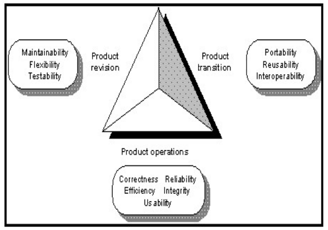
What 什么是质量

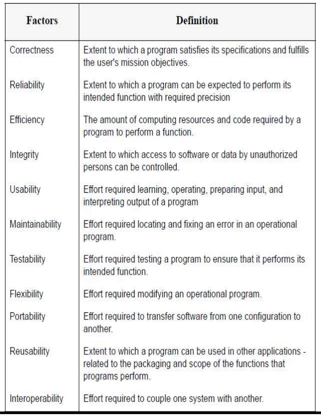
"An inherent or distinguishing characteristic"
(American Heritage Dictionary)

- Quality is what we love
  - Quality means value to somebody
  - Every software problem is of a quality problem
- Quality has different meanings to different people
  - Customer view: fit for use or meet the needs
  - Project Manager view: deliver compliant products in time
  - Developer/Tester view: bug-free
  - What's your view?

什么是质量

McCall's Quality Model





Factors	Definition
Correctness	Extent to which a program satisfies its specifications and fulfills the user's mission objectives.
Reliability	Extent to which a program can be expected to perform its intended function with required precision
Efficiency	The amount of computing resources and code required by a program to perform a function.
Integrity	Extent to which access to software or data by unauthorized persons can be controlled.
Usability	Effort required learning, operating, preparing input, and interpreting output of a program
Maintainability	Effort required locating and fixing an error in an operational program.
Testability	Effort required testing a program to ensure that it performs its intended function.
Flexibility	Effort required modifying an operational program.
Portability	Effort required to transfer software from one configuration to another.
Reusability	Extent to which a program can be used in other applications - related to the packaging and scope of the functions that programs perform.
Interoperability	Effort required to couple one system with another.

Role

	Quality Assurance（QA）	**Tester**
Orientation	**Process/Policy Oriented**	**Products/Deliverables Oriented**
Defect Goals	**Prevent Defects**	**Discover Defects**
Means	**Audit/ Governance**	**Kinds of Test Methods**
Methodology	目标	手段（之一）

按照测试方法划分

- 白盒测试
  - 语句覆盖:
  - 条件覆盖
  - 路径覆盖
  - ……
- 黑盒测试
  - 等价类法
  - 边界值法
  - ……
- 灰盒测试

按照测试的粒度或测试的范围

- Unit Test
- Integration Test
- System Test
- Acceptance Test

- Smoke Test 与 接受

- Regression Test 与 退化

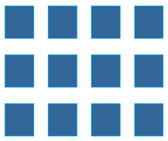
- **1.何为冒烟测试**
  冒烟测试是自由测试的一种。冒烟测试在测试中发现问题，找到了一个bug，然后开发人员会来修复这个bug。这时想知道这次修复是否真的解决了程序的bug，或者是否会对其它模块造成影响，就需要针对此问题进行专门测试，这个过程就被称为冒烟测试。在很多情况下，做冒烟测试是开发人员在试图解决一个问题的时候，造成了其它功能模块一系列的连锁反应，原因可能是只集中考虑了一开始的那个问题，而忽略其它的问题，这就可能引起了新的bug。
  冒烟测试引入到软件测试中，是指测试人员在正规测试一个新版本之前，先投入较少的人力和时间验证一个软件的主要功能，如果主要功能都没有实现，则打回开发组重新开发。这样做的好处是可以节省大量的时间成本和人力成本。

  2.何为回归测试
  回归测试是指修改了旧代码后，重新进行测试以确认修改没有引入新的错误或导致其他代码产生错误。回归测试作为软件生命周期的一个组成部分，在整个软件测试过程中占有很大的工作量比重，软件开发的各个阶段都会进行多次回归测试。在渐进和快速迭代开发中，新版本的连续发布使回归测试进行的更加频繁，而在极端编程方法中，更是要求每天都进行若干次回归测试。因此，通过选择正确的回归测试策略来改进回归测试的效率和有效性是非常有意义的。

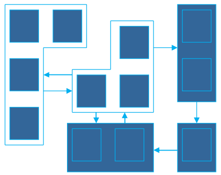
  回归测试一般是在进行软件的第二轮测试开始的，验证第一轮中发现的问题是否得到修复。当然回归也是一个循环的过程，穿插在软件测试整个生命周期里面。如果回归的问题不通过，则需要开发人员修改后再次回归，直到通过为止。

  3.两者有何区别
  冒烟测试就是完成一个新版本的开发后，对该版本最基本的功能进行测试，保证基本的功能和流程能走通。如果不通过，则打回开发那边重新开发；如果通过测试，才会进行下一步的测试(功能测试，集成测试，系统测试等等)。
  冒烟测试优点是节省测试时间，防止build失败。缺点是覆盖率还是比较低。
  回归测试我有两层理解，一是就是当你修复一个bug后，把之前的测试用例再次应用到修复后的版本上进行测试。二是当一个新版本开发好后，而且冒烟测试通过，此时可以先用上一个版本的测试用例对新版本进行测试，看是否有bug！其实回归测试用的很多，比如新增加一个功能模块等等，所以自动化测试可以高效率的进行回归测试。

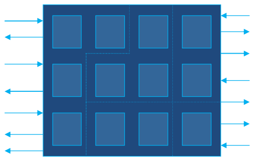
Test Techniques – By Stage



Unit Test



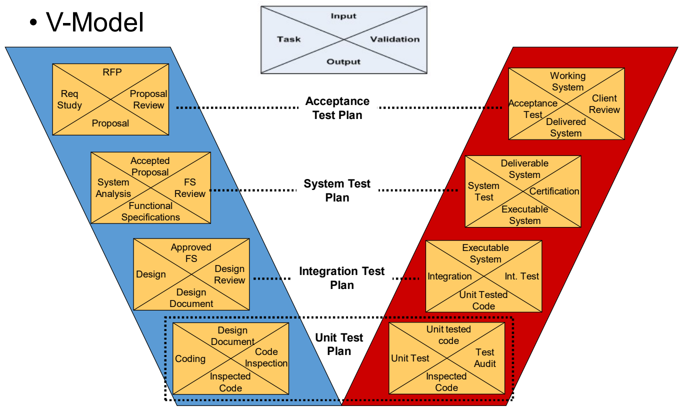
Integration Test



System & Acceptance Test

按照测试的粒度或测试的范围

- V-Model



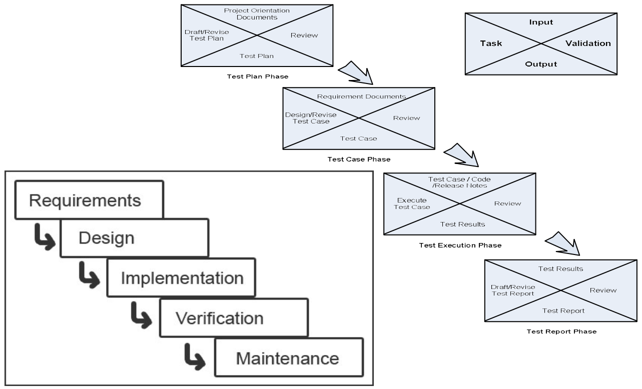
按照被测软件的评价标准

- 功能性测试
- 性能测试
- 安全性测试
- 高可用性测试

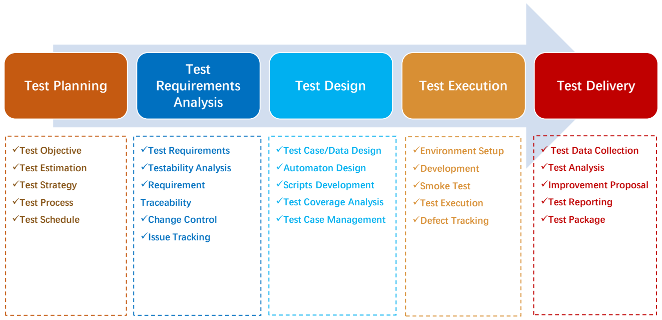
α测试和β测试

- α测试
  - 开发团队内部测试
  - 针对软件的各个方面进行测试（功能，性能，安全，可移植，可扩展，可用性）
  - 测试方法较多（黑盒，白盒）
  - 测试目标 [0.1，1.0)
- β测试
  - 最终用户测试
  - 主要针对功能测试
  - 使用黑盒测试
  - 测试目标为有限的production candidate

1，从简单的开始——瀑布模型



一个经典的瀑布模型下的瀑布式的测试流程



Test Planning
- ✓Test Objective
- ✓Test Estimation
- ✓Test Strategy
- ✓Test Process
- ✓Test Schedule

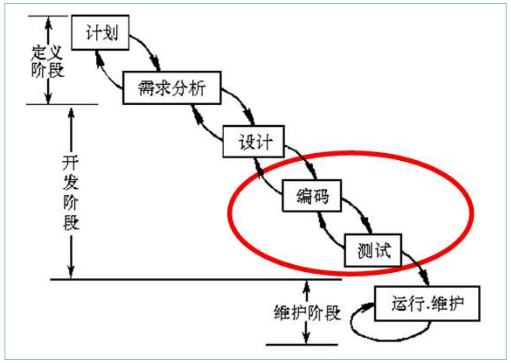
Test Requirements Analysis
- ✓Test Requirements
- ✓Testability Analysis
- ✓Requirement Traceability
- ✓Change Control
- ✓Issue Tracking

Test Design
- ✓Test Case/Data Design
- ✓Automaton Design
- ✓Scripts Development
- ✓Test Coverage Analysis
- ✓Test Case Management

Test Execution
- ✓Environment Setup
- ✓Development
- ✓Smoke Test
- ✓Test Execution
- ✓Defect Tracking

Test Delivery
- ✓ Test Data Collection
- ✓Test Analysis
- ✓Improvement Proposal
- ✓Test Reporting
- ✓Test Package

2，带回溯的瀑布模型的改进型
----回溯造成了循环



3，V模型虽然也是瀑布的一种，但强调了测试

- V-Model

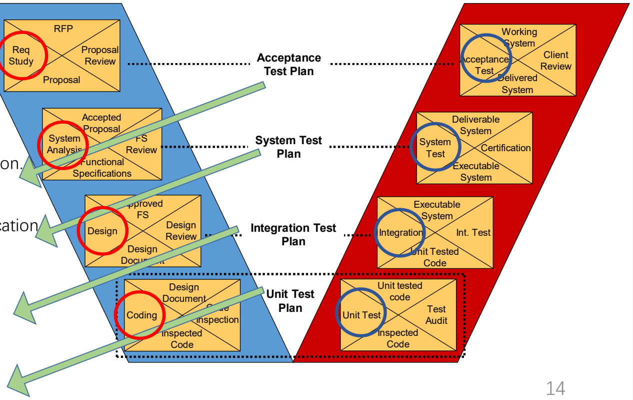
V-Model Documentation

✓ the User Requirements Specification.
✓ (URS)

✓ the System Requirements Specification
✓ (SS)

✓ the System Design Specifications,
✓ (SDS)

✓ Detailed Design Specifications,
✓ (DDS)



V-Model Documentation

- 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.

V-Model Documentation

- 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.

V-model Advantages

- 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.

V-model Disadvantages

- 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

Scrum:敏捷开发的一个增量

Scrum用到的工具

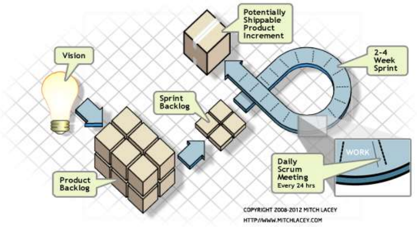
1.用户故事。迭代计划会议用到，Product Owner以用户的角度去描述需求。如，作为一个学员，我希望能在做完一份试卷后，系统能针对我的薄弱点提供相应的指导及练习。

2.Product Backlog。迭代计划会议用到，Product Owner事先将所有的用户故事按优先级排好，放到一个列表内，这个列表就是Product Backlog。

3.Sprint Backlog。迭代计划会议用到，整个开发小组通过估点将用户故事按优先级移入到迭代计划内，迭代计划中待完成的用户故事列表即为Sprint Backlog。

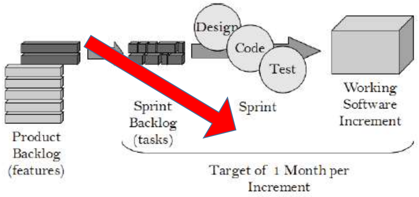
4.估点。主要用于评估用户故事的大致工作量。下一篇文章会额外介绍估点。

5.燃尽图。主要用于迭代进度的管控。下一篇文章会额外介绍燃尽图。


Vision
Potentially Shippable Product Increment
2-4 Week Sprint
Sprint Backlog
Product Backlog
WORK
Daily Scrum Meeting Every 24 hrs

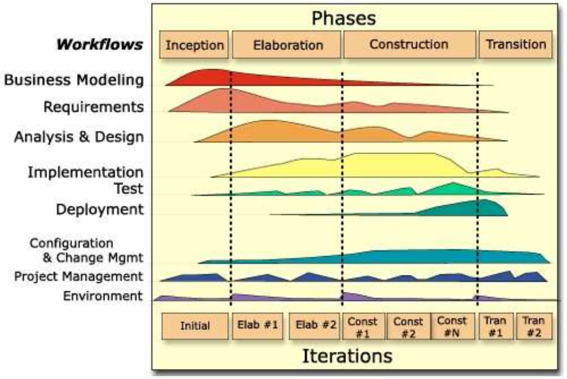
COPYRIGHT 2008-2012 MITCH LACEY
HTTP://WWW.MITCHLACEY.COM

- Backlog
- Sprint
- Product increment


Design
Code
Test
Product Backlog (features)
Sprint Backlog (tasks)
Sprint
Working Software Increment
Target of 1 Month per Increment

- The QA’s job on the business analysis task are: 1) help business analyst to make the requirement document and test the requirement from the logic aspect; 2) do the team working, knowledge sharing, training, and communication to keep the team’s understanding of the requirement is on the same line.
- The QA’s job on the release candidate submission are do the test analysis and finish the report in order to answer the question whether or not the increment of this sprint is qualified to be released to the product environment and supply service to the end user.

4，统一过程UP

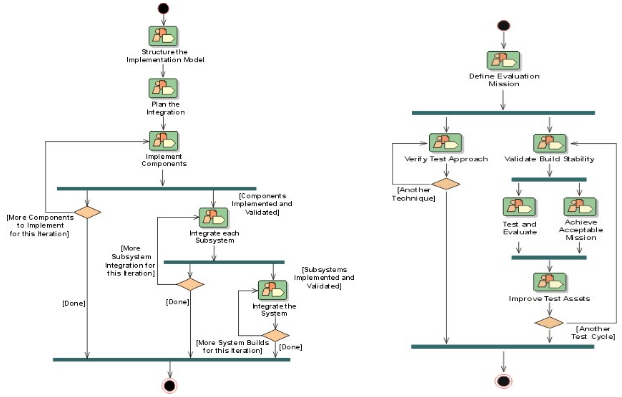


21

- 统一过程就是在软件生命周期过程中以用例为驱动、构架为中心来进行一次一次的增量式的迭代，每次迭代都是以上一次迭代为基础并生成包括构件的源代码体、需求说明、测试用例等的制品。每次的迭代又具体分为四个阶段：初始、细化、提交和转移，而在每个阶段又分为多个工作流：需求、分析、设计、实现和测试等。统一过程模型是基于面向对象方法和UML统一建模语言的，用这种方法论来指导软件开发主要可以解决两个问题：
- 软件复用问题
- 需求变化问题。
- **统一过程是用例驱动的**
- **统一过程是以构架为中心的**

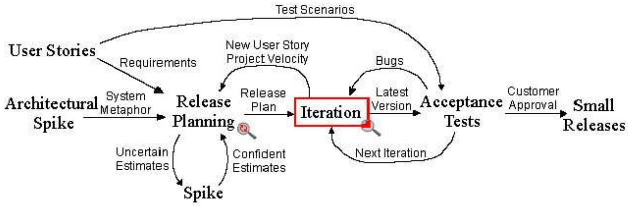
22

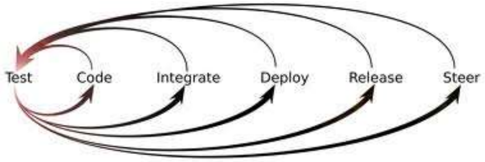
统一过程中的测试流程


Structure the Implementation Model
Plan the Integration
Implement Components
[More Components to Implement for this Iteration]
[Components Implemented and Validated]
Integrate each Subsystem
[More Subsystem Integration for this Iteration]
[Done]
[Done]
Integrate the System
[Subsystems Implemented and Validated]
[More System Builds for this Iteration]
[Done]
Define Evaluation Mission
Verify Test Approach
Validate Build Stability
[Another Technique]
Test and Evaluate
Achieve Acceptable Mission
Improve Test Assets
[Another Test Cycle]

23

5.2, XP & TDD





24

归纳一下，什么是流程

- 流程是一种运行方式
- 流程是基本概念的一簇定义
- 流程包括了：
  - 基本任务
  - 基本任务的先后次序
  - 角色与分工
  - 产出物

产出物

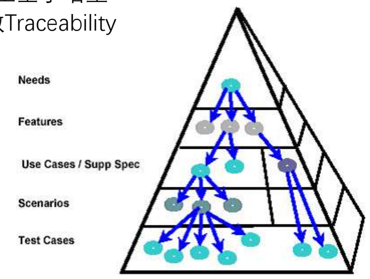
- 测试计划
  - Schedule
  - Source （Team）
  - Scope
- 测试用例
  - 测什么
  - 怎么测
- 测试报告
  - 我们处于什么位置
  - 我们该怎么办

产出物需要澄清一下

- 测试用例和测试计划的区别
- 先有测试用例还是先有测试计划
- 测试报告的级别和重要性

产出物的内在逻辑1：Traceability

- 文档化是附加产出
  - 也可能是最重要的产出
- 文档化的逻辑结构呈金字塔型
  - 其内在逻辑叫做Traceability



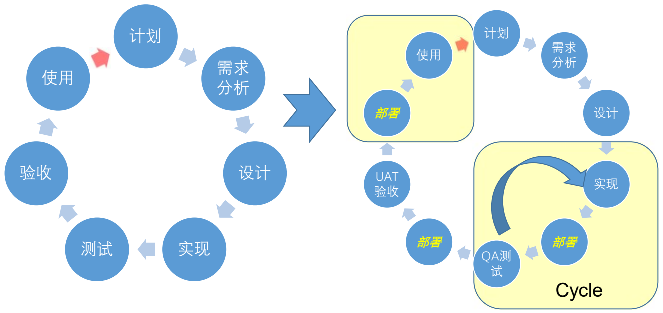
- Traceability is a technique that provides a relationship between different levels of requirements in the system. This technique helps you determine the origin of any requirement. The pyramid illustrates how requirements are traced from the top level down. Every need usually maps to a couple of features, and then features map to use cases and supplementary requirements. Use cases describe functional requirements, and supplementary specifications describe non-functional items. In addition, every use case maps to many scenarios. Mapping use cases to scenarios, then, is a one to many relationship. Scenarios map to test cases also in a one to many relationship. Between needs and features, on the other hand, there is many to many mapping.

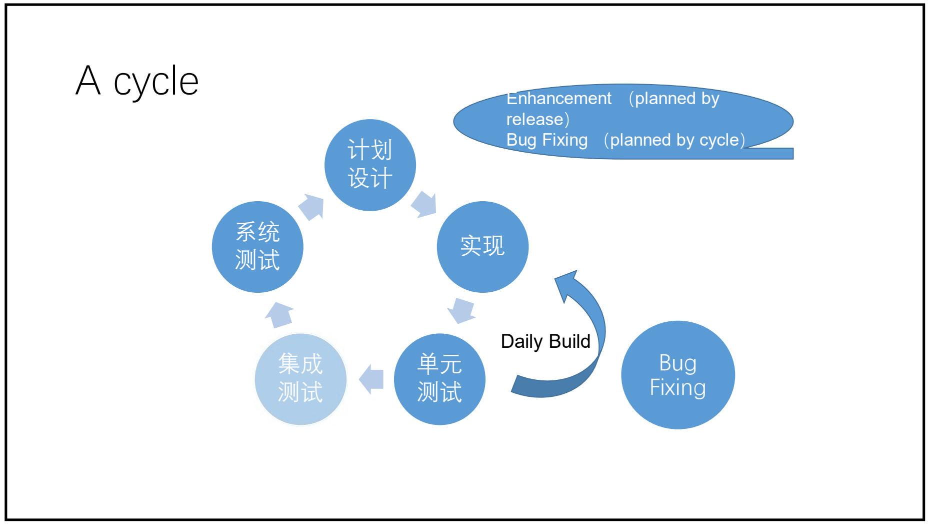
产出物的内在逻辑2： Coverage

	UC1				UC2		UC3		
TC	MF	AF1	AF2	AF3	MF	AF	MF	AF1	AF2
TC1.1	●				●		●		
TC1.2	●				●		●		
TC1.3		●			●		●		
TC1.4		●			●			●	
TC1.5		●			●				●
TC1.6			●		●				●
TC1.7			●		●				
TC2.1			●						
TC2.2				●					
TC2.3				●		●			
TC2.4				●		●			

Test coverage is defined as a technique which determines whether and how our test cases are actually covering the application original requirement, and how much requirement is exercised when we run those test cases. Frequently the coverage is measured by percentage.

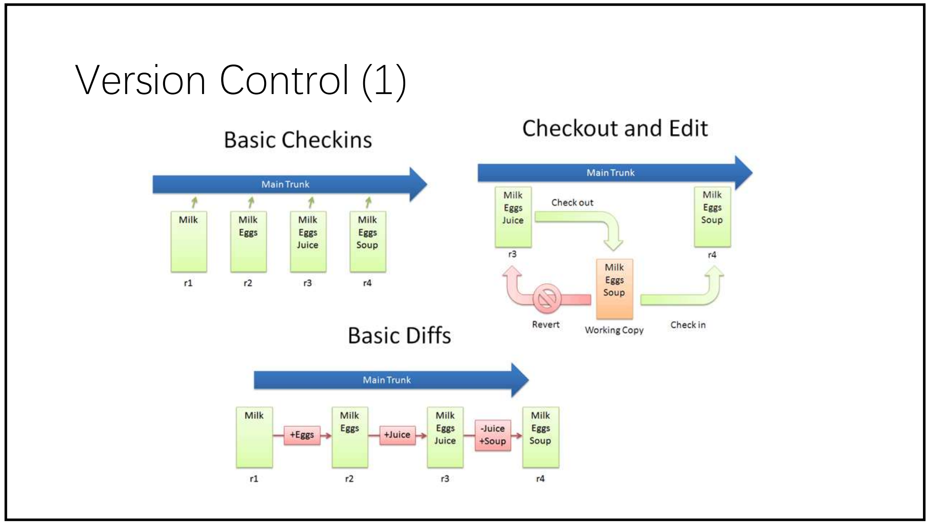
A release



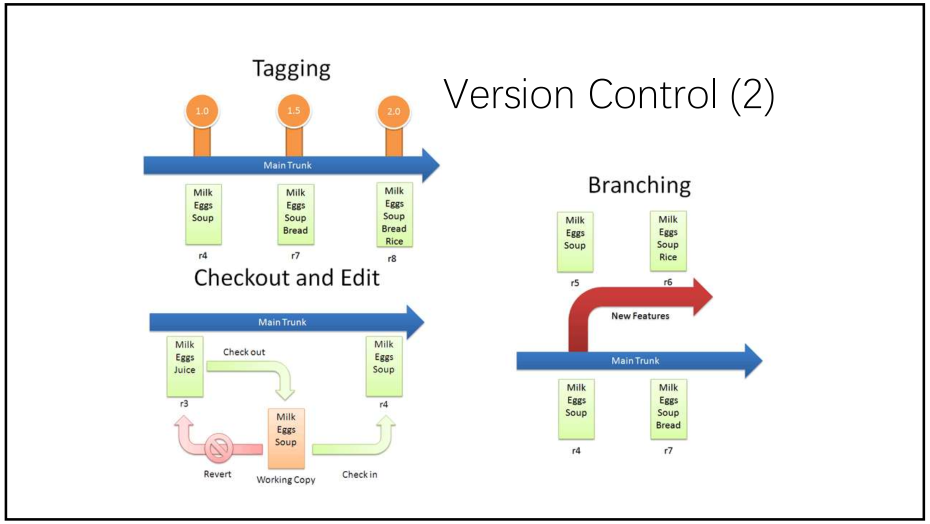


A cycle

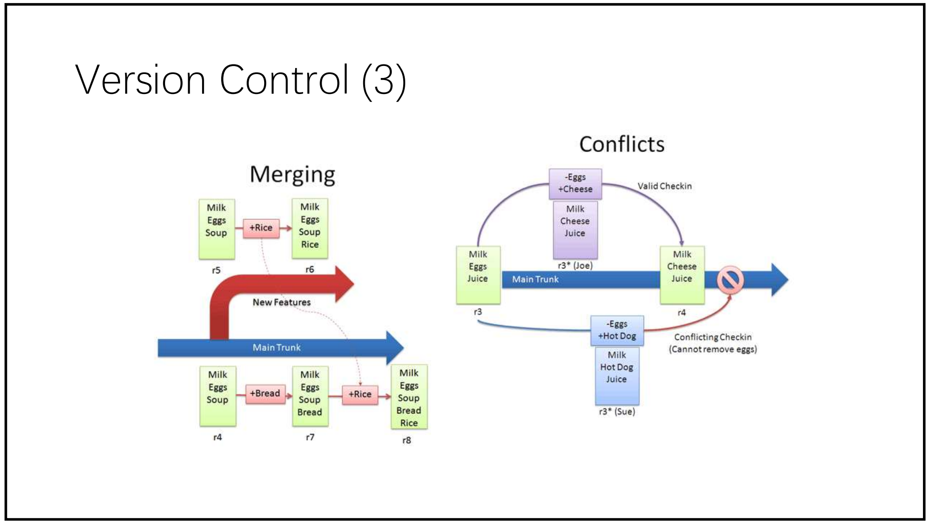
Enhancement （planned by release）
Bug Fixing （planned by cycle）



Version Control (1)



Version Control (2)



Version Control (3)

测试数据

- 功能测试不需要大量的数据
- 功能测试需要数据的覆盖率高
- 功能测试的测试数据要求尽量真实
- 性能测试需要大量的数据
- 性能测试的测试数据应尽可能的达到符合实际的数据分配

IEEE Std. 829-1998
测试环境与测试阶段的关系

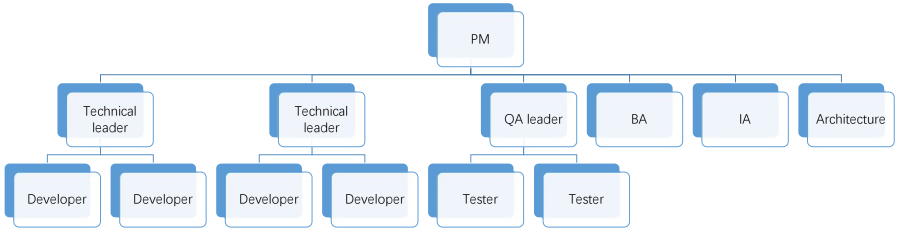
不同阶段的测试环境

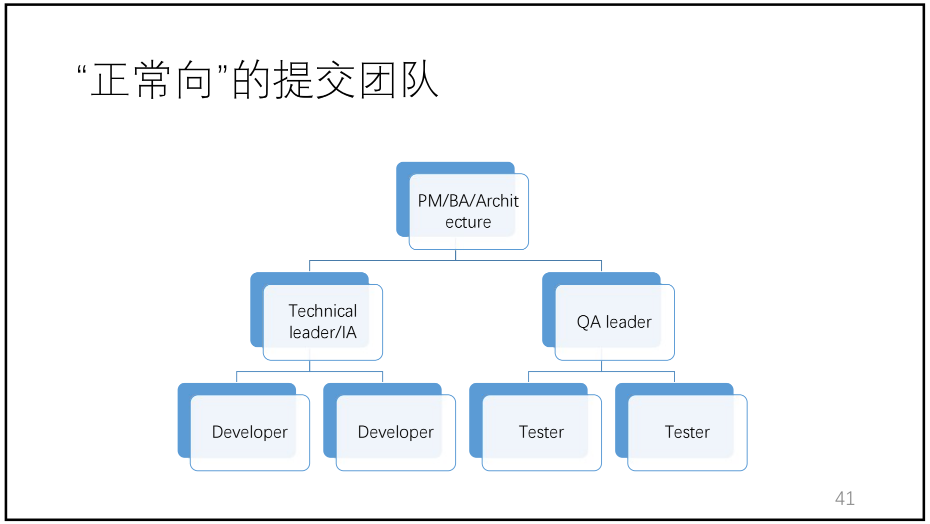
Attribute	Level			
	Unit	Integration	System	Acceptance
People	Developers	Developers & Testers	Testers	Testers & Users
Hardware O/S	Programmers' Workbench	Programmers' Workbench	System Test Machine or Region	Mirror of Production
Cohabiting Software	None	None	None/Actual	Actual
Interfaces	None	Internal	Simulated & Real	Production
Source of Test Data	Manually Created	Manually Created	Production & Manually Created	Production
Volume of Test Data	Small	Small	Large	Large
Strategy	Unit	Groups of Units/Builds	Entire System	Simulated Production

计划是什么

- 计划是一种资源组织方式
- 计划是从一个状态，经过一系列步骤，到达一个目标
  - 设定范围 Scope
    - 设定目标
    - 设定条件
  - 设定路径 Schedule
  - 组织资源 reSource

全功能提交团队





“正常向”的提交团队

PM/BA/Architecture

Technical leader/IA

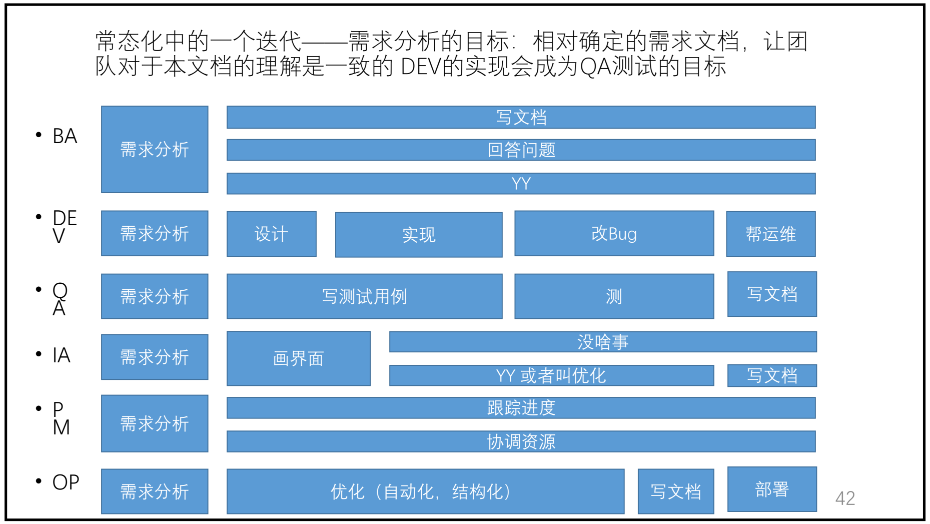
QA leader

Developer

Developer

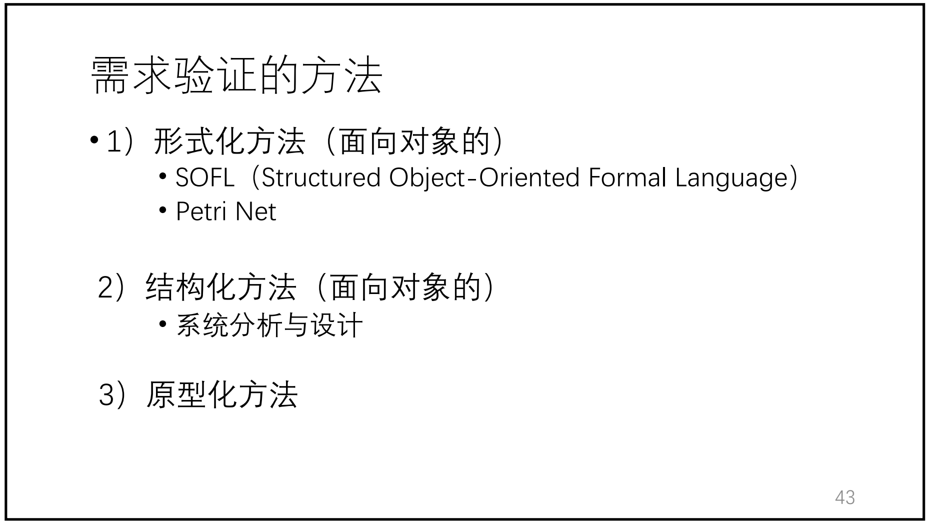
Tester

Tester



常态化中的一个迭代——需求分析的目标：相对确定的需求文档，让团队对于本文档的理解是一致的 DEV的实现会成为QA测试的目标

BA	需求分析	写文档 回答问题 YY
DEV	需求分析	设计 / 实现 / 改Bug / 帮运维
QA	需求分析	写测试用例 / 测 / 写文档
IA	需求分析	画界面 / 没啥事 YY 或者叫优化 / 写文档
PM	需求分析	跟踪进度 协调资源
OP	需求分析	优化（自动化，结构化） / 写文档 / 部署

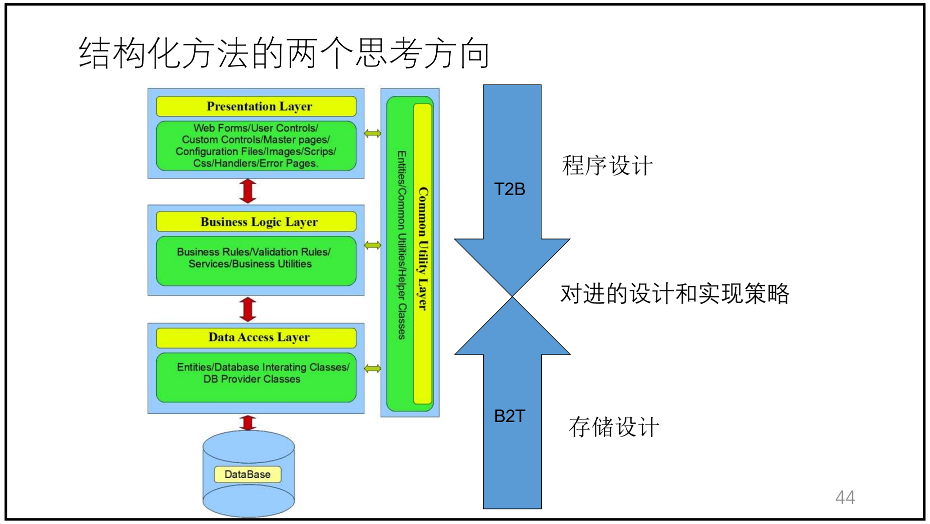


需求验证的方法

- 1）形式化方法（面向对象的）
  - SOFL（Structured Object-Oriented Formal Language）
  - Petri Net

2）结构化方法（面向对象的）
  - 系统分析与设计

3）原型化方法



结构化方法的两个思考方向

Presentation Layer

Web Forms/User Controls/
Custom Controls/Master pages/
Configuration Files/Images/Scrips/
Css/Handlers/Error Pages.

Business Logic Layer

Business Rules/Validation Rules/
Services/Business Utilities

Data Access Layer

Entities/Database Interating Classes/
DB Provider Classes

DataBase

Common Utility Layer

Entities/Common Utilities/Helper Classes

T2B

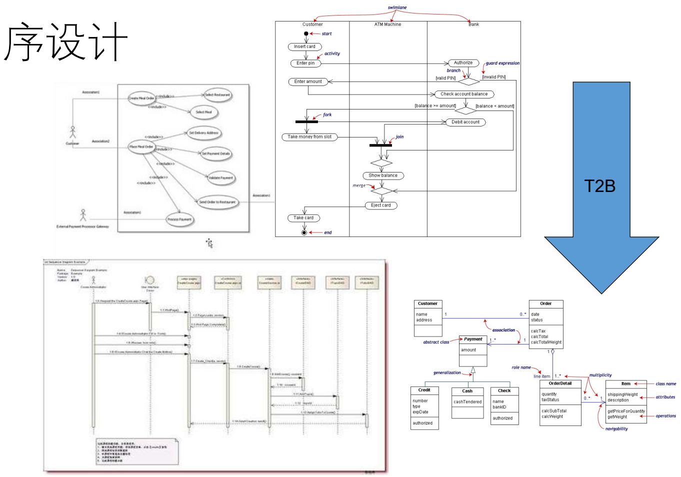
B2T

程序设计

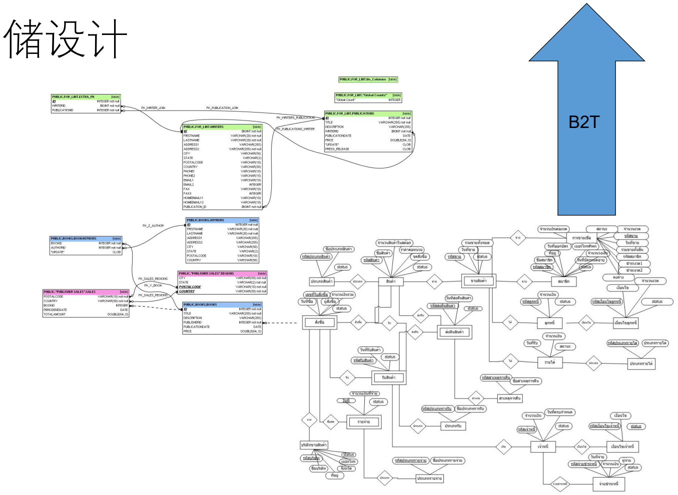
对进的设计和实现策略

存储设计

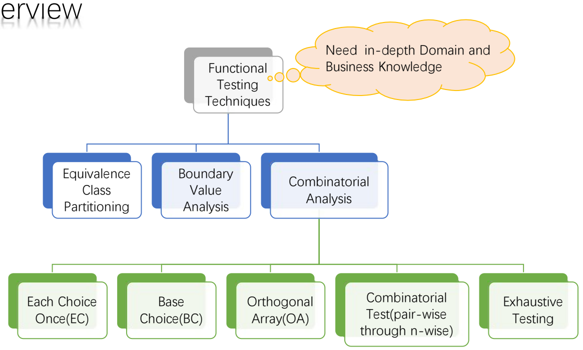
程序设计



存储设计



Overview


Functional Testing Techniques
Need in-depth Domain and Business Knowledge
Equivalence Class Partitioning
Boundary Value Analysis
Combinatorial Analysis
Each Choice Once(EC)
Base Choice(BC)
Orthogonal Array(OA)
Combinatorial Test(pair-wise through n-wise)
Exhaustive Testing

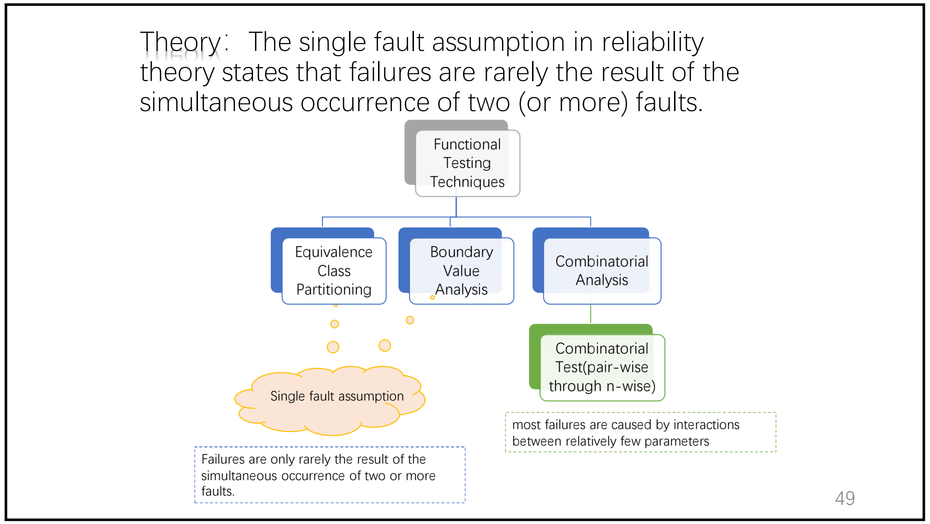
The value or effectiveness of using any test technique ultimately depends on the individual test's system and domain knowledge and ability to apply the applicable technique in the correct situation.

Exhaustively testing all possible inputs to any nontrivial software component is generally impossible due to the enormous number of variations. One approach to create a test suite with high coverage and a low number of variations is known as combinatorial testing. One common strategy, known as pairwise, tests a set of variations where every possible pair of parameters appears at least once. This method can be extended to use higher orders of combinations (3-wise, 4-wise, etc) for increased coverage at the expense of a larger test suite.

Software test techniques exist to reduce the number of tests to be run whilst still providing sufficient coverage of the system under test

There is no single technique or approach that is completely effective in software testing, so by increasing the diversity of methods used in testing and considering different perspectives, we are more likely to be successful in both exposing potential issues as well as increasing the effectiveness of our testing.

The value or effectiveness of using any test technique ultimately depends on the individual test's system and domain knowledge and ability to apply the applicable technique in the correct situation.

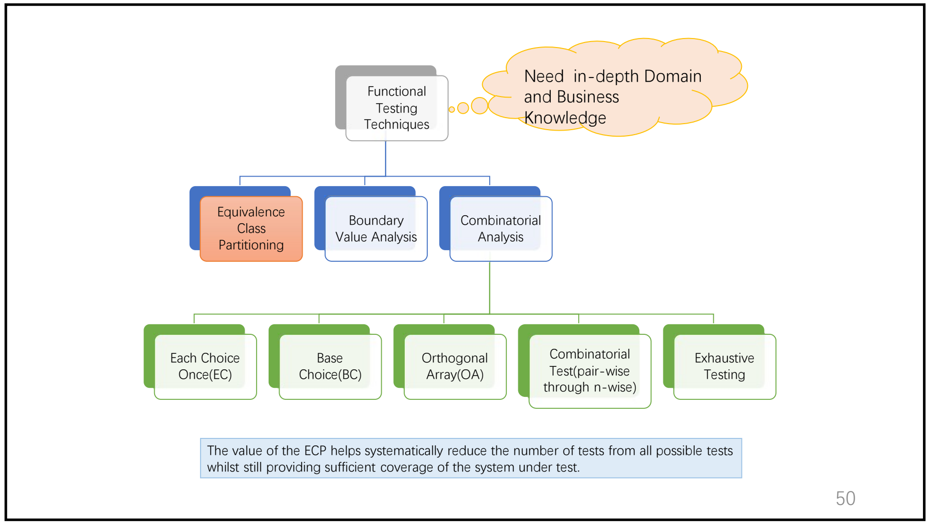


Theory: The single fault assumption in reliability theory states that failures are rarely the result of the simultaneous occurrence of two (or more) faults.

Functional Testing Techniques
Equivalence Class Partitioning
Boundary Value Analysis
Combinatorial Analysis
Combinatorial Test(pair-wise through n-wise)
Single fault assumption

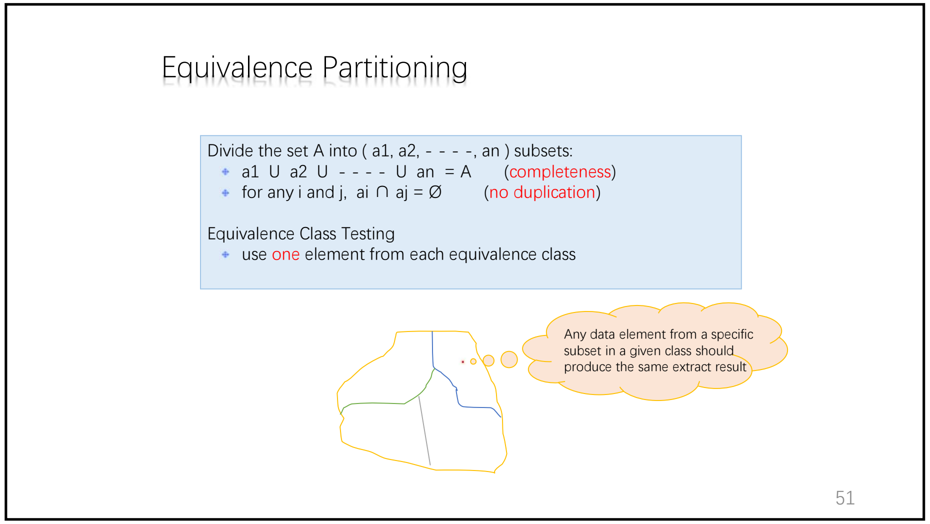
Failures are only rarely the result of the simultaneous occurrence of two or more faults.

most failures are caused by interactions between relatively few parameters



Functional Testing Techniques
Need in-depth Domain and Business Knowledge
Equivalence Class Partitioning
Boundary Value Analysis
Combinatorial Analysis
Each Choice Once(EC)
Base Choice(BC)
Orthogonal Array(OA)
Combinatorial Test(pair-wise through n-wise)
Exhaustive Testing

The value of the ECP helps systematically reduce the number of tests from all possible tests whilst still providing sufficient coverage of the system under test.

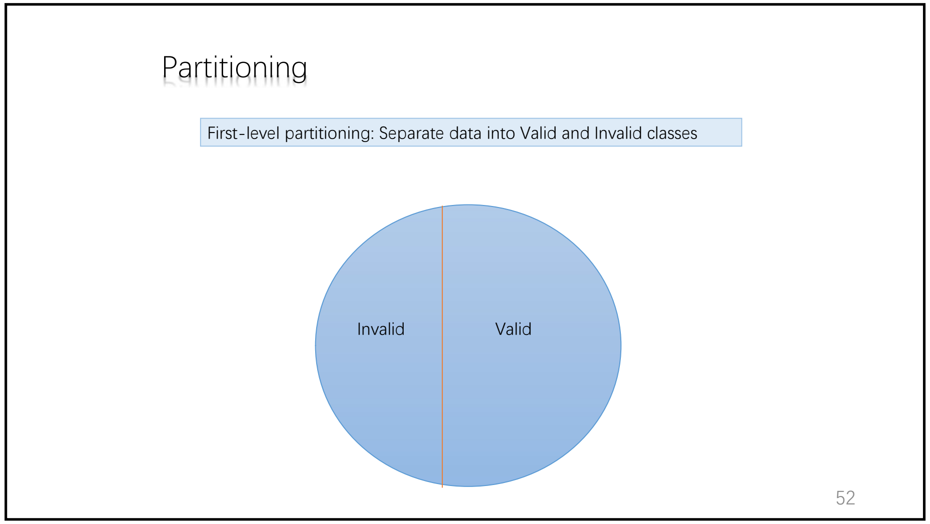


Equivalence Partitioning

Divide the set A into (a1, a2, - - - -, an) subsets:
- a1 U a2 U - - - - U an = A (completeness)
- for any i and j, ai ∩ aj = Ø (no duplication)

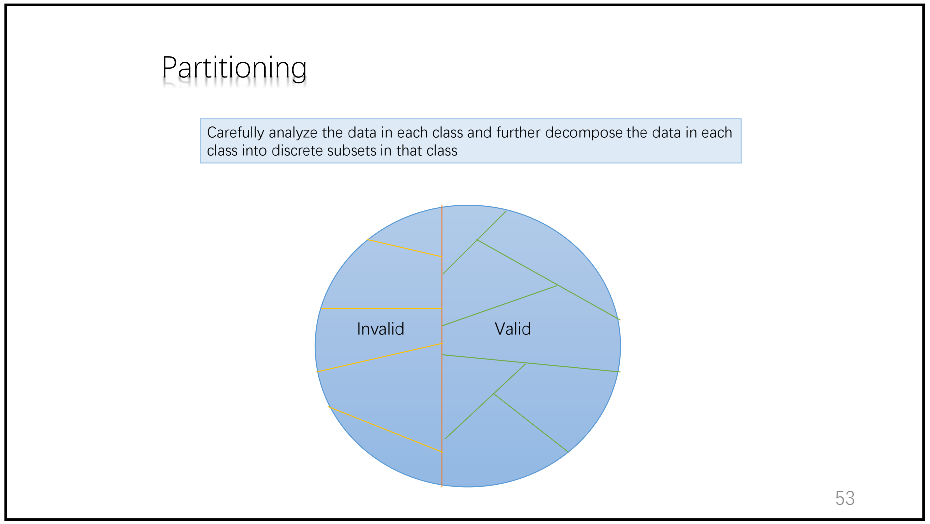
Equivalence Class Testing
- use one element from each equivalence class

Any data element from a specific subset in a given class should produce the same extract result



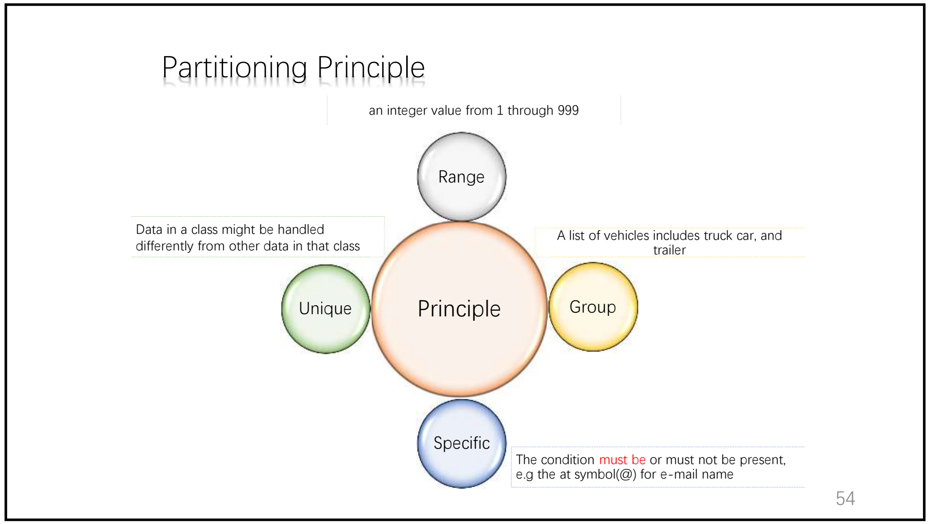
Partitioning

First-level partitioning: Separate data into Valid and Invalid classes



Partitioning

Carefully analyze the data in each class and further decompose the data in each class into discrete subsets in that class



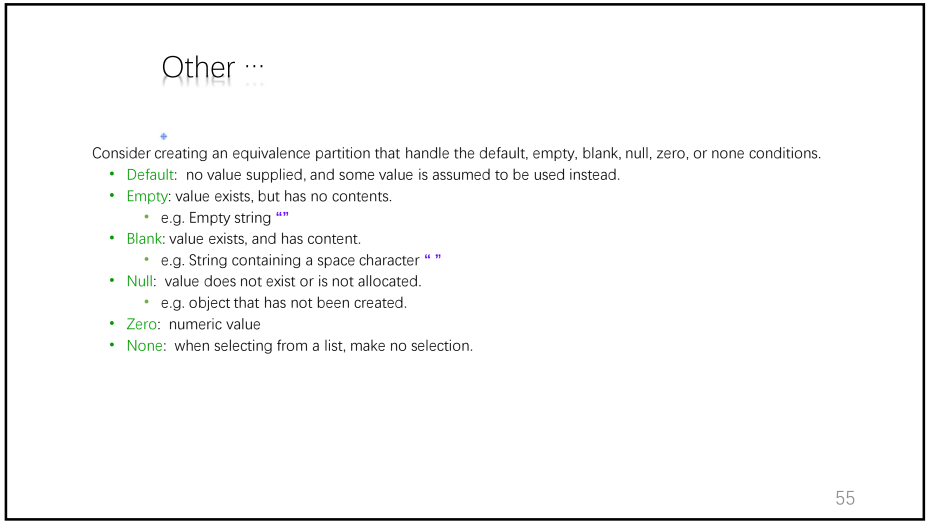
Partitioning Principle

an integer value from 1 through 999

Data in a class might be handled differently from other data in that class

A list of vehicles includes truck car, and trailer

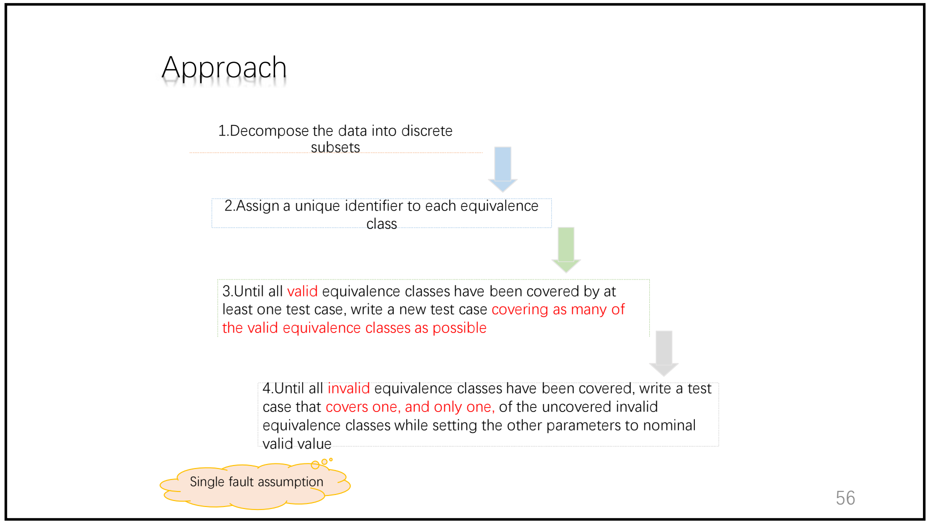
The condition must be or must not be present, e.g the at symbol(@) for e-mail name



Other …

Consider creating an equivalence partition that handle the default, empty, blank, null, zero, or none conditions.

- Default: no value supplied, and some value is assumed to be used instead.
- Empty: value exists, but has no contents.
  - e.g. Empty string “”
- Blank: value exists, and has content.
  - e.g. String containing a space character “ ”
- Null: value does not exist or is not allocated.
  - e.g. object that has not been created.
- Zero: numeric value
- None: when selecting from a list, make no selection.



Approach

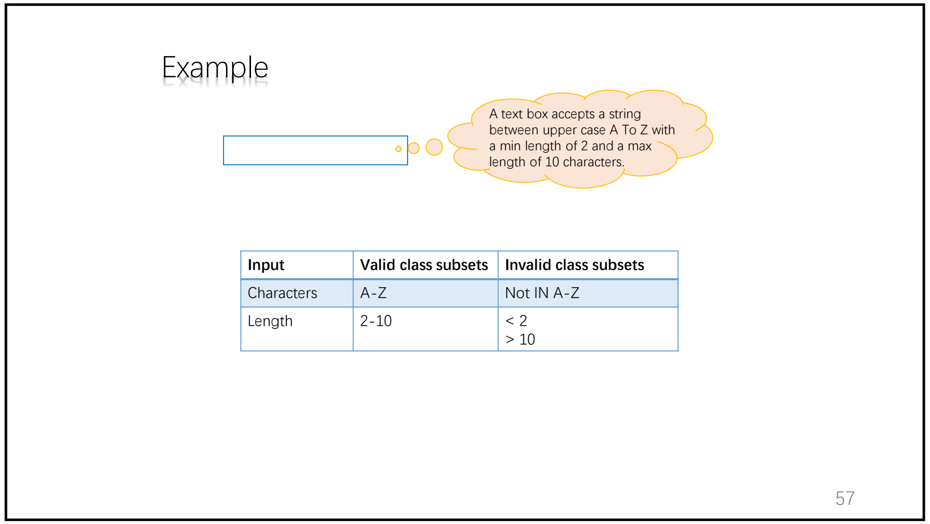
1.Decompose the data into discrete subsets

2.Assign a unique identifier to each equivalence class

3.Until all valid equivalence classes have been covered by at least one test case, write a new test case covering as many of the valid equivalence classes as possible

4.Until all invalid equivalence classes have been covered, write a test case that covers one, and only one, of the uncovered invalid equivalence classes while setting the other parameters to nominal valid value

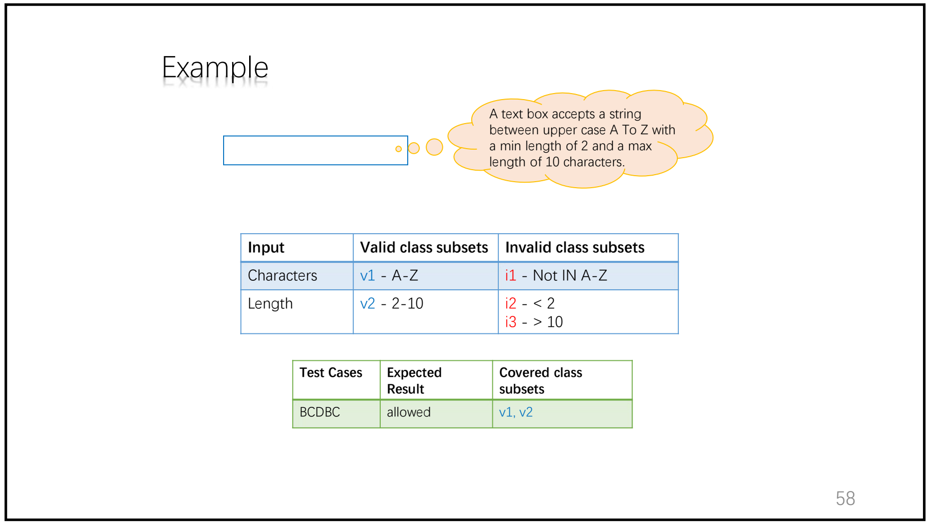
Single fault assumption



Example

A text box accepts a string between upper case A To Z with a min length of 2 and a max length of 10 characters.

Input	Valid class subsets	Invalid class subsets
Characters	A-Z	Not IN A-Z
Length	2-10	< 2 > 10

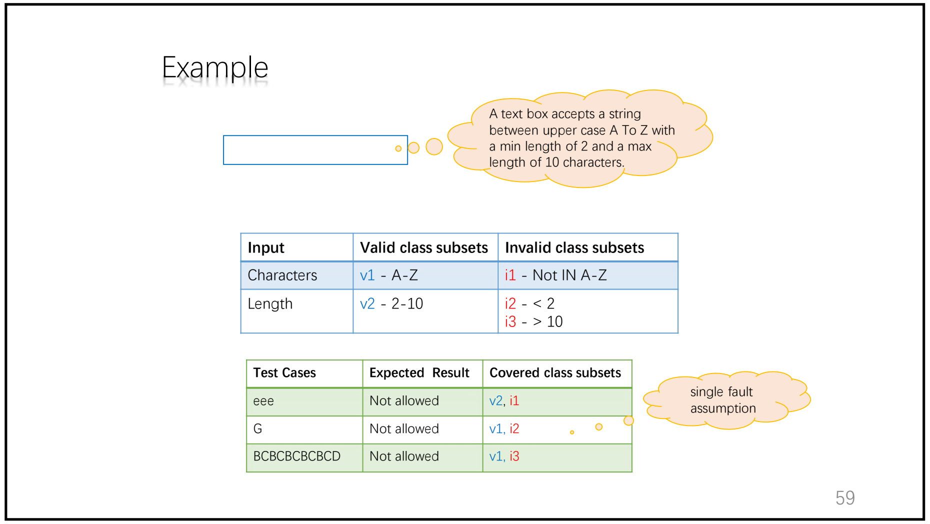


Example

A text box accepts a string between upper case A To Z with a min length of 2 and a max length of 10 characters.

Input	Valid class subsets	Invalid class subsets
Characters	v1 - A-Z	i1 - Not IN A-Z
Length	v2 - 2-10	i2 - < 2 i3 - > 10

Test Cases	Expected Result	Covered class subsets
BCDBC	allowed	v1, v2



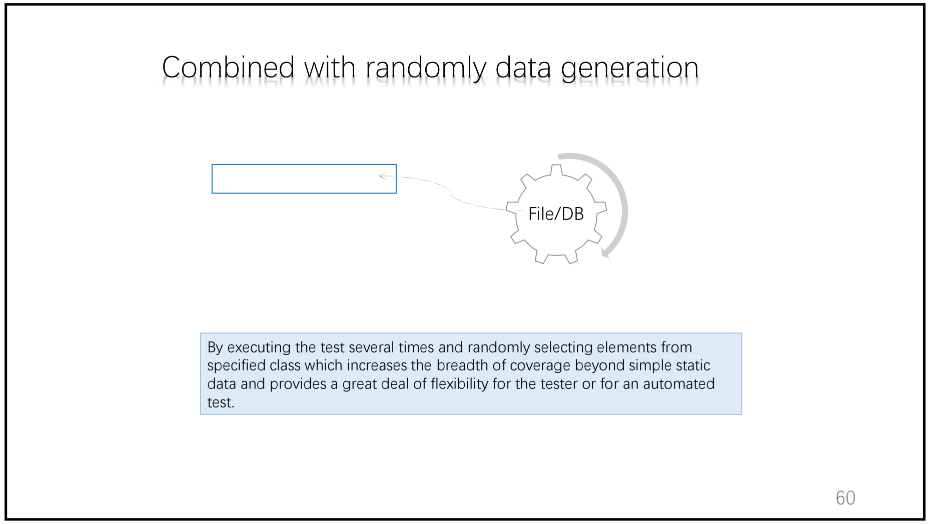
Example

A text box accepts a string between upper case A To Z with a min length of 2 and a max length of 10 characters.

Input	Valid class subsets	Invalid class subsets
Characters	v1 - A-Z	i1 - Not IN A-Z
Length	v2 - 2-10	i2 - < 2 i3 - > 10

Test Cases	Expected Result	Covered class subsets
eee	Not allowed	v2, i1
G	Not allowed	v1, i2
BCBCBCBCBCD	Not allowed	v1, i3

single fault assumption



Combined with randomly data generation

By executing the test several times and randomly selecting elements from specified class which increases the breadth of coverage beyond simple static data and provides a great deal of flexibility for the tester or for an automated test.

Case Study - Triangle

Specification

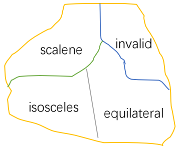
We have a triangle three sides A, B, C.

If they can form a triangle three sides must be met:

A>0, B>0, C>0, A+B>C, B+C>A, A+C>B.

1) If it is isosceles, also need to judge whether A = B, or B = C, or A = C.

2) If it is equilateral, also need to judge whether A = B, and B = C, and A = C.



Input condition	Valid class subsets	Invalid class subsets
scalene	v1 (A>0) v2 (B>0) v3 (C>0) v4 (A+B>C) v5 (B+C>A) v6 (A+C>B)	i1 (A⩽0) i2 (B⩽0) i3 (C⩽0) i4 (A+B⩽C) i5 (B+C⩽A) i6 (A+C⩽B)
isosceles	v7 (A = B and A ≠ C) v8 (B = C and A ≠ B)	i7 (A ≠ B and A ≠ C and B ≠ C)
equilateral	v10 (A = C and A = B and B = C) v9 (A = C and A ≠ B)	i8 (A ≠ B) i9 (B ≠ C) i10 (A ≠ C)

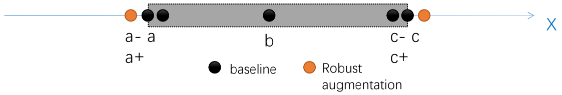
ID	[A, B, C]	Covered class subsets	Output
1	[3， 4， 5]	v1, v2, v3, v4, v5, v6	scalene
2	[3， 3， 4]	v1, v2, v3, v4, v5, v6, v7	isosceles
3	[3， 4， 4]	v1, v2, v3, v4, v5, v6, v8	isosceles
4	[3， 4， 3]	v1, v2, v3, v4, v5, v6, v9	isosceles
5	[3， 3， 3]	v1, v2, v3, v4, v5, v6, v10	equilateral
6	[0， 1， 2]	i1	invalid
7	[1， 0， 2]	i2	invalid
8	[1， 2， 0]	i3	invalid
9	[1， 2， 3]	i4	invalid
10	[1， 3， 2]	i5	invalid
11	[3， 1， 2]	i6	invalid
12	[3， 4， 6]	v1, v2, v3, v4, v5, v6, i7	scalene
13	[3， 4， 4]	v1, v2, v3, v4, v5, v6, i8	isosceles
14	[3， 4， 3]	v1, v2, v3, v4, v5, v6, i9	isosceles
15	[3， 3， 4]	v1, v2, v3, v4, v5, v6, i10	isosceles

Boundary Value Analysis

- Boundary value analysis focuses on the boundaries of the domain.

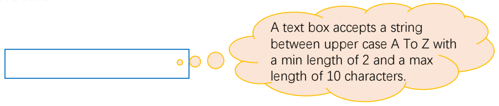
- Rationale: Historical evidence demonstrates that errors occur most frequently on or near the boundary
  a) operator <= or <
  b) For-loops, While loops and Repeat loops
  c) The requirements may not be clearly understood

Jorgensen's (6N + 1) formula for robust boundary test



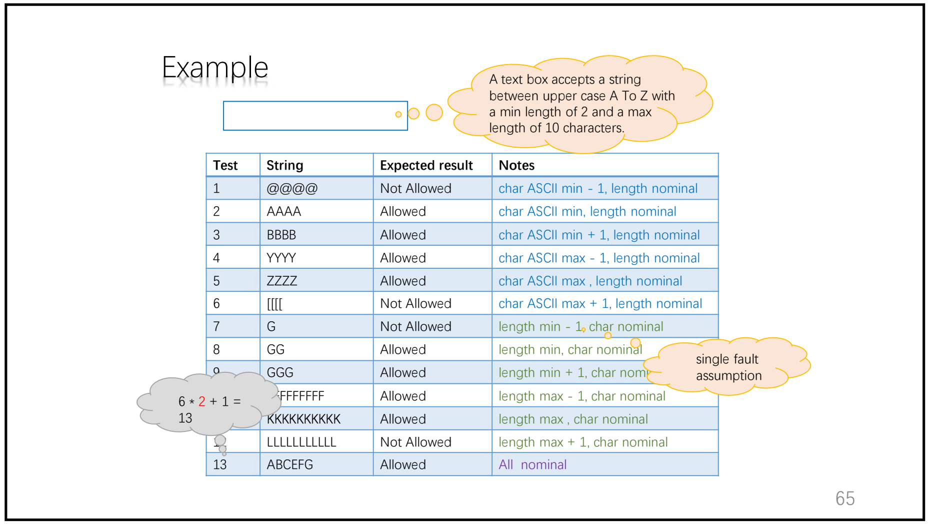
Single-Variable, Single-Range Baseline Test Cases Augmented With Robustness Tests
(shaded area indicates valid values of the variable X)

Example



A text box accepts a string between upper case A To Z with a min length of 2 and a max length of 10 characters.

Char	@	A	B	···	Y	Z	[
ASCII/Unicode	64	65	66	···	89	90	91



Example

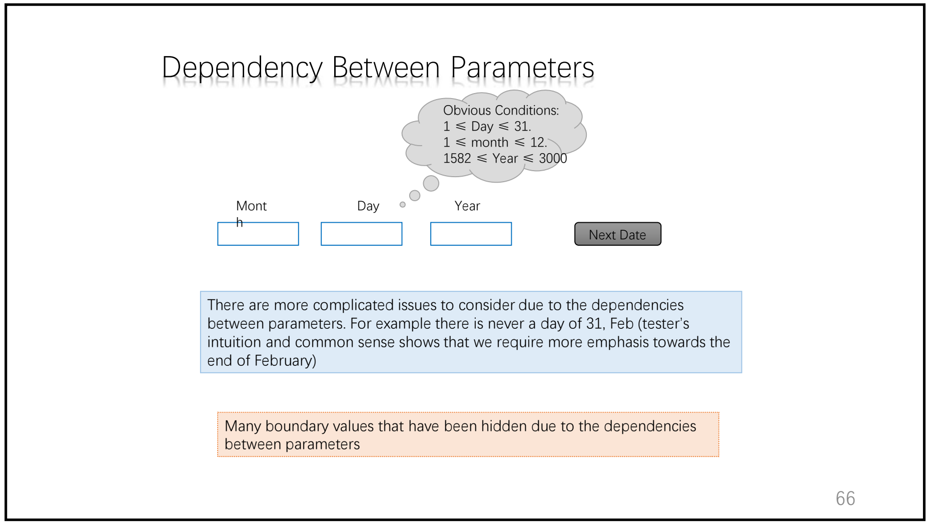
A text box accepts a string between upper case A To Z with a min length of 2 and a max length of 10 characters.

Test	String	Expected result	Notes
1	@@@@	Not Allowed	char ASCII min - 1, length nominal
2	AAAA	Allowed	char ASCII min, length nominal
3	BBBB	Allowed	char ASCII min + 1, length nominal
4	YYYY	Allowed	char ASCII max - 1, length nominal
5	ZZZZ	Allowed	char ASCII max , length nominal
6	[[[[	Not Allowed	char ASCII max + 1, length nominal
7	G	Not Allowed	length min - 1, char nominal
8	GG	Allowed	length min, char nominal
9	GGG	Allowed	length min + 1, char nomi
	FFFFFFFF	Allowed	length max - 1, char nominal
	KKKKKKKKKK	Allowed	length max , char nominal
1	LLLLLLLLLLL	Not Allowed	length max + 1, char nominal
13	ABCEFG	Allowed	All nominal

single fault assumption

6 * 2 + 1 = 13

65



Dependency Between Parameters

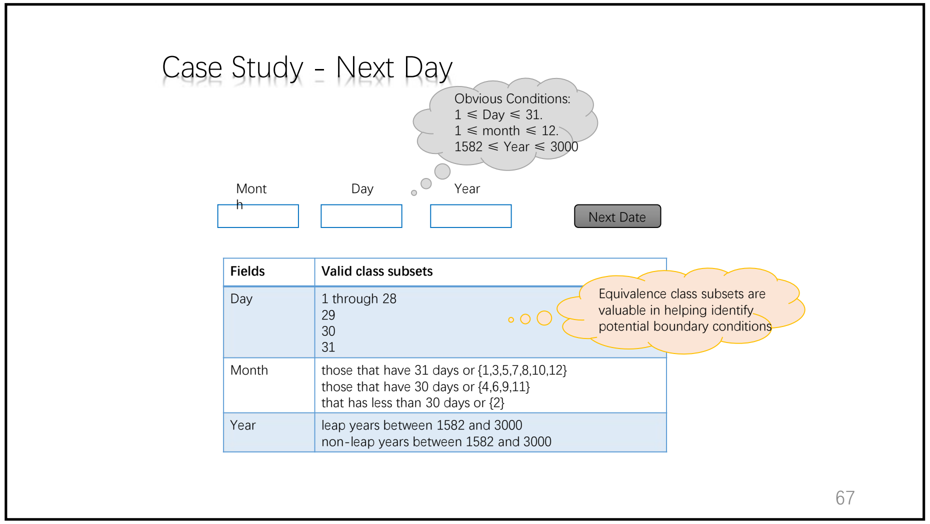
Obvious Conditions:
1 ≤ Day ≤ 31.
1 ≤ month ≤ 12.
1582 ≤ Year ≤ 3000

Month
Day
Year
Next Date

There are more complicated issues to consider due to the dependencies between parameters. For example there is never a day of 31, Feb (tester's intuition and common sense shows that we require more emphasis towards the end of February)

Many boundary values that have been hidden due to the dependencies between parameters

66



Case Study - Next Day

Obvious Conditions:
1 ≤ Day ≤ 31.
1 ≤ month ≤ 12.
1582 ≤ Year ≤ 3000

Month
Day
Year
Next Date

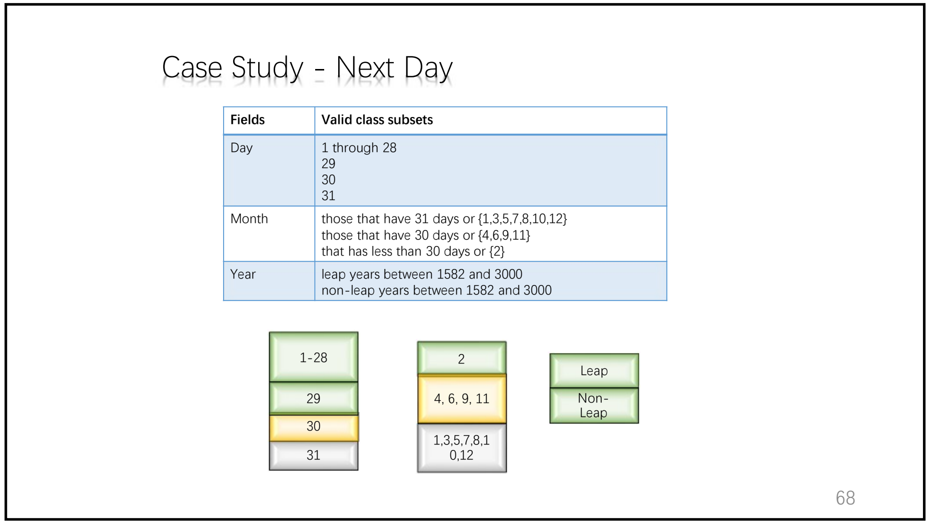
Fields	Valid class subsets
Day	1 through 28 29 30 31

Month	those that have 31 days or {1,3,5,7,8,10,12}
	those that have 30 days or {4,6,9,11}
	that has less than 30 days or {2}

Year	leap years between 1582 and 3000 non-leap years between 1582 and 3000

Equivalence class subsets are valuable in helping identify potential boundary conditions

67



Case Study - Next Day

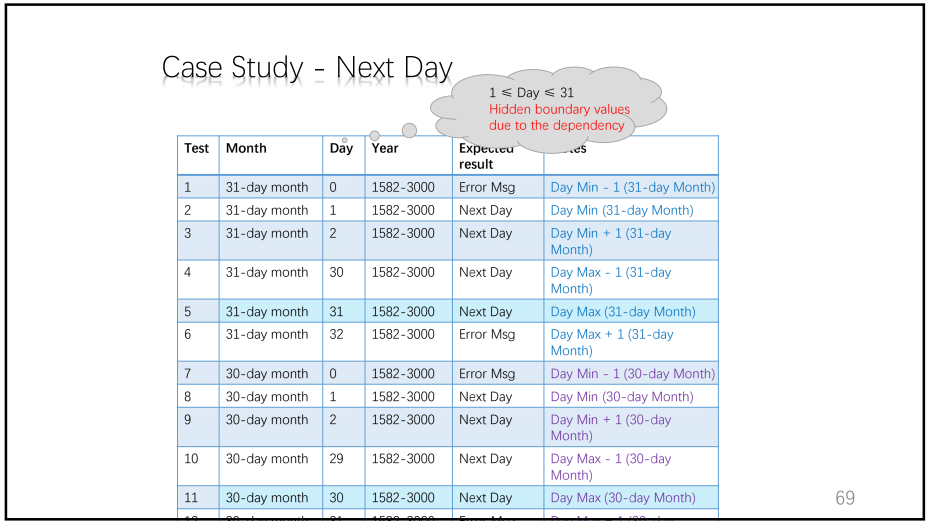
Fields	Valid class subsets
Day	1 through 28 29 30 31

Month	those that have 31 days or {1,3,5,7,8,10,12}
	those that have 30 days or {4,6,9,11}
	that has less than 30 days or {2}

Year	leap years between 1582 and 3000 non-leap years between 1582 and 3000

1-28
29
30
31
2
4, 6, 9, 11
1,3,5,7,8,10,12
Leap
Non-Leap

68

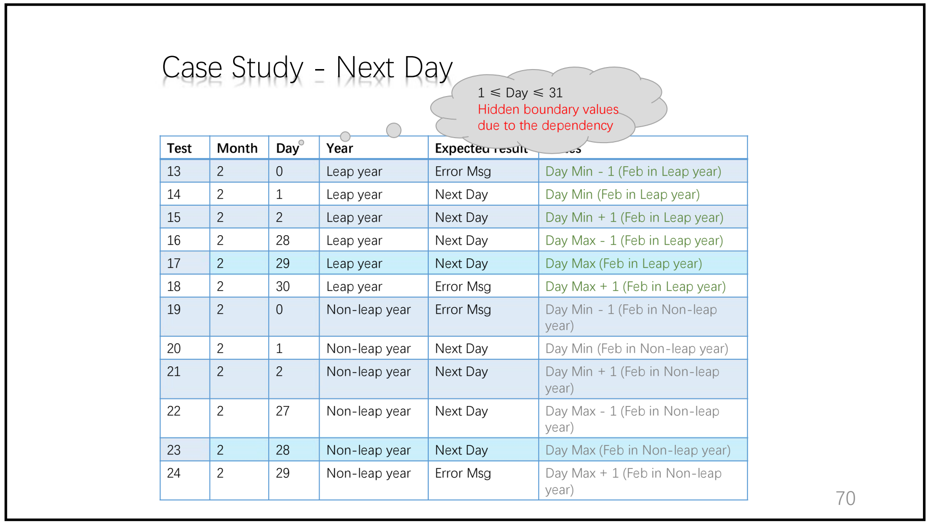


Case Study - Next Day

1 ≤ Day ≤ 31
Hidden boundary values due to the dependency

Test	Month	Day	Year	Expected result	Notes
1	31-day month	0	1582-3000	Error Msg	Day Min - 1 (31-day Month)
2	31-day month	1	1582-3000	Next Day	Day Min (31-day Month)
3	31-day month	2	1582-3000	Next Day	Day Min + 1 (31-day Month)
4	31-day month	30	1582-3000	Next Day	Day Max - 1 (31-day Month)
5	31-day month	31	1582-3000	Next Day	Day Max (31-day Month)
6	31-day month	32	1582-3000	Error Msg	Day Max + 1 (31-day Month)
7	30-day month	0	1582-3000	Error Msg	Day Min - 1 (30-day Month)
8	30-day month	1	1582-3000	Next Day	Day Min (30-day Month)
9	30-day month	2	1582-3000	Next Day	Day Min + 1 (30-day Month)
10	30-day month	29	1582-3000	Next Day	Day Max - 1 (30-day Month)
11	30-day month	30	1582-3000	Next Day	Day Max (30-day Month)
12	30-day month	31	1582-3000	Error Msg	Day Max + 1 (30-day

69

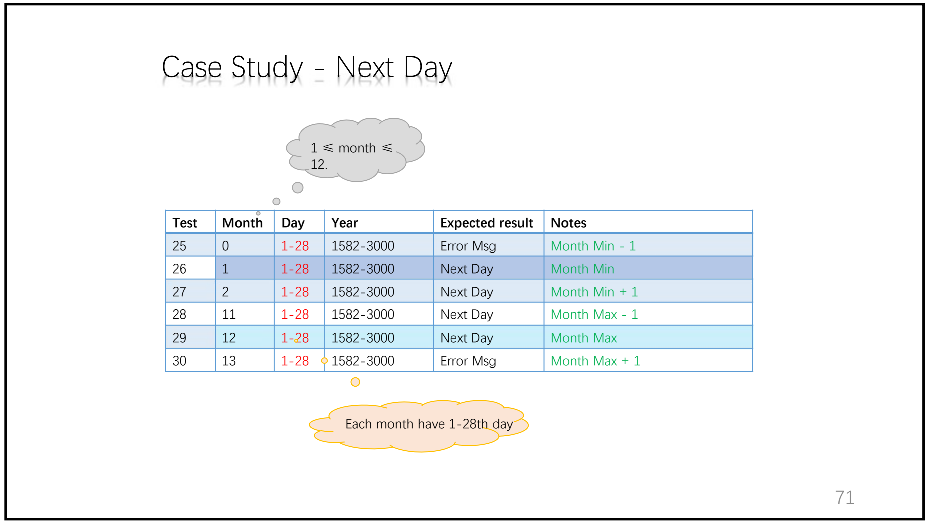


Case Study - Next Day

1 ≤ Day ≤ 31
Hidden boundary values due to the dependency

Test	Month	Day	Year	Expected result	Notes
13	2	0	Leap year	Error Msg	Day Min - 1 (Feb in Leap year)
14	2	1	Leap year	Next Day	Day Min (Feb in Leap year)
15	2	2	Leap year	Next Day	Day Min + 1 (Feb in Leap year)
16	2	28	Leap year	Next Day	Day Max - 1 (Feb in Leap year)
17	2	29	Leap year	Next Day	Day Max (Feb in Leap year)
18	2	30	Leap year	Error Msg	Day Max + 1 (Feb in Leap year)
19	2	0	Non-leap year	Error Msg	Day Min - 1 (Feb in Non-leap year)
20	2	1	Non-leap year	Next Day	Day Min (Feb in Non-leap year)
21	2	2	Non-leap year	Next Day	Day Min + 1 (Feb in Non-leap year)
22	2	27	Non-leap year	Next Day	Day Max - 1 (Feb in Non-leap year)
23	2	28	Non-leap year	Next Day	Day Max (Feb in Non-leap year)
24	2	29	Non-leap year	Error Msg	Day Max + 1 (Feb in Non-leap year)

70



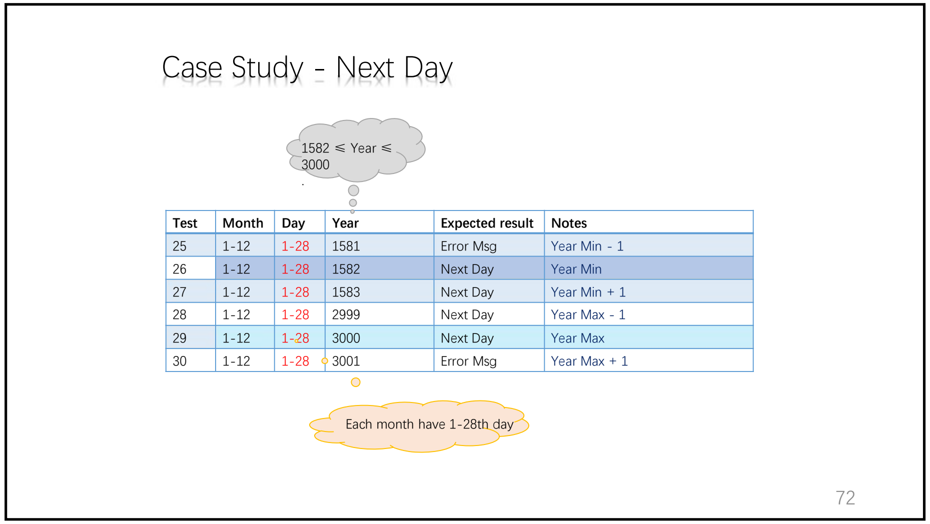
Case Study - Next Day

1 ≤ month ≤ 12.

Test	Month	Day	Year	Expected result	Notes
25	0	1-28	1582-3000	Error Msg	Month Min - 1
26	1	1-28	1582-3000	Next Day	Month Min
27	2	1-28	1582-3000	Next Day	Month Min + 1
28	11	1-28	1582-3000	Next Day	Month Max - 1
29	12	1-28	1582-3000	Next Day	Month Max
30	13	1-28	1582-3000	Error Msg	Month Max + 1

Each month have 1-28th day

71



Case Study - Next Day

1582 ≤ Year ≤ 3000

Test	Month	Day	Year	Expected result	Notes
25	1-12	1-28	1581	Error Msg	Year Min - 1
26	1-12	1-28	1582	Next Day	Year Min
27	1-12	1-28	1583	Next Day	Year Min + 1
28	1-12	1-28	2999	Next Day	Year Max - 1
29	1-12	1-28	3000	Next Day	Year Max
30	1-12	1-28	3001	Error Msg	Year Max + 1

Each month have 1-28th day

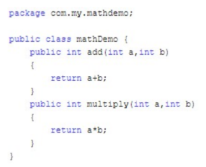
72

Case Study - Next Day

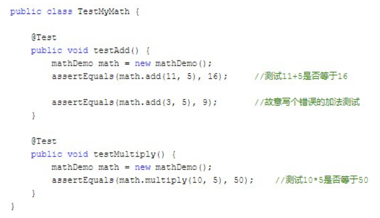
Output Boundary

Test	Month	Day	Year	Expected result	Notes
25	12	31	1581	Error Msg	Output Min - 1
26	1	1	1582	1/2/1582	Output Min
27	1	2	1583	1/3/1583	Output Min + 1
28	12	30	3000	12/31/3000	Output Max - 1
29	12	31	3000	1/1/3001	Output Max
30	1	1	3001	Error Msg	Output Max + 1

A Sample



```
package com.my.mathdemo;

public class mathDemo {
    public int add(int a,int b)
    {
        return a+b;
    }
    public int multiply(int a,int b)
    {
        return a*b;
    }
}
```



```
public class TestMyMath {

    @Test
    public void testAdd() {
        mathDemo math = new mathDemo();
        assertEquals(math.add(11, 5), 16);     //测试11+5是否等于16

        assertEquals(math.add(3, 5), 9);       //故意写个错误的加法测试
    }

    @Test
    public void testMultiply() {
        mathDemo math = new mathDemo();
        assertEquals(math.multiply(10, 5), 50);    //测试10*5是否等于50
    }
}
```

Summary

1. **void assertEquals(boolean expected, boolean actual)**
   Check that two primitives/Objects are equal
2. **void assertTrue(boolean expected)**
   Check that a condition is true
3. **void assertFalse(boolean condition)**
   Check that a condition is false
4. **void assertNotNull(Object object)**
   Check that an object isn't null
5. **void assertNull(Object object)**
   Check that an object is null
6. **void assertSame(Object object, Object object)**
   The assertSame() methods tests if two object references point to the same object
7. **void assertNotSame(Object object, Object object)**
   The assertNotSame() methods tests if two object references not point to the same object
8. **void assertArrayEquals(expectedArray, resultArray);**
   The assertArrayEquals() method will test whether two arrays are equal to each other.

assertTrue() and assertFalse()

- The assertTrue() and assertFalse() methods tests a single variable to see if its value is either true, or false.

- If the program under test returns true, the assertTrue() method will return normally. Otherwise an exception will be thrown, and the test will stop.

- If the program under test returns false, the assertFalse() method will return normally. Otherwise an exception will be thrown, and the test will stop.

assertNull() and assertNotNull()

- The assertNull() and assertNotNull() methods test a single variable to see if it is null or not null.
- If the program returns null, the assertNull() method will return normally. If a non-null value is returned, the assertNull() method will throw an exception, and the test will stop.
- The assertNotNull() method works in the opposite way to the assertNull() method,

assertEquals()

- The assertEquals() method compares two objects for equality
- If the two objects are equal, the assertEquals() method will return normally. Otherwise the assertEquals() method will throw an exception.
-
- The assertEquals() method can compare any two objects to each other, it has versions that compare primitive types like int and float to each other.

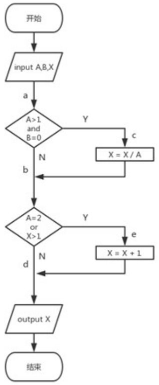
assertArrayEquals()

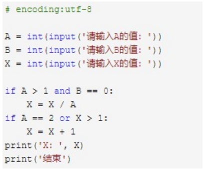
- The assertArrayEquals() method will test whether two arrays are equal to each other. In other words, if the two arrays contain the same number of elements, and if all the elements in the array are equal to each other.
- If the arrays are equal, the assertArrayEquals() will proceed without errors. If the arrays are not equal, an exception will be thrown, and the test aborted.

assertSame() and assertNotSame()

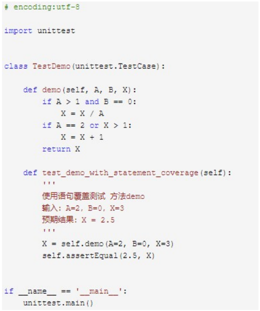
- The assertSame() and assertNotSame() methods tests if two object references point to the same object or not. It is not enough that the two objects pointed to are equals according to their equals() methods. It must be exactly the same object pointed to.

A Sample



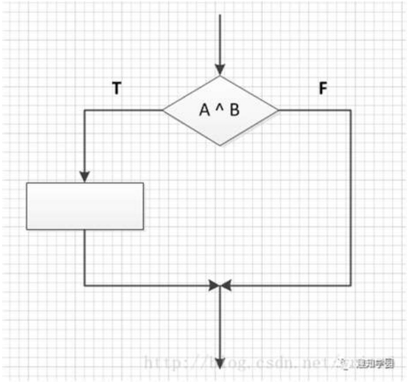


```
# encoding:utf-8

A = int(input('请输入A的值: '))
B = int(input('请输入B的值: '))
X = int(input('请输入X的值: '))

if A > 1 and B == 0:
    X = X / A
if A == 2 or X > 1:
    X = X + 1
print('X: ', X)
print('结束')
```



```
# encoding:utf-8

import unittest


class TestDemo(unittest.TestCase):

    def demo(self, A, B, X):
        if A > 1 and B == 0:
            X = X / A
        if A == 2 or X > 1:
            X = X + 1
        return X

    def test_demo_with_statement_coverage(self):
        '''
        使用语句覆盖测试 方法demo
        输入: A=2, B=0, X=3
        预期结果: X = 2.5
        '''
        X = self.demo(A=2, B=0, X=3)
        self.assertEqual(2.5, X)


if __name__ == '__main__':
    unittest.main()
```

Statement coverage

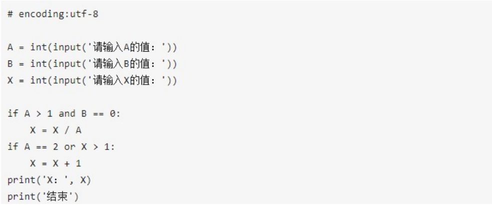


Statement coverage concept

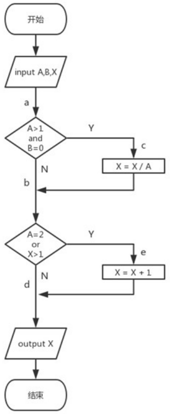
- 程序中每一个语句至少能被执行一次

- 语句覆盖是一种最弱的覆盖方法
- 无需测试程序的分支情况
  - 可以支持短路
- 无需测试程序分支判断的输入值以及输入值的组合
  - 无需划分等价类
- 无需测试程序执行的不同路径
  - 可以取巧

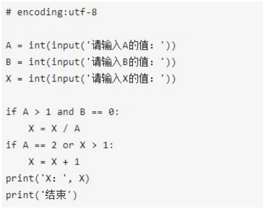
源代码与程序流程图



```
# encoding:utf-8

A = int(input('请输入A的值: '))
B = int(input('请输入B的值: '))
X = int(input('请输入X的值: '))

if A > 1 and B == 0:
    X = X / A
if A == 2 or X > 1:
    X = X + 1
print('X: ', X)
print('结束')
```

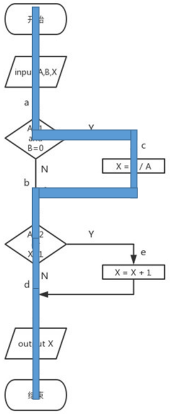


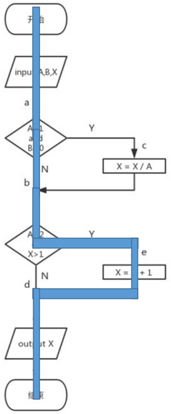
方案1



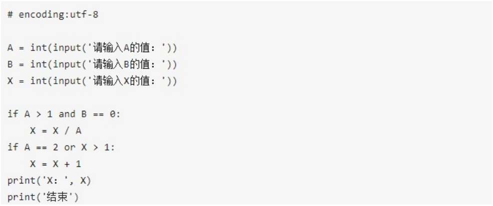
```
# encoding:utf-8

A = int(input('请输入A的值：'))
B = int(input('请输入B的值：'))
X = int(input('请输入X的值：'))

if A > 1 and B == 0:
    X = X / A
if A == 2 or X > 1:
    X = X + 1
print('X: ', X)
print('结束')
```

- 用例1：A=10，B=0，X=30
- 用例2：A=2，B=5，X=5



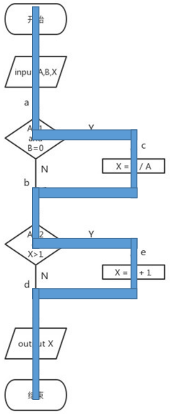


方案2



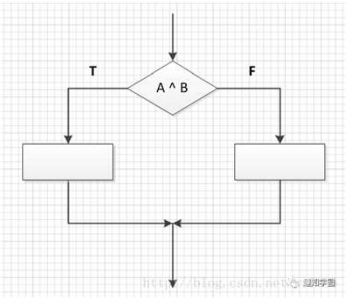
```
# encoding:utf-8

A = int(input('请输入A的值：'))
B = int(input('请输入B的值：'))
X = int(input('请输入X的值：'))

if A > 1 and B == 0:
    X = X / A
if A == 2 or X > 1:
    X = X + 1
print('X: ', X)
print('结束')
```

- 用例：A=2，B=0，X=3
- Expect result：X=2.5



Determine coverage

- Branch coverage



对于判断语句，在设计用例的时候，要设计判断语句结果为True和False的两种情况

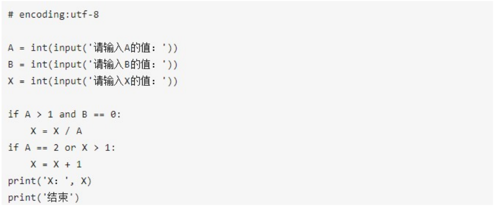
测试用例条件：
A ^ B = T
A ^ B = F

Determine coverage concept

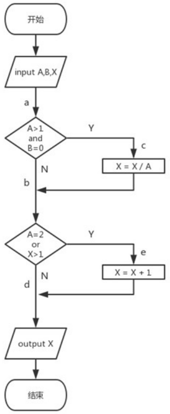
- 程序中每个判定至少有一次为真值，有一次为假值
- 使得程序中每个分支至少执行一次

- 满足判定覆盖的测试用例一定满足语句覆盖
- 对整个判定的最终取值（真或假）进行度量
- 判定内部每一个子表达式的取值不被考虑

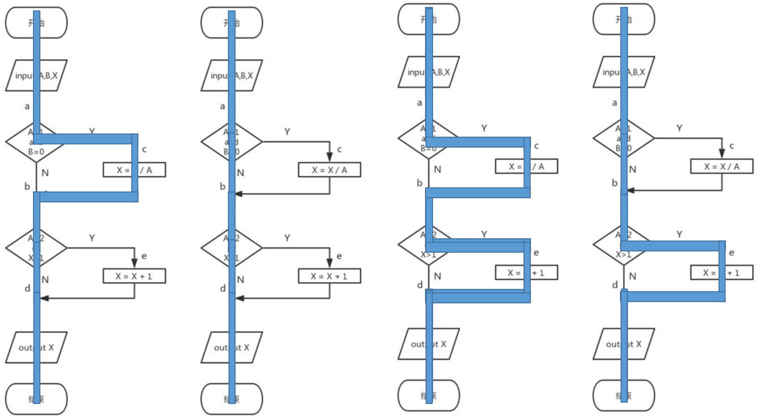
源代码与程序流程图



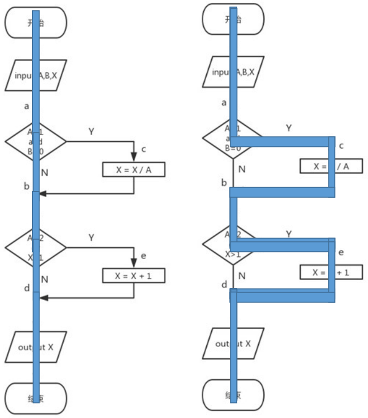
```
# encoding:utf-8

A = int(input('请输入A的值: '))
B = int(input('请输入B的值: '))
X = int(input('请输入X的值: '))

if A > 1 and B == 0:
    X = X / A
if A == 2 or X > 1:
    X = X + 1
print('X: ', X)
print('结束')
```



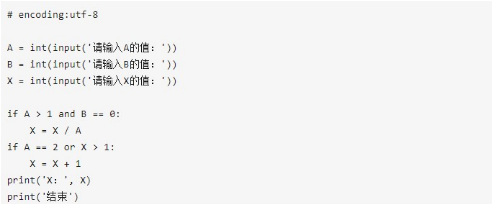
方案1



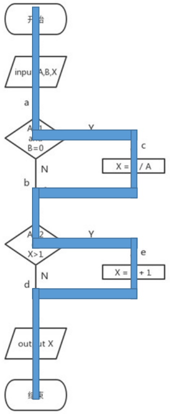
方案2



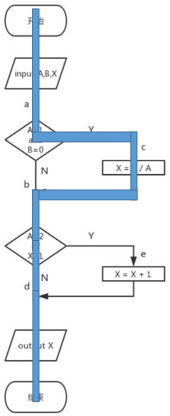
方案2

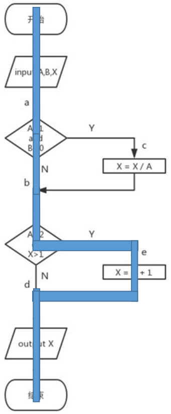


```
# encoding:utf-8

A = int(input('请输入A的值: '))
B = int(input('请输入B的值: '))
X = int(input('请输入X的值: '))

if A > 1 and B == 0:
    X = X / A
if A == 2 or X > 1:
    X = X + 1
print('X: ', X)
print('结束')
```



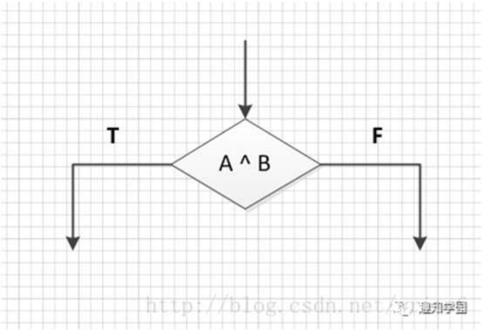
方案3





LB1
LB2

Condition coverage



设计用例时针对判断语句里面每个条件表达式true 和 false各取值一次，不考判断语句的计算结果

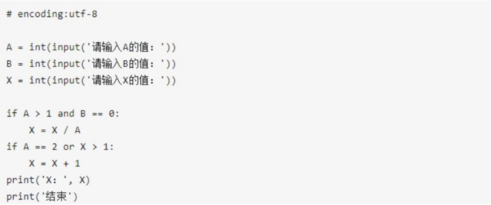
测试用例条件：

A=T	A=F
B=T	B=F

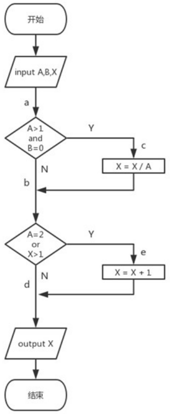
Condition coverage concept

- 一个判定（ determine ）中往往包含了若干个条件（condition
- 执行足够的测试用例，使得判定中的每个条件获得各种可能的结果（T or F）。
-
- 条件覆盖是一个比判定覆盖更强的覆盖标准
- 要同时
- 判定内部每一个子表达式的取值不被考虑

源代码与程序流程图



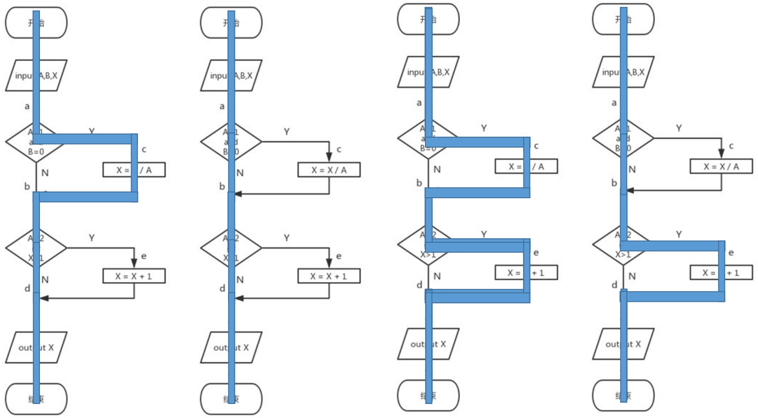
```
# encoding:utf-8

A = int(input('请输入A的值: '))
B = int(input('请输入B的值: '))
X = int(input('请输入X的值: '))

if A > 1 and B == 0:
    X = X / A
if A == 2 or X > 1:
    X = X + 1
print('X: ', X)
print('结束')
```



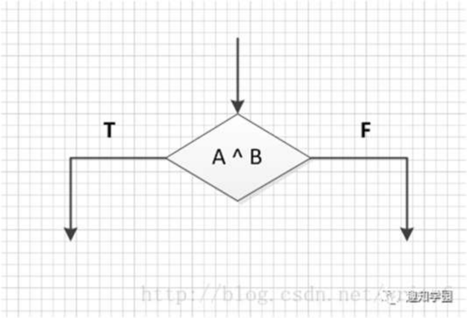
幻灯片 94

LB1	L Bruyne, 2021/1/17
LB2	L Bruyne, 2021/1/17

方案:



Branch conditional coverage



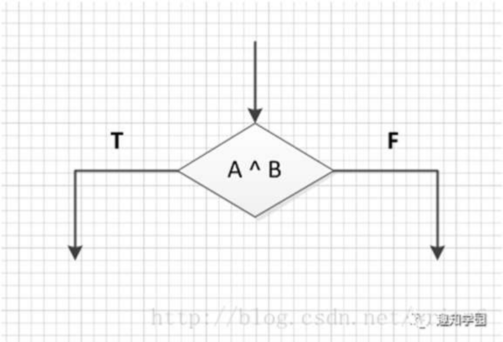
(4) 判定条件覆盖（分支条件覆盖）

设计测试用例时，使得判断语句中每个条件表达式的所有可能结果至少出现一次，每个判断语句本身所有可能结果也至少出现一次。

测试用例条件:

A ^ B = T	A ^ B = F
A=T	A=F
B=T	B=F

Conditional combination coverage



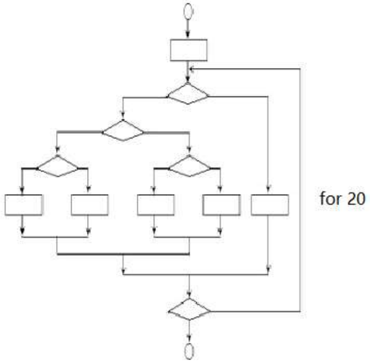
(5) 条件组合覆盖

设计测试用例时，使得每个判断语句中条件结果的所有可能组合至少出现一次

测试用例条件:

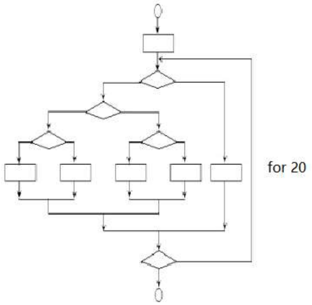
A= T	B= T
A= T	B= F
A= F	B= T
A= F	B= F

Path coverage



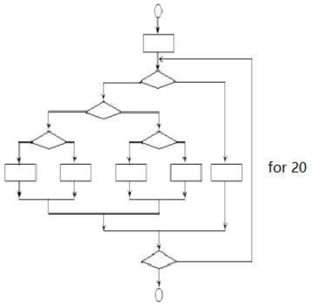
For 循环

- for fruit in fruits:
  - print(“当前水果： ”, fruit)
- for i in range(2, 10):
  - print(“当前数字：”, i)
- for i in range(2, 10):
  - print(“当前数字：”, i)
- else：
  - print(“这个数字 ”, i, “我搞不定”)



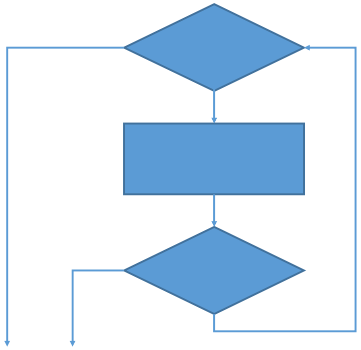
While循环

- While a<10:
  - print(a)
- while a<10:
  - print(a)
  - if a>8:
    - break



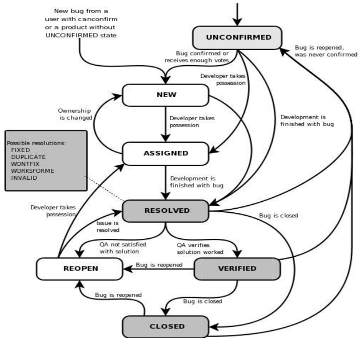
Control Flow Graph (CFG)

- 短路
- 循环体
- 循环跳出



Bug Life Cycle

Defect life cycle, also known as Bug Life cycle is the journey of a defect cycle, which a defect goes through during its lifetime. It varies from organization to organization and also from project to project as it is governed by the software testing process and also depends upon the tools used.


New bug from a user with canconfirm or a product without UNCONFIRMED state
UNCONFIRMED
Bug confirmed or receives enough votes
Bug is reopened, was never confirmed
Developer takes possession
NEW
Ownership is changed
Developer takes possession
Development is finished with bug
ASSIGNED
Possible resolutions:
FIXED
DUPLICATE
WONTFIX
WORKSFORME
INVALID
Development is finished with bug
Developer takes possession
RESOLVED
Issue is resolved
Bug is closed
QA not satisfied with solution
QA verifies solution worked
REOPEN
Bug is reopened
VERIFIED
Bug is reopened
Bug is closed
CLOSED

- **New -** Potential defect that is raised and yet to be validated.
- **Assigned -** Assigned against a development team to address it but not yet resolved.
- **Active -** The Defect is being addressed by the developer and investigation is under progress. At this stage there are two possible outcomes; viz - Deferred or Rejected.
- **Test -** The Defect is fixed and ready for testing.
- **Verified -** The Defect that is retested and the test has been verified by QA.
- **Closed -** The final state of the defect that can be closed after the QA retesting or can be closed if the defect is duplicate or considered as NOT a defect.
- **Reopened -** When the defect is NOT fixed, QA reopens/reactivates the defect.
- **Deferred -** When a defect cannot be addressed in that particular cycle it is deferred to future release.
- **Rejected -** A defect can be rejected for any of the 3 reasons; viz - duplicate defect, NOT a Defect, Non Reproducible.

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

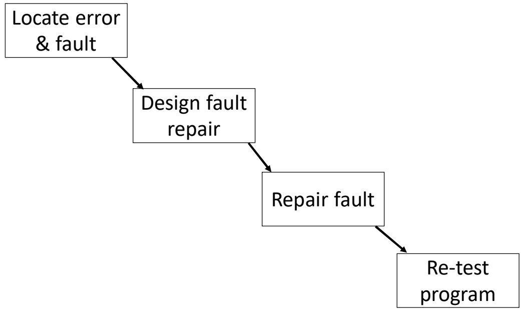
目标不同：测试的目标是发现问题，调试的目标是解决问题（发现在哪里出了问题，错误分析，缺陷定位）

优势不同：单元测试可以与jekins和git/svn这样的集成工具联动，实现自动化，更有利于团队协作和管理；debug必须手动监测变量名称，人肉执行，可以嵌入开发“中”，而不必等到事后。

对运行时的影响：无影响，必须在运行时监控代码中的变量

Value不同：单元测试可以组成不同的process，可以实现开发人员的分工，可以

Debugging Activities



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

Breakpoints

- line breakpoint
- field breakpoint
- method breakpoint
- exception breakpoint

Tools on hands

Step Into

Step Over

Step Return

Step Filter

Resume

hit count

inspect

watch

step into：单步执行，遇到子函数就进入并且继续单步执行（简而言之，进入子函数）；
step over：在单步执行时，在函数内遇到子函数时不会进入子函数内单步执行，而是将子函数整个执行完再停止，也就是把子函数整个作为一步。有一点,经过我们简单的调试,在不存在子函数的情况下是和step into效果一样的（简而言之，越过子函数，但子函数会执行）。
step out：当单步执行到子函数内时，用step out就可以执行完子函数余下部分，并返回到上一层函数。
step Filter 逐步过滤 一直执行直到遇到未经过滤的位置或断点(设置Filter:window-preferences-java-Debug-step Filtering)
Resume 重新开始执行debug,一直运行直到遇到breakpoint。　例如： A和B两个断点，debug过程中发现A断点已经无用,去除A断点，运行resume就会跳过A直接到达B断点。当debug调试跑出异常时,运行resume,重新从断点开始调试。ebug 过程中修改了某些code后--〉save&build-->resume-->重新暂挂于断点
hit count 设置执行次数 适合程序中的for循环(设置breakpoint view-右键hit count)
inspect 检查 运算。执行一个表达式显示执行值
watch 实时地监视对象、方法或变量的变化

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

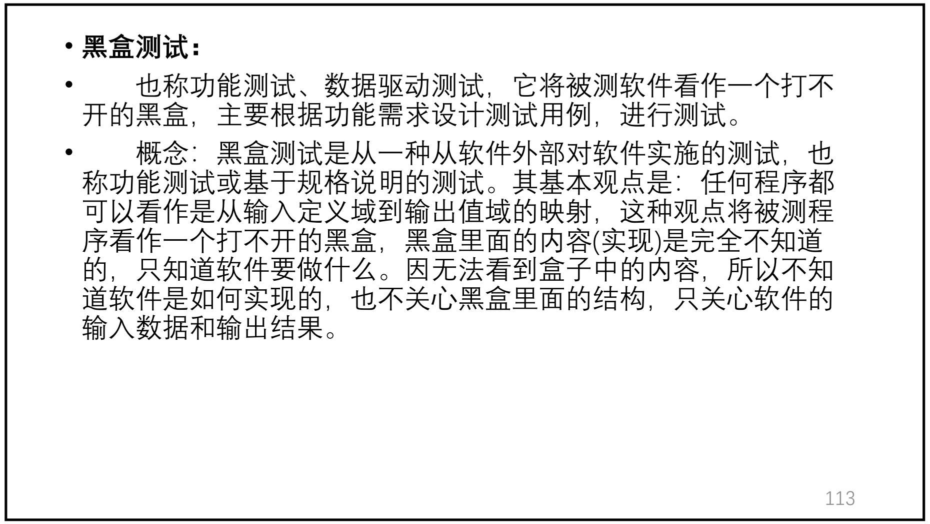
White box testing

- White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of software testing that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the expected outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT).
- White-box testing can be applied at the unit, integration and system levels of the software testing process. Although traditional testers tended to think of white-box testing as being done at the unit level, it is used for integration and system testing more frequently today. Though this method of test design can uncover many errors or problems, it has the potential to miss unimplemented parts of the specification or missing requirements.

White box testing coverage pattern

- In computer science, test coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. A program with high test coverage, measured as a percentage, has had more of its source code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to a program with low test coverage. Many different metrics can be used to calculate test coverage; some of the most basic are the percentage of program subroutines and the percentage of program statements called during execution of the test suite.

- Statement coverage – has each statement in the program been executed
- Branch coverage – has each branch (also called DD-path) of each control structure (such as in if and case statements) been executed
- Condition coverage (or predicate coverage) – has each Boolean sub-expression evaluated both to true and false



- **黑盒测试：**
- 也称功能测试、数据驱动测试，它将被测软件看作一个打不开的黑盒，主要根据功能需求设计测试用例，进行测试。
- 概念：黑盒测试是从一种从软件外部对软件实施的测试，也称功能测试或基于规格说明的测试。其基本观点是：任何程序都可以看作是从输入定义域到输出值域的映射，这种观点将被测程序看作一个打不开的黑盒，黑盒里面的内容(实现)是完全不知道的，只知道软件要做什么。因无法看到盒子中的内容，所以不知道软件是如何实现的，也不关心黑盒里面的结构，只关心软件的输入数据和输出结果。