is tested to ensure that it works correctly. This may be a single component or a compound component.
- A component might be a method, a class, or subsystem. It could be a single GUI component (e.g. button) or a collection of them (e.g. a window).
- This makes use of the programming interface of the unit.

36

Integration Testing

- Two or more units are tested to ensure that they interoperate correctly.
- This may use the programming interface or the System interface.
- Can be Top-Down, Bottom-up, or take an 'end-to-end user functionality' approach

37

System Testing

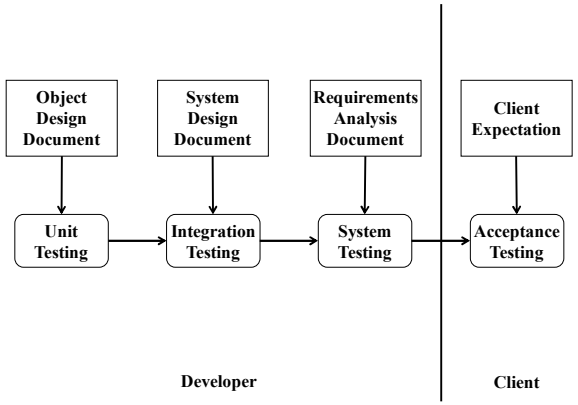
- The entire software system is tested to make sure that it works correctly and that it meets/solves the user's needs/problem.
- This uses the system interface which may be a GUI, Network interface, web interface etc…

38

Acceptance Testing

- The entire software system is tested to make sure that it meets the user's needs.
- Again, this uses the system interface.

39



40

Regression testing

- Regression testing is a form of software testing that confirms or denies a software's functionality after the software undergoes changes.
- make new regression tests as you find and correct bugs
- Ensure that you don't reintroduced errors that were previously fixed

41

Regression Testing practice

- Maintain a Strict regular Testing Schedule
- Use Test Management Software
- Categorize Your Tests so they are easily understood
- Prioritize Tests based on need, e.g. customer's requirements

42

Theory of Testing

- The goal is to identify the ideal test – that is, the minimum test data required to ensure that the software works for all inputs.

43

Exhaustive Testing

- This is generally not feasible as it would take too long or require too much memory space.
- A good test should have a high probability of finding faults, not duplicate another test, be independent in what it measures so no faults conceal one another, and test as much of the code as possible.

44

Time Allocation to create Software

- Since the 1970s, software developers began to increase their efforts on requirements analysis and preliminary design, spending 20 percent of their effort in these phases.
- More recently, software developers started to invest more time and resources in integrating the different pieces of software and testing the software pieces as units rather than as a complete entity
- Effort spent on determining the developmental requirements has also increased in importance, with 40% of software developers effort now happening in the requirements analysis phase

45

Testing Types

- Black-box testing – generate input values that exercise the specification and compare the actual output with the expected output
- White-box testing – generate input values that exercise the implementation and compare the actual output with the expected output
- Fault insertion – insert faults into the code or data to measure the effectiveness of testing or ensure that the fault is detected and handled correctly

46

Finishing Testing结束测试的标准

- A budgetary point of view: when the time or budget allocated has expired
- An activity point of view: when the software has passed all of the planned tests
- A risk management point of view: when the predicted failure rate meets some quality criteria

47

Criteria标准

- Usage-based criteria give priority to the most frequent sequences of program events.
- Risk of Release predicts the cost of future failures based on their chance of occurring.

48

Software Testing and Quality Assurance – Black Box Testing

黑盒测试

49

Black-box testing

- Black-box testing (functional testing) is based on the program specification, without consideration of internal structures of the software
- Aims to verify if the program meets the requirement specification
- Different approaches, each one has strengths and weaknesses

50

Black Box Testing types

1) Equivalence Partitioning (EP)
2) Boundary Value Analysis (BVA)
3) Combinational Testing
4) Sequential (State-Based) Testing
5) Testing with Random Data
6) Error Guessing/Expert Testing

51

White Box Test Types

7) Statement Coverage (SC)
8) Branch Coverage/Decision Coverage (BC/DC)
9) Condition Coverage (CC)
10) Decision Condition Coverage (DCC)
11) Multiple Condition Coverage (MCC)
12) Modified Condition/Decision Coverage (MCDC)
13) Path Coverage
14) Data flow (DU Pair) Coverage

52

Sequence of Testing

- In general, the normal usage of black-box and white-box testing techniques is as follows. Black-box testing is used initially to verify that the software satisfies the specification:
- Use Equivalence Partitioning to verify the basic operation of the software
- If the specification contains boundary values, use Boundary Value Analysis to verify correct operation at the boundaries
- If the specification states different processing for different combinations of inputs, use Combinational Testing to verify correct behaviour for each combination

53

Sequence of Testing

- If the specification contains state-based behaviour, or different behaviour for different sequences of inputs, then use Sequential Testing to verify this behaviour
- If there are reasons to suspect that there are faults in the code, perhaps based on past experience, then use Error Guessing/Expert Opinion to try and expose them
- If the typical usage of the software is known, then use Random test data to verify the correct operation under these usage patterns

54

Sequence of Testing

- For each of these tests, measure the statement and branch coverage. Normally the goal is to achieve 100% statement coverage and 100% branch coverage-because these are easy to measure automatically.
- If this has not been achieved, then white-box techniques can be used as follows:
- Use Statement Coverage to ensure that 100% of the statements have been executed.
- Use Branch Coverage to ensure that 100% of the branches have been taken.

55

Sequence of Testing

- Subsequently, if the code contains complex decisions, or if 100% branch coverage has not been achieved:
- Use Condition Coverage to ensure that every condition has been exercised
- Use Decision/Condition Coverage to ensure that every decision and every condition has been exercised
- Use Multiple Condition Coverage to ensure that every combination of conditions has been exercised

56

Sequence of Testing

- These white-box test techniques can be further augmented as follows:
- If the code contains complex end-to-end paths, then use Path Testing to ensure coverage of these
- If the code contains complex data usage patterns, then use DU Pair Testing to ensure coverage of these

57

Sequence of Testing

- In all cases, the decision to proceed with further tests is based on a cost-benefit trade-off: balancing the extra time and work required to do the extra tests justified against the extra confidence they will provide in the software quality.
- Often it is a judgement call as to what level of testing to execute.
- Fault Insertion can subsequently be used to measure the effectiveness of these tests in finding particular faults.

58

Note:

- Black-box tests can be written before, or in parallel with, the code (as they are based on the specifications).
- It normally serves no purpose to execute white-box tests before black-box tests.
- White-box testing can never be used as a substitute for black-box testing.
- White-box tests must be reviewed, and probably changed, every time the code is modified

59

Equivalence Partitioning

- In general, its goal is to verify the basic operation of the software
- Equivalence Partitioning is based on selecting representative values of each parameter from the equivalence partitions.
- Each equivalence partition for each of the parameters is a test case. Both the inputs and the output should be considered.
- The technique involves generating as few tests as possible: each new test should select data from as many uncovered partitions as possible.
- Error cases should be treated separately to avoid error hiding.
- The goal is to achieve 100% coverage of the equivalence partitions.

60

EP Test Cases

- Each partition for each input and output is a test case.
- It is often useful to use the prefix “EP-" for Equivalence Partition test cases.
- Note that the actual numerical values selected from the partitions are not the test cases, they are the test data.

61

EP Test Data

- Input test data is selected, based on test cases which are not yet covered. Ideally, each normal Test will include as many additional normal test cases as possible. Each error Test must only include one error test case.
- Expected output values are derived from the specification. However, the tester must ensure that all the test cases related to the output parameters are covered. It may be necessary to read the specification “backwards" to determine input values that will result in an output value being in the required equivalence partition.

62

EP Test Data

- Hint: it is usually easiest to identify test data by going through the test cases in order, selecting the next uncovered test case for each parameter, and then selecting an Equivalence Partition value.
- There is no reason to use different values from the same partition-in fact it is easier to review the test data for correctness if the one particular value is chosen from each partition, and then used throughout

63

Comment

- Equivalence Partitions provide a minimum level of black-box testing. At least one value has been tested from every input and output partition, using a minimum number of tests.
- These tests are likely to ensure that the basic data processing aspects of the code are correct. But they do not exercise the different decisions made in the code.
- This is important, as decisions are a frequent source of mistakes in the code. These decisions generally reflect the boundaries of input partitions, or the identification of combinations of inputs requiring particular processing.

64

EP 优点

- Provides a good basic level of testing.
- Well suited to data processing applications where input variables may be easily identified and take on distinct values allowing easy partitioning.
- Provides a structured means for identifying basic test cases.

65

EP 缺点

- Correct processing at the edges of partitions is not tested.
- Combinations of inputs are not tested.
- The technique does not provide an algorithm for finding the partitions or selecting the test data.

66

等价类划分的参数：显示和隐式

- Parameters
  - Input, output parameters of methods/functions
- Methods (and functions) have explicit and implicit parameters.
  - Explicit parameters are passed in the method call.
  - Implicit parameters are not: for example, in a C program they may be global variables; in a Java program, they may be attributes.
- Both types of parameter must be considered in testing. A complete specification should include all inputs and outputs.

67

Equivalence Partitions

- Equivalence Partitions (EP)
  - EP is a range of values for a parameter for which the specification states equivalent processing
  - Representative value from each partition is selected as test data.
  - Example:
    - Partition 1 (negative number): Integer.MIN_VALUE..-1
    - Partition 2 (0) : 0..0
    - Partition 3(positive number): 1.. Integer.MAX_VALUE
  - Any value in the partition is processed equivalently to any other value

68

Analysis of software specification

- Equivalence Partitions
  - Every value for every parameter is in one partition
  - No values between partition
  - Natural range of the parameter provides the upper and lower limits for partitions
  - Any one value can be selected to represent any other value in the same partition. Traditionally a value in the middle is picked
  - A single test uses a single value in a partition

69

EP

- Equivalence Partitions are useful for testing the fundamental operation of the software:
- if the software fails using EP values, then it is not worth testing with more sophisticated techniques until the faults have been fixed.

70

Test Data

- Each Test is specified by its associated test data. Each Test requires a unique identifier (unique across all tests for the method under test). The data for each test should include a unique identifier, the test cases covered, the input values, and the expected output value(s).
- Test input data is selected from arbitrary values within the equivalence partitions:
  - normally a central value is selected. Start with the first normal test case (i.e. not an error case). Then complete the tests for all the other normal test cases - for each additional test, data is selected to cover as many additional normal test cases as possible.

71

边界值分析的作用

- In general, the goal of BVA is to find faults in the software associated with decisions.
- The type of processing applied depends on the equivalence partitions of the inputs, and the correctness of the decisions tends to be associated with the boundaries of these partitions.

72

BVA Description

- Programming faults are often related to the incorrect processing of boundary conditions, so an obvious extension to Equivalence Partitioning is to select two values from each partition: the bottom and the top values.
- This doubles the number of tests, but is more likely to find boundary-related programming faults.
- Each boundary value for each parameter is a test case.

73

BVA Description

- As for Equivalence Partitioning, the number of tests is minimised by selecting data that includes as many uncovered test cases as possible in each new test.
- Error tests are as always considered separately- only one error boundary value is included per error test.
- The goal is to achieve 100% coverage of the boundary values.

74

BVA Test Cases

- Each boundary value for each partition for each input and output is a test case.
- It is good practice to give each test case for each SUT a unique identifier.
- Note that, unlike Equivalence Partitions, the values selected from the boundaries are the test cases

75

BVA Test Data

- Input test data is selected, based on test cases which are not yet covered.
- Ideally, each normal Test will include as many additional normal test cases as possible.
- Each error Test must only include one error test case.

76

BVA Comment

- There is little published evidence that using Boundary Values improves the effectiveness of testing, but experience indicates that this is likely to cover significantly more possible errors than Equivalence Partitions.
- Note that Boundary Value Analysis provides exactly the same test cases as Equivalence Partitioning for boolean and enumerated parameters

77

BVA 的优缺点

- Strengths
  - Test data values are provided by the technique.
  - Tests focus on areas where faults are more likely to be found.
- Weaknesses
  - Combinations of inputs are not tested.

78

边界值选取规则

- Some rules for picking boundary values:
- Every parameter has a boundary value at the top and bottom of every equivalence partition.
- For a contiguous data type, the successor to the value at the top of one partition must be the value at the bottom of the next.
- The natural range of the parameter provides the ultimate maximum and minimum values.
- It is important to note that boundary values do not overlap, and that there is no gap between partitions.

79

边界值分析的测试用例

- Boundary Values are the upper and lower values for each Equivalence Partition.
- Having identified the partitions, identifying the boundary values is straightforward.
- Test Cases: Each boundary value is a test case. Approach this systematically: unless there is a good reason not to, consider boundary values in increasing order (or in the order in which the values are defined).
- However, it is good practice to deal with all the error tests separately.

80

BVA Tests

- At the expense of approximately twice the number of tests, the minimum and maximum value of each Equivalence Partition has been tested at least once, using a minimum number of tests.
- However, combinations of different boundary values have not been exhaustively tested: in this example there is a limit of 128 candidate combinations (not all are possible).

81

Combinations of Values

- Some programs do simple data processing just based on the input values. Other programs exhibit different behaviour based on the combinations of input values.
- Handling complex combinations correctly is likely to be a source of faults, and they need to be analysed to derive associated test cases.

82

Combinations of Values

- There are a number of different techniques for identifying relevant combinations, such as Cause-Effect Graphs and Decision Tables.
- The recommended technique is Truth Tables- they are simplest to create, and tests can be derived directly from them.

83

组合测试

- The analysis of combinations involves identifying all the different combinations of input *causes* to the software and their associated output *effects*.
- The causes and effects are described as logical statements (or predicates), based on the specification of the software.
- These expressions specify the conditions required for a particular variable to cause a particular effect.

84

组合测试

- To identify a minimum subset of possible combinations that will test all the different behaviours of the program, a truth table is created.
- The inputs (“Causes") and outputs (“Effects") are specified as Boolean expressions (using predicate logic). Combinations of the causes are the inputs that will generate a particular response from the program.

85

组合测试

- These causes and Effects are combined in a Boolean graph or truth table that describes their relationship.
- Test cases are then constructed that will cover all possible combinations of Cause and Effect.
- For *N* independent causes, there will therefore be a total of 2^N different combinations. The truth table specifies how the software should behave for each combination.

86

Truth Tables

- A Truth Table is used to map the causes and effects through rules. Each rule states that under a particular combination of input causes, a particular set of output effects should result. It is critical to note that only one rule may be "active" at a time: the truth table is invalid if any input matches more than one rule!
- To generate the truth table, each cause is listed in a separate row, and then a different column is used to identify each combination of causes that creates a different effect. Each column is referred to as a" Rule" in the truth table{each rule is a different test case.

87

Truth Tables

- The truth tables for the three examples are shown next. Note that "T" is used as shorthand for true, and "F" for false.

88

Truth Table for inRange()

	Rules		
	1	2	3
Causes			
x<low	T	F	F
x≤high	T	T	F
Effects			
return value	F	T	F

Note that there is no Rule 4, as the combination " (x<low) and !(x ≤ high)" is not logically possible

. Rule 1 states that if (x<low) and (x≤high), then the return value is false.
. Rule 2 states that if !(x<low) and (x ≤ high), then the return value is true.
. Rule 3 states that if !(x<low) and !(x ≤ high), then the return value is false.

89

Don’t Care Conditions

- "Don't care" conditions exist where the value of a cause has no impact on the effect. These "Don't care" conditions are used to reduce the number of rules where the same output will be generated irrespective of whether the cause is true or false.
- In the worst case, if there are no "Don't care" conditions, *N* causes will create 2^N rules.
- "Don't care“ conditions are represented by using " *" for the causes in a truth table. The " *" is also used where an effect might be either true or false-the value is not determined by the specification.

90

Truth Table for condIsNeg()

	Rules		
	1	2	3
Causes			
flag	T	T	F
x<0	T	F	*
Effects			
return value	T	F	F

Rule 1 states that when x<0 and flag, the return value is true.
Rule 2 states that when !(x<0) and flag, then the return value is false.
Rule 3 states that when !ag, the return value is false.

Note the don't-care condition for the cause x<0 when the flag is false

91

Finding *Don’t Care* conditions

- To systematically introduce don’t-care conditions, find two rules with exactly the same effect, and just one different value for a cause, and merge them using a don't care for that cause.
- The rules need not be next to each other.
- Merging mutually exclusive causes is difficult, and takes great care-another reason to avoid mutually exclusive causes.

92

Partial Truth Tables

- Another technique to reduce table size is to use partial truth tables.
- These only contain a subset of the causes, the others being defined to have a constant value.
- For example, the table on the previous slide is a partial truth table for condInRange() where low≤high.

93

Partial Truth Tables and Error Hiding

- Partial truth tables are used to avoid handling error conditions.
- Due to error hiding, errors tend to hide all other behaviour. Unless there are interesting combinations of inputs that cause errors, it is usual to present truthtables without error conditions.
- This is shown for condInRange(), where the error condition low>high is not considered in the truth-table.
- However if errors are caused by a combination of input values, and not just particular values or relationships, then they should be included in the truth-table.

94

Combinational Testing

- The rules in a truth table are used to identify the output value for all the possible combinations of inputs.
- Typically error cases are not covered in a truth table, in order to reduce its size.
- However, if combinations of inputs can cause errors, then a separate table should be used, covering just the error outputs.

95

Analysis for Causes and Effects

- It takes practice to identify reasonable causes: The partitions provide a good starting point.
- There is no one right answer - typically the causes can be stated in a large number of different (but equivalent) ways.
- Note: comfortFlag is boolean, so it is redundant to state “comfortFlag==true".

96

Analysis for Causes and Effects

- The number of causes should be minimized to reduce the size of the truth table-in particular, where a parameter has a range of values that provide a particular response, this can be expressed as a single cause.
- The (non-error) causes for this program, taken from the specification, can be expressed as follows:
  - passengers $\leq$80
  - passengers $\leq$ 120
  - comfortFlag

97

Analysis for Effects

- When considering the non-error combinations for testing, only the non-error effects need be considered:
  - return value == SUCCESS
  - return value == FAILURE

98

Truth Table

- The rules must be (a) complete, and (b) independent. One rule, and exactly one rule, must be selected by any combination of the input causes. Do not include impossible combinations of causes.
- Develop the rules systematically, starting with all F at the left-hand side of the table, and ending with all T at the right-hand side (or, alternatively, starting with all T, and ending with all F).
- Use don't care conditions (indicated by a '*') when the value of a cause has no impact on the effect of the rule.
- It sometimes helps to develop the full truth table first, and then remove redundant and impossible rules
- Note that we are not considering error cases: so the following truth tables are defined to always meet the conditions: passengers>0

99

Test Cases

- Each rule from the truth-table is a test case. If an SUT includes multiple truth-tables (for example, in a class) then it is useful to give each test case a unique identifier.
- Often a truth-table will only include normal cases, due to the number of rules required to describe all the possible error cases.
- If error cases are included, then it is often in a separate table for clarity.

100

Test Data

- Input test data is selected, based on test cases (rules) which are not yet covered. Each Test will cover exactly one test case (or rule).
- Expected output values are derived from the specification (the truth-table).

101

Picking Test Data

- It is usually easiest to identify test data by going through the test cases (rules) in order, and selecting a value for each parameter that matches the required causes.
- As for equivalence partitions, it can be easier to review a test for correctness if as few different values as possible are used for each parameter.
- For expected output, the value from the specification must, if the technique has been followed properly, match the required effect.

102

Comment

- The number of tests is reduced using `don't-care" conditions where the value of a particular cause has no effect on the output.
- This means that Combinational Testing does not test all the combinations of causes

103

Comment

- Testing all the possible combinations of causes and effects would cause a very large number of tests for a typical program. Picking a minimum number of rules, based on using “don't care" conditions reduces the number of tests significantly-at the cost of reducing the test coverage.
- It should be noted that Equivalence Partitions/Boundary Values are complementary to Combinational Testing. There is again little published evidence as to the effectiveness of truth table testing, but experience in programming indicates that this is likely to cover different errors from Equivalence Partitions and Boundary Values.

104

组合测试的优缺点

- Strengths
  - Exercises combinations of test data
  - Expected outputs are created as part of the process
- Weaknesses
  - The truth tables can sometime be very large. The solution is to identify subproblems, and develop separate tables for each.
  - Very dependent on the quality of the specification - more detail means more causes and effects, which takes more time to test; less detail means less causes and effects, but less effective testing

105

Test Data

- Each test case must be covered in a separate test-it is not possible to have multiple combinations in the same test.
- Test input data is selected by picking values that satisfy the causes and effects for the rule to be tested.

106

随机测试

- Random testing can be used to achieve one or more of a number of goals-the main ones are as follows:
  - To use a probabilistic approach to search for faults not found with the previous, systematic approaches.
  - To use a probabilistic approach and demonstrate the correctness of software in a broader range of scenarios that explored with the previous, systematic approaches.
  - To estimate quality statistics, such as the mean time to failure (MTTF), by applying inputs which statistically match the expected inputs in real use, and measuring the failure rate.

107

随机测试

- Test data is generated using random number generators. The distribution may be uniform, or chosen to mimic, in a statistical sense, the type of inputs that the program will receive in real use. If the specification is clearly written and thorough, then it should be possible to find the set(s) of possible input values.
- Typically uniformly distributed random numbers are used in Unit Testing, as the code may well be expected to work in a variety of different scenarios or within different programs. Statistically relevant distributions of random numbers are more often used in System Testing, where specific scenarios or customer usage patterns may be known.

108

随机测试

- The overall goal of Random Testing is to achieve a “reasonable" coverage of the possible values for each input parameter, based on its distribution.
- This can be determined heuristically (using, for example, 10 random values), or based on a statistical sample size determined from the required confidence in the coverage

109

Test Cases

- Each test case is represented by a set of (random) input values, one for each parameter.
- If the test is fully automated, then each test case is represented by a distribution of values for a particular parameter. This will normally include the upper and lower limits, and the distribution to be used between these limits to select a random value.
- Each test case should be given a unique identifier.

110

Test Data

- Input test data is selected, normally automatically, based on the test cases.
- Expected output values are derived from the specification. This may be manual or automated. Automating the interpretation of a specification to produce the expected output is difficult. Three approaches are:
  - 1. Select the output partition at random first, then select random output values from this partition, and then select matching input values (which may be random or non-random depending on the specification).
  - 2. Write a Test Oracle in a higher-level language, which is more likely to be correct.
  - 3. Use post-conditions to determine the validity of an output after it is produced (rather than producing the expected output value before the test is run).

111

Comment

- Random test data generation is straightforward to implement and leads to a fast generation of test cases. However, calculating the Expected Output from the specification is as time consuming as for EP and BV.
- If the distribution/histogram of the real-world input data is known, then this provides a mathematical basis for selecting a set of input test case values. The measured test failure rate then provides an indication of the expected failure rate in use.
- However, if the distribution is not known, then the basis for choosing the random data may not reflect its use, and the failure rate cannot be predicted from the results.
- Furthermore, the random input data obtained may not have a sufficient set of illegal or extreme values, or even combinations of valid values, that will test the program thoroughly.

112

随机测试的优缺点

- Strengths
  - Fast generation of test cases
  - Can offer a mathematical basis for selecting an appropriate set of input values
- Weaknesses
  - Possibly an insufficient set of extreme or illegal values may be tested
  - If the distribution of the input is unknown the input values chosen may not reflect typical usage

113

回归和稳定性测试

- Randomisation can also be used to select a small set of tests from a large suite in order to execute the suite more quickly. This can be particularly useful for Regression Testing. The effectiveness of this depends on how the random tests are selected.
- More sophisticated techniques may be “directed", using feedback from each test to select data for the following test.
- Random data selection is sometimes used for Stability Testing, to ensure that no input data value causes the software to crash or raise unexpected exceptions. This technique is easy to implement in an automated manner, but is unlikely to find faults except in low-quality code.

114

随机测试

- Random data for Unit Testing can be easily generated by first randomly selecting which partition the output value is to be in, and then randomly selecting appropriate input values.
- Uniformly distributed random numbers are generally used for Unit Testing

115

Test Data

- The random values shown here have been generated (using a Java program and the Random class) as follows:
- 1. Select the result at random (SUCCESS, FAILURE, or ERROR)
- 2. Select the comfortFlag at random
- 3. Select a value for passengers from the range of values that will cause the required result:
  - ERROR: from Integer.MIN VALUE to 0
  - SUCCESS and false: from 1 to 120
  - SUCCESS and true: from 1 to 80
  - FAILURE and false: from 121 to Integer.MAX VALUE
  - FAILURE and true: from 81 to Integer.MAX VALUE

116

Test Data

ID	Inputs		Exp. Output
	passengers	comfortFlag	return value
T4.1	16	false	SUCCESS
T4.2	46	true	SUCCESS
T4.3	-1974596984	true	ERROR
T4.4	-122368221	false	ERROR
T4.5	10	true	SUCCESS
T4.6	40	false	SUCCESS
T4.7	112	false	SUCCESS
T4.8	1950430522	true	FAILURE
T4.9	74	false	SUCCESS
T4.10	-1942749054	true	ERROR

Note that random data values are normally selected at runtime using test automation, and not specified beforehand as shown here.

117

错误推测法

- The goal of error guessing, in general, is to try find faults in the software based on the experience of domain experts, or software experts.
- This is an ad-hoc approach, based on intuition and experience. Test data is selected that is likely to expose faults in the code. Some typical examples of inputs likely to cause problems are given on the next slide.

118

常见错误

- Empty or null strings, arrays, lists, and class references. These may find code that does not check for empty or non-null values before using them.
- Zero as a value, or as a count of instances or occurrences. These may find divide-by-zero faults.

119

常见错误

- Spaces or null characters in strings. This may find code that does not process strings correctly, or does not trim whitespace before trying to extract data from the string.
- Negative numbers. These may find faults in code that only expects to receive positive numbers.

120

Test Cases

- The tester selects values which are likely to produce errors. Each value is a test case.
- Each test case should have a unique identifier.
- This technique can produce both normal and error test cases.
- The values selected are those that are likely to expose faults in the code, they are not necessarily illegal values.

121

Test Data

- Input test data is selected, based on test cases which are not yet covered.
- As for other test techniques, error cases should be executed individually.
- Expected output values are derived from the specification.
- It may be required to read the specification “backwards" to determine input values for output parameter test cases.

122

Comment

- With experienced testers, this can be a very effective complement to other testing techniques.
- It depends on how well the testers know the types of mistakes that the developers are likely to make, or mistakes that have a high impact on the final product.

123

Appraisal

- Strengths
  - Intuition can frequently provide an accurate basis for finding faults.
  - The technique is very efficient, as it focuses on likely faults.
- Weaknesses
  - The technique relies on experienced testers-but they are not always available.
  - The ad-hoc nature of the approach means it is hard to ensure completeness of the testing.

124

Elimination of Duplicate Tests

- The use of 'standard' values simplifies the task of identifying identical tests.
- In cases where this has not been done, then further test elimination can be achieved by identifying equivalent tests, and making minor changes to the test data.

125

白盒测试

- PPT暂缺，只找到了PDF版本的PPT
- 需要另外进行打印

126

系统测试和集成测试

- PPT也没有，只有直接打印的PDF版本

127

Testing and Debugging

128

Testing and Debugging

- Defect testing and debugging are distinct processes
- Defect testing is concerned with confirming the presence of errors
- Debugging is concerned with locating and repairing these errors
- Debugging involves formulating a hypothesis about program behavior then testing these hypotheses to find the system error

129

Debugging: Issues

- observed bug and its cause may be geographically separated
- observed bug may disappear when another problem is fixed
- cause of bug may be due to human error that is hard to trace
- cause of bug may be due to assumptions that everyone believes
- observed bug may be intermittent because of a system or compiler error

130

Debugging: Approaches

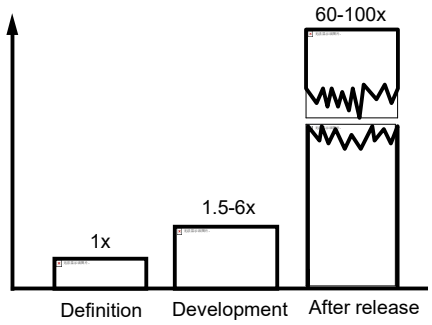
- Brute Force – hack away at the code until it is found
- Backtracking – fine for small programs
- Cause elimination – hypothesise about what is causing the bug and input test data to check this

131

Cost of Error Correction
纠正错误的cost

132

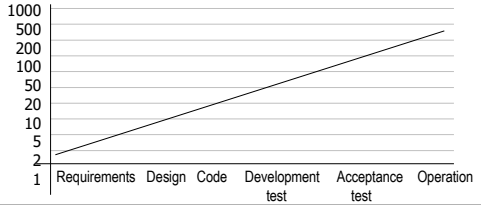
The Cost of Change



133

Cost of Error Correction

- Cost of error-correction goes up with life cycle stage
- 60% of errors introduced during design, 40% during implementation
- 2/3 of design errors are discovered when the software is operational



134

Software Testing and Quality Assurance

Testing in the Software Process

135

大爆炸开发

- Big Bang - wait until all the code has been written and then to test the finished product all at once
- It is an attractive option to developers because testing activities do not hold back the progress towards completing the product.

136

Big Bang Development Drawbacks

- However, it is a very risky strategy as the likelihood that the product will work, or even be close to working can be very low,
- Also, it is particularly dependent on program complexity and program size.
- Additionally, if tests do reveal faults in the program, it is much more difficult to identify their source. It is then necessary to search through the complete program to locate them.

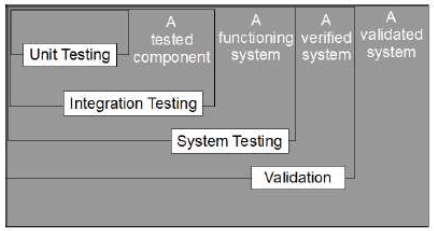
137

阶段化测试与开发Stages

- Individual modules, or software features, are tested as they are written.
- This process continues as additional software increments are produced until the product is completed.
- This may have the effect of slowing down the arrival of the final product
- However, it should produce one that has fewer errors and that the development team will have much more confidence in.

138

4个阶段



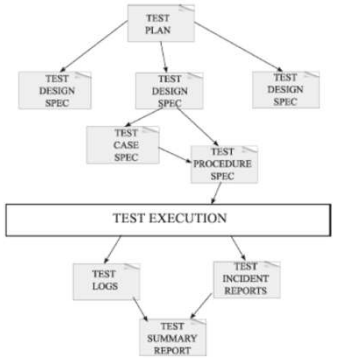
139

测试计划

- A typical Test Plan would include such information as:
  - Items to be tested
  - Tasks to be performed
  - Responsibilities
  - Schedules
  - Required resources

140

Test Planning – IEEE model



The IEEE standard 892-1998 provides a formal framework within which a plan can be prepared

141

软件开发生命周期

- The Software Development Life Cycle is a structured plan for organizing the development of a software product.
- The need for such planning arose with the growth in size and complexity of software projects.
- By adopting a plan for the development it was intended to create a repeatable and predictable software development process that would automatically improve productivity and quality.

142

瀑布模型

- This model visualizes the software development process as a linear sequence of phases 一系列线性的阶段
- 介绍瀑布模型的组成
  - It begins with a requirements analysis
  - followed by the system design
  - then coding, testing, and
  - ending with system maintenance after the software has been deployed

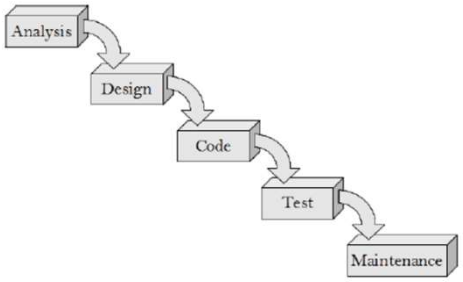
143

瀑布模型的特点

- All the planning is done at the beginning, and once created it is not to be changed. 计划好了就不能变
- There is no overlap between any of the subsequent phases. 各阶段没有重叠
- Often anyone's first chance to "see" the program is at the very end once the testing is complete.

144

The Waterfall Model



145

瀑布模型的优点

- If time is spent early on making sure that the requirements and design are absolutely correct then this will save much time and effort later.
- There is an emphasis on documentation which keeps all knowledge in a central repository and can be referenced easily by new members joining the team.

146

瀑布模型的缺点

- Few visible signs of progress until the end of the project
- It is not flexible to changes
- Time-consuming to produce all the documentation
- Tests are only carried out at the end – this could mean a compromise if time or budgetary constraints exist

147

瀑布模型的缺点

- Having to test the program as a whole could result in incomplete testing
- If testing does identify a fault that suggests a redesign it may be ignored because of the trouble involved
- If the customer is unhappy it may incur a long maintenance phase resolving their issues

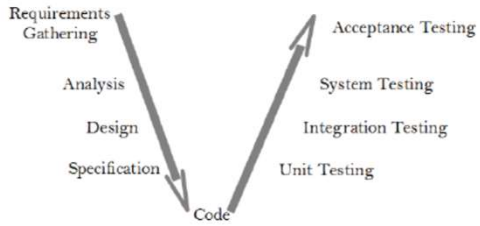
148

V字模型

- This is an extension of the Waterfall model but in contrast it emphasizes Verification & Validation by marking the relationships between each phase of the life cycle and testing activities. 瀑布模型的拓展
- Once the code implementation is finished the testing begins. 代码写完开始测试
- This starts with unit testing, and moves up one test level at a time until the acceptance testing phase is completed 从单元测试逐渐提高级别

149

V字模型



150

V模型的文档

- Each document produced is associated with pairs of phases in the model.
- These are the
  - (a) Detailed Design Specifications,
  - (b) the System Design Specifications,
  - (c) the System Requirements Specification,
  - (d) the User Requirements Specification.

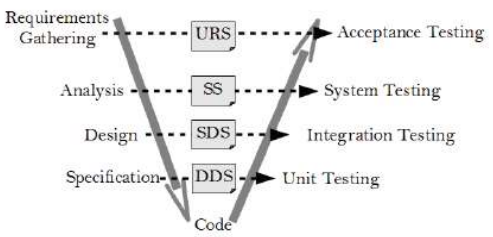
151

文档的具体要求

- Requirements Gathering produces the User Requirements Specification (URS), which is both the input to Analysis, and the basis for Acceptance Testing.
- Analysis produces the System Specification (SS) – also know as the Software Requirements Specification (SRS) – which is both the input for Software Design, and the basis for System Testing.
- Design produces the System Design Specification (SDS), which is both the input for the detailed Specification phase, and the basis for Integration Testing.
- The Specification activity produces the Detailed Design Specifications (DDS), which are both used to write the code, and also are the basis for Unit Testing.

152

V模型文档



153

V-model 优点

- It is simple and easy to manage due to the rigidity of the model,
- It encourages Verification and Validation at all phases:
- Each phase has specific deliverables and a review process.
- It gives equal weight to testing alongside development rather than treating it as an afterthought.

154

V-model 缺点

- Its disadvantages are that similarly to the Waterfall model there is no working software produced until late during the life cycle
- It is unsuitable where the requirements are at a moderate to high risk of changing.
- It has been suggested too that it is a poor model for long, complex and object-oriented projects

155

敏捷开发

- Agile methods share with other incremental development methods an emphasis on building releasable software in short time periods.
- However, Agile development differs from the other development models in that its time periods are measured in weeks rather than months and work is performed in a highly collaborative manner和其他模型的区别

156

敏捷开发

- For effective testing:
  - When the developers “negotiate” the requirements for the upcoming iteration with the customers, the testers must be full participants in those conversations.
  - The testers immediately translate the requirements that are agreed upon in those conversations into test cases.
  - When requirements change, testers are immediately involved because everyone knows that the test cases must be changed accordingly.

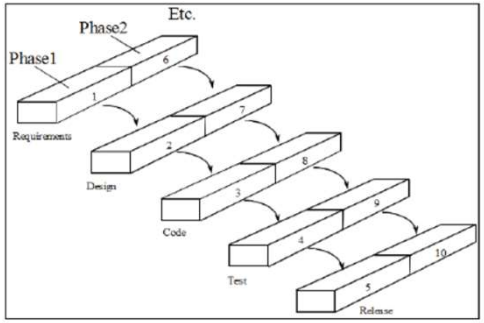
157

增量模型

- The incremental model begins with a simple implementation of a part of the software system. With each increment the product evolves with enhancements being added every time until the final version is reached.
- Testing is an important part of the incremental model and is carried out at the end of each iteration. This means that testing begins earlier in the development process and that there is more of it overall.
- Much of the testing is of the form of regression testing, and much re-use can be made of test cases and test data from earlier increments.

158

Incremental Development



159

增量模型的优点

- A major advantage of the incremental model is that the product is written and tested in smaller pieces, reducing risk and allowing for change to be included easily
- The customer or users is involved from the beginning which means the system is more likely to meet their requirements and they themselves are more committed to the system

160

增量模型的缺点

- It can be difficult to manage because of the lack of documentation in comparison to other models
- The continual change to the software can make it difficult to maintain as it grows in size.

161

Extreme Programming极限编程

- Extreme Programming (XP) is a subset of the philosophy of Agile software development. 是软件敏捷开发的子集
- 三个特点
- It emphasizes code reviews, continuous integration and automated testing, and very short iterations.强调代码审查、持续集成和自动化测试，以及非常短的迭代
- It favours ongoing design refinement (or *refactoring*), in place of a large initial design phase, keeping the current implementation as simple as possible.
- It favours real-time communication, preferably face-to-face, over writing documents, and working software is seen as the primary measure of progress.

162

极限编程特点 cont.

- The methodology also emphasizes team work. Managers, customers, and developers are all part of a team dedicated to delivering quality software.
- Programmers are responsible for testing their own work; testers are focused on helping the customer select and write functional tests, and on running these tests regularly

163

Extreme Programming - Values

- Communication: XP programmers communicate with their customers and fellow programmers 交流
- Simplicity: they keep their design simple and clean 简化
- Feedback: they get feedback by software testing from the start 反馈
- Courage: they deliver the system to customers as early as possible and implement changes as suggested, responding with courage to changing requirements 激励

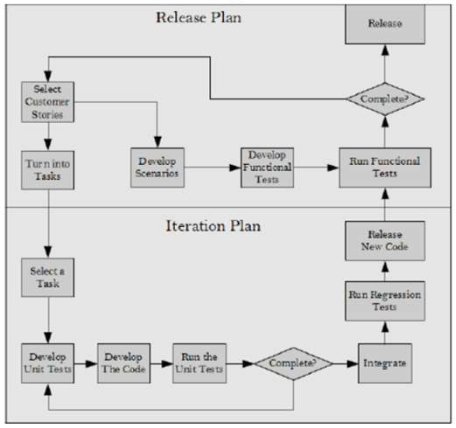
164

极限编程 开发过程

- A project begins by identifying a metaphor that describes the system. The metaphor acts as a conceptual framework, identifying key objects and providing insight into their interfaces.
- The first iteration sets the initial skeleton of the project.
- User Stories, in the format of about three sentences of text, are written by the customers. These are the features of the application that the system needs to have and are used to drive the creation of the acceptance tests later on.

165

Extreme Programmming



166

极限编程的一些特点和细节

- A Release Plan is created from the User Stories. This plan sets out the overall project. 发布计划
- Iteration plans are then created for each individual iteration, using development time estimates for each user story. 迭代计划
- The customer specifies scenarios to show that a user story has been correctly implemented. A set of functional (or acceptance) tests is developed based on these. 客户指定场景，在此基础上开发一组功能(或验收)测试
- The customers are responsible for verifying the correctness of the acceptance tests, and reviewing test scores to decide which failed tests are of highest priority. 客户责任
- Acceptance tests are also used as regression tests prior to the release of a new version of the software.接受测试

167

Extreme Programming

- Each Iteration Plan is developed in detail just before the iteration begins and not in advance. Iterations are between 1 and 3 weeks in duration. 迭代计划
- User Stories are converted into implementation tasks, recorded on task cards. A programmer takes a task card, writes the unit test cases for the task, implements the code, and tests it. 用户叙事
- When the tests pass, the programmer then integrates the new code, runs regression tests, and releases the code for full functional testing. 测试通过
- After this, there is a tested, working, software feature ready to demonstrate to the customer. 结果发布
- Eventually, after all the iterations have been completed the product will be finished. 结束

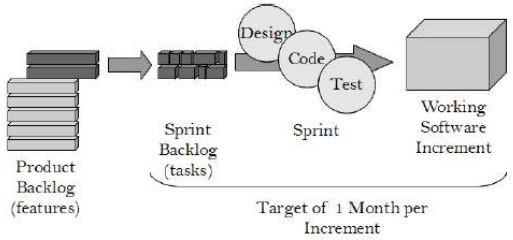
168

SCRUM 模型与XP的区别

- Scrum teams work in iterations that are called sprints. These can last a little longer than XP iterations.
- Scrum teams do not allow changes to be introduced during the sprints. XP teams are more flexible with changes within an iteration as long as work has not started on that particular feature already.
- XP implements features in a priority order decided essentially by the customer, while in SCRUM there is more flexibility for additional stakeholders to influence the ordering.
- In XP unit testing and simple design practices are built in, while in SCRUM it is up to the team to organize themselves.

169

Scrum



170

Scrum的backlog和sprint

- SCRUM starts with the Product Backlog which is a prioritized list of all product requirements.从产品待办事项列表开始
- The backlog items come from: Users, customers, sales, marketing, customer service, engineering, and anyone else that has an interest in the outcome of the project. The Product Backlog is never finalized, it emerges and evolves with the product. 待定项的来源
- SCRUM teams take on as much of the product backlog as they think they can turn into an increment of product functionality within a 30-day iteration. This is called a Sprint. 团队职责
- The team maintains a list of tasks to perform during each Sprint that is called the Sprint Backlog. 维护任务列表
- Multiple teams can take on product increments in parallel, all working from the same Product Backlog. A project will have multiple sprints. 多个团队合作

171

Scrum的sprint

- Before a sprint starts a meeting is conducted to decide what is going to be developed and delivered in that particular sprint. 开会
- After the completion of the sprint, a meeting is held to collect feedback from the team. This feedback helps in planning and working on the next sprint. sprint完成之后
- As the development team starts to work on the next sprint, the testing team carries out functional testing of the features developed in the last sprint. 下个spring开始之前
- This approach gives good results as the testing team works with the developers from the start of the project. 项目结果

172

Organizing the stories – A user story map

- This is related to the product backlog in SCRUM
- The map outlines the big picture
- The product backlog approach usually arranges the stories in a build order from the highest value to lowest value – this suits project managers!
- However, it is better to arrange a user story map so that it is clear what the system does.

173

用户叙事图

- On top are the big stories/activities. These are the essentials of the software.
- Each big story will break down into smaller stories/tasks such as “send message” and “” receive message”.
- The smaller tasks are prioritized from absolutely necessary to being a nice extra feature.

174

Devops模型

- DevOps – a combination of **Development & Operations** – is a software development methodology which looks to integrate all the software development functions from development to operations within the same cycle. 定义
- This calls for higher level of coordination within the various stakeholders in the software development process (namely Development, QA & Operations)

175

Devops

- Requires Continuous Integration combined with Continuous Delivery 需要持续集成
- This approach places great emphasis on automation of build, deployment and testing.

176

Devops

- Build tools help to achieve fast iteration, 创建工具快速迭代
- Continuous-Integration (CI) tools to merge code from multiple developers and check for faults. 持续迭代工具
- Additionally, when the software is in operation, Logging provides traces for identifying and tracking application faults 日志

177

DevOps and 项目经理

- Complex software architectures and features must be decomposed into small chunks that can be produced and deployed independently.
- visibility of the configuration and build must be ensured so that everyone is aware of what is deployed, with which versions and dependencies.
- The shift must be made from legacy practices to those that facilitate the integration of development and operations.

178

DevOps and 测试

- Developers assume responsibility for both the testing and release environment.
- The development team performs test-driven development and CI.
- The use of CI means that the system is constantly being tested
- To accelerate this activity, those responsible for quality assurance must ensure automation of all test cases and full code coverage.

179

好处、优点 of Devops

- Faster Time to market – reduced cycle times and higher deployment rates
- Increased Quality – better availability of the product as it is being created, increased change success rate and fewer failures
- Increased organizational effectiveness – more time spent on value adding activities and greater value being delivered to the customer

180

困难的地方

- DevOps requires high level of coordination between various functions of the deliverable chain.
- A need to master the various automation and continuous integration tools

181

International Comparisons

2003 IEEE Software

- **Survey:** Completed in 2002-2003
- **Objective:** Determine usage of iterative versus Waterfall-ish techniques, with performance comparisons
  - 118 projects plus 30 from HP-Agilent for pilot survey
- **Participants**

  India: Motorola MEI, Infosys, Tata, Patni

  Japan: Hitachi, NEC, IBM Japan, NTT Data, SRA, Matsushita, Omron, Fuji Xerox, Olympus

  US: IBM, HP, Agilent, Microsoft, Siebel, AT&T, Fidelity, Merrill Lynch, Lockheed Martin, TRW, Micron Tech

  Europe: Siemens, Nokia, Business Objects

182

Observations

- Most projects (64%) are not pure waterfall; 36% were
- Mix of "conventional" and "iterative" common
- Best "nominal" quality from traditional "waterfall" (fewer cycles & late changes implies less bugs)
- Best balance of quality, flexibility, cost, & speed from combining conventional & iterative practices
- However, differences in quality between waterfall & iterative disappear if a bundle of techniques of used:
  1. Early prototypes (get customer feedback early)
  2. Design reviews (continuously check quality of design)
  3. Regression tests on each build (continuously check quality of code and functional status)

183

Process related quality and standards models - CMM

- The *Capability Maturity Model* was developed initially by the Software Engineering Institute at Carnegie Mellon University in 1991 as a model based on best practices for software development.
- The CMM ranks software development organizations in a hierarchy of five levels, each with a progressively greater capability of producing quality software

184

CMM等级

- Level 1 – Initial (also referred to as Chaotic or *Ad Hoc*). Processes are typically undocumented and dynamic. They are driven in an uncontrolled, reactive manner by users or events.
- Level 2 – Repeatable. Processes are repeatable, possibly with consistent results. Process discipline is unlikely to be rigorous, but where it exists it may help to ensure that existing processes are maintained during times of stress.
- Level 3 – Defined. Defined and documented standard processes are established and subject to some degree of improvement over time. These processes are used to establish consistency of process performance across the organization.

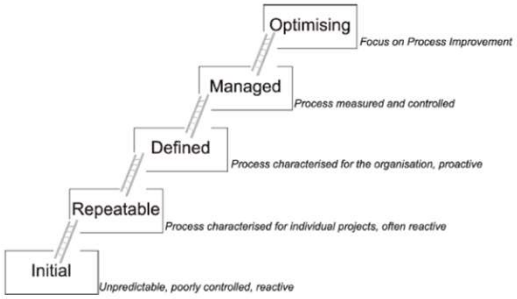
185

CMM等级

- Level 4 – Managed. Using process metrics, management effectively controls the process. This includes identifying ways to adjust and adapt the process to particular projects without measurable losses of quality or deviations from specifications.
- Level 5 – Optimizing. The focus is on continually improving process performance through both incremental and innovative technological changes/improvements.

186

CMM Levels



187

CMMI

- CMMI defines a number of roles and Software Engineering process areas.
- Testing is mainly associated with the Software Quality Assurance (SQA) and Software Quality Control (SQC) roles in the CMMI model. The main test activities are in the following Software Engineering process areas:
  - CMMI Technical Solution – Unit Testing
  - CMMI Product Integration – Integration Testing
  - CMMI Verification – System Testing
- The test results provide a measure of the process quality, used as a significant input to the Process Assessment and Improvement activity.

188