CS608
Software Testing

Dr. Stephen Brown

Room Eolas 116

stephen.brown@mu.ie

CS608

Introduction to Software Testing

(Essentials of Software Testing, Chapter 1)

Notes

- We have a lot of introductory material to cover today
- Fast paced lecture
- Take notes
- Read Chapter 1 afterwards

Approach in CS608

- Software testing techniques are introduced through **worked examples**

The Approach

- Software testing techniques introduced through worked examples
- **Practical Lab** to give you some experience in applying each technique

The Approach

- Software testing techniques introduced through worked examples
- Lab to give you some experience in applying each technique
- **Process** emphasised:
  - Analyse the testing problem
  - Design and review the tests
  - Implement the tests
  - Review the test results

The Approach

- Software testing techniques introduced through worked examples
- Lab to give you some experience
- Process: Analysis, Design, Implementation, Results
- Each technique is then explained **in more detail**
  - Limitations explored by inserting faults

The Approach

- Software testing techniques introduced through worked examples
- Lab to give you some experience
- Process: Analysis, Design, Implementation, Results
- Each technique is then explained in more detail
- **Worked examples**
  - Offer the beginner a practical, step-by-step introduction to each technique
- **Additional details**
  - Providing a deeper understanding in applying the underlying principles

Introduction to Software Testing

- Modern society is heavily reliant on software, and the correct operation of this software is a critical concern:
  - Personal: heating, mobile phone, television, hi-fi, laptop, car, e-banking, e-payments, e-shopping, social networking, …
  - Business: document preparation and exchange, contracts, e-payments, video conferencing, …
  - Government: public records, e-services (public, businesses), …

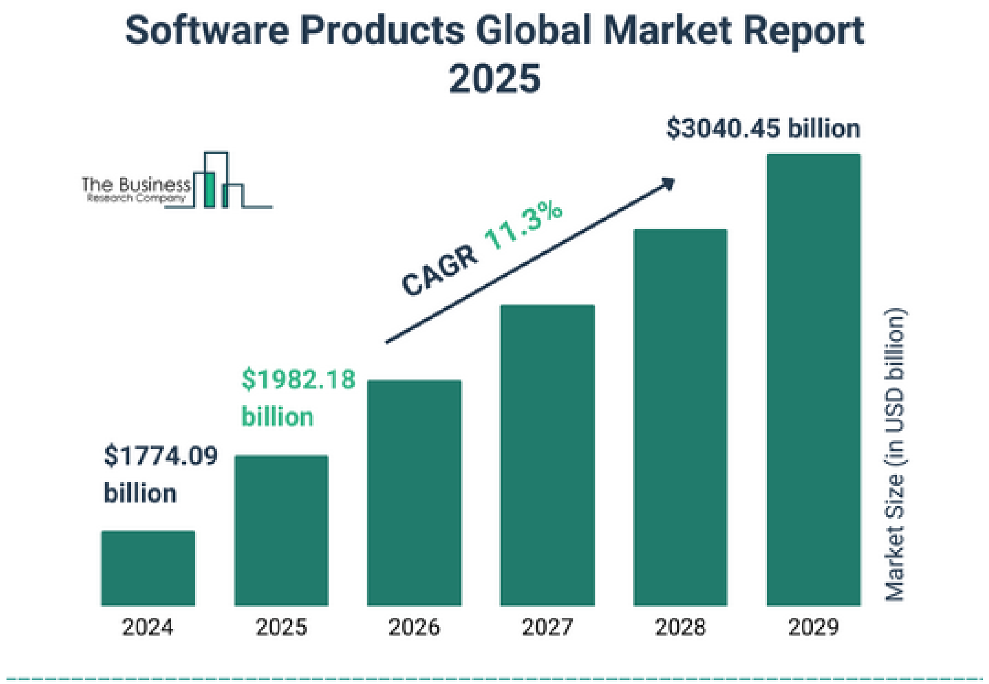
Introduction to Software Testing

- Modern society is heavily reliant on software, and the correct operation of this software is a critical concern
- The purpose of this module is to introduce you to the essential principles of software testing, enabling you to produce high quality software

Introduction to Software Testing

- Modern society is heavily reliant on software, and the correct operation of this software is a critical concern
- The purpose of this module is to introduce you to the essential principles of software testing, enabling you to produce high quality software
- Software testing can be regarded as an art, a craft, and a science: this module provides a bridge between these different viewpoints

The Software Industry

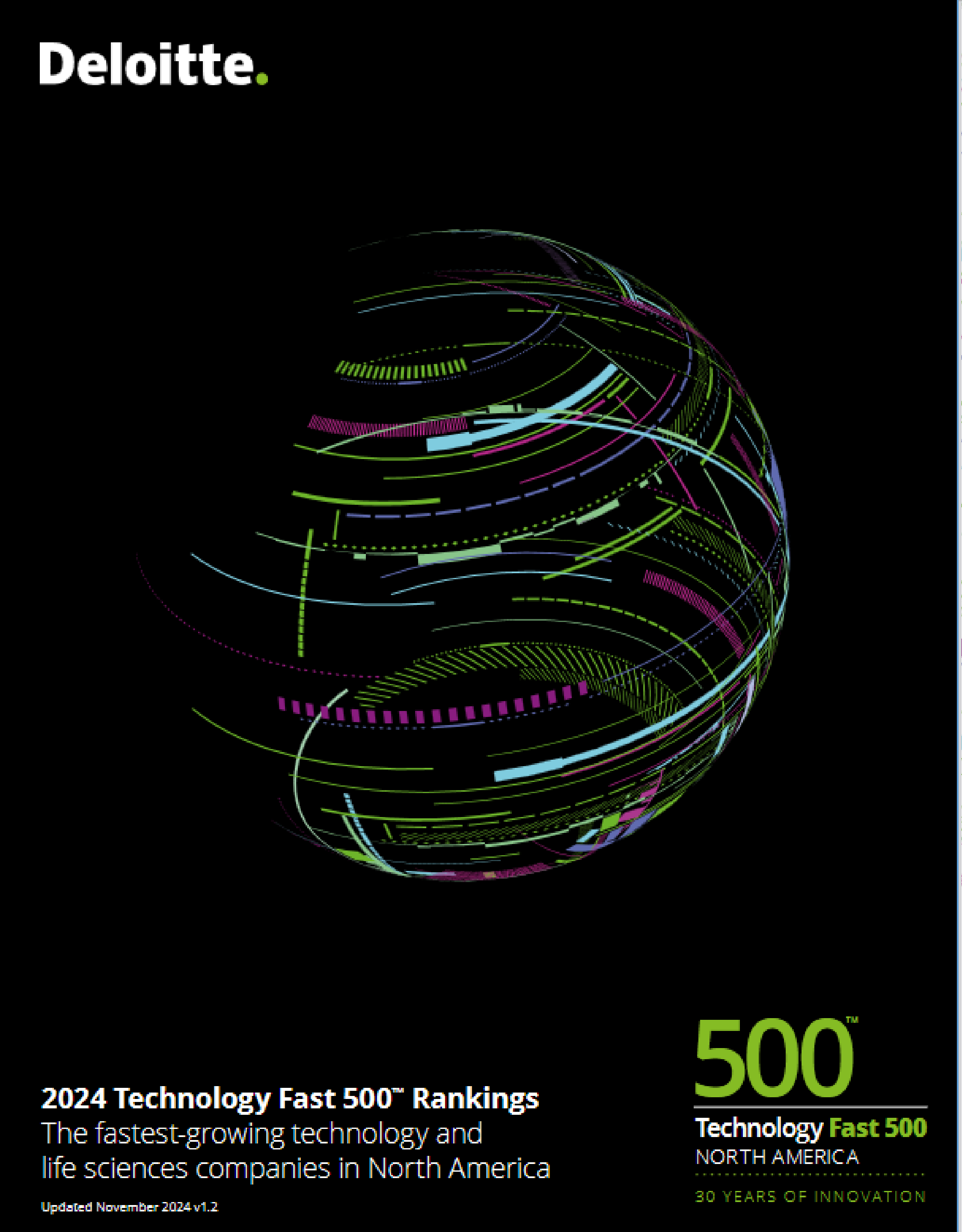


Software Products Market Size 2025 And Growth Rate

The software products market size has grown rapidly in recent years. It will grow from **$1774.09 billion in 2024 to $1982.18 billion in 2025 at a compound annual growth rate (CAGR) of 11.7%.** The growth in the historic period can be attributed to enterprise resource planning, E-Commerce boom, mobile application development, open-source movement, cybersecurity concerns, open source movement.

The Software Industry

- In 2010 the Top 500 companies in the global software industry had revenues of $492 billion, and by 2018, this had risen to $868 billion [Software Magazine]
- Global software products market: $931 billion in 2020, $968 billion in 2021, predicted $1.493 trillion in 2024 [businesswire]


Deloitte.

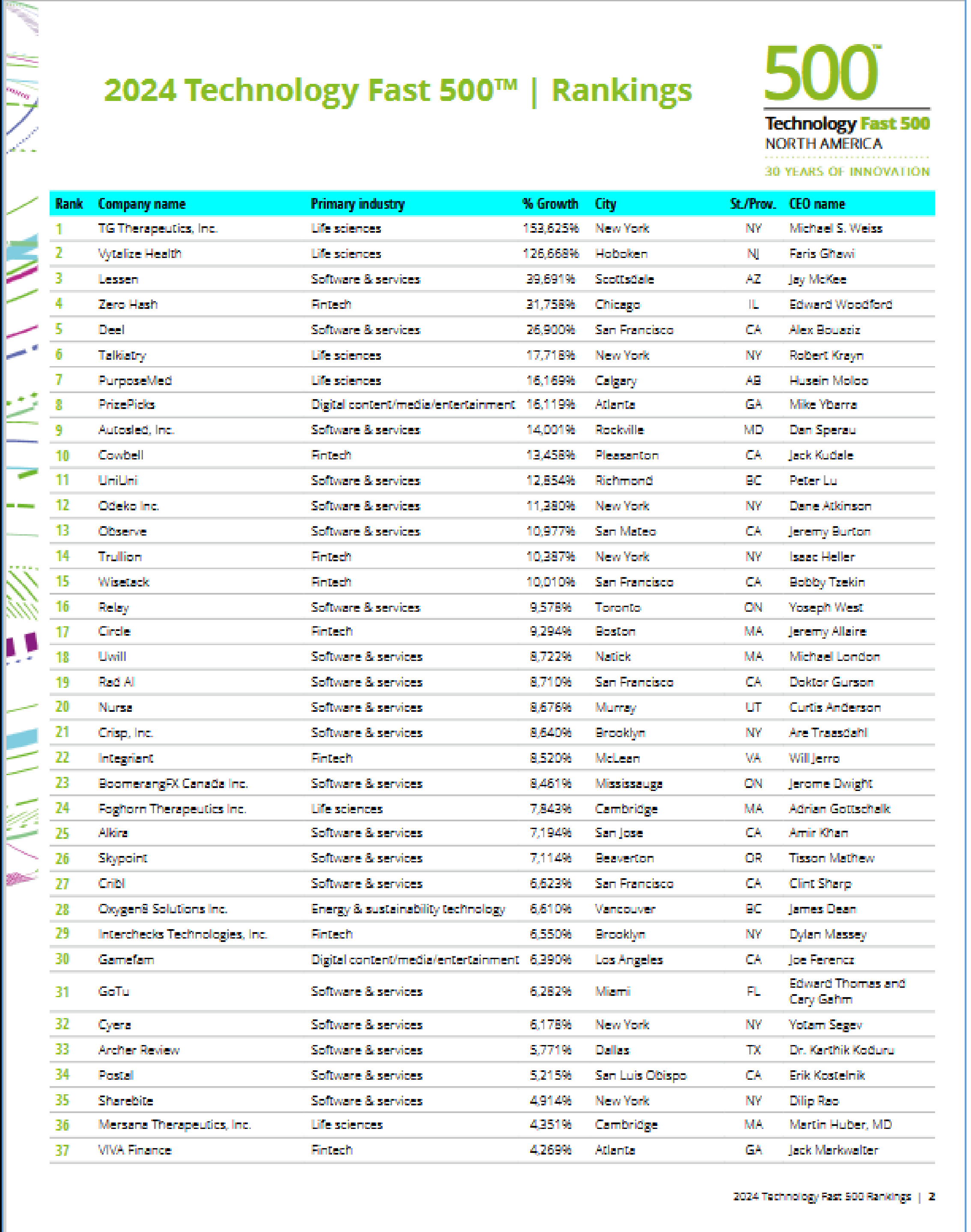
2024 Technology Fast 500™ Rankings

The fastest-growing technology and life sciences companies in North America

Updated November 2024 v1.2

500™ Technology Fast 500 NORTH AMERICA

30 YEARS OF INNOVATION

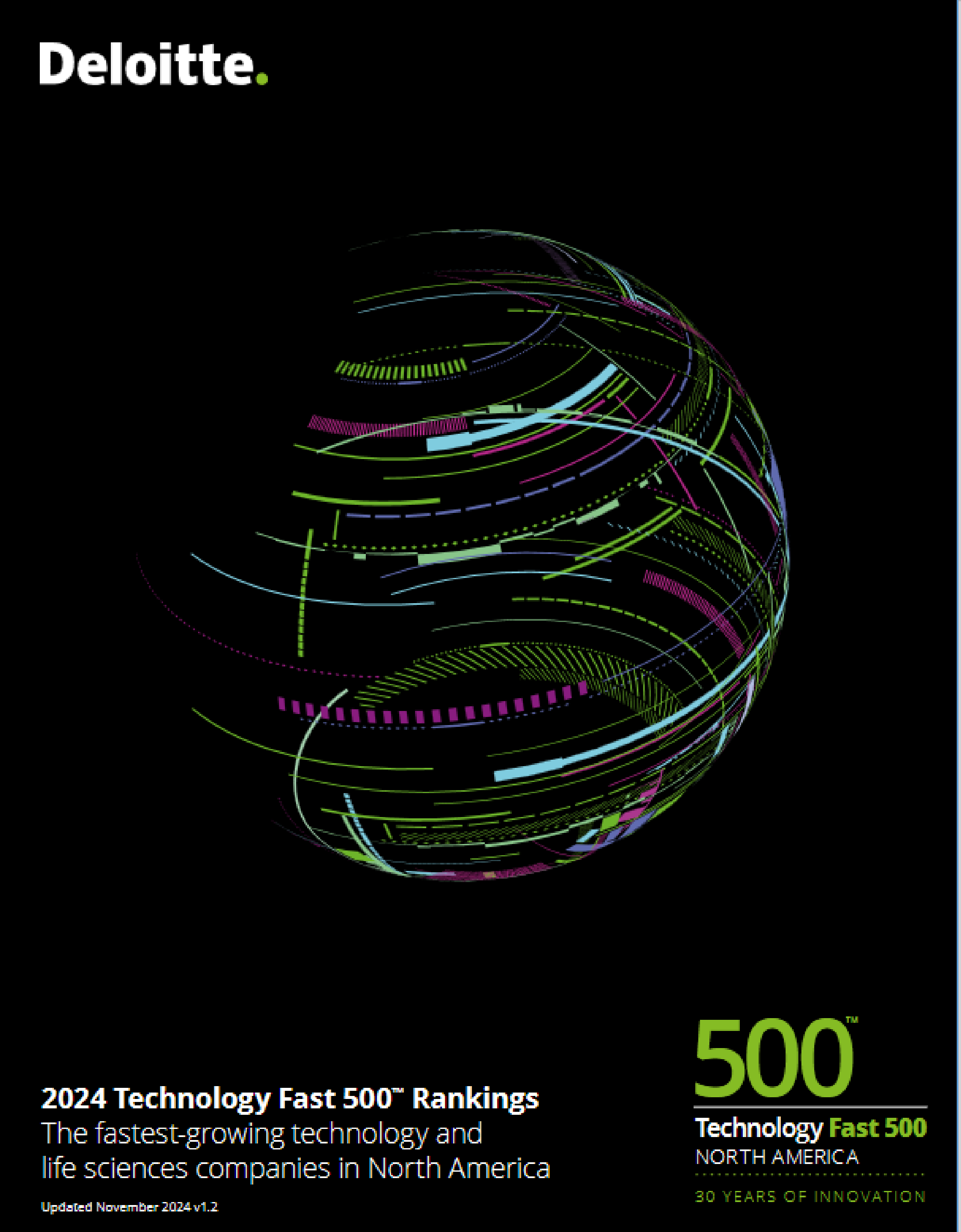

2024 Technology Fast 500™ | Rankings

500™ Technology Fast 500 NORTH AMERICA

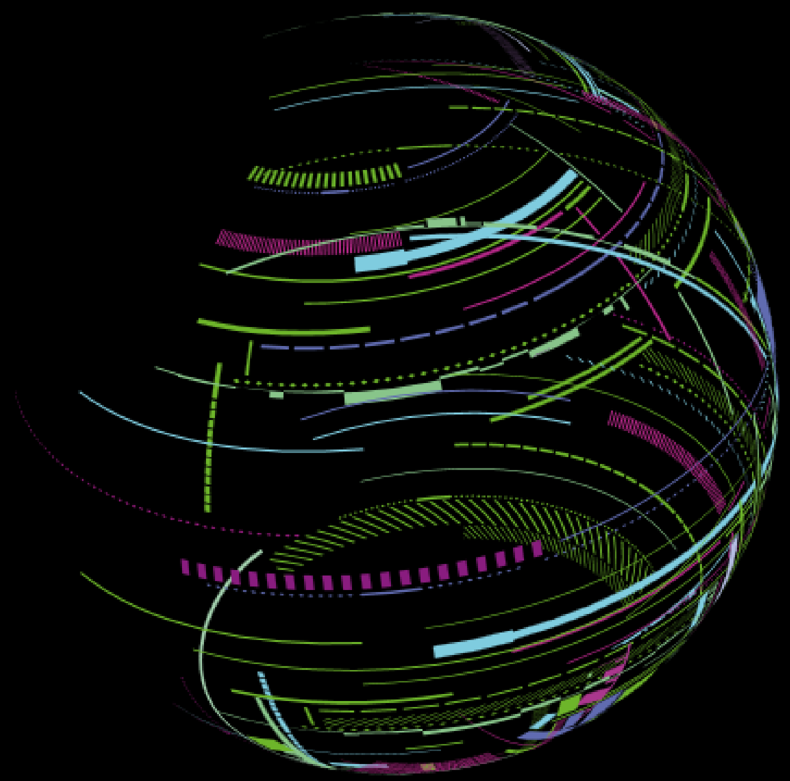
30 YEARS OF INNOVATION

Rank	Company name	Primary industry	% Growth	City	St./Prov.	CEO name
1	TG Therapeutics, Inc.	Life sciences	153,625%	New York	NY	Michael S. Weiss
2	Vytalize Health	Life sciences	126,668%	Hoboken	NJ	Faris Ghawi
3	Lessen	Software & services	39,691%	Scottsdale	AZ	Jay McKee
4	Zero Hash	Fintech	31,758%	Chicago	IL	Edward Woodford
5	Deel	Software & services	26,900%	San Francisco	CA	Alex Bouaziz
6	Talkiatry	Life sciences	17,718%	New York	NY	Robert Krayn
7	PurposeMed	Life sciences	16,169%	Calgary	AB	Husein Moloo
8	PrizePicks	Digital content/media/entertainment	16,119%	Atlanta	GA	Mike Ybarra
9	Autosled, Inc.	Software & services	14,001%	Rockville	MD	Dan Sperau
10	Cowbell	Fintech	13,458%	Pleasanton	CA	Jack Kudale
11	UniUni	Software & services	12,854%	Richmond	BC	Peter Lu
12	Odeko Inc.	Software & services	11,380%	New York	NY	Dane Atkinson
13	Observe	Software & services	10,977%	San Mateo	CA	Jeremy Burton
14	Trullion	Fintech	10,387%	New York	NY	Isaac Heller
15	Wisetack	Fintech	10,010%	San Francisco	CA	Bobby Tzekin
16	Relay	Software & services	9,578%	Toronto	ON	Yoseph West
17	Circle	Fintech	9,294%	Boston	MA	Jeremy Allaire
18	Uwill	Software & services	8,722%	Natick	MA	Michael London
19	Rad AI	Software & services	8,710%	San Francisco	CA	Doktor Gurson
20	Nurse	Software & services	8,676%	Murray	UT	Curtis Anderson
21	Crisp, Inc.	Software & services	8,640%	Brooklyn	NY	Are Traasdahl
22	Integrient	Fintech	8,520%	McLean	VA	Will Jerro
23	BoomerangFX Canada Inc.	Software & services	8,461%	Mississauga	ON	Jerome Dwight
24	Foghorn Therapeutics Inc.	Life sciences	7,843%	Cambridge	MA	Adrian Gottschalk
25	Alkira	Software & services	7,194%	San Jose	CA	Amir Khan
26	Skypoint	Software & services	7,114%	Beaverton	OR	Tisson Mathew
27	Cribl	Software & services	6,623%	San Francisco	CA	Clint Sharp
28	Oxygen8 Solutions Inc.	Energy & sustainability technology	6,610%	Vancouver	BC	James Dean
29	Interchecks Technologies, Inc.	Fintech	6,550%	Brooklyn	NY	Dylan Massey
30	Gamefam	Digital content/media/entertainment	6,390%	Los Angeles	CA	Joe Ferencz
31	GoTu	Software & services	6,282%	Miami	FL	Edward Thomas and Cary Gahm
32	Cyera	Software & services	6,178%	New York	NY	Yotam Segev
33	Archer Review	Software & services	5,771%	Dallas	TX	Dr. Karthik Koduru
34	Postal	Software & services	5,215%	San Luis Obispo	CA	Erik Kostelnik
35	Sharebite	Software & services	4,914%	New York	NY	Dilip Rao
36	Mersana Therapeutics, Inc.	Life sciences	4,351%	Cambridge	MA	Martin Huber, MD
37	VIVA Finance	Fintech	4,269%	Atlanta	GA	Jack Markwalter

2024 Technology Fast 500 Rankings | 2



Deloitte.

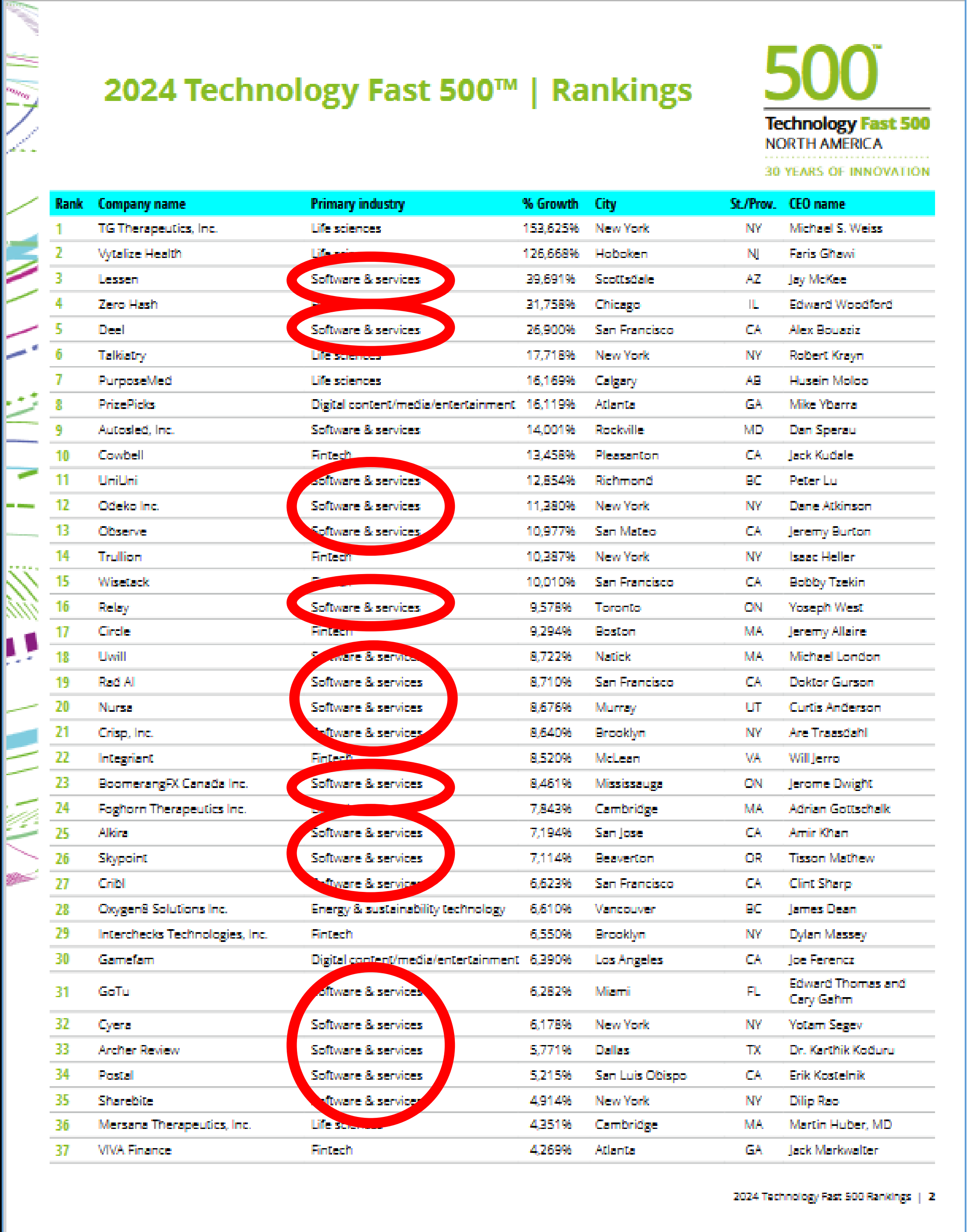


2024 Technology Fast 500™ Rankings

The fastest-growing technology and life sciences companies in North America

500™ Technology Fast 500 NORTH AMERICA 30 YEARS OF INNOVATION

Updated November 2024 v1.2



2024 Technology Fast 500™ | Rankings

500™ Technology Fast 500 NORTH AMERICA 30 YEARS OF INNOVATION

Rank	Company name	Primary industry	% Growth	City	St./Prov.	CEO name
1	TG Therapeutics, Inc.	Life sciences	153,625%	New York	NY	Michael S. Weiss
2	Vytalize Health	Life sciences	126,668%	Hoboken	NJ	Faris Ghawi
3	Lessen	Software & services	39,691%	Scottsdale	AZ	Jay McKee
4	Zero Hash	[illegible]	31,758%	Chicago	IL	Edward Woodford
5	Deel	Software & services	26,900%	San Francisco	CA	Alex Bouaziz
6	Talkiatry	Life sciences	17,718%	New York	NY	Robert Krayn
7	PurposeMed	Life sciences	16,169%	Calgary	AB	Husein Moloo
8	PrizePicks	Digital content/media/entertainment	16,119%	Atlanta	GA	Mike Ybarra
9	Autosled, Inc.	Software & services	14,001%	Rockville	MD	Dan Sperau
10	Cowbell	Fintech	13,458%	Pleasanton	CA	Jack Kudale
11	UniUni	Software & services	12,854%	Richmond	BC	Peter Lu
12	Odeko Inc.	Software & services	11,380%	New York	NY	Dane Atkinson
13	Observe	Software & services	10,977%	San Mateo	CA	Jeremy Burton
14	Trullion	Fintech	10,387%	New York	NY	Isaac Heller
15	Wisetack	Fintech	10,010%	San Francisco	CA	Bobby Tzekin
16	Relay	Software & services	9,578%	Toronto	ON	Yoseph West
17	Circle	Fintech	9,294%	Boston	MA	Jeremy Allaire
18	Uwill	Software & services	8,722%	Natick	MA	Michael London
19	Rad AI	Software & services	8,710%	San Francisco	CA	Doktor Gurson
20	Nurse	Software & services	8,676%	Murray	UT	Curtis Anderson
21	Crisp, Inc.	Software & services	8,640%	Brooklyn	NY	Are Traasdahl
22	Integriant	Fintech	8,520%	McLean	VA	Will Jerro
23	BoomerangFX Canada Inc.	Software & services	8,461%	Mississauga	ON	Jerome Dwight
24	Foghorn Therapeutics Inc.	Life sciences	7,843%	Cambridge	MA	Adrian Gottschalk
25	Alkira	Software & services	7,194%	San Jose	CA	Amir Khan
26	Skypoint	Software & services	7,114%	Beaverton	OR	Tisson Mathew
27	Cribl	Software & services	6,623%	San Francisco	CA	Clint Sharp
28	Oxygen8 Solutions Inc.	Energy & sustainability technology	6,610%	Vancouver	BC	James Dean
29	Interchecks Technologies, Inc.	Fintech	6,550%	Brooklyn	NY	Dylan Massey
30	Gamefam	Digital content/media/entertainment	6,390%	Los Angeles	CA	Joe Ferencz
31	GoTu	Software & services	6,282%	Miami	FL	Edward Thomas and Cary Gahm
32	Cyera	Software & services	6,178%	New York	NY	Yotam Segev
33	Archer Review	Software & services	5,771%	Dallas	TX	Dr. Karthik Koduru
34	Postal	Software & services	5,215%	San Luis Obispo	CA	Erik Kostelnik
35	Sharebite	Software & services	4,914%	New York	NY	Dilip Rao
36	Mersana Therapeutics, Inc.	Life sciences	4,351%	Cambridge	MA	Martin Huber, MD
37	VIVA Finance	Fintech	4,269%	Atlanta	GA	Jack Markwalter

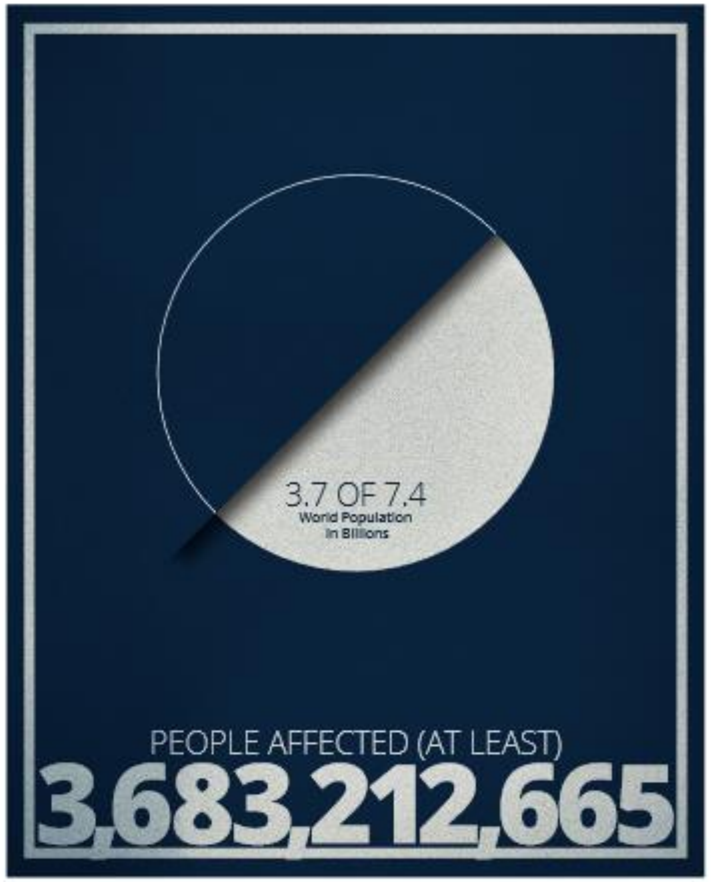
The Software Industry

- Value: $1.7 trillion+
- Industry is extremely dynamic and continually undergoing rapid change as new innovations appear. Unlike some other industries (e.g. transportation) it is still in many ways an immature industry. It does not, in general, have a set of quality standards that have been gained through years of hard-won experience

The Software Industry

- Value: $1.7 trillion+
- Industry is extremely dynamic and continually undergoing rapid change as new innovations appear. Unlike some other industries (e.g. transportation) it is still in many ways an immature industry. It does not, in general, have a set of quality standards that have been gained through years of hard-won experience
- Numerous examples exist of the results of failures in software quality and the costs it can incur

2017 Highlights

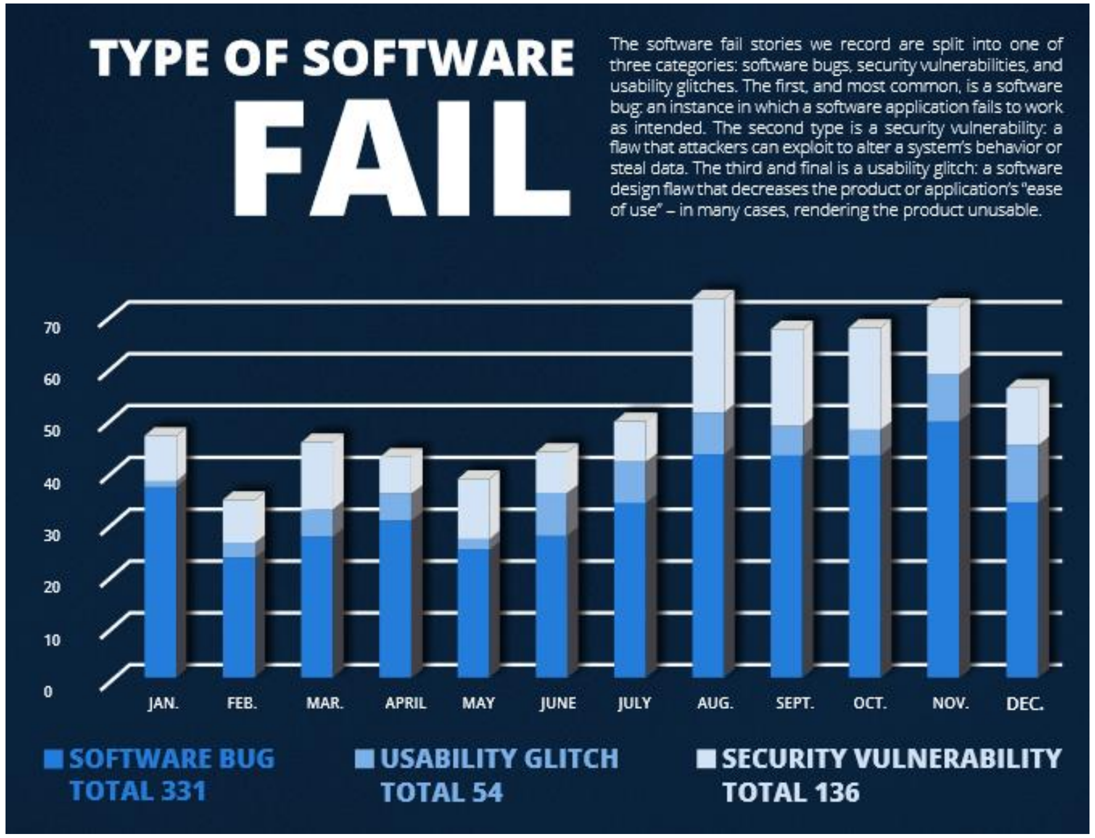


[TRICENTIS Software Fail Watch: 2016 and 2017]

LO$$E$ FROM SOFTWARE FAILURES (USD)

1,715,430,778,504

ONETRILLIONSEVENHUNDREDFIFTEENBILLIONFOURHUNDREDTHIRTYMILLIONSEVENHUNDREDSEVENTY-EIGHTTHOUSANDFIVEHUNDREDFOUR



The software fail stories we record are split into one of three categories: software bugs, security vulnerabilities, and usability glitches. The first, and most common, is a software bug: an instance in which a software application fails to work as intended. The second type is a security vulnerability: a flaw that attackers can exploit to alter a system's behavior or steal data. The third and final is a usability glitch: a software design flaw that decreases the product or application's "ease of use" – in many cases, rendering the product unusable.

Software Quality is Important

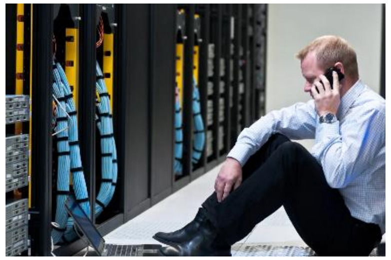
Knight Capital is in the process of being taken over after losing $461.1m in a trading software glitch

Business Computing World, 2012

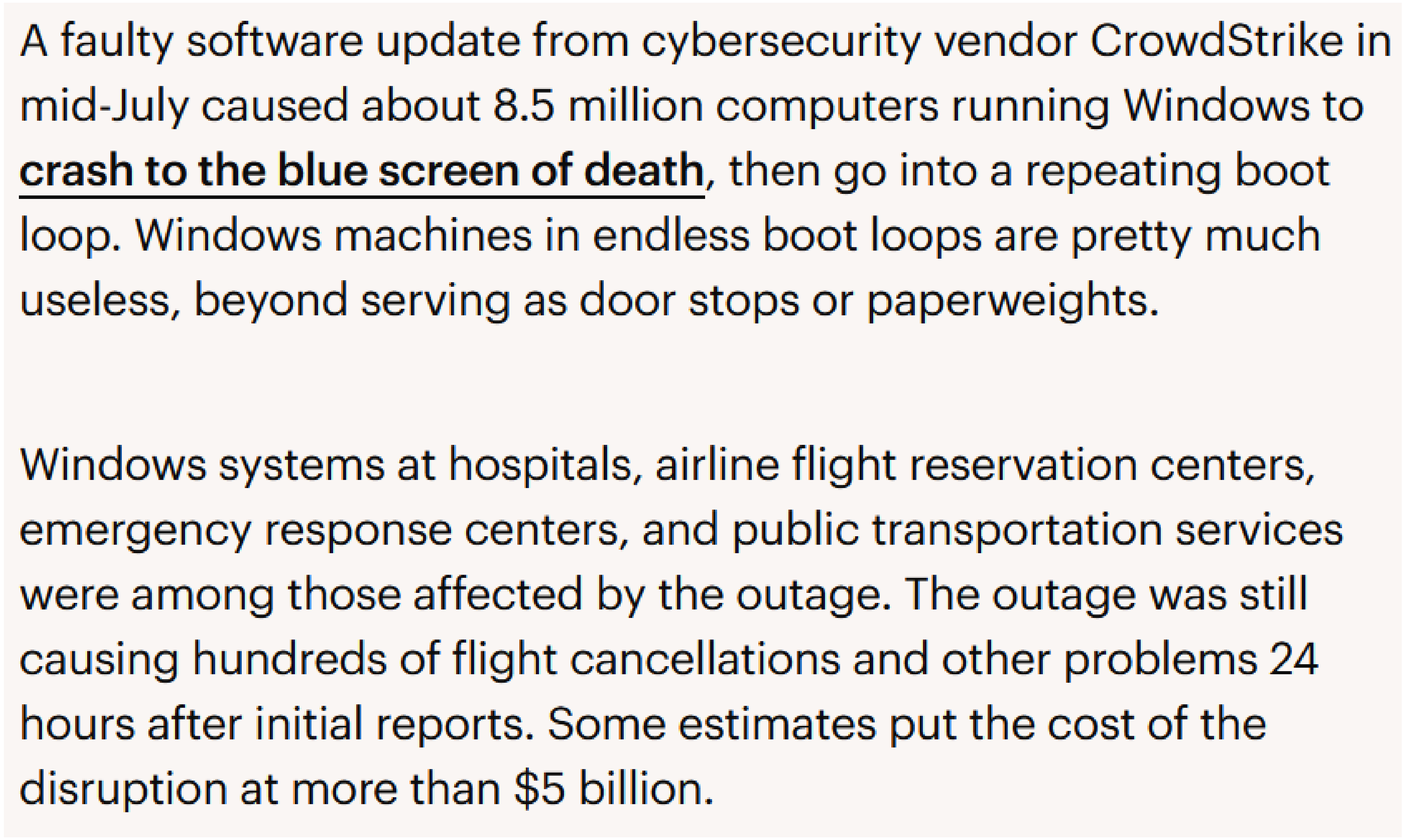
Formerly one of the largest electronic market makers on the New York Stock Exchange, Knight Capital suffered heavy losses when a computer glitch inadvertently led to the purchase of billions of dollars of shares in 148 stocks. Unwinding the mistaken positions cost $461m.

Financial Times, August 1, 2012

CrowdStrike (July 19, 2024)



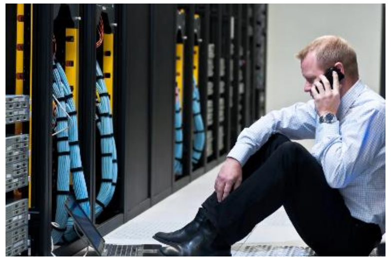
www.cio.com

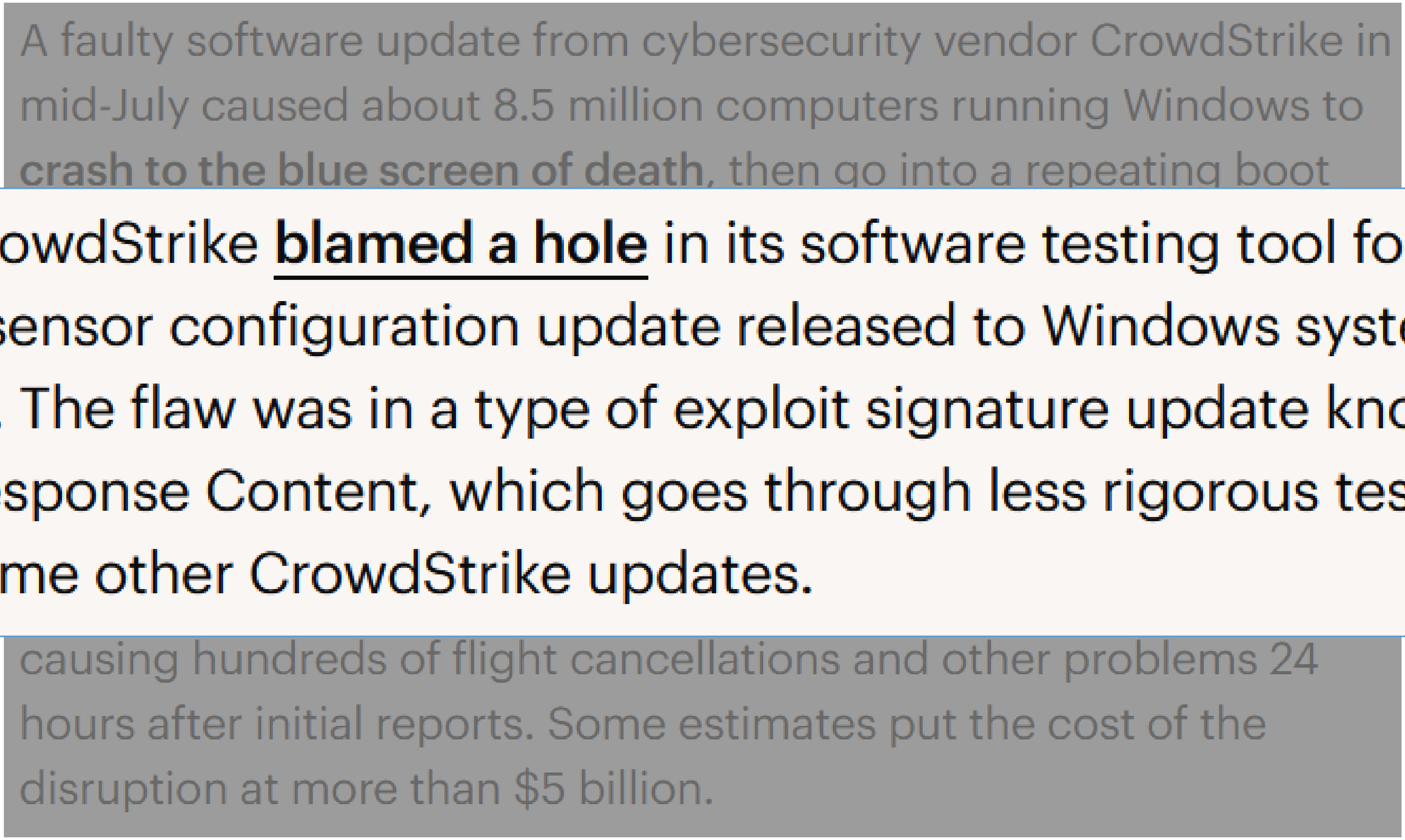


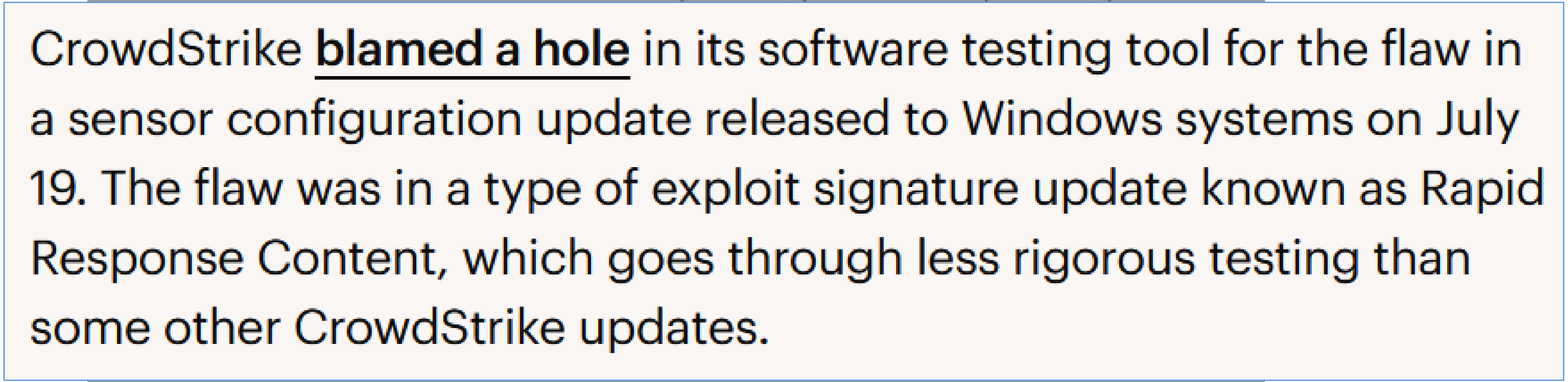
A faulty software update from cybersecurity vendor CrowdStrike in mid-July caused about 8.5 million computers running Windows to **crash to the blue screen of death**, then go into a repeating boot loop. Windows machines in endless boot loops are pretty much useless, beyond serving as door stops or paperweights.

Windows systems at hospitals, airline flight reservation centers, emergency response centers, and public transportation services were among those affected by the outage. The outage was still causing hundreds of flight cancellations and other problems 24 hours after initial reports. Some estimates put the cost of the disruption at more than $5 billion.

www.cio.com

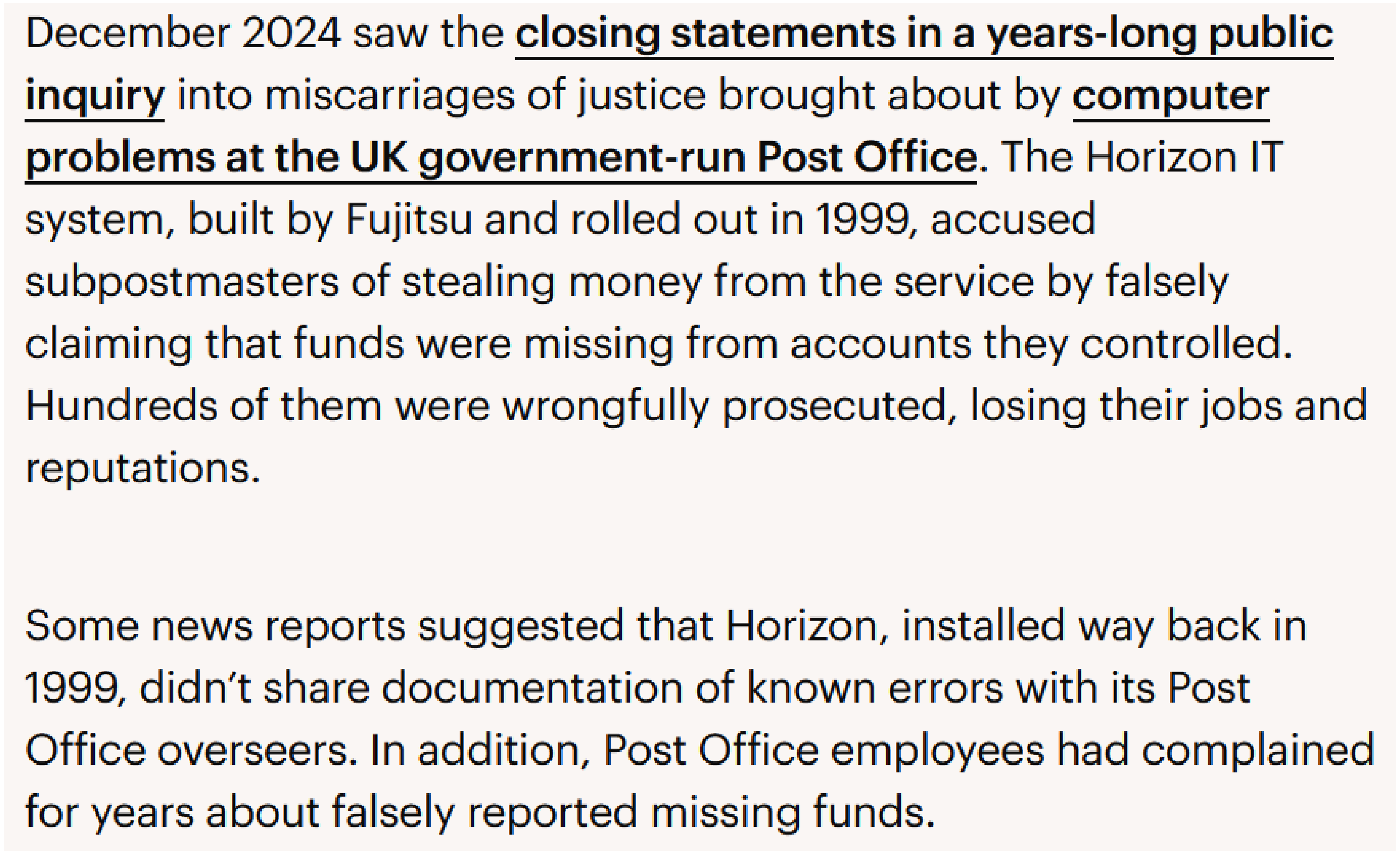

www.cio.com


A faulty software update from cybersecurity vendor CrowdStrike in mid-July caused about 8.5 million computers running Windows to **crash to the blue screen of death**, then go into a repeating boot


CrowdStrike **blamed a hole** in its software testing tool for the flaw in a sensor configuration update released to Windows systems on July 19. The flaw was in a type of exploit signature update known as Rapid Response Content, which goes through less rigorous testing than some other CrowdStrike updates.

causing hundreds of flight cancellations and other problems 24 hours after initial reports. Some estimates put the cost of the disruption at more than $5 billion.

UK Post Office/Horizon IT System



December 2024 saw the **closing statements in a years-long public inquiry** into miscarriages of justice brought about by **computer problems at the UK government-run Post Office**. The Horizon IT system, built by Fujitsu and rolled out in 1999, accused subpostmasters of stealing money from the service by falsely claiming that funds were missing from accounts they controlled. Hundreds of them were wrongfully prosecuted, losing their jobs and reputations.

Some news reports suggested that Horizon, installed way back in 1999, didn't share documentation of known errors with its Post Office overseers. In addition, Post Office employees had complained for years about falsely reported missing funds.

www.cio.com

More Examples

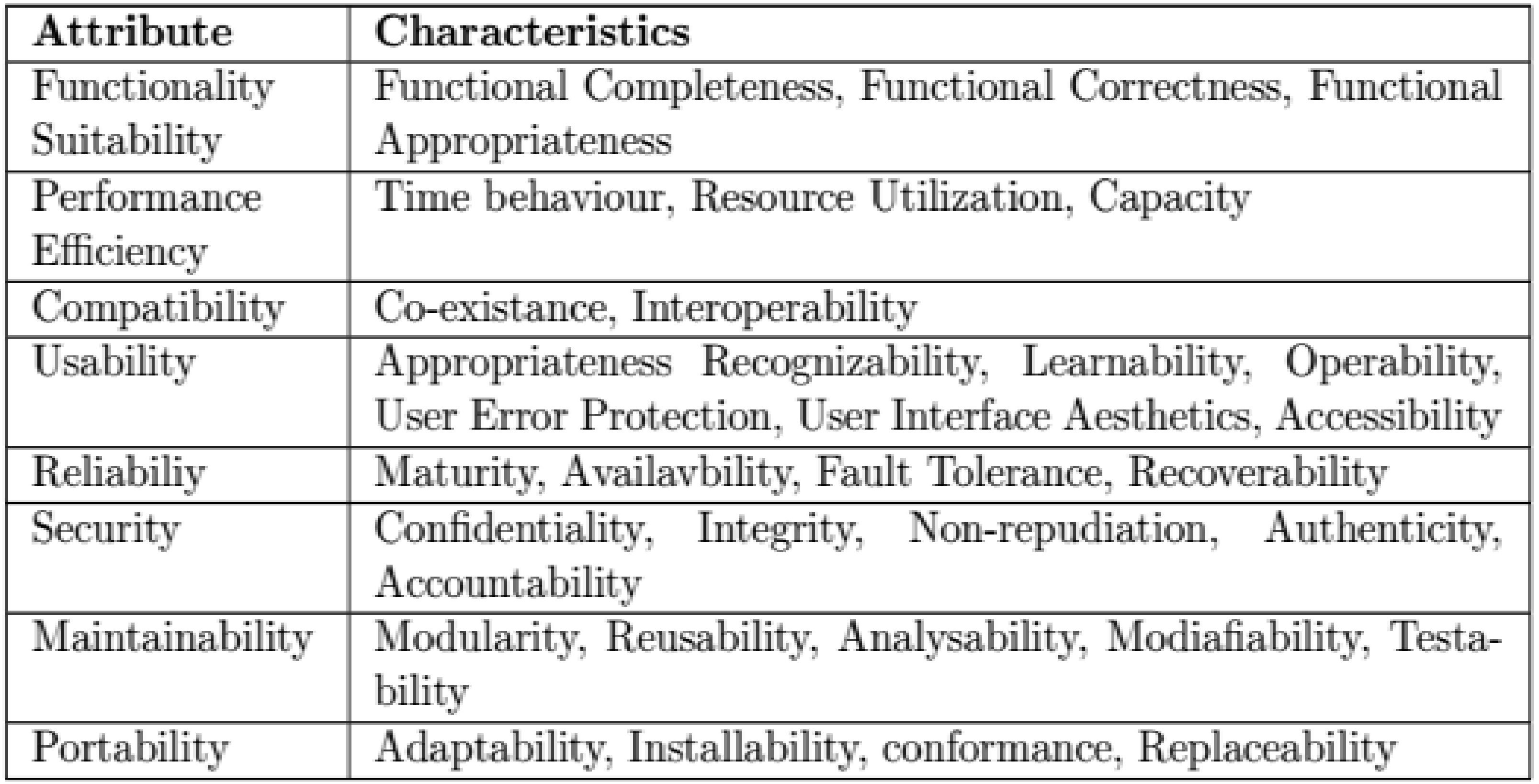
- Read up on these as independent study
  - Tesla recall, Tik-Tok zero followers, Log4j web server vulnerability, iPhone 15 Pro overheating, British Postal Service/Horizon (Fujitsu) scandal
- And find more examples yourself
- Make sure they are well authenticated
  - e.g. official reports, newspapers, journals, research papers
  - You can use web searches, Wikipedia, and blogs to **find** them – but then find a more reliable source to **confirm** the details

Software Testing and Quality

- Software testing reduces the **risks** associated with software development
- Modern programs often:
  - Are very complex
  - Have millions of lines of code
  - Have multiple interactions with other software systems
- Causes of inadequate software testing, leading to reduced quality:
  - Imprecise requirements; external deadlines; budget pressures
- Poor quality=>software failures=>increased development costs=>delays
- Business problems: loss of reputation; reduced market share; legal claims

Software Quality (ISO/IEC 25010)

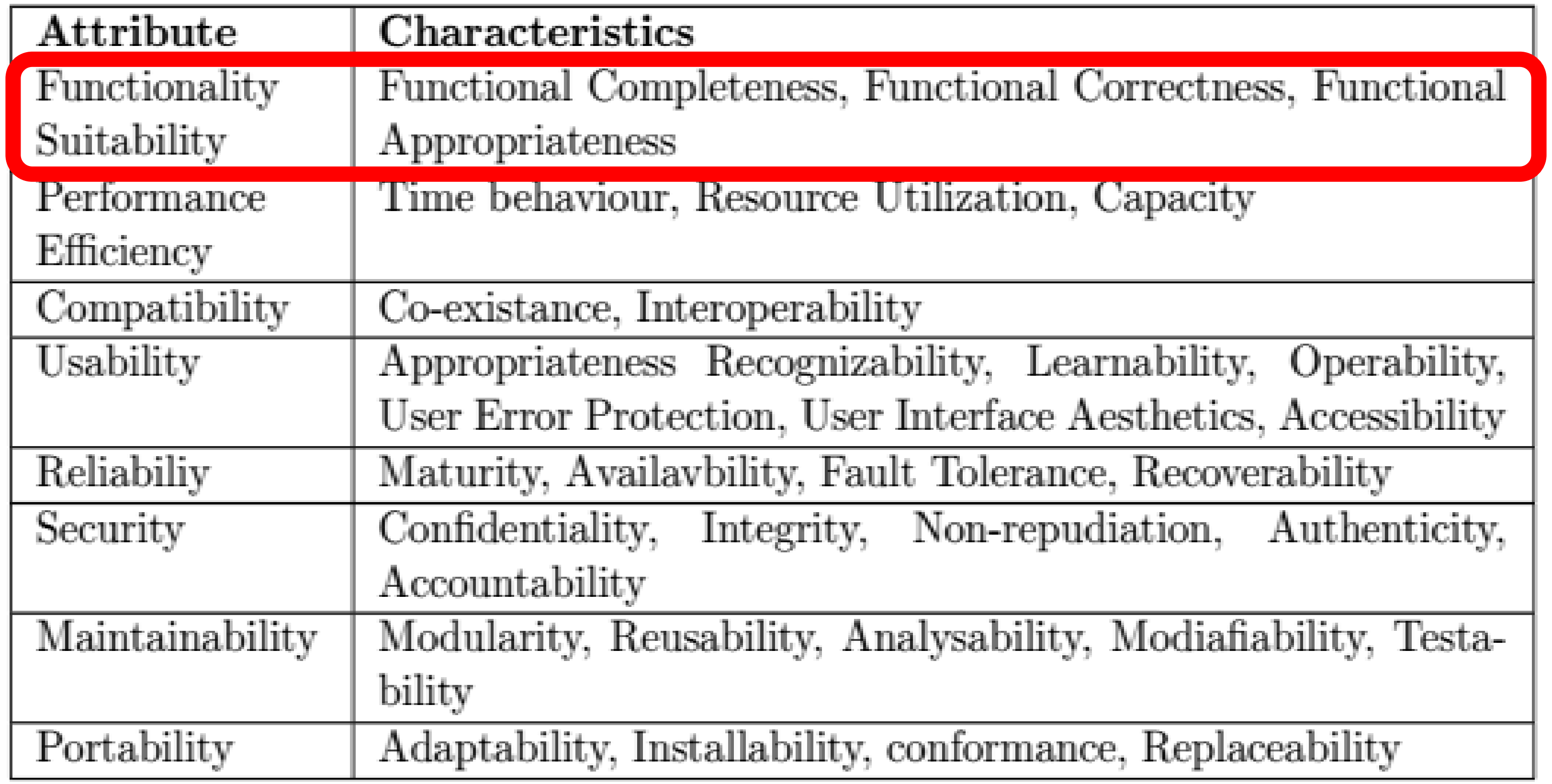
- International standard
- Product quality model
- 8 quality characteristics
- Software/product quality is a large topic
- This course focusses on **Functionality**



Attribute	**Characteristics**
Functionality Suitability	Functional Completeness, Functional Correctness, Functional Appropriateness
Performance Efficiency	Time behaviour, Resource Utilization, Capacity
Compatibility	Co-existance, Interoperability
Usability	Appropriateness Recognizability, Learnability, Operability, User Error Protection, User Interface Aesthetics, Accessibility
Reliabiliy	Maturity, Availavbility, Fault Tolerance, Recoverability
Security	Confidentiality, Integrity, Non-repudiation, Authenticity, Accountability
Maintainability	Modularity, Reusability, Analysability, Modiafiability, Testa-bility
Portability	Adaptability, Installability, conformance, Replaceability

Software Quality (ISO/IEC 25010)

- International standard
- Product quality model
- 8 quality characteristics
- Software/product quality is a large topic
- This course focusses on **Functionality**



Attribute	Characteristics
Functionality Suitability	Functional Completeness, Functional Correctness, Functional Appropriateness
Performance Efficiency	Time behaviour, Resource Utilization, Capacity
Compatibility	Co-existance, Interoperability
Usability	Appropriateness Recognizability, Learnability, Operability, User Error Protection, User Interface Aesthetics, Accessibility
Reliabiliy	Maturity, Availavbility, Fault Tolerance, Recoverability
Security	Confidentiality, Integrity, Non-repudiation, Authenticity, Accountability
Maintainability	Modularity, Reusability, Analysability, Modiafiability, Testa-bility
Portability	Adaptability, Installability, conformance, Replaceability

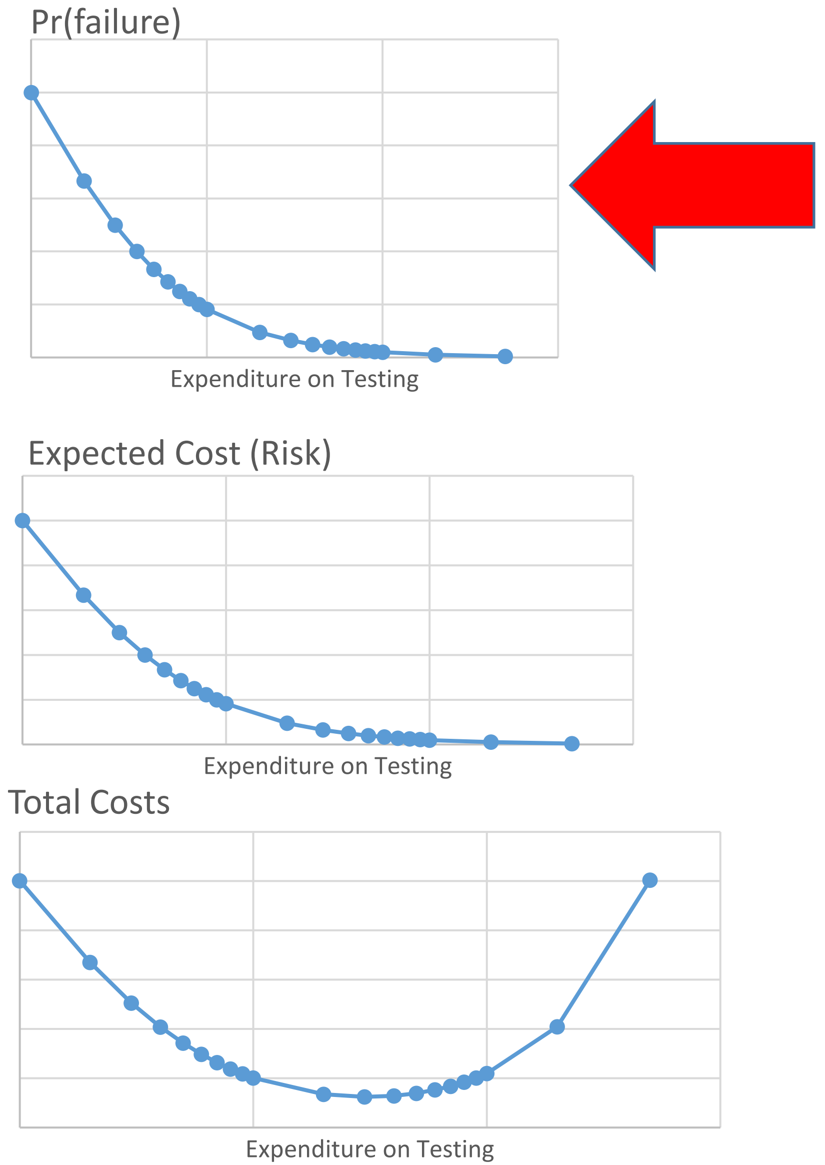
Software Testing and Risk Management

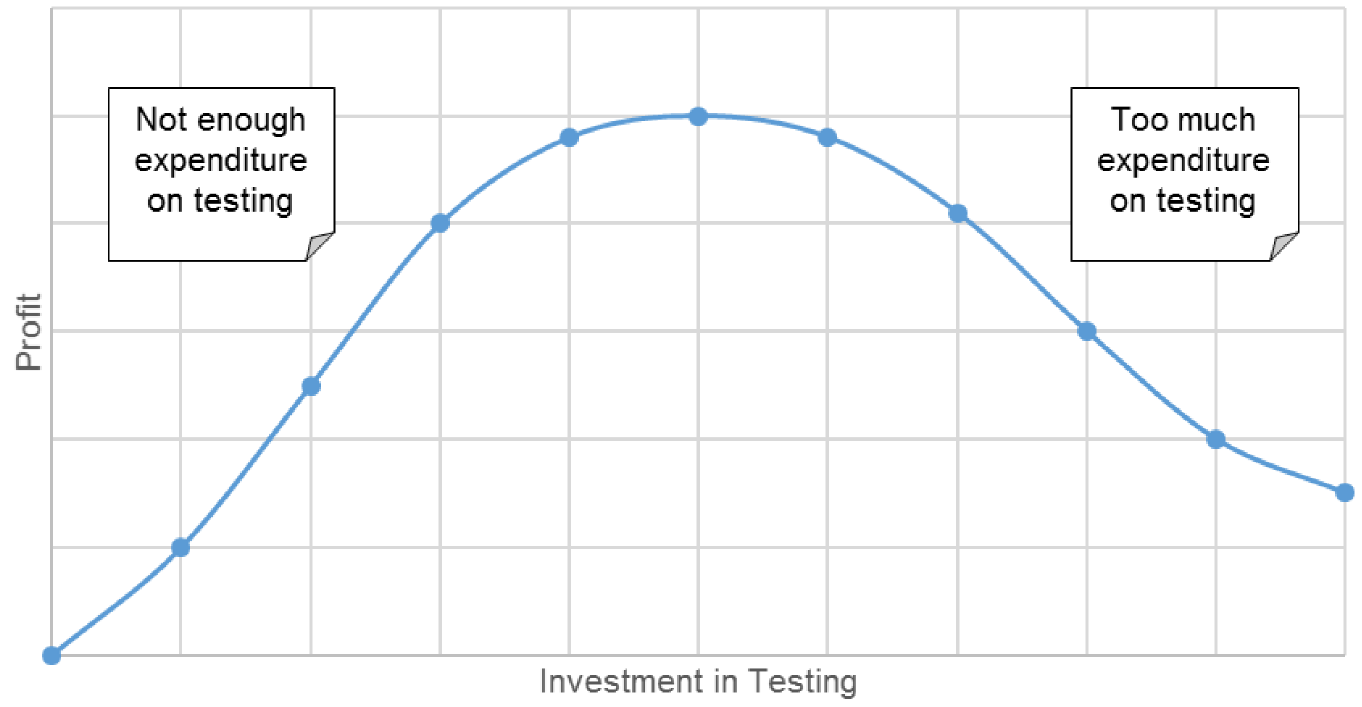
- Testing costs money
- But so do software failures
- One financial measure of risk is the extra expected cost associated with low quality software

*Expected-cost = probability(failure) * cost(failure)*

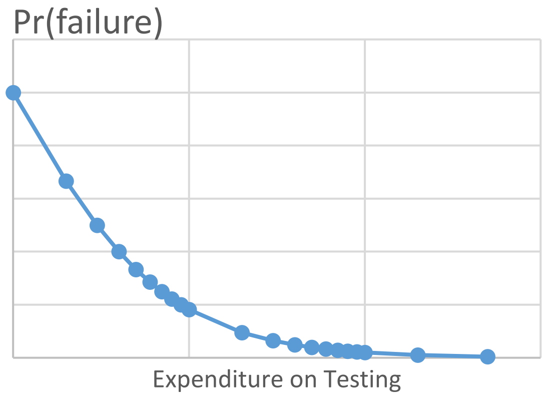
- Each business has optimal point to minimize costs & maximize profit

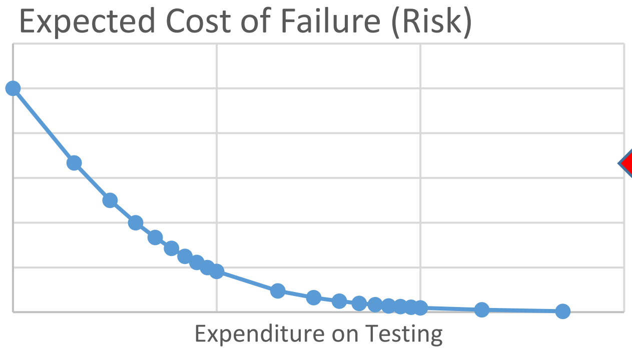
Software Testing and Risk Management

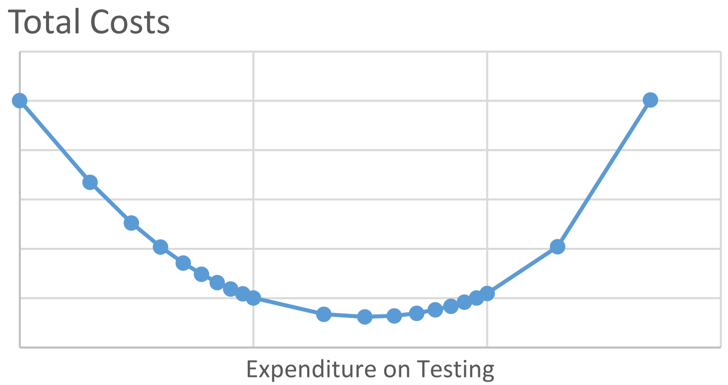


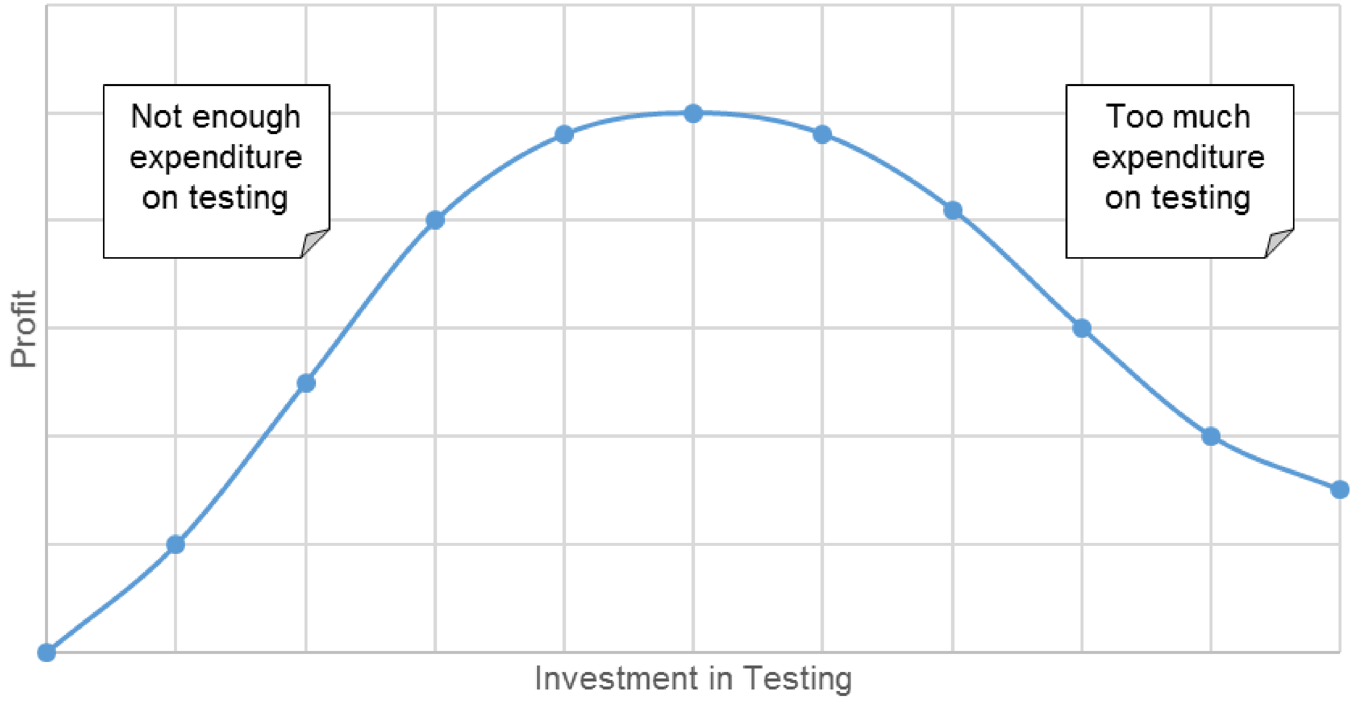


Software Testing and Risk Management

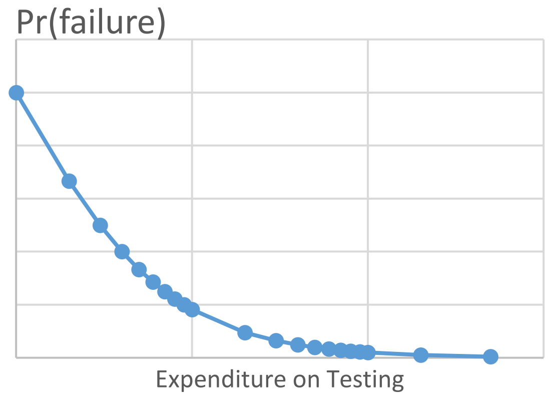


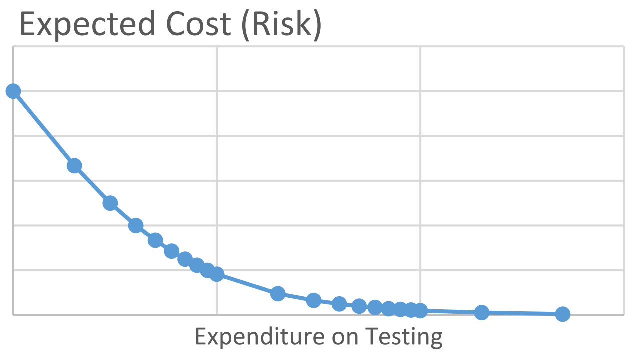


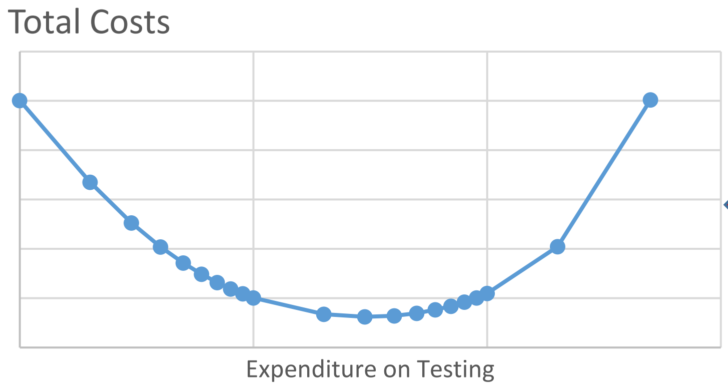


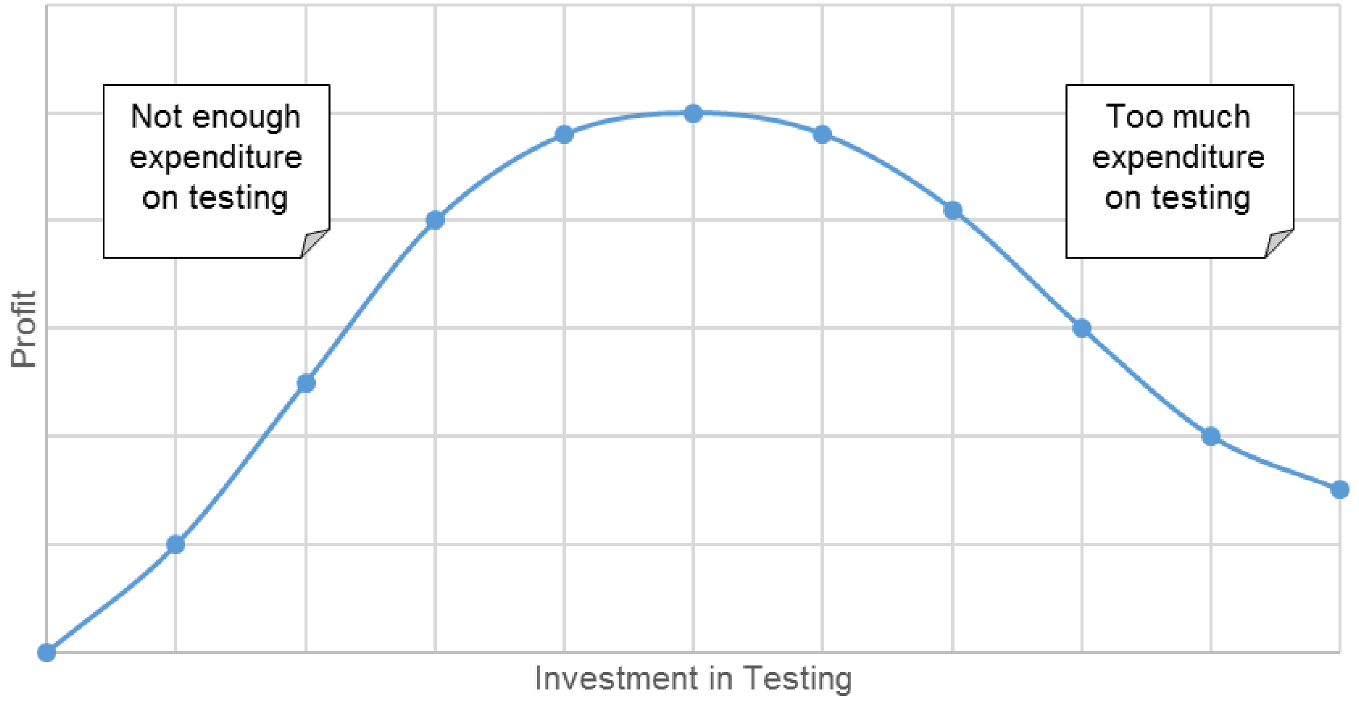


Software Testing and Risk Management

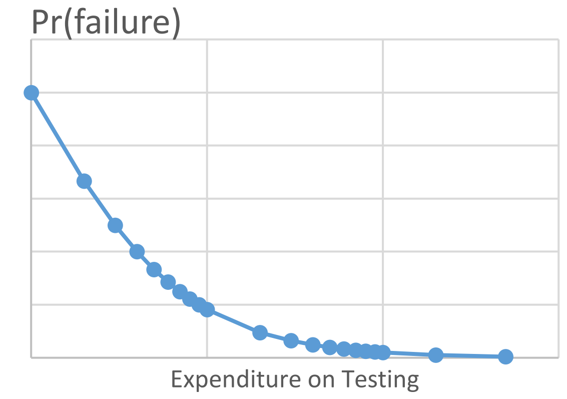


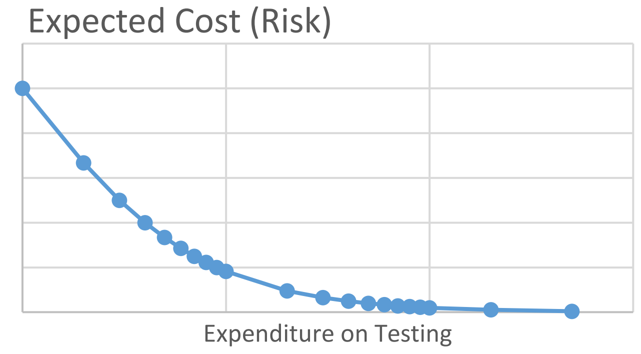


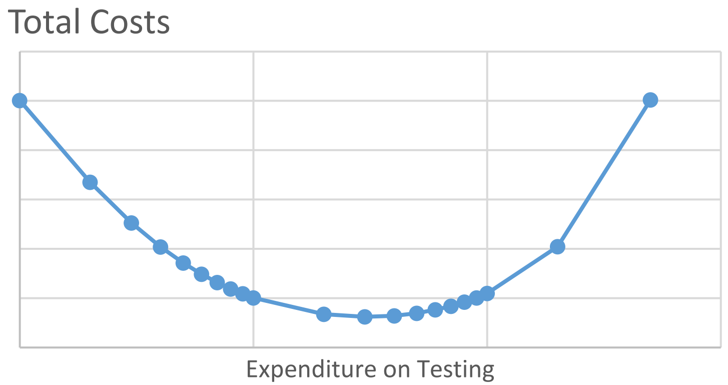


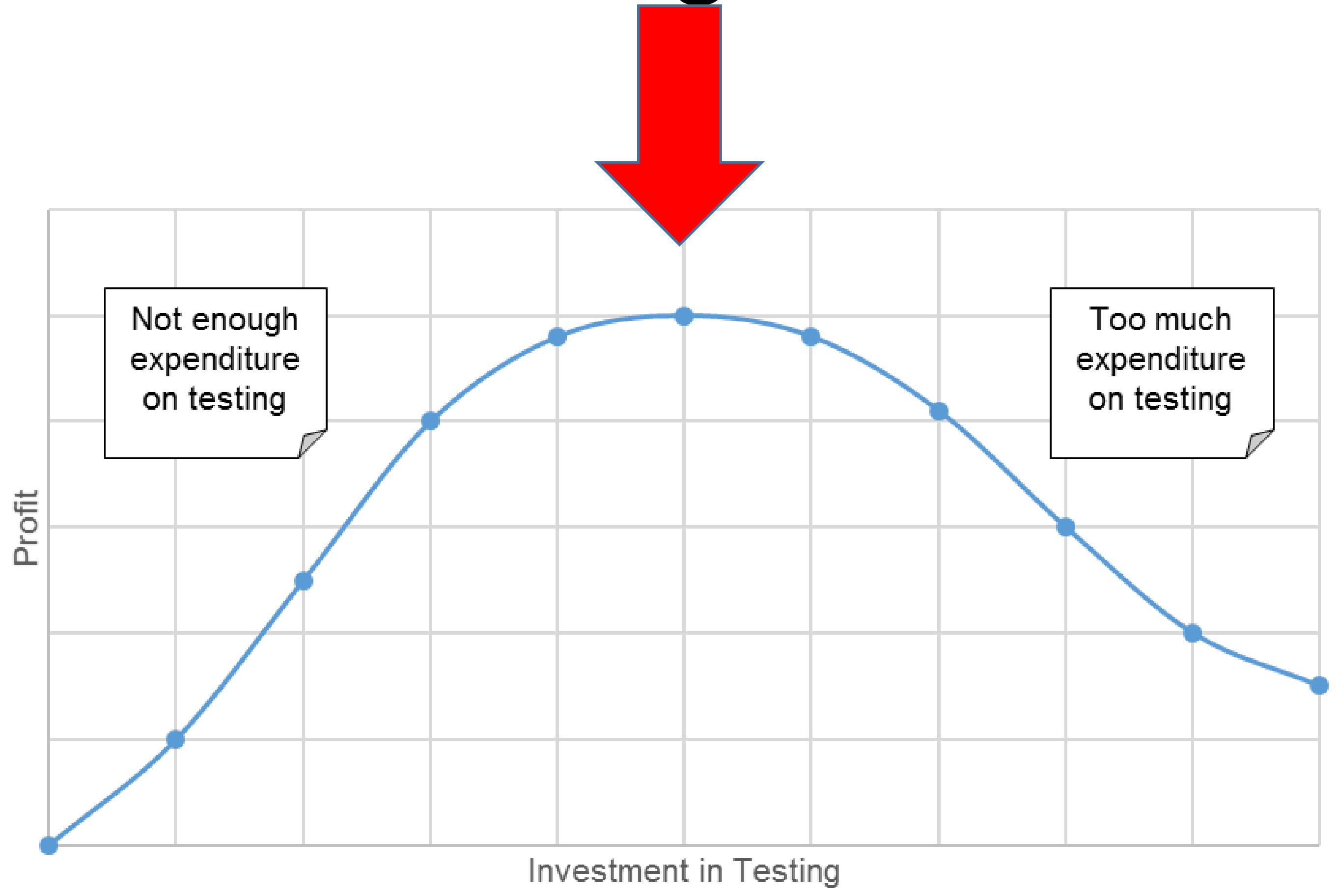


Software Testing and Risk Management



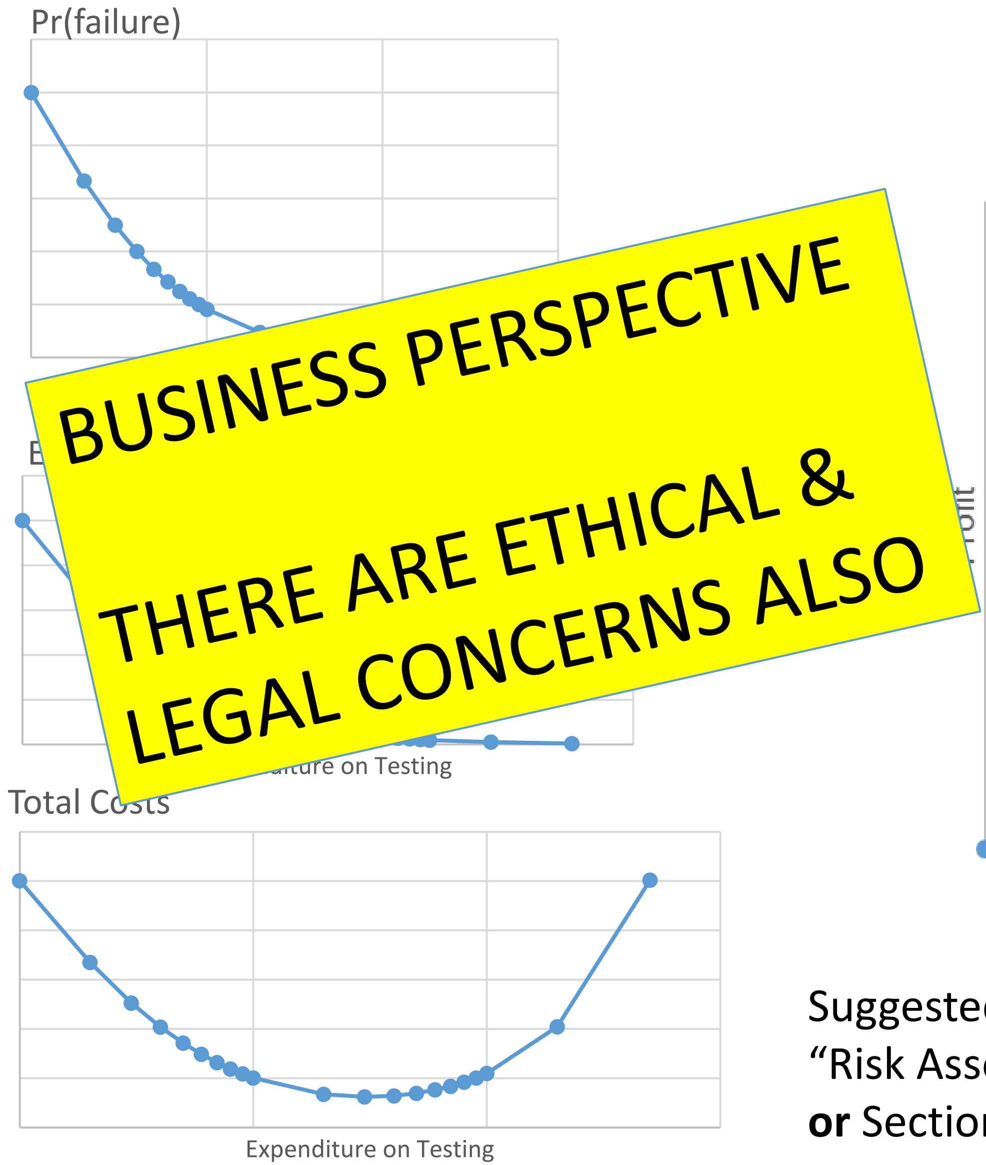






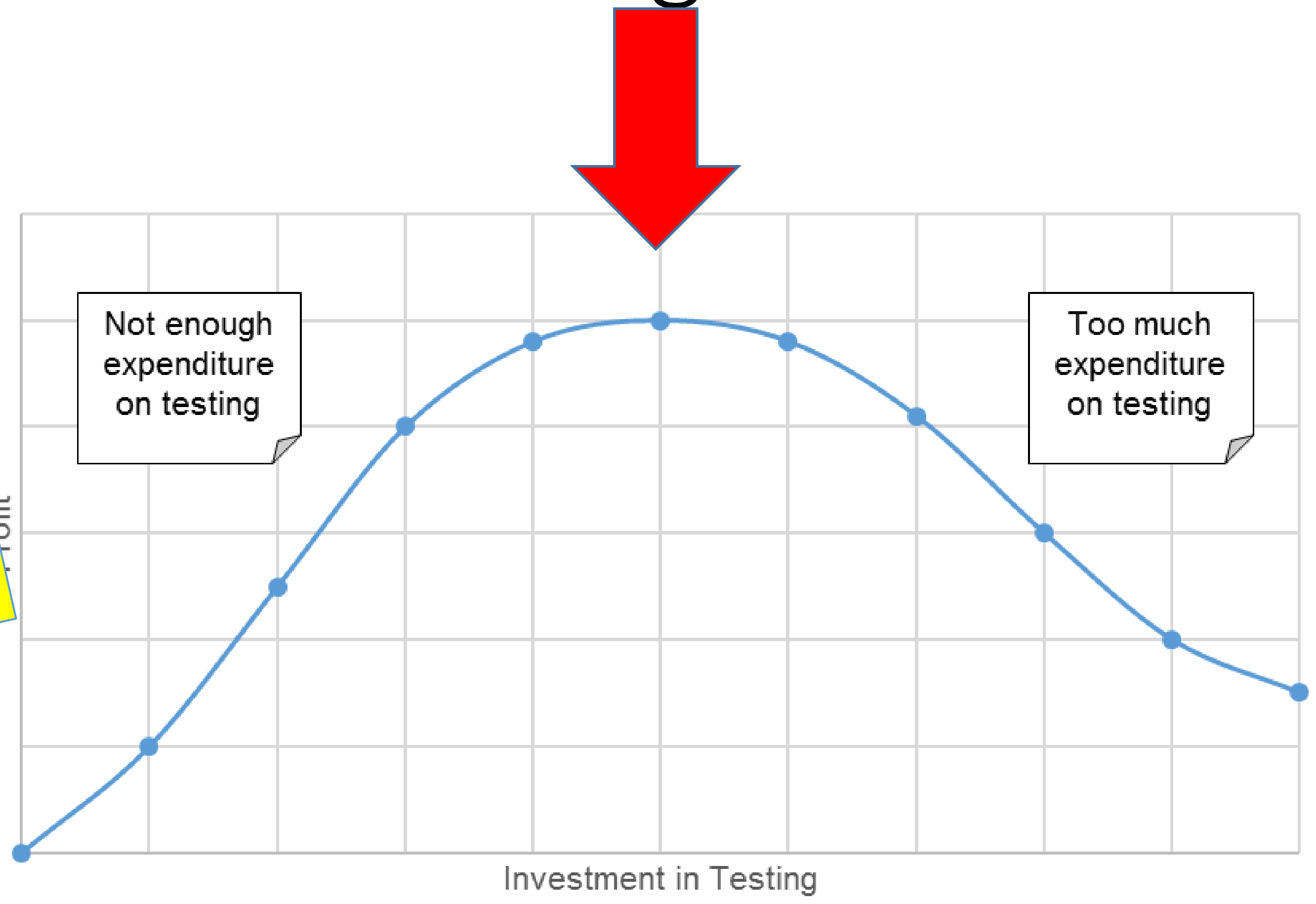
Suggested reading:
"Risk Assessment", Chapter 186, *The Engineering Handbook*, 1st Edn, CRC PRESS 1996 **or** Section XXVIII in 2nd Edition, 2008

Software Testing and Risk Management



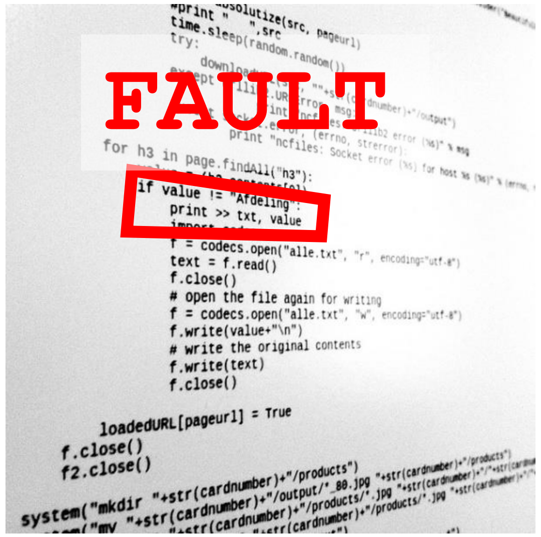
BUSINESS PERSPECTIVE

THERE ARE ETHICAL & LEGAL CONCERNS ALSO



Suggested reading:
"Risk Assessment", Chapter 186, *The Engineering Handbook*, 1st Edn, CRC PRESS 1996 **or** Section XXVIII in 2nd Edition, 2008

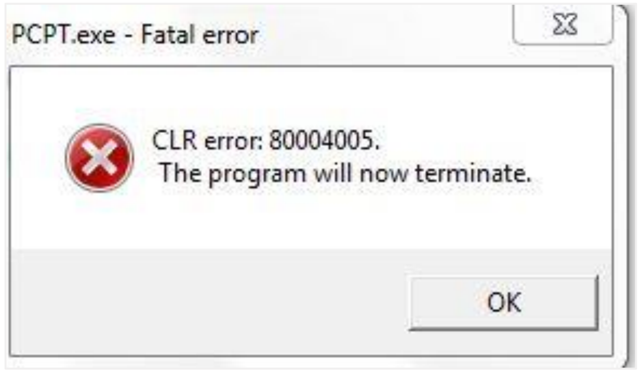
Mistakes, Faults & Failures



```
#print "   ",src
time.sleep(random.random())
try:
    download[illegible]
except [illegible]
        print "ncfiles: Socket error (%s) for host %s (%s)" % (errno, [illegible]
for h3 in page.findAll("h3"):
    if value != "Afdeling":
        print >> txt, value
        f = codecs.open("alle.txt", "r", encoding="utf-8")
        text = f.read()
        f.close()
        # open the file again for writing
        f = codecs.open("alle.txt", "w", encoding="utf-8")
        f.write(value+"\n")
        # write the original contents
        f.write(text)
        f.close()

    loadedURL[pageurl] = True
f.close()
f2.close()
system("mkdir "+str(cardnumber)+"/products")
```

FAILURE



PCPT.exe - Fatal error

CLR error: 80004005.
The program will now terminate.

OK

Mistakes, Faults & Failures

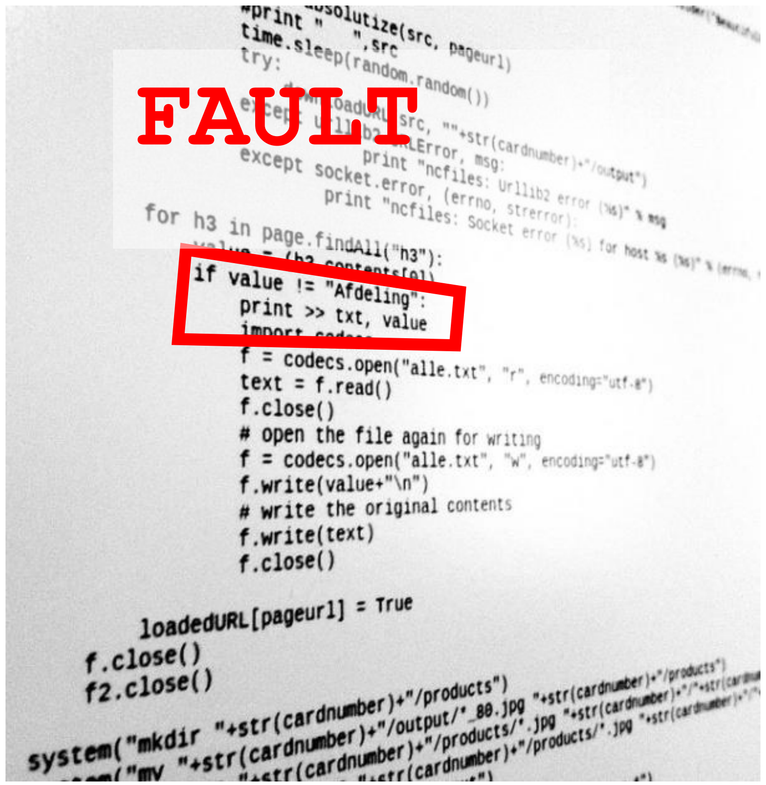
- **Mistakes**: these are made by software developers. They exist in the mind, and can result in one or more faults in the source code.

Mistakes

- Misunderstanding in communication
  - Poor documentation; imprecise discussion; ambiguous statements
  - Example: Confuse metric & imperial measurements
- Misinterpretation
  - Lack of attention; lack of experience; lack of knowledge
  - Example: swap over the order of parameters in a method call
- Incorrect Assumptions
  - Example:
    - Assuming default values (e.g. Java int=0 vs C++ which has no default value)

Mistakes, **Faults** & Failures

- **Faults**: these are flaws in the source code, and can be the product of one or more mistakes. Faults can lead to failures during program execution.



```
#print "      absolutize(src, pageurl)
time.sleep(random.random())
",src
try:
    except urllib2.URLError, msg:
        print "ncfiles: Urllib2 error (%s)" % msg
    except socket.error, (errno, strerror):
        print "ncfiles: Socket error (%s) for host %s (%s)" % (errno,
for h3 in page.findAll("h3"):
    if value != "Afdeling":
        print >> txt, value
        f = codecs.open("alle.txt", "r", encoding="utf-8")
        text = f.read()
        f.close()
        # open the file again for writing
        f = codecs.open("alle.txt", "w", encoding="utf-8")
        f.write(value+"\n")
        # write the original contents
        f.write(text)
        f.close()

    loadedURL[pageurl] = True
  f.close()
  f2.close()

system("mkdir "+str(cardnumber)+"/products")
```

Faults

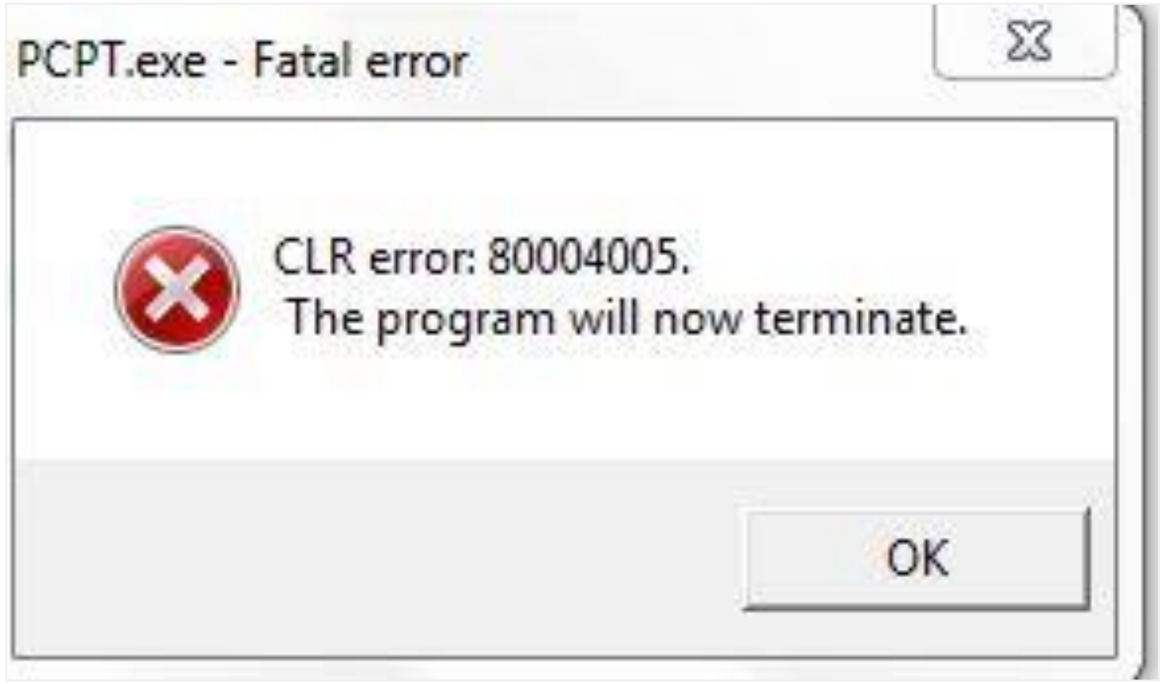
- Faults exist in the source code
- They are the result of a mistake by the developer
- And may lead to failures during execution
- Fault classification can be used:
  - When analysing, designing, or coding software: a checklist of faults to avoid
  - When developing software tests: as a guide to likely faults
  - When undergoing software process evaluation or improvement: input

Fault Classification

- Classification of faults can be useful in preventing future faults
- Examples:
  - **Algorithmic:** Software algorithm produces wrong output
  - **Computation and Precision:** Result not to the expected accuracy or precision
  - **Stress or Overload:** System fails when overloaded
  - **Capacity and Boundary:** System fails when data stores overflow
  - **Timing or Co-ordination:** Failure of concurrent activity (multi-threaded or real time)
  - **Throughput or Performance:** System fails to meet performance requirements
  - **Recovery:** System does not recover as expected after a fault is detected and corrected
  - **Standards and Procedure:** A team member does not follow the standards making it difficult for other members to extend the system or solve problems
  - Also: **documentation**, **syntax**

Mistakes, Faults & **Failures**

- **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. When a failure is detected by software, it is often indicated by an error code.

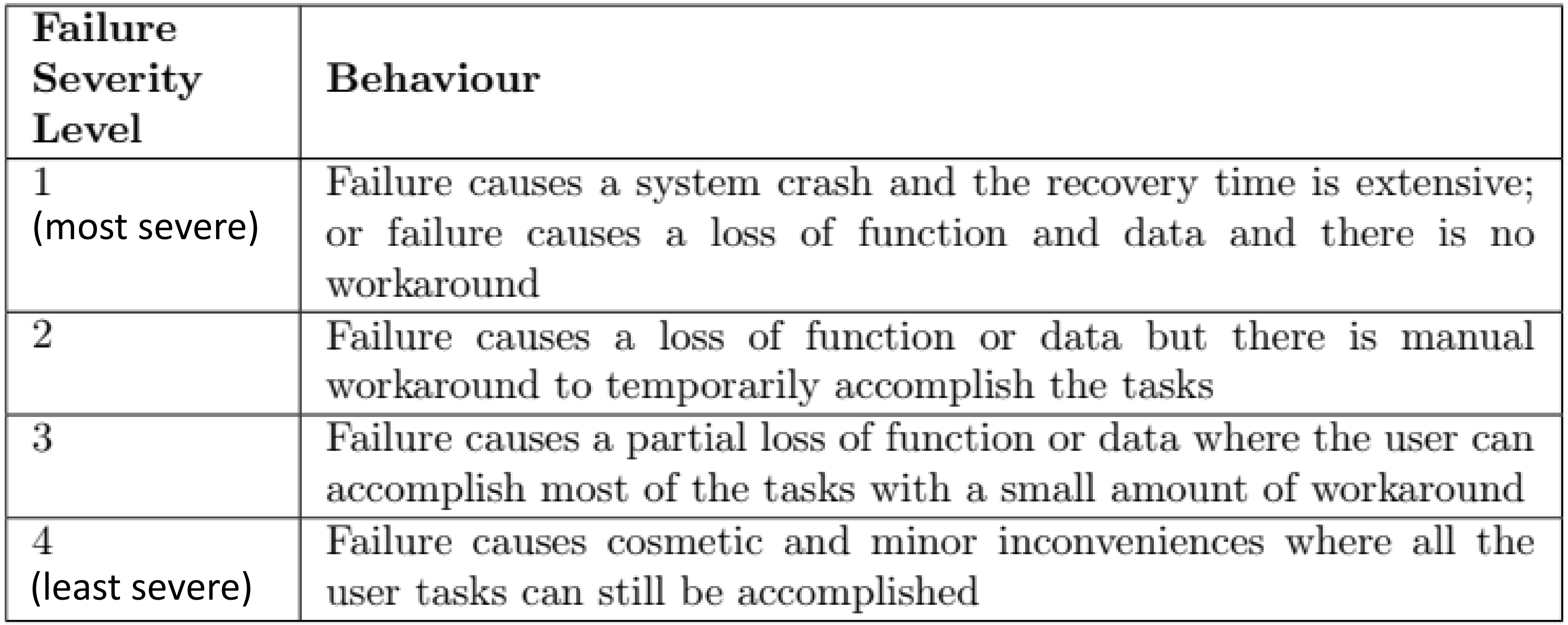

PCPT.exe - Fatal error

CLR error: 80004005.
The program will now terminate.

Software Failures

- Classifying the severity of failures that result from particular faults is more difficult because of their subjective nature, particularly with failures that do not result in a program crash
- One user may regard a particular failure as being very serious, while another may not feel as strongly about it

Example Classification of Software Failures



Failure Severity Level	**Behaviour**
1 (most severe)	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 the user can accomplish most of the tasks with a small amount of workaround
4 (least severe)	Failure causes cosmetic and minor inconveniences where all the user tasks can still be accomplished

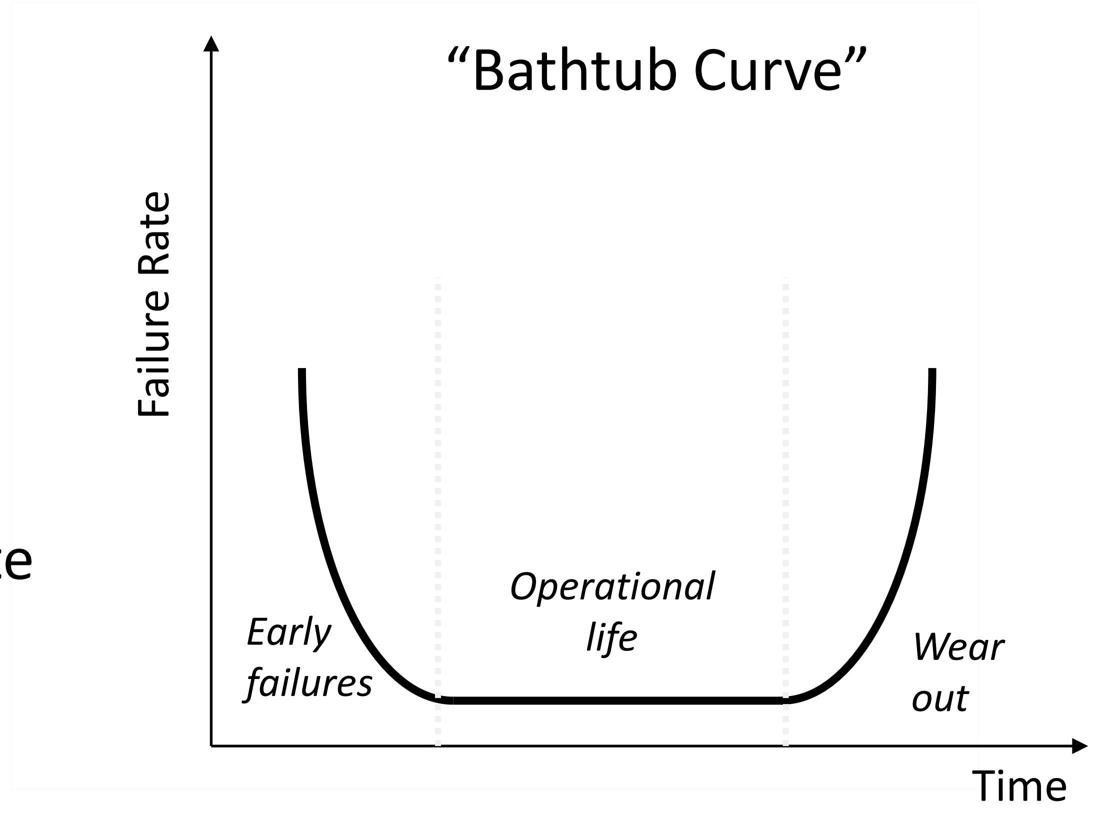
Industry Figures

- It is very difficult to find industry figures relating to software faults and failures
- One example is a study by Hewlett Packard on the frequency of occurrence of various fault types, which found that 50% of the faults analysed were either *Algorithmic* or *Computation and Precision*

Hardware Failures

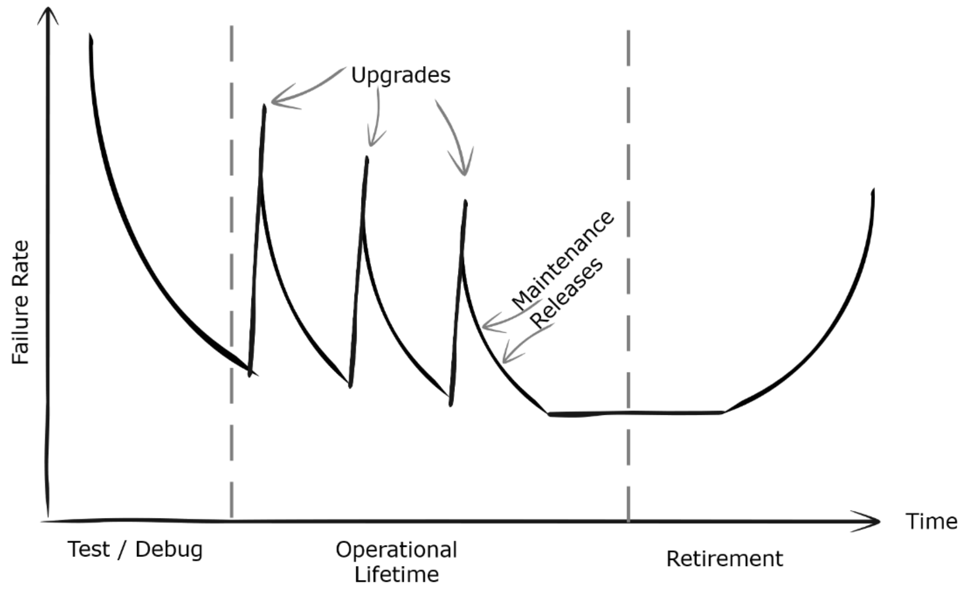
- Typical “bathtub curve”
  - High failure rate initially
  - Followed by relatively low failures
  - Eventually failure rate rises again
- The early failures are caused by manufacturing issues, and handling and installation errors
- As these are ironed out, during the main operational life of a product, the failure rate stays low
- Eventually, however, hardware ages and wears out, and the failure rates rise again

- Most consumer products follow this curve.

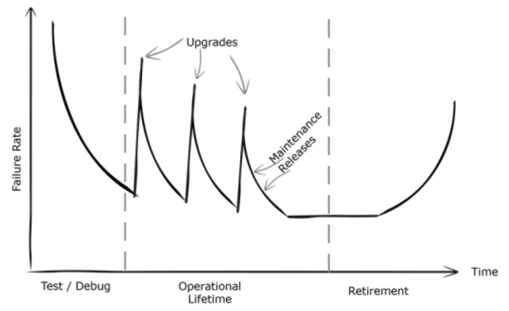


Software Failures

- Software failures demonstrate a similar pattern, but for different reasons as software does not physically wear out
- During initial development, the failure rate is likely to fall rapidly during the test and debug cycle
- During operational lifetime, upgrades tend to introduce new failures or expose latent faults
- Maintenance releases lower the rate
- Finally software is retired (no longer maintained)
- Eventually changes to the environment, or to software dependencies, may lead to higher failure rates again



Applicable to Modern, Agile Environments



- With continuous integration, new software features added on a regular basis, and software is frequently redesigned (referred to as refactoring)
- The changes introduced by this rapid rate of upgrades are likely to lead to new faults being introduced
- And most software is eventually replaced, or ceases to be maintained as a supported version, leaving existing users with an eventual rise in the failure rate. Examples:
  - Python 2 libraries not compatible with Python 3
  - Support for Windows 7 ended recently, so a rise in failure rate to be expected until these systems are replaced with Windows 10 (and now 11)

Need for Testing

- Why does software have faults?

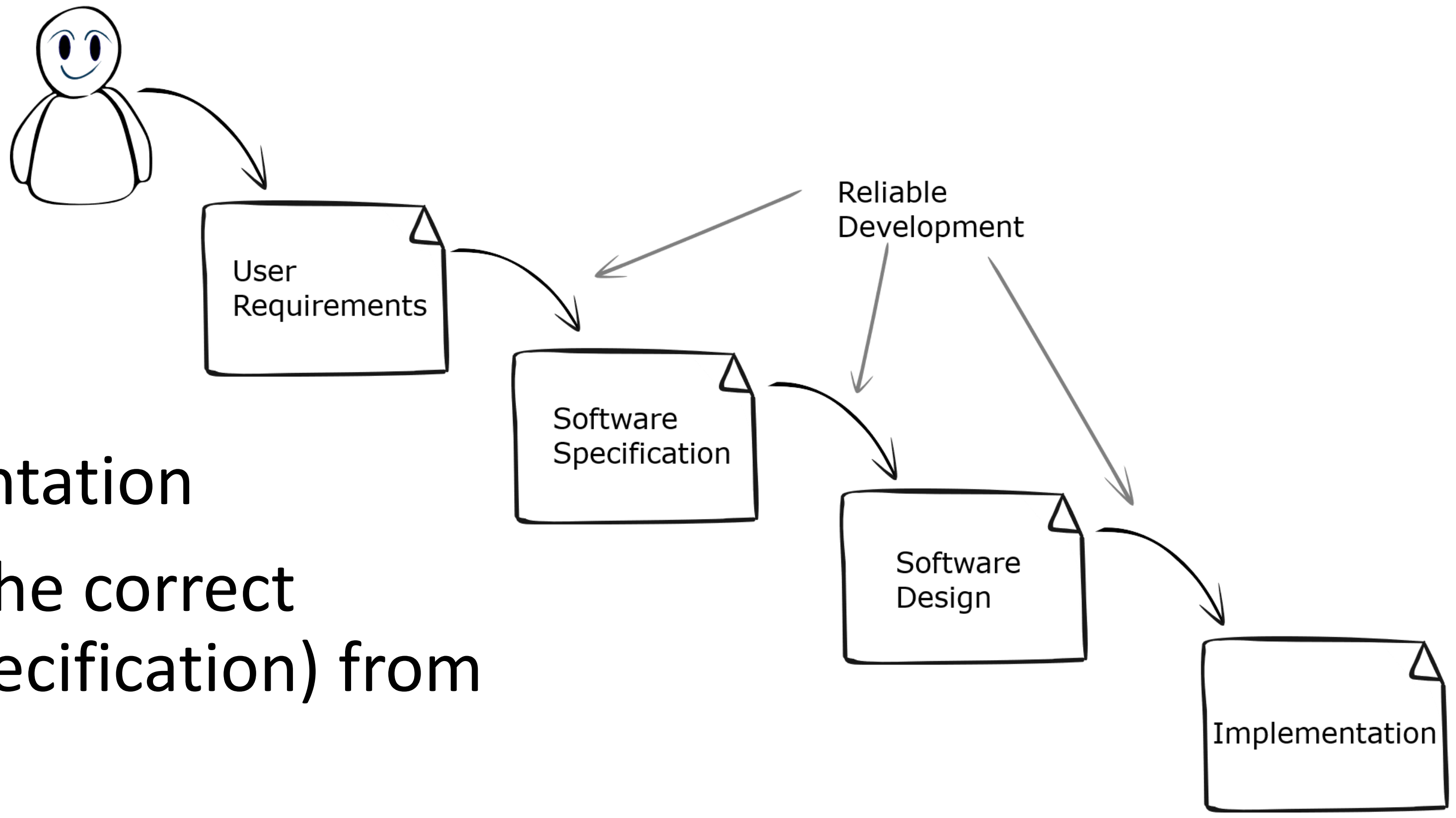
Need for Testing

- Why does software have faults?
- It is difficult to:
  - Collect the user requirements correctly
  - Specify the required software behaviour correctly
  - Design software correctly
  - Implement software correctly
  - Modify software correctly

Engineering Approaches for Correct Systems

- Forward Engineering
  - “Get it right first time”
- Feedback-based
  - “Keep fixing it until it works correctly”
  - Verification: make sure each step in the process has been done correctly
  - Validation: make sure the implementation meets the user’s needs

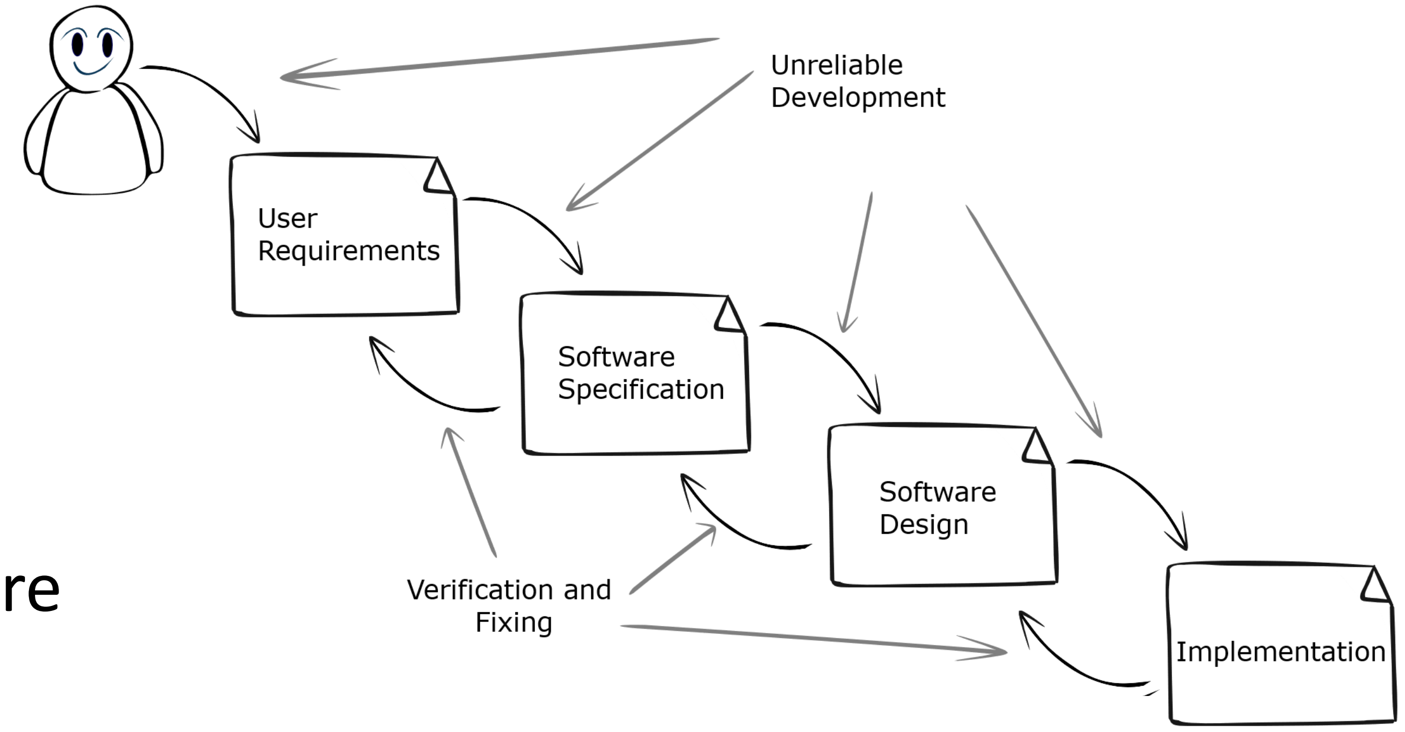
Ideal Project with Forward Engineering



- From User to Correct Implementation
- Reliable: each activity creates the correct outputs based on its inputs (specification) from the previous activity
- The end product thereby matches its specification and meets the user’s needs

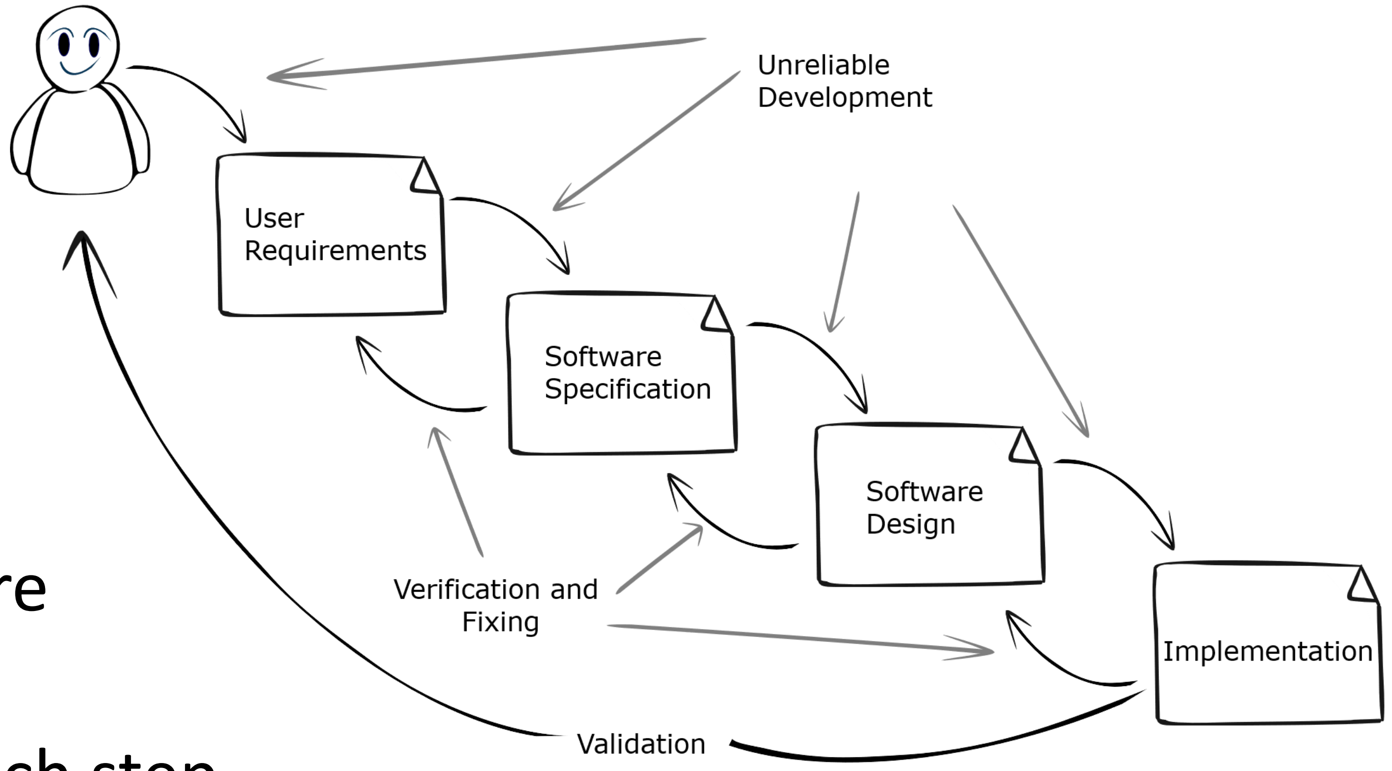
Realistic Project with Verification & Validation

- In practice, subject to mistakes and ambiguities, leading to incorrect results
- To resolve this, each step is checked
- Opportunity to fix mistakes before proceeding
- **Verification** and fixing follows each step



Realistic Project with Verification & Validation

- In practice, subject to mistakes and ambiguities, leading to incorrect results
- To resolve this, each step is checked
- Opportunity to fix mistakes before proceeding
- **Verification** and fixing follows each step
- **Validation** is performed on the final implementation against the user's needs



The Role of Specifications

- Definitions
- Their importance

Software Specification

- **Specification**

  “A document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or component and often the procedures for determining whether these provisions have been satisfied.”

  [NIST SP 800-37]

Software Specification

- **Specification**

  “A document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or component and often the procedures for determining whether these provisions have been satisfied.”

  [NIST SP 800-37]

- **Software Requirements Specification**

  “Structured collection of the requirements (functions, performance, design constraints, and attributes) of the software and its external interfaces”

  [ISO/IEC/IEEE Standard 24198]

The Role of Specifications

- A software specification identifies the **precise and detailed behaviour** that the software is required to provide
- It gives a model of what the software is supposed to do
- These specifications play a key role in testing. In order to be tested, the correct behaviour of software must be known
- This implies the need for detailed specifications (or software requirements)
- Note: *software requirements* are different from *user requirements*
    - User requirements = what the user wants
    - Software requirements = what the software should do to meet the user requirements

The Role of Specifications

- A software specification identifies the precise and detailed behaviour that the software is required to provide
- It gives a **model** of what the software is supposed to do
- These specifications play a key role in testing. In order to be tested, the correct behaviour of software must be known
- This implies the need for detailed specifications (or software requirements)
- Note: *software requirements* are different from *user requirements*
  - User requirements = what the user wants
  - Software requirements=what the software should do to meet the user requirements

The Role of Specifications

- A software specification identifies the precise and detailed behaviour that the software is required to provide
- It gives a model of what the software is supposed to do
- These specifications play a key role in testing. In order to be tested, the **correct behaviour of software must be known**
- This implies the need for detailed specifications (or software requirements)
- Note: *software requirements* are different from *user requirements*
  - User requirements = what the user wants
  - Software requirements=what the software should do to meet the user requirements

The Role of Specifications

- A software specification identifies the precise and detailed behaviour that the software is required to provide
- It gives a model of what the software is supposed to do
- These specifications play a key role in testing. In order to be tested, the correct behaviour of software must be known
- This implies the need for **detailed specifications** (or software requirements)
- Note: *software requirements* are different from *user requirements*
    - User requirements = what the user wants
    - Software requirements=what the software should do to meet the user requirements

The Role of Specifications

- A software specification identifies the precise and detailed behaviour that the software is required to provide
- It gives a model of what the software is supposed to do
- These specifications play a key role in testing. In order to be tested, the correct behaviour of software must be known
- This implies the need for detailed specifications (or software requirements)
- Note: *software requirements* are different from *user requirements*
    - **User requirements** = what the user wants
    - **Software requirements** = what the software should do to meet user requirements

The Role of Specifications

- Specifications play a key role in testing
- In order to be tested, the correct behaviour of software must be known both under ***normal*** and ***error*** conditions

The Role of Specifications

- Specifications play a key role in testing
- In order to be tested, the correct behaviour of software must be known both under ***normal*** and ***error*** conditions
- Testing based on “reasonable" or “expected” behaviour of software is unreliable: not every developer, tester, or user will have the same expectations
    - Invalid input: ignore it? return error? raise an exception? write to error log?
    - Should temperature be in in degrees Centigrade(°C), Kelvin(°K), or Fahrenheit(°F)?
    - Is zero a positive number? negative number? or neither?
    - Angle have a value of 0°? Shape have an area of 0 cm^2? Or negative area?

Need Exact Specifications for Testing

- In general terms, the expected behaviour of a program in handling errors in the input data is to indicate that this has occurred: return error code or raise exception
- To test the software, the **exact** details of this must be defined
- These specify exactly what an error in the input is, and how such errors should be notified to the caller
- **To test software properly, detailed specifications are necessary**
- Often a tester will need to convert a specification from a “written English“ form to one that is easier to create tests from

A First Look at Software Testing

A First Look at Software Testing

- Manual Test Example
- The Theory of Software Testing
- Test Heuristics (Random Testing, Black-box and White-box Testing, Experience-based Testing, Fault Insertion)
- When to Finish Testing
- Static and Dynamic Testing (including Program Proving)
- Testing in the Software Development Process
- Software Testing Activities
- Test Artefacts
- Fault Models

Manual Test Example

- Consider a discount program for an online shop
- Customers collect bonus points
- Customers can sign up to be gold customers
- The program (“check”) determines whether a customer should get a price discount on their next purchase, based on their bonus points to date, and whether they are a gold customer

Specifications

- Bonus points accumulate as customers make purchases
- Gold customers get a discount once they have exceeded 80 bonus points
- Other customers only get a discount once they have more than 120 bonus points
- All customers always have at least one bonus point

Manual Test Example

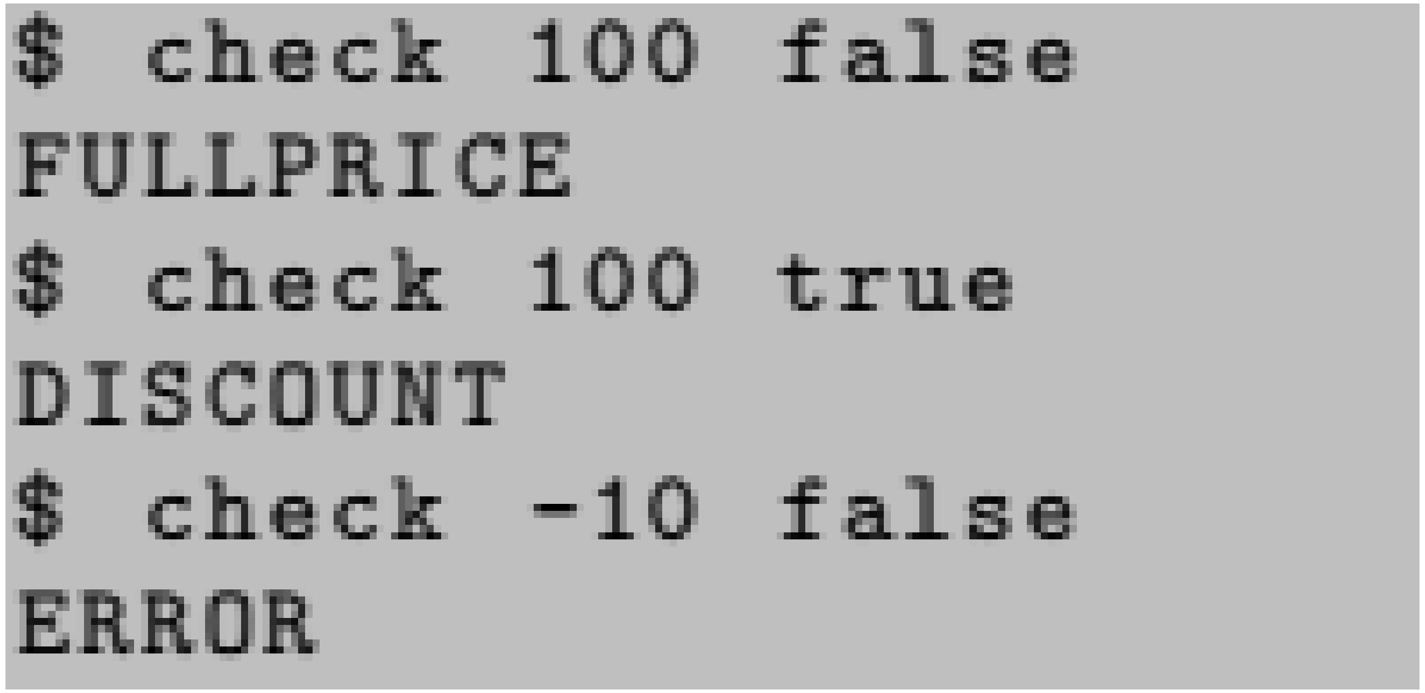
- The program returns one of the following:
  - **FULLPRICE** which indicates that the customer should be charged the full price
  - **DISCOUNT** which indicates that the customer should be given a discount.
  - **ERROR** which indicates an error in the inputs
    (bonusPoints is invalid if it is less than 1)

Manual Test Example

- The inputs to the program are:
  - the number of bonusPoints the customer has
  - a flag indicating whether the customer is a gold customer (true) or not (false)
- For simplicity, we will ignore illegal input values at the moment:
  - the first input is not a number
  - the second input is not a boolean

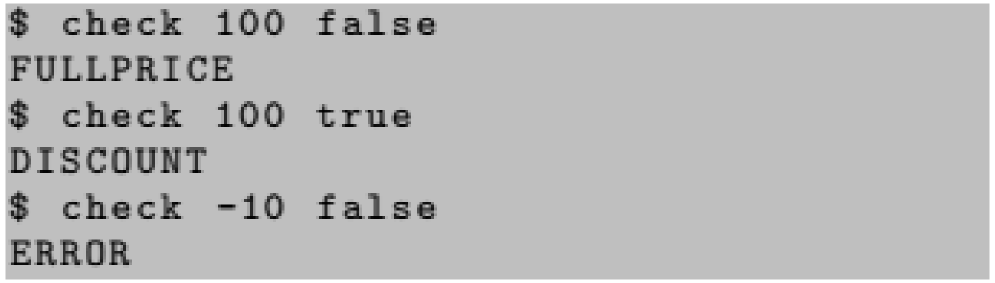
Manual Test Example

DEMO



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

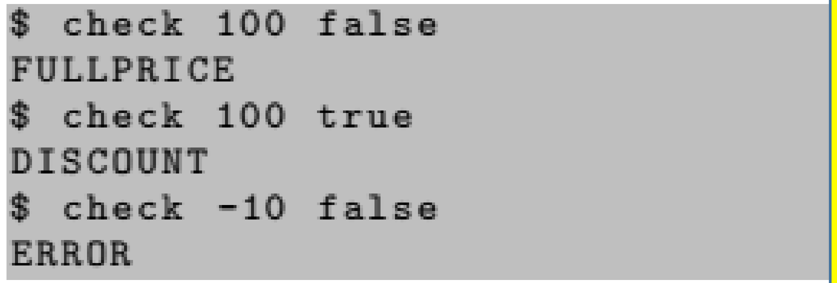
Is the Program Correct?



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

- These results look correct:
  - Run 1: a regular customer with 100 points is charged the full price
  - Run 2: a Gold Customer with 100 points is given a discount
  - Run 3: -10 is identified as an invalid number of points.
- So does this program work correctly?
- Does it work for every possible set of input values?

Is the Program Correct?



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

Examine the theory of software testing to try and answer these questions

- These results look correct:
  - Run 1: a regular customer with 100 points
  - Run 2: a Gold Customer with 100 points is
  - Run 3: -10 is identified as an invalid numbe
- So does this program work correctly?
- Does it work for every possible set of input values?

Theory of Software Testing*

- Summary:
  - For a test to be successful, all test data within the test set must produce the results as defined by the specification
  - The test data selection criteria is reliable if it consistently produces test sets which are successful, or it consistently produces test sets which are not successful
  - If a set of test data is chosen using a criterion that is **reliable** and **valid**, then the successful execution of that test data implies that the program will produce correct results over its entire input domain
  - **The only criterion that can be guaranteed to be reliable and valid is one that selects each and every value in the program domain** (“exhaustive testing”)

(*) Goodenough and Gerhart, *Toward a Theory of Test Data Selection*

Feasibility of Exhaustive Testing

- Input **goldCustomer** can have 2 possible values: true and false
- A Java long is defined as a 64-bit, signed integer, so the input **bonusPoints** can have 2^{64} possible values
- Total number of possible input combinations is 2^{65} or 3.7E+19
- Exhaustive testing would require running 2^{65} tests
  - 36,893,488,000,000,000,000 or 36 million, million, million tests
- And also checking the output value for each is correct!

Footnote: 3.7E+19=3.7 * 10^{19}

This is Not Feasible

- **Execution Time**: executing this number of tests on a single CPU at 60,000 tests per second, would take approximately:
  - 6E+14 or 600,000,000,000,000 seconds
  - Or approx. 20 million years (approx. 30 million seconds in a year)
- **Design Time**: the correct answer must be worked out for each

Implications

- Larger Examples:
  - A class with 64 bytes of data can have 2^{512} states
  - A database with 1,000,000 records of 128 bytes each can have approx. $2^{1,000,000,000}$ states
  - Both require infeasible numbers of tests
- Implications:
  - Testing with large numbers of combinations is infeasible: it would take too long to execute the tests
  - And, working out the correct output for each is not feasible manually
  - Writing a program to do this introduces the *Test Oracle* problem
    how do you test that the test oracle program works correctly?
    It has exactly the same specification as the program you are testing!!!

Test Heuristics

- Infeasible to test all possible combinations (and sequences) of inputs
- Need to reduce the number of tests significantly without significantly reducing the effectiveness of the testing
- Good tests:
  - High probability of finding faults
  - Don’t duplicate one another
  - Be independent (to prevent faults concealing each other)
  - Test as much of the code as possible
- Difficult. Subject of research.
- Select a subset:
  - To find faults effectively
  - No well established theory => use *Test Heuristics* based on principles and practice

Consider 4 Heuristics

1. Random Testing
2. Black-Box and White-Box Testing
3. Experience-Based Testing
4. Fault-Insertion

1. Random Testing

- Simplest heuristic
- Large number of inputs selected at random (scales well)
- Invariably automted
- Approximates to exhaustive testing

Random Testing Code for check()

```java
public class RandomTest {

   public static void checkRandomTest(long loops) {

      Random r=new Random();
      long bonusPoints;
      boolean goldCustomer;
      String result;

      for (long i=0; i<loops; i++) {
        bonusPoints = r.nextLong();
        goldCustomer = r.nextBoolean();
        result = Check.check( bonusPoints, goldCustomer );
        System.out.println("check("+bonusPoints+","+
        goldCustomer + ")->" + result);
      }

   }

   public static void main(String[] args) {
      checkRandomTest(1000);
   }

}
```

Random Test Execution

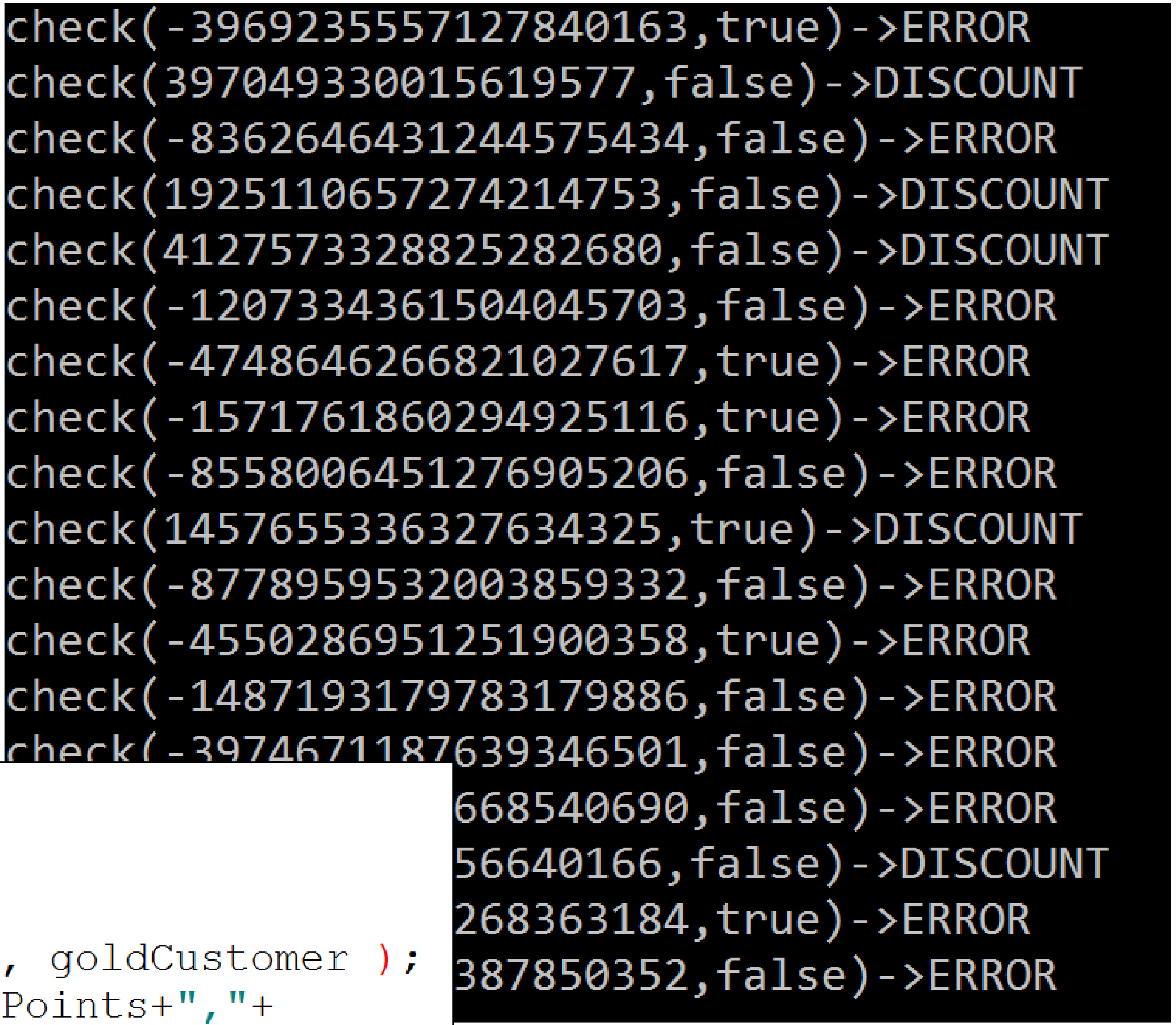
- Executing this program with loops=1,000,000 results in check() being called a million times with different random inputs, and generates a large amount of output
- On my desktop, this takes about 7 seconds to execute, mainly due to writing the output to the screen
- Without the output, the execution time is under 100 milliseconds
- On my laptop it takes too long, so I've changed loops to 1,000

Random Testing

DEMO

Note: I’ve changed loops to 1,000 for my laptop

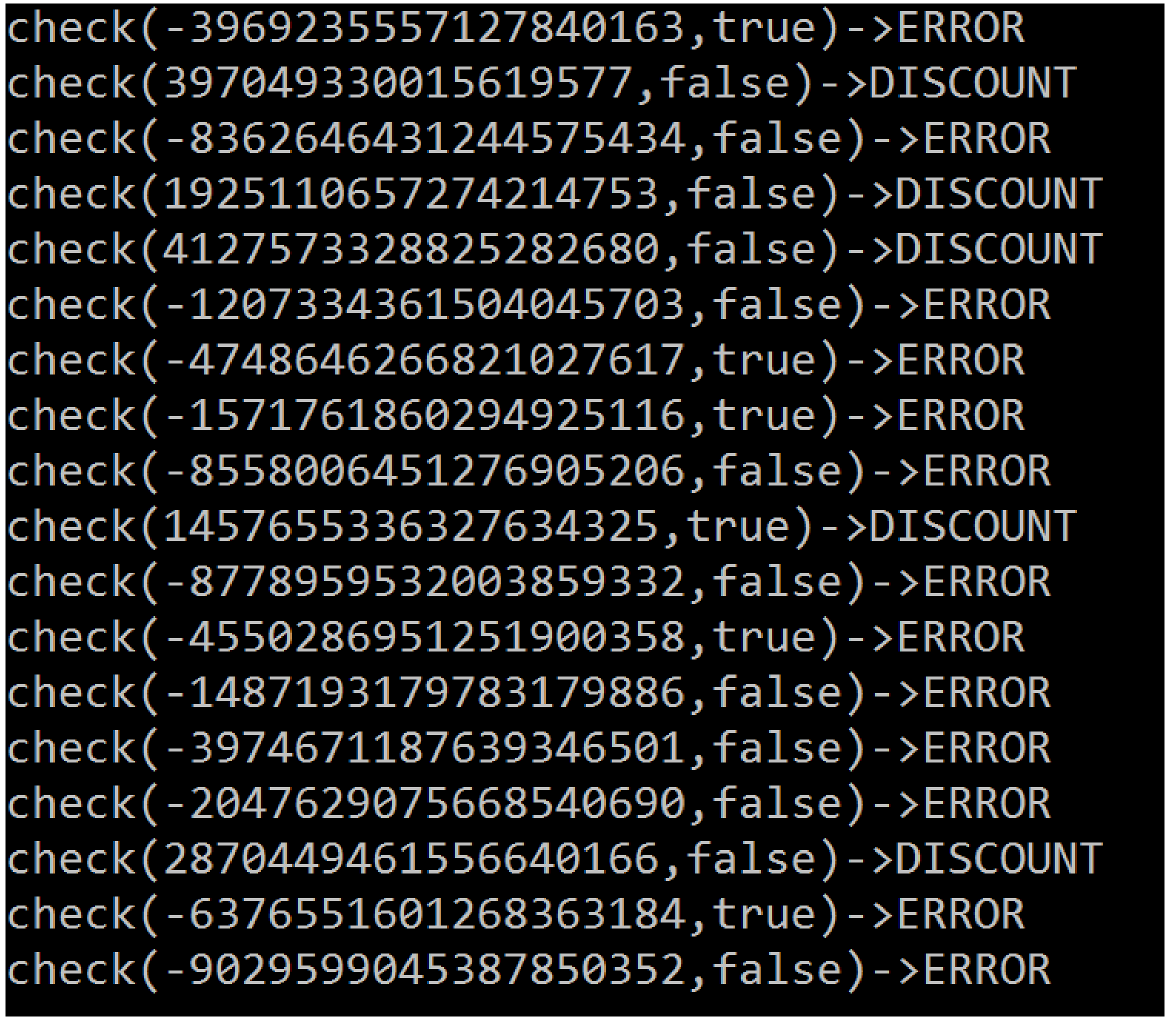
Results



```
check(-3969235557127840163,true)->ERROR
check(3970493300156l9577,false)->DISCOUNT
check(-8362646431244575434,false)->ERROR
check(1925110657274214753,false)->DISCOUNT
check(4127573328825282680,false)->DISCOUNT
check(-1207334361504045703,false)->ERROR
check(-4748646266821027617,true)->ERROR
check(-1571761860294925116,true)->ERROR
check(-8558006451276905206,false)->ERROR
check(1457655336327634325,true)->DISCOUNT
check(-8778959532003859332,false)->ERROR
check(-4550286951251900358,true)->ERROR
check(-1487193179783179886,false)->ERROR
check(-3974671187639346501,false)->ERROR
668540690,false)->ERROR
56640166,false)->DISCOUNT
268363184,true)->ERROR
387850352,false)->ERROR
```

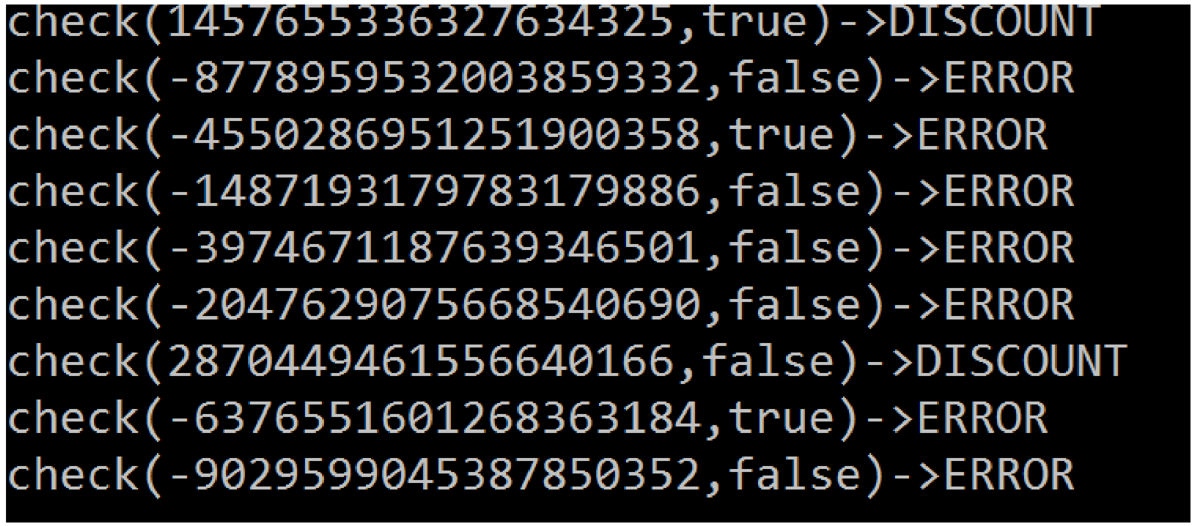
```java
for (long i=0; i<loops; i++) {
  bonusPoints = r.nextLong();
  goldCustomer = r.nextBoolean();
  result = Check.check( bonusPoints, goldCustomer );
  System.out.println("check("+bonusPoints+","+
  goldCustomer + ")->" + result);
}
```

So does this verify the program is correct?



```
check(-3969235557127840163,true)->ERROR
check(3970493300156l9577,false)->DISCOUNT
check(-8362646431244575434,false)->ERROR
check(1925110657274214753,false)->DISCOUNT
check(4127573328825282680,false)->DISCOUNT
check(-1207334361504045703,false)->ERROR
check(-4748646266821027617,true)->ERROR
check(-1571761860294925116,true)->ERROR
check(-8558006451276905206,false)->ERROR
check(1457655336327634325,true)->DISCOUNT
check(-8778959532003859332,false)->ERROR
check(-4550286951251900358,true)->ERROR
check(-1487193179783179886,false)->ERROR
check(-3974671187639346501,false)->ERROR
check(-2047629075668540690,false)->ERROR
check(2870449461556640166,false)->DISCOUNT
check(-6376551601268363184,true)->ERROR
check(-9029599045387850352,false)->ERROR
```

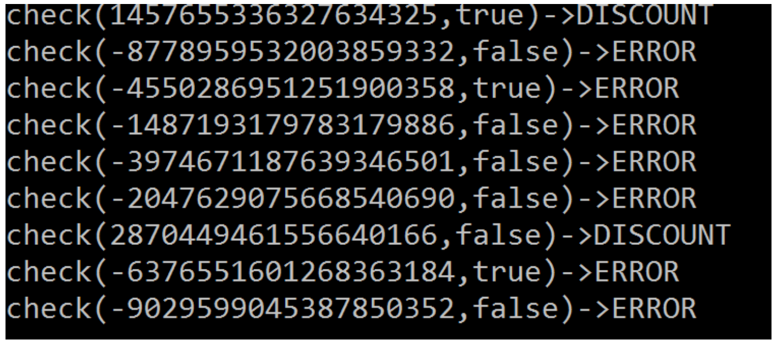
So does this verify the program is correct?



```
check(1457655336327634325,true)->DISCOUNT
check(-8778959532003859332,false)->ERROR
check(-4550286951251900358,true)->ERROR
check(-1487193179783179886,false)->ERROR
check(-3974671187639346501,false)->ERROR
check(-2047629075668540690,false)->ERROR
check(2870449461556640166,false)->DISCOUNT
check(-6376551601268363184,true)->ERROR
check(-9029599045387850352,false)->ERROR
```

1. It **EXERCISES** the code: demonstrates it does not crash, but does not **TEST** the program by checking each output is correct
2. It generates a **POOR DISTRIBUTION** of input test data:
    - Half negative, generating ERROR
    - Most positive inputs generate DISCOUNT
    - Very few between 80 and 100 – approx. 1 in 100,000,000 – generate FULLPRICE
3. Is **ENDING THE TEST** after 1,000,000 (or 1,000) loops optimal?

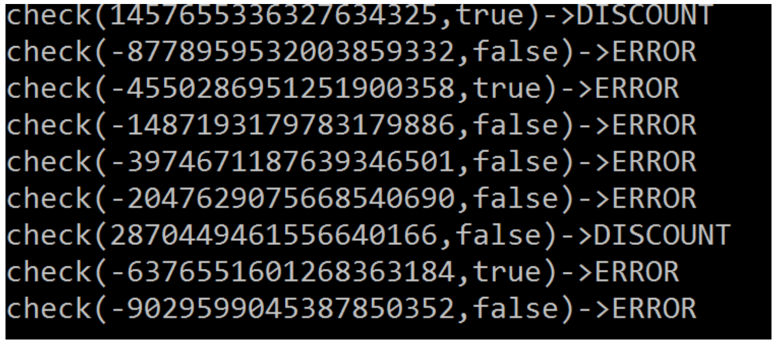
Demonstrates 3 Key Problems



```
check(1457655336327634325,true)->DISCOUNT
check(-8778959532003859332,false)->ERROR
check(-4550286951251900358,true)->ERROR
check(-1487193179783179886,false)->ERROR
check(-3974671187639346501,false)->ERROR
check(-2047629075668540690,false)->ERROR
check(2870449461556640166,false)->DISCOUNT
check(-6376551601268363184,true)->ERROR
check(-9029599045387850352,false)->ERROR
```

- The Test Oracle Problem
- The Test Data Problem
- The Test Completion Problem

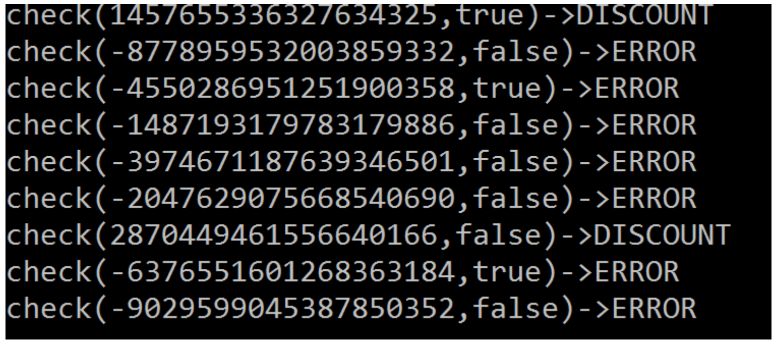
Demonstrates 3 Key Problems



```
check(1457655336327634325,true)->DISCOUNT
check(-8778959532003859332,false)->ERROR
check(-4550286951251900358,true)->ERROR
check(-1487193179783179886,false)->ERROR
check(-3974671187639346501,false)->ERROR
check(-2047629075668540690,false)->ERROR
check(2870449461556640166,false)->DISCOUNT
check(-6376551601268363184,true)->ERROR
check(-9029599045387850352,false)->ERROR
```

- The Test Oracle Problem
  - How to tell each result is correct?
  - Doing this manually would be very tedious and take too long
  - Writing a program to do it does not make sense – how would it be tested? In fact, we've already written a program to do this, and are trying to test it!
- The Test Data Problem
- The Test Completion Problem

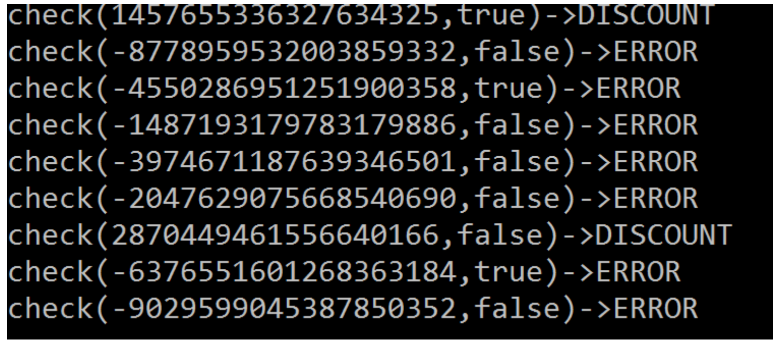
Demonstrates 3 Key Problems



```
check(1457655336327634325,true)->DISCOUNT
check(-8778959532003859332,false)->ERROR
check(-4550286951251900358,true)->ERROR
check(-1487193179783179886,false)->ERROR
check(-3974671187639346501,false)->ERROR
check(-2047629075668540690,false)->ERROR
check(2870449461556640166,false)->DISCOUNT
check(-6376551601268363184,true)->ERROR
check(-9029599045387850352,false)->ERROR
```

- The Test Oracle Problem
- The Test Data Problem
  - This code is generating random values, not random data
  - This results typically in both generating too many error input values
  - And also results typically in missing some required input values
- The Test Completion Problem

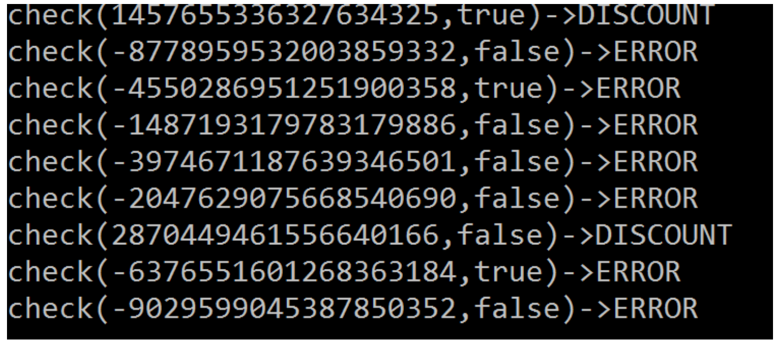
Demonstrates 3 Key Problems



```
check(1457655336327634325,true)->DISCOUNT
check(-8778959532003859332,false)->ERROR
check(-4550286951251900358,true)->ERROR
check(-1487193179783179886,false)->ERROR
check(-3974671187639346501,false)->ERROR
check(-2047629075668540690,false)->ERROR
check(2870449461556640166,false)->DISCOUNT
check(-6376551601268363184,true)->ERROR
check(-9029599045387850352,false)->ERROR
```

- The Test Oracle Problem
- The Test Data Problem
- The Test Completion Problem
  - In this example the random tests run for 1,000,000 iterations
  - There is no real basis for this number
  - Perhaps 1000 tests would have provided adequate assurance of correctness
  - Perhaps 1,000,000,000 would be required

Demonstrates 3 Key Problems



```
check(1457655336327634325,true)->DISCOUNT
check(-8778959532003859332,false)->ERROR
check(-4550286951251900358,true)->ERROR
check(-1487193179783179886,false)->ERROR
check(-3974671187639346501,false)->ERROR
check(-2047629075668540690,false)->ERROR
check(2870449461556640166,false)->DISCOUNT
check(-6376551601268363184,true)->ERROR
check(-9029599045387850352,false)->ERROR
```

- The Test Oracle Problem
- The Test Data Problem
- The Test Completion Problem

- Note that these problems exist with all forms of testing
- But in random testing their solutions are invariably automated, which requires further analysis (they are difficult to automate)
- We'll spend a week on random testing later in the semester

2. Black-Box and White-Box Testing

- Widely used heuristics
- In general, there are two approaches to selecting a subset
- Select test data that “exercises" the specification
    - Black-Box Testing
    - Or testing with specification coverage criteria
- Select test data that “exercises” the implementation
    - Usually the source code
    - White-box testing
    - Or testing with structural coverage criteria

Black-Box Testing

- Exercising the specification means to have generated enough tests that every possible different specified type of processing has been executed
- This not only means that every different category of output has been generated
- But also that every different category of output has been generated for every different input cause.

White-Box Testing

- Exercising the implementation means to have generated enough tests so that every component that makes up the implementation has been demonstrated to produce a valid result when executed
- At its simplest, each line of source code is regarded as a separate component
- There are more sophisticated ways of identifying components that provide for fuller coverage of a program’s possible behaviours

Testing vs Exercising

- In all tests, just exercising the code or the specification is not enough
- A test must not only do this, it must also **verify** that *actual result* matches the *expected results*
- A test passes if and only if this verification succeeds
- We will start looking at black-box testing next weel

3. Experience-Based Testing

- A*d-hoc* approach ("error guessing" or "expert opinion")
  - tester uses their experience to develop tests
- Goal
  - usually find faults in the code
- Tests based on:
  - end-user experience
  - operator experience
  - business or risk experience
  - legal experience
  - maintenance experience
  - programming experience
  - experience in the problem domain
  - etc.

Strengths and Weaknesses

- Strengths:
  - Experienced intuition frequently finds faults
  - Tends to be quite efficient

- Weaknesses:
  - Heavily reliant on the quality & relevance of the experience
  - *Ad-hoc* approach means difficult to ensure rigour/completeness of testing

4. Fault Insertion

- Philosophically different approach to testing
- Insert faults and verify they are found:
  - Into the software
  - Into the data
- Foundations in hardware testing:
  - Hardware connection will be artificially held at a high or low voltage
    - representing a “stuck at 1" or “stuck at 0" condition
  - The hardware is then run, typically through simulation:
    - ensure the fault is detected
    - ensure the fault is handled correctly

Software Fault Insertion

- Fault inserted into data
  - As for hardware testing, the purpose is to ensure that the fault is detected and handled correctly by the software
- When inserted into code
  - The purpose is different: it can be used is to measure the effectiveness of existing tests
  - Often referred to as ‘Mutation Testing’ – existing tests run against mutations of the code (with faults inserted) to verify the tests produce a different result

When to Finish Testing

- Theory of testing identifies one end point: complete exhaustive testing
- Therefore, a number of viewpoints:
  - Budget: when the time or budget allocated for testing has expired
  - Activity: when the software has passed all of the planned tests
  - Risk management: the predicted failure rate meets the quality criteria
- Software often does not pass all tests prior to release:
  - “Usage-Based Criteria” – give higher priority to expected inputs in use
  - “Risk of release” – calculate the expected cost of future failures
- Potential ethical and legal issues involved in making this decision
  - Especially for mission-critical or life-critical systems
  - *Information Technology Law in Ireland* (Kelleher and Murray)

When to Finish Testi

- Theory of testing identifies o testing
- Therefore, a number of view
  - Budget: when the time or bu
  - Activity: when the software
  - Risk management: the predi
- Software often does not pa
  - "Usage-Based Criteria" – gi
  - "Risk of release" – calculate
- Potential ethical and legal issues involved in ecision
  - Especially for mission-critical or life-critical systems

Safety-critical or life-critical systems may pose a threat to health or life if they fail.

Mission-critical systems may pose a threat to a mission if they fail.

Examples include software for space exploration, the aerospace industry, and medical devices.

Static and Dynamic Testing

- The previous examples have all used dynamic testing where the code has is executed and the output verified
- An alternative approach is static verification (“static analysis” or “static testing”) – the code is verified without executing it
- There are two approaches to static verification:
  1. Review-based techniques,
  2. Proving programs mathematically
- Both of these may be manual, semi-automated, or fully automated

Review-Based Techniques

- (Least formal) **pair programming**:
  - Two programmers work in tandem, with one writing the code while the other continuously reviews their work. This has been shown to be very effective, and is a core element of many modern software engineering processes (e.g. Agile)

Review-Based Techniques

- (Most formal) **code review/**walkthrough:
  - review team is presented with the specifications (product requirements, design documents, software requirements, etc.) and the code in advance
  - Formal meeting for code walkthrough checking that the code meets its requirements
  - Code inspection checklist
  - Formal report produced (basis for remedial work)
  - Highly effective method in software engineering, especially for high quality requirements.
  - Automated code analysis tools can be also used: these might find potential security flaws, to identify common mistakes in the code, or to confirm that the software conforms to a particular coding standard

Program Proving

- Mathematical techniques: “**Formal Methods**” or “**Formal Verification**” based heavily on set theory
- Prove that a program meets its specifications
  - Manually: very time-consuming and requires a high level of expertise
  - Semi-automated: where a tool evaluates the code, and provides “proof obligations” to be manually proven
  - Fully automated:
    - Programmer/designer produces formal requirements for the code
    - Code run through a program prover, to prove works or find “counter examples” where it fails
    - Requires internal specifications in the code, and one of the most challenging of these are “loop invariants”
    - Requires less skill and experience than manual proofs; significantly more challenging than dynamic testing

Program Proving

- Significant benefits of a proof: a program to be proven to work in all circumstances (equivalent to exhaustive testing)
- The program proof can be incorporated with higher level design proofs (such as model checking), providing an integrated proof framework for a full computer-based system
- Languages and tools: many research tools in development, ready for industrial use?
- For example: JML for Java programs, Viper for Python, etc. (see Book, Section 14.3.8)
- Reference module CS603: Rigorous Software Process

Testing in the Software Development Process

Testing in the Software Development Process

Software of any degree of complexity has three key characteristics:

1. **User requirements**, stating the needs of the software users

2. A **functional specification**, stating what the software must do to meet those needs

3. A number of **modules**, integrated to form the final system

Testing in the Software Development Process

Unit Testing: A software ‘unit’ is tested: a single or compound component: function, method, class, GUI component (such as a button), or a collection of them (such as a window). Unit testing almost invariably makes use of the programming interface of the unit (API)

Testing in the Software Development Process

Unit Testing: A software ‘unit’ is tested: a single or compound component: function, method, class, GUI component (such as a button), or a collection of them (such as a window). Unit testing almost invariably makes use of the programming interface of the unit (API)

Regression Testing: Check nothing is broken when make a change/add a module

Integration Testing: Check two or more modules work correctly together

Subsystem Testing: Check a subsystem

Testing in the Software Development Process

Unit Testing: A software ‘unit’ is tested: a single or compound component: function, method, class, GUI component (such as a button), or a collection of them (such as a window). Unit testing almost invariably makes use of the programming interface of the unit (API)

Regression Testing: Check nothing is broken when make a change/add a module

Integration Testing: Check two or more modules work correctly together

Subsystem Testing: Check a subsystem

System Testing: Entire software system is tested as a whole to make sure it works correctly (e.g. smoke testing: “turn it on and see if it smokes”). This testing uses the system interface: application GUI, server web-interface, etc. Test data is selected to ensure the system satisfies the specification

Testing in the Software Development Process

Unit Testing: A software ‘unit’ is tested: a single or compound component: function, method, class, GUI component (such as a button), or a collection of them (such as a window). Unit testing almost invariably makes use of the programming interface of the unit (API)

Regression Testing: Check nothing is broken when make a change/add a module

Integration Testing: Check two or more modules work correctly together

Subsystem Testing: Check a subsystem

System Testing: Entire software system is tested as a whole to make sure it works correctly (e.g. smoke testing: “turn it on and see if it smokes”). This testing uses the system interface: application GUI, server web-interface, etc. Test data is selected to ensure the system satisfies the specification

Acceptance Testing: Entire system is tested to make sure it meets the user’s needs

Focus of the Module

- Activities can use black-box or white-box techniques to develop tests
- White-box techniques are seldom used except in unit testing
- The focus in this module is dynamic software verification
- With a particular emphasis on using the test design techniques for:
  - Unit testing (including object-oriented testing)
  - System/Application testing

Software Testing Activities

Modules uses a systematic and rigorous approach to aid learning:

1. **Analysis**
2. **Identify Test Coverage Items**
3. **Identify Test Cases**
4. **Test Design Verification**
5. **Test Implementation**
6. **Test Execution**
7. **Review Test Results**

Quick Overview/Analysis

1. **Analysis** - the specification and the software implementation are analysed to extract the information required for designing the tests
2. **Test Coverage Items**
3. **Test Cases**
4. **Test Design Verification**
5. **Test Implementation**
6. **Text Execution**
7. **Test Results**

Quick Overview/Test Coverage Items

1. **Analysis**
2. **Test Coverage Items** - these are the criteria that the tests are required to address (or cover), and are derived using the results of the analysis
3. **Test Cases**
4. **Test Design Verification**
5. **Test Implementation**
6. **Text Execution**
7. **Test Results**

Quick Overview/Test Cases

1. **Analysis**
2. **Test Coverage Items**
3. **Test Cases** - these specify the data required to address the test coverage items
4. **Test Design Verification**
5. **Test Implementation**
6. **Text Execution**
7. **Test Results**

Quick Overview/Test Design Verification

1. **Analysis**
2. **Test Coverage Items**
3. **Test Cases**
4. **Test Design Verification** - the test cases are reviewed to ensure that every test coverage item has been included
5. **Test Implementation**
6. **Text Execution**
7. **Test Results**

Quick Overview/Test Implementation

1. **Analysis**
2. **Test Coverage Items**
3. **Test Cases**
4. **Test Design Verification**
5. **Test Implementation:**
    1. Tests usually implemented using automated test tool libraries (“TestNG”)
    2. Sometimes, a manual test procedure may be defined, but we won’t do this
6. **Text Execution**
7. **Test Results**

Quick Overview/Test Execution

1. **Analysis**
2. **Test Coverage Items**
3. **Test Cases**
4. **Test Design Verification**
5. **Test Implementation**
6. **Text Execution** - the tests are executed: the code being tested is called, using the selected input data values, and checking that the results are correct
7. **Test Results**

Quick Overview/Test Results

1. **Analysis**
2. **Test Coverage Items**
3. **Test Cases**
4. **Test Design Verification**
5. **Test Implementation**
6. **Text Execution**
7. **Test Results** - the test results are examined to determine whether any of the tests have failed. In a formal test process, this may result in the production of Test Incident Reports

Software Testing Activities in Practice

- 7-steps: Analysis, Test Coverage Items, Test Cases, Test Design Verification, Test Implementation, Text Execution, Test Results
- Worked examples to demonstrate the application of these activities
- In practice, an experienced tester may perform many of the steps mentally without documenting the results
- Software in mission-critical or safety-critical systems requires high quality:
  - Test process generates significant amounts of documentation
  - This documentation supports detailed reviews of the test activities as part of the overall software quality assurance (SQA) process
- In this module, all the results are fully documented as a learning guide

Software Testing Activities in Practice

- Consider in more detail

1. Analysis
2. Test Coverage Items
3. Test Cases
4. Test Design Verification
5. Test Implementation
6. Text Execution
7. Test Results

Analysis

- All test design requires some form of analysis to be undertaken
- Source code, specifications (user requirements and the software functional specifications)
- Sources of test information are referred to as the Test Basis
- The results of the analysis are referred to as Test Conditions, and are used to determine Test Coverage Items
- In practice, the analysis results may not be fully documented, if at all - but doing so allows the test coverage items to be easily identified and reviewed

Identify Test Coverage Items

- Test coverage items are particular items to be covered by a test
- They are generated by reviewing the results of the analysis using the test design technique that has been selected
- Some examples of test coverage items are:
  - A particular value, or range of values, for an input/output parameter to take
  - A particular relationship between two parameters to be achieved
  - A particular line of code to be executed
  - A particular path in the code to be taken

Test Coverage Items (used in Test Cases)

- A test case may cover multiple test coverage items
- A goal of most test techniques is to reduce the number of test cases by covering as many coverage items as possible in each
- This reduces the time it takes to execute the tests
  - For a small number of tests this is not important
  - Large software systems may have tens of thousands of tests, and the execution time may run into multiple days

Test Coverage Items (Values)

- Any values used in defining a test coverage item should be as generic as possible
- Reference constants by name
- State relationships rather than actual values where possible
- This allows maximum flexibility in designing the test cases
- Allowing a single test case to cover as many coverage items as possible

Test Coverage Items (Identifiers)

- Each test coverage item requires an identifier
- So that it can be referenced when the test design is being verified
- This identifier must be unique for each test item
- To achieve this, as the test coverage items are specific to each test technique, this book uses a prefix based on the test technique being used (e.g. EP-001 as an identifier for test coverage item **1**, using **E**quivalence **P**artitions as shown next week)

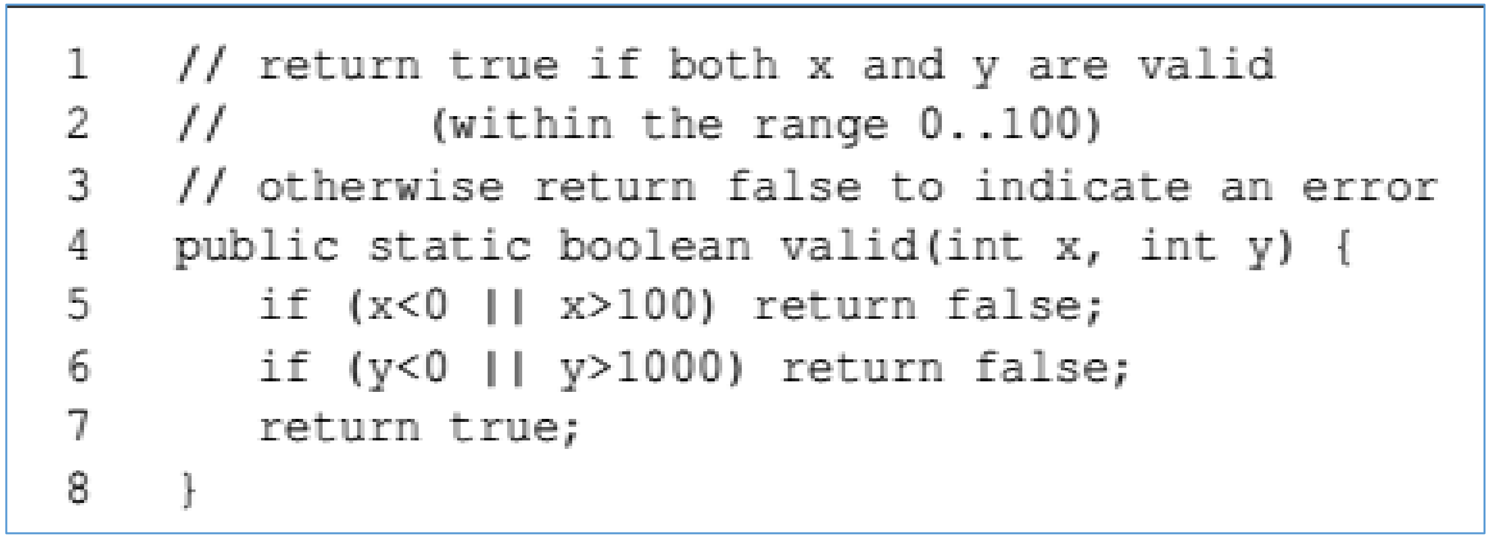
Test Coverage Items (Implicit/Explicit Arguments)

- In simple testing:
  - All the inputs are passed as arguments to the code
  - The output is returned as a value
  - Example: `int x=max(45,1000,Integer.MIN_VALUE)`
- In practice, it is not unusual for there to be both:
  - Explicit arguments (passed in the call to the code)
  - Implicit inputs (Java example: class attributes read by the code)
- Also there may be both:
  - An explicit return value
  - And other implicit outputs (such as, in Java, class attributes modified by the code)
- **These should all be included** in the test coverage items

Test Coverage Items and Error Hiding

- Two categories of test coverage items: normal and error cases
- Separate these:
  - multiple input **normal** coverage items may be incorporated into one test case
  - multiple input **error** coverage items may not
- This is because of error hiding
  - Most software does not differentiate between all the possible error causes
  - In order to test that each input error is identified and handled correctly only one such error may be tested at a time
- We use an asterisk (*) signify input error test coverage items and error test cases

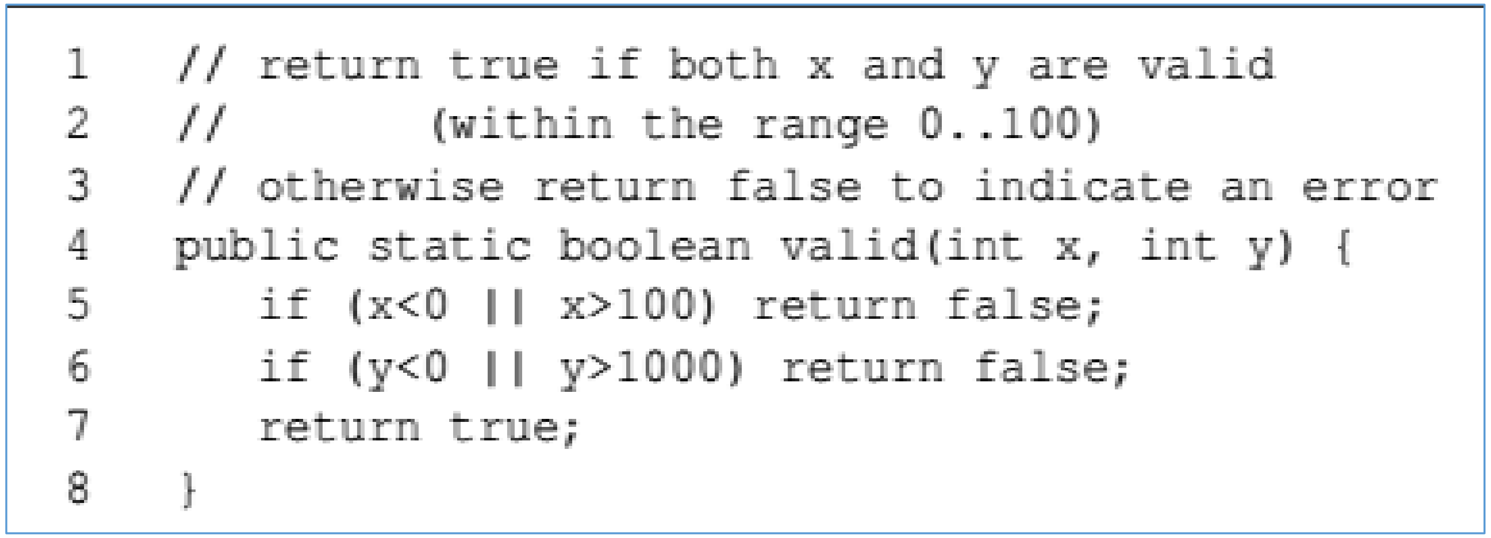
Error Hiding (Example)



```
1   // return true if both x and y are valid
2   //          (within the range 0..100)
3   // otherwise return false to indicate an error
4   public static boolean valid(int x, int y) {
5      if (x<0 || x>100) return false;
6      if (y<0 || y>1000) return false;
7      return true;
8   }
```

- Both x=500 and y=500 should return false to indicate an error
- But a test with the simultaneous inputs x=500 and y=500 does not verify that errors in both parameters are identified
- The code on line 5 will correctly detect the first error case, and return false
- But Line 6 is not executed: so the **fault** on line 6 is not detected – it has 100 instead of 1000

Error Hiding (Example)



```
// return true if both x and y are valid
//         (within the range 0..100)
// otherwise return false to indicate an error
public static boolean valid(int x, int y) {
   if (x<0 || x>100) return false;
   if (y<0 || y>1000) return false;
   return true;
}
```

- This is called "Error Hiding" – where one error hides another
- Two separate tests are required:
  - A test with x=50 and y=500
  - A test with x=500 and y=50
- This ensures that each error value is tested individually, so that an earlier fault does not hide a subsequent fault

Identify Test Cases

- Test data for each test case is based on using **uncovered** test coverage items
- For **normal** test coverage items, in order to reduce the number of tests, each test case should cover as many (new) test coverage items as possible
- For **error** test coverage items, each test case must only cover exactly one test coverage item
- Selecting data to minimize the number of tests is a skill: make sure to cover all the test coverage items
- In this module, be systematic and consistent (see lab instructions)

Test Case (Specification)

- An **identification of the test item**: for example, the method name/signature, and version number from the version control system
- A unique **identifier** for the test case (e.g. T001 for Test Case 1)
  - This aids debugging: a specific test can be run to reproduce a fault
- A **list of the test coverage items** covered by each test case
- The test data consisting of:
  - Input Values: these should be specific values
  - Expected Results (always from the specification, never from the source code)

Test Design Verification

- Test Design=analysis, defining TCI, developing test cases/data
- Document:
  - the test coverage items that are covered by each test case
  - the test cases that cover each test coverage item
- This allows the test design to be easily reviewed, ensuring that every test coverage item is efficiently covered
- Formal reviews can be used to ensure the quality of the test designs. In practice, much of this work is frequently done mentally and not documented in detail (and cannot be reviewed)

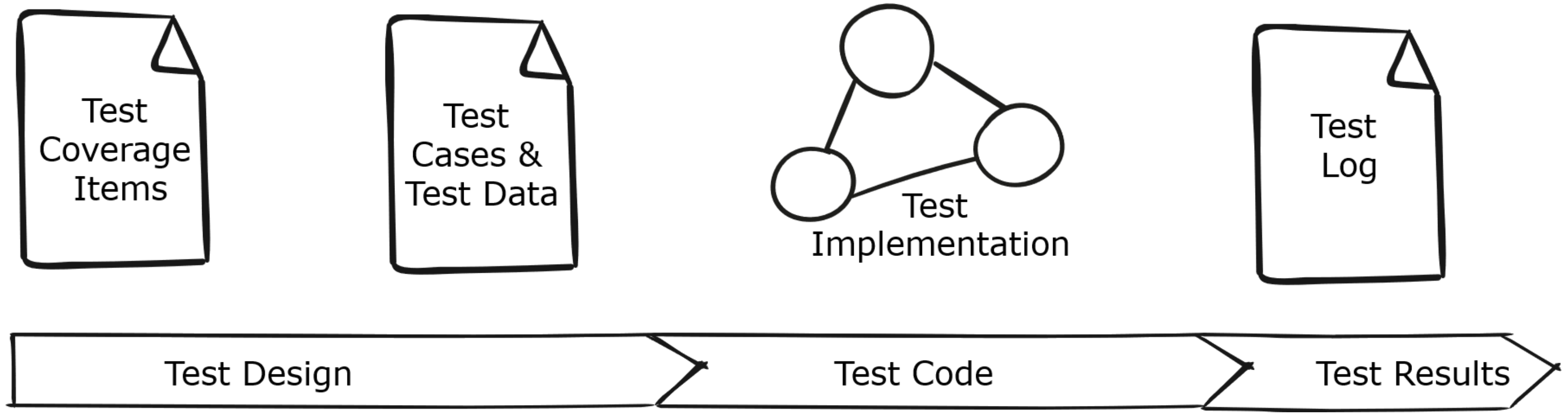
Test Implementation

- Tests may be implemented:
  - As code (for automated testing) – used on this module
  - Or as procedures (for manual testing)
- Recent trends are to automate as many tests as possible. However, there is still a place for manual testing. This requires a test procedure to be documented for executing the test. The documentation must describe how to setup the correct environment, how to execute the code, the inputs to be provided, and the expected results.
- Unit tests are invariably automated; integration tests usually are; system tests are where possible

Test Implementation (Code)

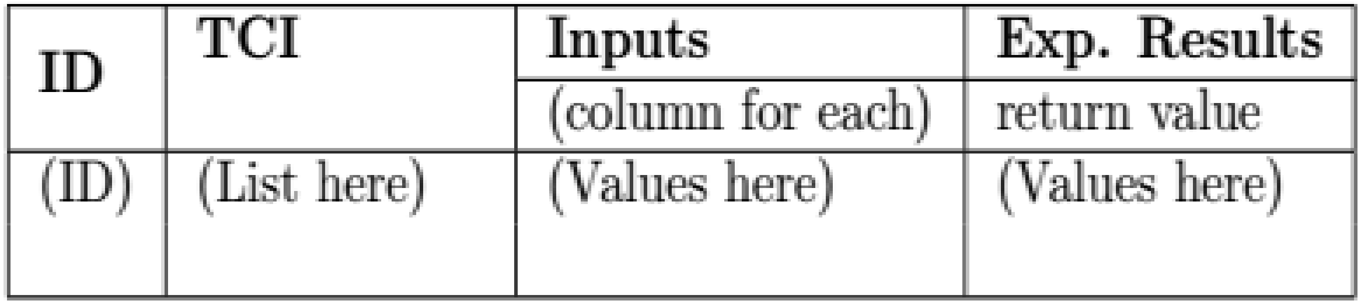
- Implementing an automated test involves writing code to:
    - Invoke the Implementing an a test item with the specified input parameters
    - Comparing the actual results with the expected results
- It is good practice to use method names in the test code that reflect the matching test case identifier
- Each specified test case should be implemented separately (this allows individual failures to be clearly identified)
- Tests may be grouped into test suites (or test sets) which allow a tester to select a large number of tests to be run as a block (more on this later in the module)

Test Artefacts



- The Test Design activity produces:
  - The Test Coverage Items - the items that the test data must exercise
  - The Test Cases and Test Data (test identifiers, values for input data & expected results, test coverage items covered by each test case)
- The Test Code activity produces the Test Implementation. This is based on the test cases
- The Test Results are produced by executing the Test Implementation, producing a Test Log

Test Cases Template



ID	TCI	Inputs	Exp. Results
		(column for each)	return value
(ID)	(List here)	(Values here)	(Values here)

- On this module, test cases are specified using the template shown
- There is no standard – consistency allow tests specifications to be reviewed easily
- The **ID** column provides a unique Test Case ID for each test case (e.g. T001)
- The **TCI** column documents the test coverage items that this test case covers. It is recommended to identify the new test coverage items covered by this test case separately from those already covered by previously defined test cases
- The **Inputs** column gives the required input data values for each parameter
- The **Expected Results** (Exp. Results) column documents the output values which are expected based on the specification of the software being tested. For Java, this is usually the return value from the method called

Test Cases (Avoiding Duplication)

- It is a key goal in developing tests to avoid duplication
- The experienced tester may be able to do this while developing data for the test cases
- In this module, we will document the required data for each test case as it is designed, and then subsequently identify and discard test cases with duplicate data

Fault Models

- As exhaustive testing is not feasible, every test approach is a compromise, designed to find particular classes of faults
- These classes are referred to as **fault models**
- Each type of testing is designed around a particular fault model
- In this module, we will start with simple fault models and gradually introduce more complex ones
- Understanding the classes of faults introduced by the software developers can lead to a more effective emphasis on tests most likely to find the faults

Order of Testing

- Usually unit testing is followed by integration testing, and then by system/application testing
- Within unit testing, black-box testing is invariably used first, followed by white-box testing, in order to selectively increase coverage measures, such as the percentage of statements executed
- This approach maximises test coverage, while reducing the duplication of tests
- We will not cover integration testing in detail, due to the additional complexities it introduces (there are many books on the tools, but no systematic books on the principles) – in practice

Documenting Test Design (in Practice)

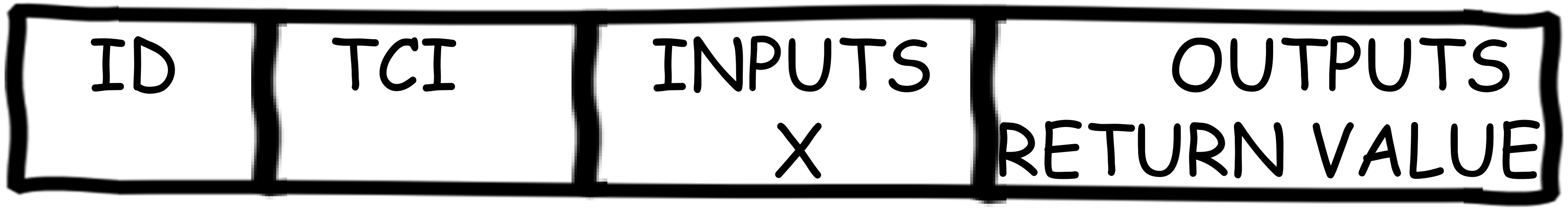
- In practice, most testers will not document their tests to the degree of detail shown in this module,
- Except in situations where high software quality is required
- However, the tester must do the same mental work to develop tests systematically, and while learning to test it is a good practice to write down all the intermediate results as shown in this book

Using Tables

- Very concise way of documenting your test design
- Recommend: draw by hand (initially)

Using Tables

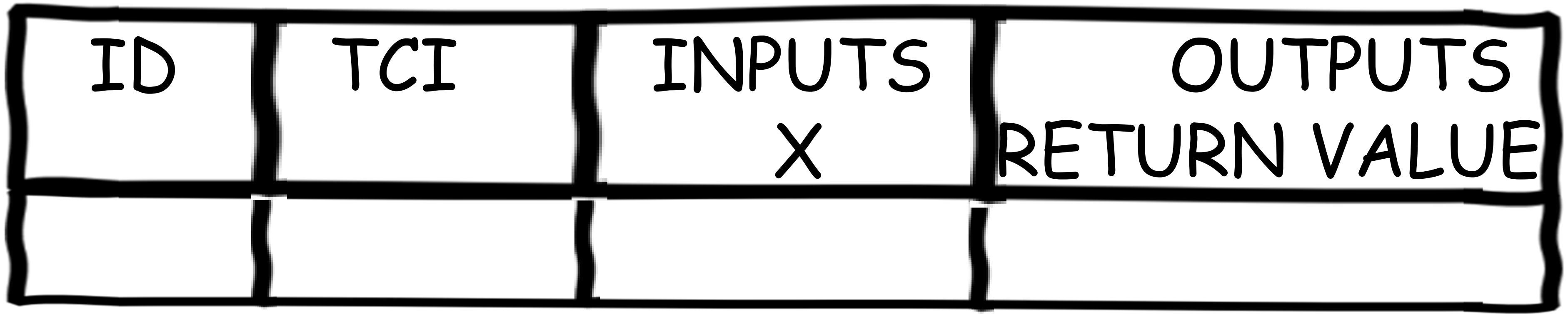
- Very concise way of documenting your test design
- Recommend: draw by hand (initially)
- Add Column Titles



ID	TCI	INPUTS	OUTPUTS
		X	RETURN VALUE

Using Tables

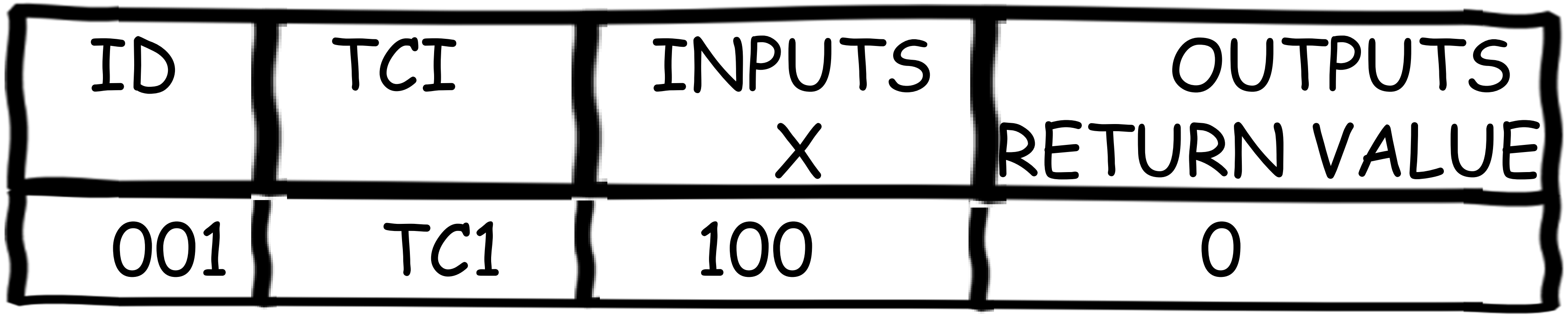
- Very concise way of documenting your test design
- Recommend: draw by hand (initially)
- Add column titles
- Add each row as needed



ID	TCI	INPUTS	OUTPUTS
		X	RETURN VALUE

Using Tables

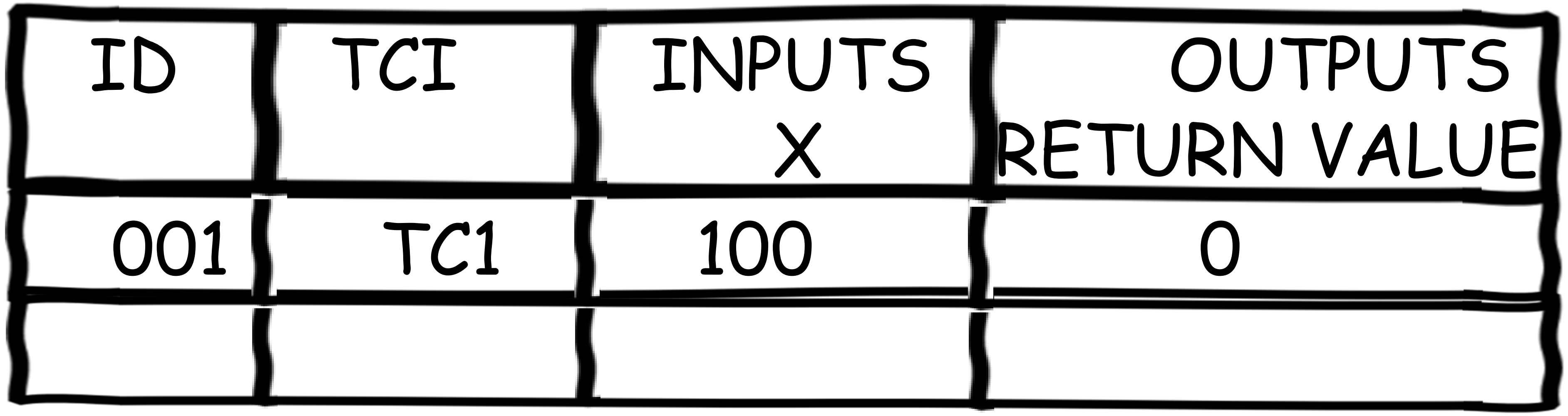
- Very concise way of documenting your test design
- Recommend: draw by hand (initially)
- Add column titles
- Add each row as needed – and fill



ID	TCI	INPUTS X	OUTPUTS RETURN VALUE
001	TC1	100	0

Using Tables

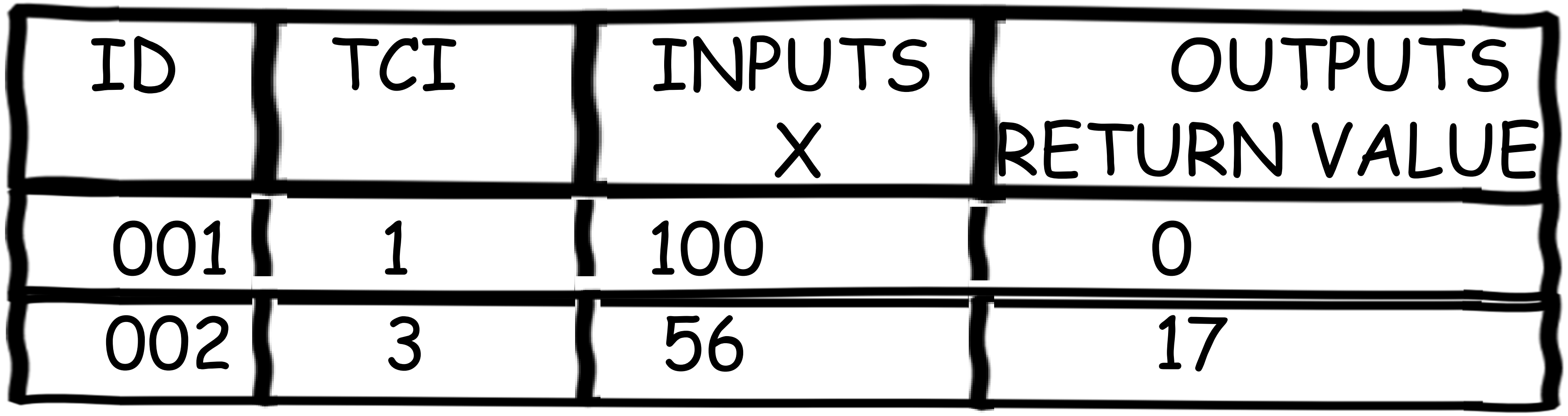
- Very concise way of documenting your test design
- Recommend: draw by hand (initially)
- Add column titles
- Add each row as needed



ID	TCI	INPUTS X	OUTPUTS RETURN VALUE
001	TC1	100	0

Using Tables

- Very concise way of documenting your test design
- Recommend: draw by hand (initially)
- Add column titles
- Add each row as needed – and fill - etc



ID	TCI	INPUTS X	OUTPUTS RETURN VALUE
001	1	100	0
002	3	56	17

Programming Language

- The Java programming language (Java 21) has been used for all examples in the book
- The principles shown, and the approach used for Java, serves as an example for testing in other procedural or object-oriented languages

Software Test Tools

- The key goal of the module is that you should understand the principles of software testing, and be able to apply them in practice
- The software test tools you will use have been selected as good examples for demonstrating test automation
- Not necessarily the most popular or up-to-date examples of each
- May not be the best tools to use in particular real-world testing situations
- This book does not endorse or recommend any particular tool
- Only an essential subset of the tool features are covered in this book: the reader should refer to the tool-specific documentation for more details

Terminology

- **BEWARE**: little consistent use of software testing terminology

Terminology

- **BEWARE**: little consistent use of software testing terminology
- Many books, research papers, white-papers, test automation documentation, and web pages, use different or **conflicting** terms

Terminology

- **BEWARE**: little consistent use of software testing terminology
- Many books, research papers, white-papers, test automation documentation, and web pages, use different or **conflicting** terms
- This module uses simplified terminology, that is consistent with the terminology used in the **ISO International Standard 29119** (*ISO/IEC/IEEE 29119*)

Terminology

- **BEWARE**: little consistent use of software testing terminology
- Many books, research papers, white-papers, test automation documentation, and web pages, use different or **conflicting** terms
- This module uses simplified terminology, that is consistent with the terminology used in the **ISO International Standard 29119** (*ISO/IEC/IEEE 29119*)
- In particular, three terms as used as follows:
  - **Test coverage item:** the test goals
  - **Test case:** the data needed to achieve those goals
  - **Test implementation:** the code which implements the test

This Afternoon

- Warm-up lab – run examples for:
  - Manual Testing
  - Random Testing
  - Automated Testing
- Tools:
  - Command line
  - Java JDK21 – must be installed
  - Chrome web browser – must be installed
  - Automatic dependencies: Gradle (TestNG, JaCoCo, Selenium)

Running the Book Examples

- Checked with Java JDK 21
- Create a new folder: note it must not have spaces in the path
- Extract the book .zip file:
  - book-examples-gradle-2025.zip

Files

```
<DIR>          .
<DIR>          ..
<DIR>          .gradle
<DIR>          ch01
<DIR>          ch02
<DIR>          ch03
<DIR>          ch04
<DIR>          ch05
<DIR>          ch06
<DIR>          ch07
<DIR>          ch09
<DIR>          ch10
<DIR>          ch11
<DIR>          ch12
<DIR>          example
<DIR>          gradle
        44,745 gradle-run-menu-linux.sh
        44,742 gradle-run-menu-macos.sh
        40,594 gradle-run-menu-win.bat
         8,496 gradlew
         2,868 gradlew.bat
         6,490 readme.txt
         2,071 settings.gradle

 7 File(s)
16 Dir(s)
```

Running the Book Examples

- Open a command window
  - There are many ways to do this in windows
  - Or use **ocw-win.bat** which you can double click to open a command window
- Run
  - **@gradle-run-menu-win.bat**
  - Note: gradle is included in the .zip file
- Note: Linux and macOS also supported
  - Must give execute permission to some files first (instructions on Moodle)

Independent Study

- Read Chapter 1 of the book
- Find more recent examples of software failures & the associated costs
- Complete the lab exercises
- Extra exercise - exhaustive Testing
  - modify the Random Test to systematically produce every possible combination of inputs (use nested for loops) and measure how long it takes to execute!