Software Testing

Joe Timoney

ZJU 2019

1

Module Outline

- Introduction to Testing
- The Principles of Software Testing
- Static Verification
- Black Box Software Testing – Methods and Examples
- White Box Software Testing – Methods and Examples
- Integration and System testing
- Testing in the Software Process

2

Textbook

- Software Testing – Principles and Practice by S. Brown, J. Timoney, T. Lysaght and D. Ye, China Machine Press

3

What is Software Testing?

- Software Testing is a process or strategy of finding, assessing, predicting defects in software products with the aim to reliably build large-scale and high quality applications.

4

First a little history on the programmer

- At the beginning of the computer revolution it was women that were the first programmers
- Ada Lovelace is thought to have written the first code
- She wrote a program for Babbage's Analytical engine

5

ENIAC

- In 1943 the first general-purpose computer was built in the US at the University of Pennsylvania
- The programming team of six were all female

6

1960s

- By the 1960s men from established fields like physics, mathematics and electrical engineering left their old professions to become programmers
- This was a new job that had no professional identity, no professional organisations, and no means of screening potential members

7

Early Programmers

- Programming was viewed as an art form
- The programmer often learned his craft by trial and error. There was no such thing as Stack Overflow….
- The software world was completely undisciplined

8

Development of the programming profession

- This changed as the user had to specify the requirements,
- And the programmer developed the programs
- A division of labour occurred as programs become more sophisticated

9

The Software Industry

- Throughout the 1970s there was an expansion of automated information-processing tasks in companies
- The importance of programming to companies increased and tools appeared to support the programmer's productivity
- The introduction of the personal computer and its widespread adoption after 1980 accelerated the demand for software and programming

10

Software is everywhere

- Software currently implements key, market-differentiating capabilities in sectors as diverse as automobiles, air travel, consumer electronics, financial services, and mobile phones and is adding major value in domestic appliances, house construction, medical devices, clothing and social care
- It's what lets us get cash from an ATM, make a phone call, drive our car
  - A typical smartphone contains Millions of lines of code
- The scale of the software-dependent industry, often termed the secondary sector, extends far beyond the conventional software sector.

11

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

12

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…

13

The true 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)

14

Issues with producing software

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

15

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

16

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

17

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

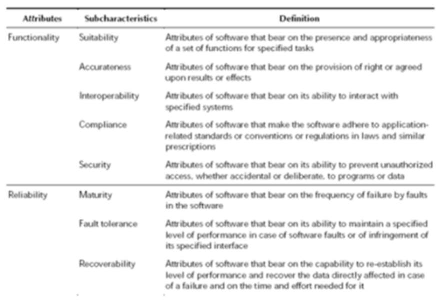
18

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

19

Detailed Description



Attributes	Subcharacteristics	Definition
Functionality	Suitability	Attributes of software that bear on the presence and appropriateness of a set of functions for specified tasks
	Accurateness	Attributes of software that bear on the provision of right or agreed upon results or effects
	Interoperability	Attributes of software that bear on its ability to interact with specified systems
	Compliance	Attributes of software that make the software adhere to application-related standards or conventions or regulations in laws and similar prescriptions
	Security	Attributes of software that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs or data
Reliability	Maturity	Attributes of software that bear on the frequency of failure by faults in the software
	Fault tolerance	Attributes of software that bear on its ability to maintain a specified level of performance in case of software faults or of infringement of its specified interface
	Recoverability	Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it

20

Software: Errors, Faults and Failures

- These can be broken down into three elements of Errors, Faults and Failures

21

Errors, Faults and Failures

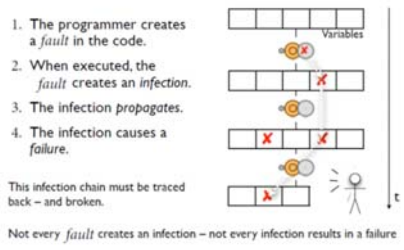
1. Errors: these are mistakes made by software developers. They exist in the mind and can result in one or more faults in the software.

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.

22

From Fault to Failure


1. The programmer creates a *fault* in the code.
2. When executed, the *fault* creates an *infection*.
3. The infection *propagates*.
4. The infection causes a *failure*.
This infection chain must be traced back – and broken.
Variables
t
Not every *fault* creates an infection – not every infection results in a failure

23

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).

24

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

25

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

26

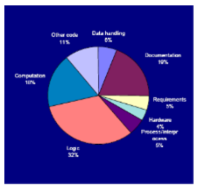
Software Faults - Categories

- 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.

27

Software Faults

- A study by Hewlett Packard on the frequency of occurrence of various fault types, found that 50% of the faults were either Algorithmic or Computation and Precision



28

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

29

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

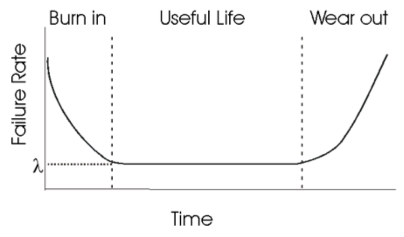
30

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

31

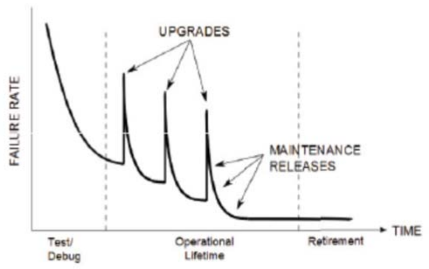
Failure Curve for Hardware



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.

32

Curve for Failure Rate over Product lifecycle



33

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.

34

Curve for Failure Rate over Product lifecycle

- Initially, the first version has a high failure rate. With debugging and testing, as the faults causing this is 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.

35

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

36

Ariane 5

- On June 4, 1996, the maiden flight of the European Ariane 5 launcher crashed about 40 seconds after takeoff.
- The rocket flipped 90 degrees in the wrong direction, and less than two seconds later, aerodynamic forces ripped the boosters apart from the main stage at a height of 4km. This caused the self-destruct mechanism to trigger, and the spacecraft was consumed in a gigantic fireball of liquid hydrogen.

37

Ariane 5

38

Ariane 5

- The disastrous launch cost approximately $370m (uninsured).
- The Ariane 5 launch is widely acknowledged as one of the most expensive software failures in history.

39

Ariane 5

- The CNES (French National Center for Space Studies) and the European Space Agency immediately appointed an international inquiry board who produced their report in hardly more than a month

40

Ariane 5

- It is a remarkably short, simple, clear and forceful document. Its conclusion: the explosion was the result of a software error -- possibly the costliest in history.
- The error came from a piece of the software that was *not* needed during the crash. It has to do with the Inertial Reference System, (termed SRI in the report).

41

Ariane 5

- The rocket used this system to determine whether it was pointing up or down, which is formally known as the horizontal bias, or informally as a BH value. This value was represented by a 64-bit floating variable.
- Before lift-off certain computations are performed to align the SRI. It caused an exception, which was not caught after takeoff.

42

Ariane 5

- The exception was due to a floating-point error: a conversion from a 64-bit integer to a 16-bit signed integer (which should only have been applied to a number less than 2^15) was erroneously applied to a greater number.
- For the first few seconds of flight, the rocket's acceleration was low, so the conversion between these two values was successful

43

Ariane 5

•However, as the rocket's velocity increased, the 64-bit variable exceeded 65000, and became too large to fit in a 16-bit variable. It was at this point that the processor encountered an operand error.

•There was no explicit handler to catch the exception, so it crashed the entire software, hence the on-board computers.

44

Ariane 5

- Several factors make this failure particularly embarrassing:
- Firstly, this was a legacy from the codebase from the rocket's predecessor, the Ariane 4
- Secondly, code which would have caught and handled these conversion errors had been disabled due to performance constraints on the Ariane 4 hardware which actually did not apply to Ariane 5.
- Third, there was a change in user requirements. The Ariane 5 launched with a much steeper trajectory than the Ariane 4, which resulted in greater vertical velocity, increasing the certainty of a conversion error.

45

Therac-25

- The Therac-25 was a radiation therapy machine manufactured by AECL in the 80s, which offered a revolutionary dual treatment mode. It was also designed from the outset to use software based safety systems rather than hardware controls.
- Eleven Therac-25s were installed: five in the US and six in Canada. Six accidents involving massive overdoses to patients occurred between 1985 and 1987. The machine was recalled in 1987 for extensive design changes, including hardware safeguards against software errors.

46

Therac-25

- The Therac-25 was a linear accelerator with two modes of operation. Firstly, it could fire a beam of low-energy electrons, which do not penetrate far into the body, and are therefore well-suited at killing shallow tissues, such as in skin cancer.
- The second mode of operation delivers radiation via a beam of higher-energy X-Ray photons. These particles travel further and are best suited to treating deeper tissues, such as cancer of the lungs.

47

Therac-25

- Dual-mode operation was truly revolutionary at the time. Hospitals would not need to maintain two separate machines, reducing their maintenance costs, and logistics could be simplified, as patients would not need to be moved from one room to the next.
- The low-power mode used scanning magnets to spread the electron beam, whereas the high-power mode was activated by rotating four components into the beam. This process took around 8 seconds to complete, and afterwards, it would spread and direct a beam of the appropriate strength towards its target.

48

Therac-25

- Therac-25 relied on software controls to switch between modes, rather than physical hardware.
- Preceding models used separate circuits to monitor radiation intensity, and hardware interlocks to ensure that spreading magnets were correctly positioned.
- Using software instead would in theory reduce complexity, and reduce manufacturing costs.

49

Therac-25

- Over the course of several weeks, one radiology technician had become very quick at typing commands into the Therac-25 machine. One fateful day, they accidentally entered 'x' for X-Ray rather than 'e' for Electron, so pressed the up key to choose the correct mode.
- Upon starting the program, the machine shut down, displaying the error "Malfunction 54." Due to the frequency at which other malfunctions occurred, and that "treatment pause" typically indicated a low-priority issue, the technician resumed treatment.

50

Therac-25

- The patient was receiving his 9th treatment, and immediately knew something had gone terribly wrong. He reported hearing a buzzing sound, which was later determined to be the machine delivering radiation at maximum capacity.
- After a few days, the patient suffered paralysis due to radiation overexposure, and ultimately died.

51

Therac-25

- The bug was finally reproduced when the same technician operated the machine on another patient, who also died from radiation overexposure.
- The dose of delivered radiation was in the range of 10-20,000 rads, which was over 100 times the expected dose, and more than enough to kill a grown adult.

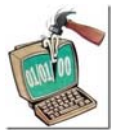
52

Therac-25 Observations

- Firstly, users will ignore cryptic error messages, particularly if they occur often. "Malfunction 54"does not convey the severity of the machine state, and the average user certainly won't consult the accompanying physical manual to find out what it means.
- Secondly, in safety-critical systems, code should be subject to formal analysis from independent parties from those who developed it. Some level of automated testing at the unit level is also needed, rather than only testing the system as a whole.

53

Software Failures



- **Y2K (1999)**
- **Cost:** €350 billion
  - **Disaster:** One man's disaster is another man's fortune, as demonstrated by the infamous Y2K bug. Businesses spent billions on programmers to fix a glitch in legacy software. While no significant computer failures occurred, preparation for the Y2K bug had a significant cost and time impact on all industries that use computer technology.
  - **Cause:** To save computer storage space, legacy software often stored the year for dates as two digit numbers, such as "99" for 1999. The software also interpreted "00" to mean 1900 rather than 2000, so when the year 2000 came along, bugs would result.

54

Y2K Bug on youku

- https://v.youku.com/v_show/id_XMzE1ODM2Mjk0NA==.html?spm=a2h0k.11417342.soresults.dtitle

55

Software Failures

- **Mars Climate *Orbiter* (1999)**
  - **Disaster:** After a 286-day journey from Earth, the Mars Climate Orbiter fired its engines to push into orbit around Mars. The engines fired, but the spacecraft fell too far into the planet's atmosphere, causing it to crash on Mars.
  - The $235m craft was intended to be the first ever weather satellite for Mars, which would monitor the planet's atmosphere with its high-resolution camera. Over the course of 687 days, or one Martian Year, the orbiter was supposed to monitor the planet's temperature, and relay photographs of dust storms back to ground control on Earth.

56

Mars Climate Orbiter

- There was a bug in the ground control software supplied by Lockheed Martin, which displayed the force in certain maneuvers. The software calculated the value in an imperial unit, pound-seconds, whereas the software built by NASA expected the value to be in a metric unit, newton-seconds. As these values were not correctly converted, this led to small discrepancies in the position of the spacecraft, which compounded over a course of millions of miles.
- Quality Assurance had not found the use of an imperial unit in external software, despite the fact that NASA's coding standards at the time mandated use of metric units.

57

Software Failures

- Communication between teams and training of the operations staff is cited as a contributing factor in NASA's investigation report.
- Navigation staff were also controlling three separate missions simultaneously, which may have diluted their attention from discrepancies in the trajectory.

58

Software Failures

Knight

- Knight Capital Group
  - On August 1st, 2012, Knight Capital deployed a new software update to their production servers. At around 08:01AM, staff in the firm received 97 email notifications stating that *Power Peg*, a defunct internal system that was last used in 2003, was configured incorrectly. This was the first warning sign.
- What Happened
  - At 09:00AM, the New York Stock Exchange opened for trading, and Knight Capital's first retail investor of the day placed an instruction to buy or sell their investment holdings. Just 45 minutes later, Knight Capital's servers had executed 4 million trades, losing the company $460 million and placing it on the verge of bankruptcy. Some shares on the NYSE shot up by over 300%, as High Frequency Trading algorithms from other firms exploited the bug. Ultimately, Knight Capital was fined an additional $12 million by the Securities Exchange Commission, due to various violations of financial risk management regulations.

59

Software Failures

Knight

- What Happened
  - A stock exchange works by pairing an individual who wants to buy a stock, with another individual who wants to sell that stock. The seller will state an *ask price*, which is how much they are willing to sell it for, and the buyer will state a *bid price*, which is how much they're willing to pay. The difference between these two values is known as the *bid/ask spread*, and typically hovers around a few cents.
  - Conventional financial wisdom also states that an individual should buy shares when the price is low, and sell when the price is high.
  - Unfortunately, an algorithm deployed on one of Knight's production servers was designed to do exactly the opposite of that, as fast as possible. The *Power Peg* program was designed to buy a stock at its ask price, and then immediately sell it again at the bid price, losing the value of the spread. Although a few cents may not seem like much, when a computer is executing thousands of trades a second, it quickly adds up to catastrophic losses.

60

Software Failures

- Buy Low, Sell high
  - In a test environment, *Power Peg* would drive up the price of stocks, allowing QA to verify that other features of the software was working correctly.
  - In a production environment, *Power Peg* would result in C-Level employees spending their weekends frantically searching for Wall Street Investors to cover a multi-million dollar black hole that had just appeared in the budget.

61

Software Failures

- Cause of Failure
  - The cause of the failure was due to multiple factors. However, one of the most important was that a flag which had previously been used to enable Power Peg was repurposed for use in new functionality. This meant the program believed it was in a test environment, and executed trades as quickly as possible, without caring about losing the spread value.
  - Secondly, the dev-ops team failed to deploy the updated program to one of the eight production servers. This server now had an outdated version of the software, and a flag stating that Power Peg should be enabled.
  - Finally, Power Peg had been obsolete since 2003, yet still remained in the codebase some eight years later. In 2005, an alteration was made to the Power Peg code which inadvertently disabled safety-checks which would have prevented such a scenario. However, this update was deployed to a production system at the time, despite no effort having been made to verify that the Power Peg functionality still worked.

62

Software Failures

- Final Report
  - Many factors were highlighted by the Security Exchange Commission's report. The most damning factors are that there was no formal code review or QA process, and that processes were not in place to check that software had deployed correctly.
  - Additionally, no pre-set capital thresholds were in place to stop automated trading after a certain amount of losses.

63

Windows Vista

- For 19 hours on August 24, 2007, anyone who tried to install Windows was told, by Microsoft's own antipiracy software (called Windows Genuine Advantage) that they were installing illegal copies.
- If you'd bought Windows Vista, you discovered certain features shut off as punishment. The bug this time was both human and traditional: Someone accidentally installed a buggy, early version of the Genuine Advantage software on Microsoft's servers.

64

Software Failures

- **Royal Bank of Scotland (2012)**
- **Cost:** unknown, still to be determined
  - **Disaster:** A software update was applied on 19 June 2012 to RBS CA-7 software which controls the payment processing system. Customer wages, payments and other transactions were disrupted. Some customers were unable to withdraw cash using ATMs or see bank account details. Others faced fines for late payment of bills because the system could not process direct debits. It took until the 16th July before it was fixed.
  - **Cause:** The software upgrade was corrupted

65

RBS Software Upgrade Consequences

- People could not withdraw cash from the ATMs
- Bills could not be paid because direct debits could not be processed, furthermore, customers faced fines for not having bills paid on time
- Wages were not being paid into accounts
- Social welfare payments were not going through
- Completion of new home purchases were delayed
- Others were stranded abroad

66

British Airways Terminal 5 Opening in 2008

- Terminal 5 opened at Heathrow airport on March 27th 2008
- During the first five days, BA misplaced more than 23,000 bags, cancelled 500 flights and made losses of £16m.
- The CEO, Willie Walsh, revealed that IT problems and a lack of testing played a large part in the trouble. But he said the airline could have coped if IT had been the only issue.

67

British Airways Terminal 5 Opening in 2008

68

British Airways Terminal 5 Opening in 2008

- BAs' written evidence showed how many IT problems staff had to contend with.
- To begin with, loading staff could not sign on to the baggage-reconciliation system to link passengers and bags. They had to reconcile bags manually, causing flight delays.
- Problems with the wireless Lan at some check-in stands meant that staff could not enter information on bags into the system using their handheld devices.

69

British Airways Terminal 5 Opening in 2008

- During testing on the baggage system, technicians installed software filters in the baggage system.
- Their job was to prevent specimen messages generated by the baggage system during the tests being delivered to the "live" systems elsewhere in Heathrow.
- They were accidently left in place after the terminal opened.

70

British Airways Terminal 5 Opening in 2008

- As a result, the Terminal 5 system did not receive information about bags transferring to British Airways from other airlines.
- The unrecognised bags were automatically sent for manual sorting in the terminal's storage facility

71

British Airways Terminal 5 Opening in 2008

- On Saturday 5 April - a week and a half after opening - the reconciliation system failed for the whole day. Bags missed their flights because the faulty system told staff that they had not been security screened.

72

British Airways Terminal 5 Opening in 2008

- As the errors built up, more bags went unrecognised by the system, missed their flights, or had to re-booked on new flights.
- The baggage-handling system froze after becoming unable to cope with the number of messages generated by re-booking flights, forcing managers to switch off the automated re-booking system.

73

British Airways Terminal 5 Opening in 2008

- BA puts the failure to spot the IT issues down to inadequate system testing, caused by delays to BAA's construction work.
- Construction work was scheduled to finish on 17 September 2007. The delays meant BA IT staff could not start testing until 31 October.
- Several trials had to be cancelled, and BA had to reduce the scope of system trials because testing staff were unable to access the entire Terminal 5 site.

74

In September 2016…

- British Airways passengers have been hit by long delays after an IT glitch affected worldwide check-in systems.
- Angry travellers were forced queued for hours, as airport staff manually processed flight checks-ins. Some passengers posted photographs on social media of hand-written boarding passes.

75

BA September 2016

76

Just for reassurance…

- British Airways is not the only major airline to have experienced technical difficulties in 2016, as Delta Air Lines was forced to ground flights worldwide in early August 2016 after a power outage knocked out its computer systems around the world

77

The explanation

- "Airline computers juggle multiple systems that must interact to control gate, reservations, ticketing and frequent fliers. Each of those pieces may have been written separately by different companies. Even if an airline has backup systems, the software running those likely has the same coding flaw," he said.
- "Tracking down a software flaw can be very difficult. It's like investigating crime; there is a lot of data they've got to sift through to figure out what happened."

78

Interesting Quotes

- "If Microsoft made cars instead of computer programs, product-liability suits might now have driven them out of business."
- if cars were like software, they would crash twice a day for no reason, and when you called for service, they'd tell you to reinstall the engine
- "Thank the programmer", Airline pilot after a smooth landing

79

Interesting Quotes

- There is not now, and never will be, a language in which it is the least bit difficult to write bad programs
- Your problem is another's solution; Your solution will be their problem

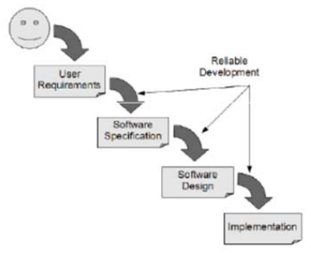
80

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.

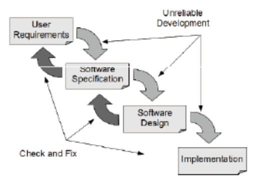
81

Forward Engineering



82

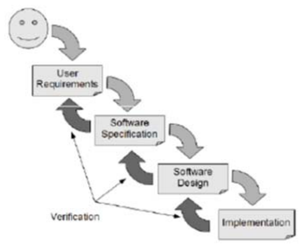
Check and Fix



In practice, all development steps must be subject to a check to ensure it has been carried out properly and any mistakes encountered are fixed.

83

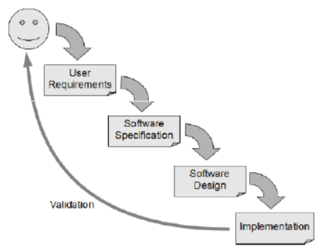
Including verification in the development process



Check that each step meets its specifications

84

Validation in the development process



Check that the software meets the users needs

85

Specifications

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

86

Specifications

- ‘Reasonable' behaviour has issues
  - What is a reasonable behaviour if a parameter is missing from a function call?
    - Ignore or Exception?
  - What is a reasonable return value for a method that calculates temperature?
    - Celsius or Fahrenheit?

87

Importance of Specifications

- The tester needs to
  - Read and understand the user requirements
  - Turn them into a structured form (e.g. a table)

88

Example Specification

- A program 'check' for an online shop determines whether a **customer gets a discount**, based on **points to date** and whether the customer is a **gold member**. Bonus points accumulate as customers make purchases.
  - Gold customers will get a discount once they accumulate 80 points.
  - Other customers only get a discount once they have more than 120 points.
  - All customers always have at least one point.

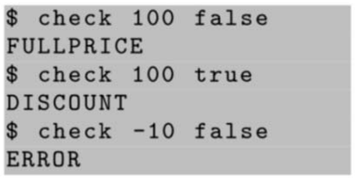
89

Return

- FULLPRICE: customer needs to pay in full
- DISCOUNT: customers gets discount
- ERROR: input parameter invalid (bonusPoints < 1)

90

How to Test?


```
$ check 100 false
FULLPRICE
$ check 100 true
DISCOUNT
$ check -10 false
ERROR
```

91

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

92

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.

93

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

94

System Testing

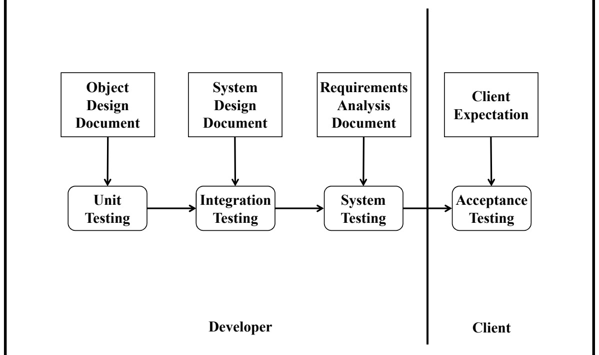
- The entire software system is tested to make sure that it works correctly and that it agrees with the specification
- This uses the system interface which may be a GUI, Network interface, web interface etc…

95

Acceptance Testing

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

96



97

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 reintroduce errors that were previously fixed

98

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

99

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.

100

Theory of Testing

- To find the **minimum of test data** that works with all inputs

1. In a successful test all test data in a test set must produce results as defined in specification
2. Test data selection criteria is reliable if it consistently produces test data that is (not) successful
3. If test data chosen with a reliable and valid criterion, then successful test means that program produces correct results for all input domain
4. Result: **Only criterion** that guarantees to be reliable and valid is the one that selects **each and every value** in the program domain -> **exhaustive testing**

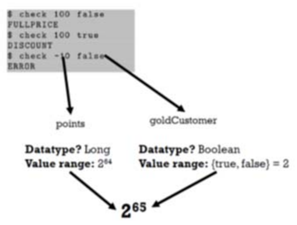
101

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.

102

Consider the first example



```
$ check 100 false
FULLPRICE
$ check 100 true
DISCOUNT
$ check -10 false
ERROR
```

103

Exhaustive Testing Example

- Consider a method bound( ) as defined below:

```
// return a value of x bounded by the
  upper and lower values
// return lower if x<=lower
// return upper if x>=upper
// return x if lower<x<upper
long bound(int lower, int x, int upper);
```

104

Exhaustive Testing Example

- If *int* is defined as a 32-bit, signed integer, then each input parameter has 2^{32} possible input values.
- Exhaustive testing would require using every possible combination of input values, leading to 2^{96} tests.
- If, on average, each test took 1 nano-second to execute – possible on a modern, multi-core processor – then the test would take 2^{96} (or about 10^{29}) nano-seconds to complete. This is approximately 10^{12} years!
- The sun is only $4.6 * 10^9$ years old. So clearly this is not feasible, even for a simple function.

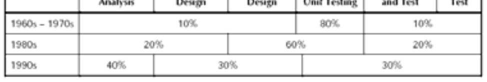
105

Time Spent on Testing

- 50%~Brooks/Myers, 1970s
- 80%~Arthur Andersons' Director of testing in North America, 1990s

106

Time Allocation to create Software



	Requirements Analysis	Preliminary Design	Detailed Design	Coding and Unit Testing	Integration and Test	System Test
1960s – 1970s	10%			80%	10%	
1980s	20%		60%		20%	
1990s	40%	30%			30%	

Source: Andersson, M., and J. Bergstrand. 1995. "Formalizing Use Cases with Message Sequence Charts." Unpublished Master's thesis. Lund Institute of Technology, Lund, Sweden.

107

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

108

Time Spent on Testing

	Planning	Code & Unit Test	Integration & Test
Commercial DP	25%	40%	35%
Internet Systems	55%	15%	30%
Real-time Systems	35%	25%	40%
Defense Systems	40%	20%	40%

DP: Desktop Publishing

Bennatan, E.M, "On Time Within Budget", 2000

109

Time Spent on Testing

Activity	Small Project (2.5K LOC)	Large Project (500K LOC)
Analysis	10%	30%
Design	20%	20%
Code	25%	10%
Unit Test	20%	5%
Integration	15%	20%
System test	10%	15%

McConnell, Steve, "Rapid Development", 1996

110

Software Testing and Debugging

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

111

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

112

Static Testing

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

113

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.

114

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.

115

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

116

How to do it

- 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

117

Test the tests

- 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

118

InternationalComparisons

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

119

“Conventional” Good Practices

	India	*Japan*	*USA*	*Europe etc*	***Total***
Number of Projects	*24*	*27*	*31*	*22*	*104*
Architectural Specs %	83%	70%	55%	73%	69%
Functional Specs %	96%	93%	74%	82%	86%
Detailed Design %	100%	85%	32%	68%	69%
Code Generators -- Yes	63%	41%	52%	55%	52%
Design Reviews - Yes	100%	100%	77%	77%	88%
Code Reviews -- Yes	96%	74%	71%	82%	80%

120

"Newer" Iterative Practices

	India	Japan	USA	Europe etc	Total
No. of Projects	24	27	31	22	104
Subcycles -- Yes	79%	44%	55%	86%	64%
Beta tests -- Yes	67%	67%	77%	82%	73%
Pair Testing -- Yes	54%	44%	35%	32%	41%
Pair Programmer -- Yes	58%	22%	36%	27%	35%
Daily Builds at project start	17%	22%	36%	9%	22%
In the middle	13%	26%	29%	27%	24%
At the end	29%	37%	36%	41%	36%
Regression test each build	92%	96%	71%	77%	84%

121

"Crude" Output Comparisons

		India	Japan	USA	Europe etc.	TOTAL
Projects		24	27	31	22	***104***
LOC/ Month	median	209	469 cf. *389* in 1990	270 cf. *245* in 1990	436	***374***
faults/ 1000 LOC	median	.263	**.020** cf. .20 in 1990	.400 cf. .80 in 1990	.225	***.150***

122

Observations

- Most projects (64%) are not pure waterfall; 36% were
- Mix of "conventional" and "iterative" common
  use of functional specs, design & code reviews, but with subcycles, regression tests on frequent builds
- Customer-reporting of defects improved
  over past decade in US and Japan; LOC "productivity" may have improved a little
- Japanese still report best quality & productivity
  but what does this mean? Preoccupation with "zero defects"? Need more lines of code to write same functionality per day as US & Indian programmers?
- Indian projects strong in process and quality

123

Observations

- 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)

124

Finishing Testing

- 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

125

Criteria

- 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.

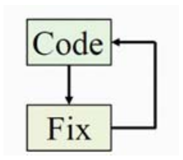
126

Software Development Lifecycles

- Code and Fix
- Waterfall
- V-model
- Sashmi
- Incremental development
- Extreme Programming
- SCRUM
- DevOps

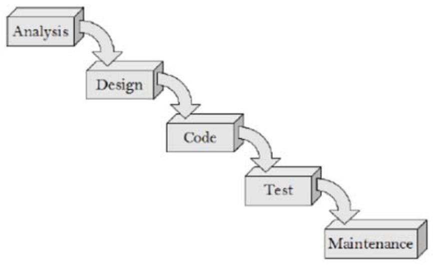
127

Code and Fix



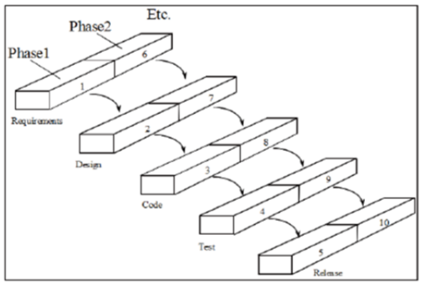
128

Waterfall



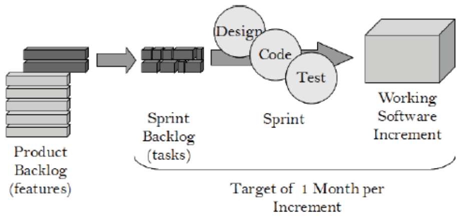
129

Incremental



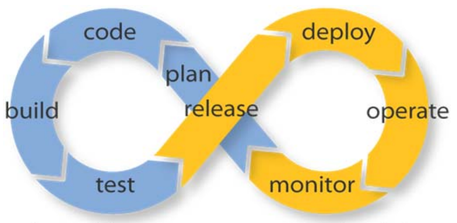
130

SCRUM


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

131

DevOps



Endless Possibilities: DevOps can create an infinite loop of release and feedback for all your code and deployment targets.

132