Undergraduate Topics in Computer Science

Neil Walkinshaw

Software Quality Assurance

Consistency in the Face of Complexity and Change

Springer

Undergraduate Topics in Computer Science

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

More information about this series at http://www.springer.com/series/7592

Neil Walkinshaw

Software Quality Assurance

Consistency in the Face of Complexity and Change

Springer

Neil Walkinshaw
Department of Computer Science
University of Leicester
Leicester
UK

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-64821-7 ISBN 978-3-319-64822-4 (eBook)
DOI 10.1007/978-3-319-64822-4

Library of Congress Control Number: 2017947829

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Emma, Iona, and Dougie.

Preface

> "Let's think the unthinkable, let's do the undoable. Let us prepare to grapple with the ineffable itself, and see if we may not eff it after all."
>
> Douglas Adams, *Dirk Gently's Holistic Detective Agency*

This book is an introduction for students to the main principles and some of the most popular techniques that constitute 'software quality assurance'. It is worth emphasising from the outset that this book is *not* a reference book. There are already plenty of excellent comprehensive Software Engineering reference books in print.

Instead, this book seeks to provide a focus on Quality Assurance that typical, more generic Software Engineering reference books do not. The goal is to to do so in such a way that the book can be read from cover to cover throughout the course of a typical university module. Specifically, this book aims to be:

- **Concise:** It aims to be small enough to be readable in its entirety over the course of a typical software engineering module.
- **Explanatory:** When topics are covered, it is important not merely to describe *what* they are, but also *why* they are the way they are – describing what events, technologies, and individuals or organisations helped to shape them into what they are now.
- **Applied:** Topics will be covered with a view to giving the reader a good idea of how they can be applied in practice, and by pointing where possible to evidence about their efficacy.

Quality Assurance is often presented and discussed in somewhat utilitarian terms, as a set of necessary, occasionally tedious, techniques; required reading for anybody who aspires to become a capable, reliable Software Engineer. This brings us to the final, slightly more nebulous objective of this book: To convince the reader that there is much, much more to Quality Assurance than that.

We inhabit a world in which software is increasingly pervasive – controlling everything from light bulbs in homes to smart phones, cars, planes, power stations, and

voting machines. Failures in software quality can have and have had disastrous consequences. There is an urgent need for a widespread appreciation of how precarious software quality can be, and how it can improved and ensured.

Although the application of Quality Assurance techniques can become 'tedious', this misses what are (for the author at least) the real attractions. The subject is not only necessary, but academically fascinating too. There is no way of *guaranteeing* that a software system will 'succeed' - that it will not contain bugs, satisfy the customer, and be delivered on time and at cost. The task of building complex systems according to complex, continuously changing requirements, in a limited amount of time, within a limited budget, whilst managing large teams of developers, is enormously challenging. There is no single 'best' solution, and there are so many open (often surprising) problems.

Acknowledgements

This book is an extension of the course notes for the "Software Quality Assurance and Metrics" course at the University of Leicester, jointly taught to under- and postgraduate students. The course was originally taught by Helge Janicke (now at De Montfort University) until I took over as convenor in 2013. Although the course has changed in several respects, I am very grateful to Helge for developing the initial structure, and thus setting the direction for a large portion of the subject-matter covered in this book.

Throughout the writing of the book, several undergraduate and postgraduate students have been kind enough to provide valuable feedback on its contents. Many thanks especially to Cara Bateman, Anita Lad, Shriya Malhotra, and Sylvester Saracevas.

Finally, I owe a debt of gratitude to the editorial team at Springer, and Ralf Gerstner in particular, for their valuable support, feedback, and patience.

Contents

Chapter 1
Introduction

The term 'software quality assurance' rarely evokes much excitement. It is often perceived as a form of applied pedantry, lacking the challenge and creativity that is required to actually design and build a software system. In short, it is often seen as a necessary chore. Nobody wants to read and learn about necessary chores. To some, a book on quality assurance is about as enticing as a book on dish-washing.

In reality such preconceptions are wildly misguided (surely something you would hope for, given that you have nine chapters on the topic ahead of you). The idea that quality assurance stands separately from other software development activities is not true. Nor is the perception that it is particularly pedantic or lacks creativity (though pedantry can be useful!). It is not a chore, and if it is treated as an afterthought, any non-trivial software project is bound to fail.

In this book we will show how quality assurance and software development are inextricably linked. A software system cannot succeed and be sustained without the framework of practices and concepts to continuously assess, ensure, and improve its quality. The real challenge of software development is not merely to assimilate the right source code instructions, but to do so in a way that the code can be readily understood, maintained, and shown to be correct, and to achieve all of this within appropriate time and cost bounds.

1.1 Consistency, Complexity, and Change

Software quality assurance is, as we shall see, notoriously difficult to pin down as a concept. Every software development activity, from drawing up the requirements all the way through to deployment and maintenance, can at some level contribute to (or detract from) the quality of the final software system. Loosely put, the term 'quality assurance' refers to the various strategies and techniques that can be adopted to convey a certain level of confidence in the quality of the final system.

A successful quality assurance strategy should also ensure a degree of repeatability and reliability. It should provide safeguards and mechanisms that will not depend

N. Walkinshaw, *Software Quality Assurance*, Undergraduate
Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_1

on the capabilities of the individual developers. In other words, one overarching goal is to ensure *consistency*.

The challenge is to achieve this in the face of complexity and change. Complexity comes in a variety of forms – from the problem domain for which the software is being developed, the existing base of software libraries and frameworks, and the organisational structure within which the software is being developed. Change also comes in several forms. Capricious customers can continuously change their minds about requirements and priorities. Development teams can comprise large numbers of developers, often located in different locations and time-zones, tweaking the code base at the same time. During development, some activities can take longer than expected, others less.

A key challenge of quality assurance is to ensure that processes are in place that are capable of detecting, accommodating and controlling these various forms of complexity and change. This has to be achieved in a manner is not counter-productive. Quality assurance processes must not hamper development by placing an unnecessary burden on the developers, being overly restrictive in terms of flexibility, or causing unsustainable increases in time or expense.

Finding or developing a suitable quality assurance process is in and of itself a challenge. There are a huge number of potential frameworks, techniques, and tools for every aspect of software development. The most appropriate combination of these approaches invariably depends on various factors that are specific to a given organisation or project. Finding a suitable process therefore relies upon a capacity for introspection – the ability to experiment with different tools and techniques, to determine which approaches work, and which ones need to be refined or replaced.

1.2 Synopsis

This book aims to provide a concise introduction to the approaches to quality assurance that span every aspect of software development. To ensure concision, the book places an emphasis on the underlying foundations of modern quality assurance techniques – on emphasising the *whys* instead of the *hows*. Although the book will delve into some specific techniques, it does not seek to provide a comprehensive reference. The main objective is to provide the reader with a comprehensive understanding of where software quality fits into the development life-cycle (spoiler: everywhere), and what the key quality assurance activities are.

Although software development is relatively new as a discipline, many of the quality assurance principles and techniques have roots in other, more established disciplines. In order to fully appreciate these techniques it is necessary to be aware of their back-stories. This is not only useful from a pedagogical perspective. An appreciation of the historical roots of areas such as software process improvement and agile development supports an often under-appreciated but very important point: software development today has been largely shaped by some of the major advances in manufacturing that shaped the world as we know it.

The reader is encouraged to read the book in a sequential fashion from start to end. The chapters have been structured in such a way that we start from some fo the most general notions (e.g. quality and development process), and gradually home-in on the more specific activities, assuming knowledge of basic notions established in prior chapters.

Chapter 2: What is Software Quality, and Why Does it Matter?

It is impossible to understate the importance of software quality. Software is increasingly pervasive, controlling many devices that we depend upon, both as individuals and as a society. Poor software quality can have wide-ranging, and often unexpected consequences. In this chapter we look at some notable examples. We also use these examples to illustrate just how broad the term "software quality" can be, and explain why it is consequently so notoriously difficult to pin down. The rest of the chapter then looks at some of the key attempts over the last 40 years to formalise such definitions as software quality models.

Chapter 3: Software Development Processes and Process Improvement

The 'quality' of a product is dependent upon the process that was used to develop it – a realisation that underpinned the revolution in industrial manufacturing across the globe. In this chapter we start with some of the key innovations within manufacturing that shaped the principles that have influenced software engineering development processes. We then go on to cover three particularly popular software engineering processes: The Waterfall model, Iterative and Incremental Development, and finally Agile software development (with an emphasis on the SCRUM methodology). The chapter concludes by introducing frameworks to tailor and improve processes within the context of a particular organisation, with a focus on the CMMI framework.

Chapter 4: Managing Requirements and Code

Software development is ultimately about the ability to turn a set of (implicit or explicit) requirements into source code. Given that the requirements can be complex, and the development process can involve many developers, this process needs to be carefully managed. This chapter is split into two parts - the first part examines the challenge of managing requirements: How to elicit and capture them, dealing with questions such as security, how to prioritise them, and trace them to the various development artefacts. The second part is concerned with managing the corresponding source code as it evolves by adopting design and coding conventions and using version repositories.

Chapter 5: Planning Activities and Predicting Cost

Ultimately, sound management of a project requires a degree of planning. A typical project involves lots of different types of activities, that can require varying degrees of time and effort. It is necessary to predict how long different activities will take, and to figure out how to schedule them. Poor planning can lead to cost-overruns or even abandoned projects. In this chapter we look at some of the most popular techniques that can serve to attenuate this risk. The chapter starts by presenting two generic planning tools - the PERT technique and Gantt charts. This is followed by various techniques that can be used to predict the effort or cost required for a project by either using historical data with linear regression techniques, the off-the-shelf COCOMO model, and Planning poker.

Chapter 6: Testing

Testing - the execution of a software system to identify failures or faults - is a particular activity that is most obviously linked with quality assurance. As such, this chapter treats testing with a greater degree of detail. It introduces the key concepts that are involved in software testing. It then proceeds to cover white-box testing (testing techniques that are predicated upon access to the source code), including the various notions of code coverage that can be used to assess test adequacy, test generation techniques that aim to achieve coverage, the cases for and against said coverage metrics, and mutation testing. It concludes with an overview of black-box testing techniques, which cater for the scenario that the source code is not available. This part introduces specification-based testing, random testing, and Fuzz-testing techniques.

Chapter 7: Inspections, Reviews, and Safety Arguments

Software inspection sits alongside software testing as an activity that is specifically geared to identify faults in the source code. In contrast to testing, inspection does not depend on the ability to execute the code. Although the activity has its roots in the traditional ‘Fagan inspections’, this book only briefly covers these in favour of focussing on the less formal but more widespread activities that are broadly referred to as ‘Modern Code Review’, where version repository-based tools such as Gerrit and Pull-requests are used to orchestrate light-weight code reviews. This is followed briefly by an introduction to specific automated and manual code review approaches. The chapter concludes with an overview of an inspection approach that is specific to safety-critical systems, by introducing the concept of safety-arguments, and a graphical language that can be used to express these arguments.

Chapter 8: Measurement

Quality assurance relies fundamentally on the ability to measure: measuring the progress of the development, or the size, complexity, and cost of the system, the complexity of the code-base, etc. This chapter starts off by presenting some general background theory to measurement - what constitutes a valid measurement, and the different types of scales that can be used. It then moves on to cover a variety of software metrics, which analyse software size, modularity, and maintainability. It then concludes with a section that discusses the validity of metrics specifically in the context of software measurement, and how the Goal Question Metric technique can be used to ensure validity.

Chapter 9: Future Challenges and Opportunities

Software development is evolving at a fast pace. Software systems are continuously growing in terms of complexity and expense, which poses enormous challenges in terms of the overhead required by quality assurance activities. Software is increasingly incorporating data-intensive Machine Learning algorithms, the behaviour of which can be very difficult to constrain and validate. These algorithms are being placed into devices that have an increasing bearing on safety and society, with self-driving cars being a prime example. On the other hand, software development has also changed to alleviate some of the long-standing problems. This chapter examines some of these changes, and discusses what they mean for software development.

Key Points and Exercises

Each chapter will be concluded by a section called “Key Points” which summarises the main points that have been covered within the chapter.

Throughout the book you will find “exercises”:

> Exercises will be presented in a highlighted box, such as this one.

These are to be treated as reading-aids, to remind you of relevant parts in the book that have been covered previously, or to give you the opportunity to reflect on a particular topic and refer to related references.

Chapter 2
What Is Software Quality, and Why Does it Matter?

If you have ever debated the relative merits of an operating system, a programming language, or even a text editor[1], you will immediately appreciate the difficulties that arise when trying to assess and communicate about software quality. Software has many 'qualities', some of which are easy to assess (e.g. cost or ease by which it is installed), whereas others lie within the eye of the beholder (e.g. the aesthetics of a user interface). Different users can also priorities these qualities in different ways, can have very different expectations, and can easily come up with contradictory assessments.

Though it may be hard to characterise, software quality directly affects us all. Digital systems are pervasive; they control cars, aircraft, military weapons systems, our communication infrastructure, financial markets, etc. Stories of how the faulty behaviour of these systems can affect thousands of people appear in the news on an increasingly regular basis.

In this chapter we discuss first of all why we should care about software quality. We analyse some of the drivers that compel developers and organisations to push for quality assurance, and examine some of the predominant efforts to define software quality.

2.1 Why Care about Software Quality?

Digital technology pervades everyday life. Almost every aspect of our daily lives is affected or controlled at some level by a piece of software. On an individual level, most of us own a smart phone, and are signed up to social networks such as Facebook or Twitter. If you drive a car, a huge proportion of its functionality – acceleration, breaking, steering, air bag deployment, entertainment system, navigation, etc. – are controlled by software. Systems that are essential for everyday life –

[1] https://www.reddit.com/r/programming/comments/2v9uzx/vim_vs_emacs_is_it_really_a_competition/

N. Walkinshaw, *Software Quality Assurance*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_2

transport and communications networks and financial markets – are to a large extent controlled by software systems.

There have been countless instances where the (mis-)behaviour or escalating cost of a software system has led to severe (sometimes potentially disastrous) consequences. Huge amounts of money have been lost, private data has been released, and people have died. Software failures are ultimately the result of failures in quality assurance.

We will now examine some notable examples of software failures. You may be familiar with some of these from news reports about them. The goal is to illustrate two of the key motivations for caring about software quality: (1) That software can affect a broad range of stakeholders (who are often unaware that they are stakeholders), (2) that "poor quality" can manifest itself in a variety of ways.

NORAD's Nuclear Missile Defence System

We start by considering this passage of events, chronicled by Borning [24]:

> On Tuesday, June 3, 1980, at 1:26 a.m., the display system at the command post of the Strategic Air Command (SAC) near Omaha, Nebraska, indicated that two submarine-launched ballistic missiles (SLBMs) were headed toward the United States. Eighteen seconds later, the system showed an increased number of SLBM launches. SAC personnel called the North American Aerospace Defense Command (NORAD), who stated that they had no indication of attack.
>
> After a brief period, the SAC screens cleared. But, shortly thereafter, the warning display at SAC indicated that Soviet ICBMs had been launched toward the United States. Then the display at the National Military Command Center in the Pentagon showed that SLBMs had been launched. The SAC duty controller directed all alert crews to move to their B-52 bombers and to start their engines, so that the planes could take off quickly and not be destroyed on the ground by a nuclear attack. Land-based missile crews were put on a higher state of alert, and battle-control aircraft prepared for flight. In Hawaii, the airborne command post of the Pacific Command took off, ready to pass messages to US warships if necessary.

This obviously alarming situation was especially precarious because, once the alarm had been raised, a decision had to be taken immediately whether to respond with a counter-attack. This critical decision was based primarily on the evidence as presented by the computer system. As it turned out, the computer system was wrong:

> Fortunately, there were a number of factors which made those involved in the assessment doubt that an actual attack was underway. Three minutes and twelve seconds into the alert, it was canceled. It was a false alert.
>
> NORAD left the system in the same configuration in the hope that the error would repeat itself. The mistake recurred three days later, on June 6 at 3:38 p.m., with SAC again receiving indications of an ICBM attack. Again, SAC crews were sent to their aircraft and ordered to start their engines.
>
> The cause of these incidents was eventually traced to the failure of a single integrated circuit chip in a computer which was part of a communication system. To ensure that the communication system was working, it was constantly tested by sending filler messages which had

the same form as attack messages, but with a zero filled in for the number of missiles detected. When the chip failed, the system started filling in random numbers for the "missiles detected" field.

The problems were summarised in a report (with the worrying title) *NORAD's missile warning system: What went wrong* [37] , detailing a further litany of software / hardware errors that led to other false alarms.

We start with this example for the obvious reason that it is a system, controlled by software, where malfunctions have the potential of leading to a genuine disaster that can affect *anybody*, as opposed to a narrow group of defined stakeholder 'users' or 'operators'. Of course, the root cause of the failure was ultimately hardware (the faulty chip), but the faulty messages that it injected into the network were allowed to percolate because there was no error-checking in the communications infrastructure [24, 37], something that one would expect should be absolutely critical for messages as critical as 'the number of missiles detected in an incoming nuclear attack' .

Star Wars Missile Defence System

Despite the various technical challenges that had been faced by the NORAD Missile Defence Systems, in the mid-80s the US government sought to achieve a yet more ambitious feat. The new system, dubbed 'Star Wars' would not only detect a nuclear missile attack. It would also, upon detection, fire missiles that would automatically intercept and destroy any incoming missiles.

The system would have been highly dependent upon software. Software would have been required to target and steer missiles travelling at many times the speed of sound to hit other missiles travelling at a similar speed. Computation would have been guided by a vast network of sensors and other computers. The system would have to respond within very hard real-time constraints.

The programme never came to fruition for various reasons. However, what is of particular interest from this book's perspective is the fact that one of the core reasons was the perception that it was impossible to construct a software system that was going to be sufficiently reliable. This case was made forcefully by David Parnas [106] (a prominent Computer Scientist whose work we will encounter later on), who resigned from the panel of scientists responsible for implementing the system.

Toyota Unintended Acceleration Bug

In 2010 several news articles reported instances of Toyota cars failing to stop when drivers attempted to break[2]. Toyota suggested that this could be due to faults with the mats underneath the breaks (the break pedal perhaps getting stuck under the mat), or that it was simply driver error that resulted in the crashes. After numerous

[2] c.f. `http://www.bbc.co.uk/news/business-12072394`

deaths that apparently arose from this problem[3], there was a growing suspicion that the problem was not as trivial as Toyota were suggesting. Numerous victims and their relatives took Toyota to court, arguing that the fault lay with the car's software.

The court cases brought about a formal investigation of the source code itself (by a team at NASA [85]), and an analysis of the software development procedures, software metrics, and functionality by Professor Koopman of Carnegie Mellon University [86][4]. These investigations were not able to reveal the specific bug in the source code that would have led to the unintended accelerations. However, they were damning about the software quality - highlighting the fact that the source code was inscrutable, and that the software developers had evidently failed to adopt even some of the most basic procedures to guard against faults (such as versioning, or avoiding poor programming practice).

Volkswagen Dieselgate

In 2015, Volkswagen was rocked by the "Dieselgate" scandal. Investigators in the US discovered discrepancies, indicating that a large number of their cars (the number eventually came to around 11 million [44]) produced lower measures of harmful Diesel emissions (NO_x) during laboratory tests than they did during conventional use. After further investigation it turned out that Volkswagen had fitted the affected cars with 'defeat devices'.

The "devices" were ultimately embedded software components that controlled the Engine Control Unit (ECU). The ECU is a chip-set that, for the affected models, was supplied by Bosch. The ECU takes as input a range out sensor-readings from the car and, under control of the software embedded by the car manufacturer, modulates the behaviour of the engine. The 'ideal' ECU settings depend on a raft of factors, such as the engine temperature, atmospheric pressure, air temperature, etc. In the case of Volkswagen, the software controlling the ECU guessed when the car was undergoing a laboratory test and changed the parameters to the ECU to artificially reduce the NO_x emissions.

Was the "quality" of the software really at fault here? This depends on how we choose to define and measure quality. Ultimately, the software did what it was designed to do. It behaved "correctly". However, in doing so, it served only a small fraction of its stakeholders (i.e. Volkswagen). However, in doing so it also caused a lot of damage. The defeat devices have been estimated to have caused 59 early deaths in the US alone, and to have led to a 'social cost' of $450m [14]. A full recall of the affected cars by the end of 2016 is estimated to prevent approximately 130 early deaths, and an estimated $850m in social costs.

[3] `http://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/`

[4] Koopman's presentation on his investigation [86] is very engaging.

Boeing 787 "Reboot" Problem

In 2015 the US Department for Transportation issued an "Airworthiness Directive" (AD). The directive pertained to the Boeing 787 Dreamliner, which had been introduced in 2011. The abstract of the directive contains the following summary [49]:

> This AD requires a repetitive maintenance task for electrical power deactivation on Model 787 airplanes. This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode. This condition is caused by a software counter internal to the GCUs that will overflow after 248 days of continuous power. We are issuing this AD to prevent loss of all AC electrical power, which could result in loss of control of the airplane.

The "repetitive maintenance task" in essence amounts to 'rebooting' the plane periodically. The "overflow" occurs when a number (perhaps incremented continually throughout operation of the software without being reset or decremented) becomes too large to represent in memory. Overflows can result in exceptions, which can lead to behaviour which is generally unanticipated. In this case, it would lead to a complete loss of power in (and therefore control of) the aircraft.

The fault is surprising. For one, software in aircraft (especially software that is potentially critical to safety) is subject to some of the most stringent standards in existence (c.f. DO178-B/C [1]). Overflow errors are also sufficiently commonplace that they are routinely checked for, especially within the various mechanisms stipulated by the aforementioned standards. Nevertheless, even in the presence of such standards and scrutiny, it remains possible for such 'typical' faults to avoid detection and to be deployed on modern aircraft.

The fault is also, again, an illustration of why software quality assurance is so important. There is of course the obvious reason of avoiding any harm. However, there is also the practical problem here: software can be embedded into hardware circuitry, within planes that are distributed across the globe. This can make the act of fixing or updating software prohibitively expensive. This latter point of course also applies to the Vokswagen Dieselgate scandal, where the world-wide recall of the affected cars has run to billions of dollars.

Vulnerabilities and Cyber-attacks

The term 'software exploit' amounts to the deliberate abuse of a software bug, to perhaps gain unauthorised access to a software system, either to steal its data, or to influence its behaviour. As we have discussed previously, computer systems permeate everything – from individual smart-phones to transport / communications infrastructures (not to speak of missile defence systems). As such, the ability to gain unauthorised control of these systems can convey a huge amount of power to the attacker; exploits can be used to spy on individuals, or to disrupt the functioning of entire sectors of a government.

It is no surprise that, as a result, a market has emerged in software exploits. An undocumented software exploit (otherwise referred to as a 'Zero-Day') can be sold on the black market for tens or hundreds of thousands of dollars. Obtaining, discovering, and sometimes even creating such vulnerabilities has become a key tool for government agencies across the world. This was highlighted in 2017 when WikiLeaks released a tranche of thousands of confidential documents pertaining to the cyber-attack capabilities of the US Central Intelligence Agency (which they named Vault7[5]). This included dozens of zero-day exploits that could enable access to data on every-day consumer devices, from Apple and Android phones to the interception of audio recorded by smart TVs. This was accompanied by a series of leaks by a separate group (who called themselves 'Shadow Brokers'), which included a list of zero-day exploits and tools used by the US National Security Agency[6]. This list included a range of exploits for implementations of the SMB protocol (popular on Windows and Linux), along with utilities to collect information about potentially exploited or exploitable PCs.

The ramifications of this leak became apparent when, the following month, the WannaCry malware was released, which took advantage of one of the exploits released by the Shadow Brokers group. WannaCry ransomware was able to spread through networks through an exploit within the SMB protocol that resided on huge numbers of older Windows machines that had not been maintained with recent security patches. Once infected, the malware would prohibit access to the computer whilst demanding payment of a ransom of $300. The attack affected hundreds of thousands of computers across the globe, but took a particular hold within the UK National Health Service (which has a particularly large number of legacy Windows XP PCs), ultimately leading to a paralysis that saw many thousands of patients being turned away from GPs, and many cancellations of operations on patients in hospitals.

The WannaCry malware shared several hallmarks with the notorious StuxNet malware, which had been deployed several years earlier, in 2010 in Iran. StuxNet [88] found its way onto the computer infrastructure that controlled centrifuges, which were key to processing Uranium for the Iranian nuclear programme. The malware caused the centrifuges to spin out of control, causing significant damage to the centrifuges and significantly denting Iran's nuclear programme in the process. The attack became notorious because of its complexity; the number of ways in which it could spread from one computer to another, and sophistication by which it ended up manipulating the hardware of the centrifuges. A dossier on StuxNet by Symantec documented (amongst sundry others) the following features:

- Self-replicates through removable drives exploiting a vulnerability allowing auto-execution.
- Spreads in a LAN through a vulnerability in the Windows Print Spooler.
- Spreads through SMB by exploiting the Microsoft Windows Server Service Vulnerability.
- Updates itself through a peer-to-peer mechanism within a LAN.

[5] `https://wikileaks.org/ciav7p1/`

[6] `https://github.com/misterch0c/shadowbroker/`

- Exploits a total of four unpatched [zero-day] Microsoft vulnerabilities, two of which are previously mentioned vulnerabilities for self-replication and the other two are escalation of privilege vulnerabilities that have yet to be disclosed.
- Contacts a command and control server that allows the hacker to download and execute code, including updated versions.
- Fingerprints a specific industrial control system and modifies code on the Siemens PLCs to potentially sabotage
- Hides modified code on PLCs, essentially a rootkit for PLCs.

Whereas WannaCry had merely exploited one particular vulnerability, StuxNet was particularly surprising for its time because it combined several zero-day vulnerabilities in a highly targeted way. This highlights the power that can be afforded to governments by building up repositories of zero-day exploits. However, the release of the WannaCry malware (which had been created from a leak of such exploits) also highlighted the enormous danger that such exploits can pose to society (as well as the ethical questions that have to be confronted by agencies who decide not to disclose their existence).

Software quality and security go hand-in-hand. Exploits in essence take advantage of bugs – lapses in quality that were not prevented, detected, or tracked by the developers. The existence of a thriving market in exploits demonstrates that there is a will (both by some individuals and organisations) to undermine the privacy and even safety of users. This places an onus on developers to ensure that the opportunities for such exploits are minimal.

Code Quality and Maintainability

Successful software tends to evolve at a rapid pace. Large numbers of developers can end up simultaneously updating large numbers of code files, often to different, even conflicting, ends. If this change is not managed properly, the design and readability of the code base can start to deteriorate. It can become harder to understand, increasing the risk of introducing bugs, and the cost of maintaining the system further down the line.

One extreme example of the scale and complexity of a code base can be found at Google [111]. The Google code base contained in 2016 approximately 1 billion files, including 9 million files of source code, comprising approximately 2 billion lines of source code. It had a history of approximately 35 million commits to the version repository spanning Google's 18 years of existence. In 2016 it was being updated by approximately 40,000 commits per day.

Although Google has managed its code quality, there are numerous examples of organisations where the opposite has happened. A notorious example of this was Denver Airport's automated baggage handling system [40], which was introduced with a degree of fanfare in 1994, at an initial cost of $186 million. The system was unfortunately beset by glitches, and none of the airlines apart from United (the airport's busiest airline) switched over to the system. However, the continued problems with the system led to daily maintenance costs of $1 million. Ultimately, United de-

cided to switch the system off, and to resort to normal 'manual' baggage-handling, because they realised that this would save them $1 million per month.

2.2 What Drives Software Quality Assurance?

There are many reasons that an organisation might seek to ensure that its software is of a high quality. These often become apparent when quality fails (as we have seen above); poor software quality can have a myriad of consequences for users, businesses, and other stakeholders. Some of the key drivers are listed below.

Exercise: *Before we read some suggestions, try to list what you consider to be key, overriding reasons that might drive an organisation to ensure that the software it develops is of a high quality.*

Reputation

Software developers and their organisations rely on reputation. A poor quality product (or family of products) can be hugely damaging for business. Software bugs can have immediate impacts on custom, especially in customer-facing industries. The automotive software problems with Volkswagen and Toyota have led to an enormous amount of negative publicity.

Limiting Technical Debt

Cost is an overriding factor in software development. Poor quality software tends to be expensive to develop and to maintain, which can have a detrimental effect the organisations that end up maintaining the software in the longer term. These costs are often referred to as 'Technical Debt'; the organisation in charge of the software needs to invest a disproportionate amount of resources into maintaining and running the software to make up for (and to try and remedy) poor design and implementation decisions.

Software Certification

Depending on the domain of the software (e.g. software for civilian aircraft or nuclear power stations), the development and use of software might require some form of certification, which can often require evidence of the application of various quality control and assessment measures. For example, software in modern civilian air-

craft often has to be certified to the DO178 standard [1], which imposes requirements on every aspect of the software development lifecycle.

Organisational Certification

As we shall see, the procedures and structures that are employed for software development within an organisation can have a huge bearing on the quality of the software that it produces. There are various ways by which to categorise the extent to which an organisation employs good practice. International certification procedures and standards such as CMMI[7] and ISO9001[8] (more about these later) exist, so that software development organisations can ensure and continuously improve their "capability" to develop high quality software. Being certified to such standards can play an important role when companies bid for software development contracts. For example, the US Department of Defence has requirements, grounded in CMMI, on the maturity of the software development processes employed by its contractors.

Legality

Depending on the country, there may be overriding legal obligations that apply to organisations that use software. For example, in the UK, organisations have to demonstrate that the risk posed by their technology (this includes software) is "As Low As Reasonably Practicable" or "ALARP"[9]. In other words, every "practicable" measure must have been taken to demonstrate that (in our case) the software system does not pose a risk to its users.

Moral / ethical codes of practice

Even in cases where a software system is not covered by industrial certification and legislation, and where its failure is not necessarily business or safety-critical, there can remain a moral obligation to the users. Professional organisations such as the American Computer Society (ACM) have explicit ethical guidelines and codes of practice[10], with statements such as "Software engineers shall act consistently with the public interest". This clearly implies that they ought to do whatever possible to maximise the quality of their software and to prevent it from containing potentially harmful bugs.

---

[7] `https://www.sei.cmu.edu/cmmi/`

[8] `http://www.iso.org/iso/home/standards/management-standards/iso_9000.htm`

[9] `http://www.hse.gov.uk/risk/theory/alarpglance.htm`

[10] `http://www.acm.org/about/se-code`

Exercise: *Read through the ACM ethical guidelines, and think about how these align with the development of the Volkswagen Dieselgate software mentioned above.*

2.3 Defining "Software Quality"

Exercise: *Write down 6 attributes of a product (possibly a software system) that in your opinion relate to its quality.*

Software quality is notoriously difficult to capture. Although you might have found the above exercise straightforward, it is likely that your definition would not necessarily fit with that of your colleagues. Different people value different attributes of a software system. Some favour reliability, others favour usability, or the number of features, or the cost. In this subsection we will look at some of the overriding attempts to define quality, and some mechanisms that can be used to capture it.

2.3.1 The Challenge of Defining Quality

In 1931, Walter Shewhart [121] noted that the notion of "quality" is multidimensional. When we speak of the "quality" of a product, we are commonly referring to a multitude of individual qualities. He used the example of a relay switch, where its 'goodness' might be reflected in the qualities of capacity, inductance, and resistance.

The challenge of defining quality ultimately lies in determining what these individual 'qualities' are. The wrong choice of qualities can ultimately undermine a product, because limited resources are focussed on the wrong areas. Does one aim for a product with lots of features and accept that some of them may not function perfectly? Or does one aim to produce a focus with relatively few features, but that are implemented well? Should focus be placed on security or usability?

Over the past century, several schools of thought have emerged to answer such questions. Chief amongst these are the philosophies of Joseph Juran (a colleague of Shewhart's), and Phil Crosby, two key figures in the broad area of quality assurance. The former placed an emphasis on satisfying the user, whereas the latter placed an emphasis on satisfying fixed, objective requirements:

- **Fitness for use (Joseph Juran):** Joseph Juran had been a contemporary of Walter Shewhart, and embodied the idea that the quality product revolves around its fitness for use [78]. He argued that, ultimately, the value of a product depends on the customer's needs. Crucially, it forces product developers to focus on those

Fig. 2.1 Walter Shewhart (1891-1967) was an American Physicist and Statistician. He is known as the father of 'statistical quality control' and his work had a profound impact on quality control in the engineering and manufacturing sector, and eventually on software engineering as well. © Nokia Corporation, reused with permission.

aspects of the product that are especially crucial (the *vital few* objectives) as opposed to the *useful many*.

- **Conformance to Requirements (Phil Crosby):** Phil Crosby embodied a different tone. He defined quality as "conformance to requirements" [38]. His opinion was that quality can be achieved by the disciplined specification of these requirements, by setting goals, educating employees about the goals, and planning the product in such a way that defects would be avoided.

On the one hand we have Crosby's view that quality is an intrinsic property of the product. On the other hand we have Juran's view that quality is perceived by the user. As such, Crosby's view tends to be referred to as 'product-centric', whilst Juran's view is referred to as 'user-centric'.

Occasionally, these two definitions can come into tension. This is nicely illustrated by the Volkswagen Dieselgate scandal discussed previously. Viewed from a strictly product-centric perspective, the software is doing what it is supposed to do. However, viewed from a user-centric perspective, the opposite is true. Users clearly did not want a car that had cheated its way through emissions tests. This is corroborated by the fact that, in the UK at the time of writing, an "10,000 owners had already expressed an interest in suing VW"[11].

[11] `http://www.bbc.co.uk/news/business-38552828`

Exercise: *Take the six quality attributes you wrote down earlier, and divide them into what you would consider to be product-centric properties, or user-centric.*

2.3.2 Quality Models - a Historical Perspective

It is important for software developing organisations to have a well-defined set of principles that can be used as a basis for discussing and assessing software quality. Given the various ways in which quality can be defined, there have been a multitude of efforts to formalise definitions. These formalisations are called *quality models*. In this section, we will cover some of the most prominent models.

The term 'formalisation' is perhaps a bit of an overstatement. Quality models are rarely formal in the mathematical sense. Instead, they tend to take the shape of a structured hierarchy - a tree. Terms at a higher level in the tree tend to correspond to more abstract concepts that are deemed to be of relevance from a quality perspective, and these tend to be subdivided into more granular, low-level concepts.

Over the years there have been many different attempts to create 'definitive' models. The first attempt was made by Jim McCall in 1977 [36], whose model was developed for software development within the US Airforce. He defined a (hierarchical) set of "Quality Factors", shown in Figure 2.2.

- **Product Operation Factors**
  - Reliability
  - Efficiency
  - Integrity
  - Usability
  - Correctness
- **Product Revision Factors**
  - Maintainability
  - Flexibility
  - Testability
- **Product Adaptability Factors**
  - Portability
  - Reusability
  - Interoperability

Fig. 2.2 McCall's Quality Model

A couple of years later, in 1979, Barry Boehm extended McCall's model [22]. He contended that, although McCall's model was useful because it made the various quality concerns explicit, it was difficult to use. Specifically, he argued that it was difficult to quantify the extent to which a product fulfilled its quality model - (i.e. to answer the question "how good is it?").

To address this, he suggested that the low-level nodes in the hierarchy should be attached to specific *metrics* – techniques that could be used to provide a quantitative value for a given aspect of the system (we will cover these in Chapter 8). He broadly maintained the three high-level categories proposed by McCall, but substituted some of the leaf nodes with definitions that were more explicitly measurable (whilst eliminating those nodes that were not). His model is shown in Figure 2.3.

His important contribution was not so much the model itself (as you will see, these have changed over the years). It was however this principle that, in order to be useful, a model had to be *measurable*. That any aspect of a model had to be quantifiable for the model to be of value to an organisation.

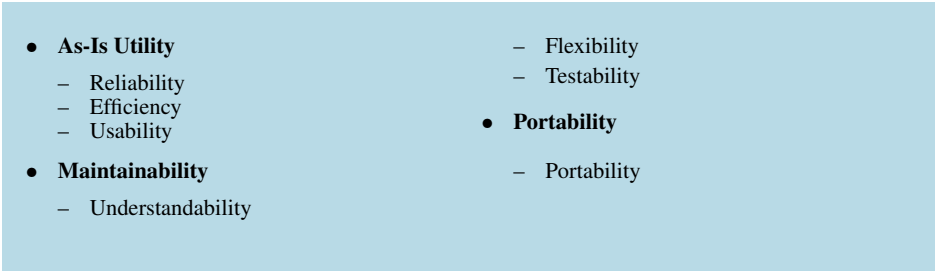

Fig. 2.3 Boehm's Q-Model

Boem's Q-Model and McCall's prior model formed the basis for subsequent international standards that were used to define software quality. One notable example is ISO9126, which was published in 1991. This is shown in Figure 2.4.

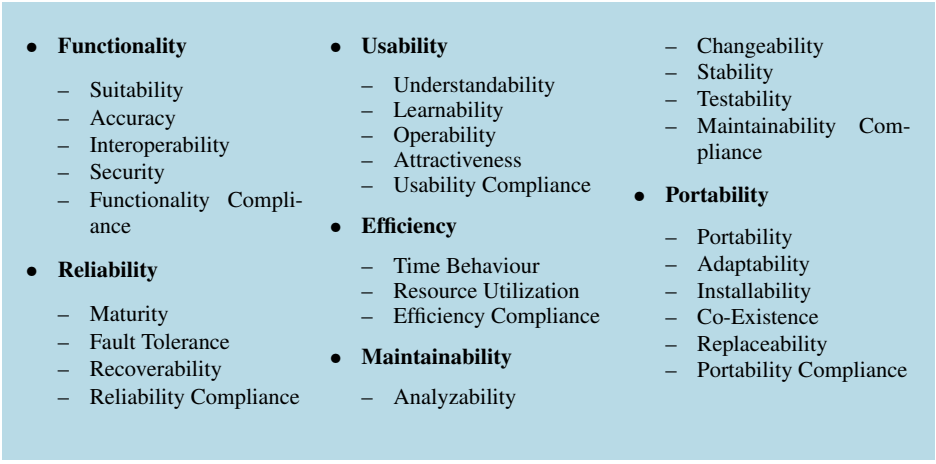

Fig. 2.4 ISO 9126

- **Functional Suitability**
  - Functional Completeness
  - Functional Correctness
  - Functional Appropriateness
- **Performance Efficiency**
  - Time Behaviour
  - Resource Utilisation
  - Capacity
- **Compatibility**
  - Co-existence
  - Interoperability
- **Usability**
  - Appropriateness
  - Realisability
  - Learnability
  - Operability
  - User Error Protection
  - User Interface Aesthetics
  - Accessibility
- **Reliability**
  - Maturity
  - Availability
  - Fault Tolerance
  - Recoverability
- **Security**
  - Confidentiality
  - Integrity
  - Non-repudiation
  - Authenticity
  - Accountability
- **Maintainability**
  - Modularity
  - Reusability
  - Analysability
  - Modifiability
  - Testability
- **Portability**
  - Adaptability
  - Installability
  - Replaceability

Fig. 2.5 ISO/IEC25010

The ISO9126 standard was later replaced by ISO/IEC25010 [5, 97], shown in Figure 2.5. This has since, again, been revised and built upon in recent years. There are too many models to provide a complete reference in this book, and indeed that would miss the point. The bottom line is that the list of software quality concerns is continuously growing to respond to changes in technology and the way in which it is used. Quality models, which are often enshrined in international standards, have therefore become increasingly elaborate.

- **Safety** - Ability to operate free from risk of causing unacceptable levels of harm.
- **Reliability** - Behaviour of the (sub-) system in a manner that is predictable and not liable to lead to faulty behaviour.
- **Availability** - Ability to perform tasks when requested.
- **Resilience** - Sensors are coherent and system is able to function in the case of a fault, damage, or error.
- **Security** - Ability to function without inadvertent or intentional external influence.

Fig. 2.6 PAS-754 Software Trustworthiness Aspects

So far, the quality standards listed have sought to encompass every conceivable angle of quality (it is the proliferation of these qualities that explains their continual growth). The final standard that we include here bucks this trend, by only focussing on those aspects of quality that are of concern to the *operation* of the software in

question. The PAS-754 standard, shown in Figure 2.6 is not presented as a hierarchy of factors (though one can easily see how the high-level factors such as safety could be broken down into sub-factors).

As is the case with most aspects of quality assurance, there is no single model that represents the "best" choice. Quality requirements can vary depending on the usage domain, the nature of the users, and the software development organisation. xAs a consequence there has been a gradual movement towards the development of customised quality models. Approaches such as QUAMOCO [131] have been developed which, starting from established models such as ISO/IEC25010 (Figure 2.5), enable organisations to refine and customise quality models to their specific context and needs.

2.4 Key Points

- **Software is pervasive.** Everyday devices, from personal / household items such as phones and TVs are controlled by software. Software is central to financial markets, energy, communications, and transport infrastructure.
- **Poor software quality can have far-reaching consequences.** Faults in a software system can lead to incorrect software behaviour. Some of the examples we covered illustrate just how far-reaching those consequences can be – from enabling far-reaching cyber-attacks to raising false alarms of a nuclear strike. Even if the software is functionally correct, a poorly constructed software system can become difficult and expensive to maintain, and raise the risk of the introduction of critical faults over the longer term.
- **There are several drivers for software quality.** Software has many responsibilities to many stakeholders. Developers and organisations that deploy software systems have ethical and legal responsibilities towards the users. There are also business considerations (poor quality software can, if it is customer-facing, deter customers, and can become increasingly expensive to run over the longer term).
- **There is no single canonical definition for "software quality"**. The question of what constitutes a "high quality" system is continuously shifting, and depends to a large extent on the broader context within which it is deployed.
- **There are two prevailing perspectives on software quality**: Juran's user-centric perspective, and Crosby's product-centric view. Both of these viewpoints pre-date software-engineering as a discipline, and were used to assess quality in a manufacturing context.
- **Software quality is commonly formalised in software quality models.** There have been numerous models that have been proposed over the decades. These models have grown over time. It is important to be able to, for a given model, be able to link specific concepts of quality to measurable artefacts in the system (we will explore this in more detail in Chapter 8).

Chapter 3
Software Development Processes and Process Improvement

Software systems can be extremely complex. There can be numerous stakeholders, with different (often conflicting) requirements. There can be, as we have already seen, a raft of (often conflicting) quality concerns that need to be satisfied. However, time and resources are often limited.

For a software development project to be successful, it is vital that the development team are able to organise themselves in such a way that they are able to maximise their productivity within the constraints of a project. This must include some guidance in addressing the key questions that arise, such as:

- What are the (most important) requirements?
- What specific tasks need to be achieved in order to fulfil these requirements?
- By when (and in what order) should these tasks be carried out to ensure that the system is delivered on time?
- What is the modus-operandi; what is the specific sequencing of development activities within the team? For example, how should a design be produced, or how should developers make their actual contributions to the code base in such a way that conflicts with other contributions are minimised?

This must all be accomplished in the face of a raft of challenges. The circumstances under which a software system is to be deployed can change, or a capricious client can change their mind, leading to sudden changes in requirements. Development activities can take longer than anticipated. Staff may be making contributions to the code base from geographically disparate locations, and may not be in-sync with each other.

Many of the essential organisational questions can be addressed by adopting a 'development process'. A development process (often referred to as a "Software Development Lifecycle") refers to a particular way of organising the activities that span the development (and sometimes deployment) of a software system. They set out an idealised work-flow that can be used to orchestrate activities within an organisation or development team.

The choice of development process is critical when it comes to assuring the quality of a software system. A failure in quality – the existence of bugs, or cost-

N. Walkinshaw, *Software Quality Assurance*, Undergraduate
Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_3

overruns, can usually be blamed (at least in part) on a poor choice of development process, or a lack of adherence to it. Accordingly, if an organisation wishes to improve the quality of the software it produces, an obvious starting point is to look at improving the underlying software development processes.

Ultimately, this chapter is about the relationship between the quality of a product, and the process that was used to develop it. One key to ensuring quality lies in adopting a scientific approach to the development process; devising a process, trying it, observing the quality of the result, and using this as a basis for refining the process. It was this systematic approach that fired the revolutions in manufacturing that spurred much of the technological progress throughout the 20th century.

This chapter starts by providing a historical introduction to manufacturing, focussing on the innovations in mass-production that linked processes to the quality of the products they produced. In this context we focus on concepts such as PDCA - linking iteration to process improvements, which went on to form the basis for iterative software development techniques such as Agile software development. We then introduce three popular software development process families - the Waterfall process, Iterative and Incremental software development and its offspring - the Spiral model and agile software development. We conclude the chapter with an overview of Software Process Improvement (relating this to manufacturing process improvement frameworks which we cover at the beginning of the chapter).

3.1 Process and Process Improvement in Manufacturing

In the abstract, software engineering can simply be seen as the product of a process – a sequence of activities that culminate in a software system. From this perspective, software engineering is comparable to any other manufacturing activity. Crucially, the very innovations that led to step-changes in manufacturing – breaking complex processes down into simple ones, and continuously improving these processes – have strongly influenced today's most successful software development practices. It is for this reason that we start this chapter with a (very) brief look at some of the key points in the history of manufacturing.

3.1.1 The Industrial Revolution

The birth of manufacturing as we know it today – the industrial production of complex products and materials – occurred with the Industrial Revolution in Great Britain in the late 18th and early 19th centuries. The arrival of steam power enabled complex products to be produced by machines instead of humans. Factories were constructed to manufacture a huge range of materials and products (especially iron, steel, and textiles).

The key to this leap was mechanisation. What had been complex, fiddly activities carried out by humans (e.g. weaving thread into a textile) were rethought in such a way that they could be carried out automatically, by machines. If a task remained too fiddly to automate, then it would still be sufficiently simple for a relatively unskilled human to carry out. Reducing dependence upon skilled humans made the activity cheaper.

Exercise: *Aside from being cheaper to run, what might the other benefits of using machines to manufacture products instead of relying upon skilled humans?*

In the early 19th century, many of the innovations that spurred the British Industrial Revolution were replicated around the world. In the US this was epitomised by the emergence of the *assembly line*. As had happened with mechanisation throughout the industrial revolution, assembly lines enabled a complex activity to be broken down into simple steps. However, assembly lines lifted this to a higher level – instead of focussing on specific activities (such as weaving), assembly lines enabled entire manufacturing processes to be broken down into their simple, constituent steps.

Fig. 3.1 Workers on the first moving assembly line put together magnetos and flywheels for Ford cars. Taken by unknown photographer in 1913[2].

One of the early proponents of the assembly line was Henry Ford. Ford used assembly lines to systematise the entire production of cars (illustrated in Figure 3.1). Although much of the production was not fully automated, the individual steps that required human input were sufficiently simple that they did not require skilled

[2] Source:
https://www.mediawiki.org/wiki/File:Ford_assembly_line_-_1913.jpg

labour. Assembly lines would (often on a rolling band) deliver partly assembled components to a worker, who would carry out a specific (often menial) task, before it was passed along to the next worker, for the next task.

In this context of mass-production, it soon became apparent that there was a direct link between the profitability of a company and its manufacturing processes. Unnecessary steps in the manufacturing process could render the product more expensive or complex than it needed to be. Failures to safeguard against mistakes would lead to a poor-quality product, which would make it less competitive against rivals.

3.1.2 Plan Do Check Act

It was against this backdrop that Walter Shewhart (as introduced in Chapter 2) worked on quality assurance, and specifically the issue of process improvement. His employers at Bell Telephone had been struggling to address problems with some of their transmission equipment (which was usually buried underground). They noted that the problems were especially difficult to pinpoint and eliminate because the process that had been used to manufacture the equipment varied across the company.

Shewhart observed that the key to eliming these manufacturing problems was to, at least at first, eliminate the variability in the manufacturing process. If a *consistent* process was used to manufacture the product, it would follow that any problems with the resulting product would have a specific cause, making them easier to eliminate by refining the process.

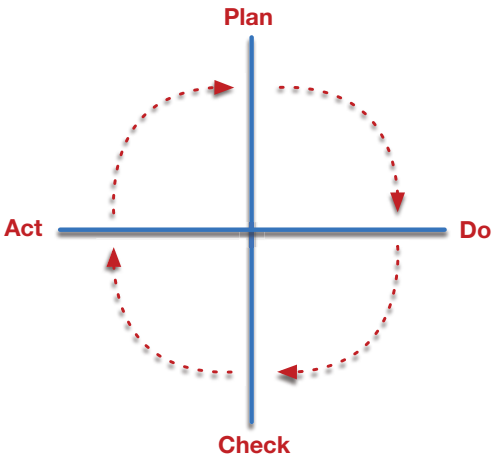

Fig. 3.2 Shewhart's PDCA cycle. This forms the essential foundation for any process improvement framework.

One of Shewhart's key innovations was the "Shewhart Cycle", which was later popularised as the *Plan-Do-Check-Act (PDCA)* cycle, shown in Figure 3.2.

- **Plan:** Set out the goals of the project, and the set of procedures you intend to apply to achieve them.
- **Do:** Carry out the planned set of procedures. Take note of any problems you encounter along the way.
- **Check:** Review the problems that were encountered during the project, along with any problems that have arisen with the final product. For each problem, try to pinpoint a reason or cause.
- **Act:** Try to identify changes to the procedures that would eliminate or mitigate the problems that you have identified.

Although Shewhart is credited with PDCA, it was brought to wide attention by a colleague of his, Edwards Deming (also a leading figure in Quality Assurance). As we will see in later sub-sections, this approach to retrospective assessment and improvement forms the basis for today's software process improvement techniques.

3.1.3 Quality-Driven Manufacturing in Japan

In the wake of the second world war, Japan made a concerted effort to rebuild the country as an economic power (as opposed to a military one). International experts in manufacturing were invited to consult for Japanese corporations, run courses in quality management, and were often offered positions in Japanese universities. Many luminaries in quality assurance migrated from the US to Japan, including familiar names – Walter Shewhart, Joseph Juran, and Edwards Deming. They (especially Deming) were to a large part credited with spurring what became known as the Japanese "economic miracle".

One especially celebrated quality assurance 'framework' to arise from this period is the Toyota Production System. In the aftermath of the war, Toyota were struggling as an organisation, and there was a suspicion that this was due to inefficiencies in its manufacturing processes. This was put into perspective when their manager at the time visited Ford's Rogue manufacturing plant in the US. Whereas Toyota had constructed 2,500 vehicles in their entire history, he observed with astonishment how Ford were producing 8,000 vehicles per day.

As a consequence of this realisation, there was a drive within Toyota to overhaul its manufacturing processes to increase production capacity. This was achieved with an overriding aim of *eliminating waste*. The Toyota Production System (TPS) consisted of a variety of maxims and guidelines that revolved around this aim, and remain an active part of the development ethos to this day. TPS is underpinned by the following 12 principles[3].

[3] `http://blog.toyota.co.uk/13-pillars-of-the-toyota-production-system`

- 'Kaizen' - Continuous Improvement: Staff are encouraged to continuously look for areas in which procedures can be improved and refined.
- 'Just In Time' - this term was coined for the TPS, and means that products should only be developed if they are required and when they are required. This gave rise to the idea of so-called "pull-through manufacturing".
- 'Jidoka' - develop techniques to capture faults as close to their source as possible.
- 'Poka Yoke' - prevent errors from occurring in the first place by embedding prevention mechanisms into the manufacturing process.
- 'Hansei' - learn from mistakes in order to prevent them from recurring.
- 'Andon' - enable workers to immediately highlight what they perceive to be a threat to [vehicle] quality by providing alert mechanisms.
- 'Hejunka' - have the correct number of parts required to build a vehicle.
- 'Genchi Genbutsu' - the best way to solve a problem is to be present and to see it for yourself.
- 'Nemawashi' - decisions should be arrived at as a team, not dictated by individuals.
- 'Kanban' - a board to communicate the state of the manufacturing system[4].
- 'Muda, Muri, Mura' - eliminate waste.
- 'Genba' - understand the work load on individuals, and ensure that processes are as transparent as possible to facilitate this understanding.

Exercise: *Many of the TPS maxims are recognisable in modern software development (in various guises). Based on your current knowledge, see if you can relate any of them to current software engineering practice.*

Exercise: *Relate relevant parts of TPS to PDCA.*

In the 80s, and the aftermath of the Japanese "economic miracle", there was a significant stagnation in the economies of Europe and the US. In the face of competition from highly successful Japanese manufacturers the UK became (for the first time since the industrial revolution) a net-importer of goods. This led to much introspection, both within the UK and the US. There was an admiration of the disciplined use of quality assurance and improvement techniques in Japan, and an acknowledgement that a similar level of discipline and innovation was required in Europe and the US if their economies were going to compete.

As a good illustration of this introspection (and of the acknowledgement of the role of quality management), watch the 1980 NBC documentary *If Japan Can, Why Can't We?*[118]. The film features several leading figures who were so instrumental in the success of the Japanese economy. At one point early on in the documentary,

[4] Anybody from an Agile software development background will be familiar with the notion of a Kanban board. This was directly inspired by the TPS, and is something we will come back to later on in this book.

the narrator states the following, which nicely captures the sentiment of the documentary:

> In a recent American study of one type of integrated circuit [. . .] the *best* American product failed *six* times more often than the best Japanese product - *six* times. Built in quality and reliability pushed Japanese productivity up, and American productivity down.

Fig. 3.3 Aftermath of the Challenger explosion. NASA, 1986, reprinted with permission[5].

This crisis of confidence was brought into sharp relief in January 1986, when the Challenger space shuttle exploded shortly after lift-off, killing its seven crew members (six astronauts and a school teacher). In the aftermath, a commission was formed, including notable pilots, astronauts, politicians, and one scientist, to investigate the cause of the explosion. The scientist in question was Richard Feynman, a nobel-prize winning physicist. Whereas the commission were reluctant to publish findings that might embarrass NASA, Feynman embarked on a forensic examination of the accident[6]

Feynman's findings did indeed end up causing considerable embarrassment. They did so because they unflinchingly exposed failures with the *processes* that were used within NASA. In particular, he exposed the absence of proper lines of communication, whereby it became routine for different groups of staff (e.g. engineers and management) to have entirely contrasting opinions about the safety of a shuttle, or component within the shuttle.

---

[5] Source:
`https://www.mediawiki.org/wiki/File:Challenger_explosion.jpg`

[6] In 2013 the BBC produced *The Challenger*, an excellent documentary of Feynman's part in the Rogers Commission.

Exercise: *Read the appendix. It is brilliantly written, managing to communicate the various complexities and technicalities to a general audience, but managing to remain forensic and convincing throughout.*

Here is the opening paragraph of his final report (this was only included in the final Rogers Report [27] as an appendix to minimise embarrassment to NASA):

> It appears that there are enormous differences of opinion as to the probability of failure with loss of vehicle and of human life. The estimates range from roughly 1 in 100 to 1 in 100,000. The higher figures come from working engineers, and the very low figures come from management. What are the causes and consequences of this lack of agreement? Since 1 part in 100,000 would imply that one could put a shuttle up each day for 300 years expecting to lose only one, we could more properly ask "What is the cause of management's fantastic faith in the machinery?"

Exercise: *List some of the TPS maxims that, if implemented, might have prevented the Challenger disaster.*

3.1.4 Total Quality Management

In Europe and the US it became generally accepted that, in order to compete economically with Japan and to prevent disasters such as the Challenger disaster from recurring, there needed to be a concerted effort to improve quality assurance processes. One technique to arise from this drive became known as *Total Quality Management* (TQM). Although TQM has taken on a number of meanings, it broadly represents an approach to management that aims for long-term success by linking quality with customer satisfaction. Its essential principles can be summarised as follows [80]:

- **Customer focus:** The objective is to achieve total customer satisfaction. Customer focus includes studying customer's wants and needs, gathering customers' requirements, and measuring and managing customers' satisfaction.
- **Process:** The objective is to reduce process variations and to achieve continuous improvement. This element includes both the business process and the product development process. Through process improvement, quality will be enhanced.
- **Human side of quality:** The objective is to create a company-wide quality culture. Focus areas include leadership, management commitment, total participation, employee empowerment, and other social, psychological, and human factors.
- **Measurement and analysis:** The objective is to drive continuous improvement in all quality parameters by a goal-oriented measurement system.

Exercise: *Can you relate elements of PDCA and Toyota's TPS to the TQM principles?*

Exercise: *In what way does TPS relate to the Joseph Juran's take on quality?*

Exercise: *Read Feynman's appendix of the Rogers Commission Report. Attempt to identify aspects of TQM that might* not *have been adopted within NASA at the time.*

TQM is the embodiment of a 'Process Improvement' framework. There is a focus on what customers value, and this is used to drive improvements in the underlying processes used to manufacture products. Employees are seen as intrinsic to the manufacturing process, and are relied upon to assess a process, and to implement change if required.

The use of TQM per-se has waned in the last couple of decades. It has however inspired several derivative processes that have become widespread, especially in Europe and in the US. These include Motorola's Six Sigma approach[7] and the ISO9000 standard[8]. These are beyond the scope of this book. However, in the software development domain, TQM also influenced the development of the widely used CMMI approach, which is discussed later in this chapter.

3.2 The Software Development Process

In the earlier days of software development (pre-70s), the very idea that software was something that had to be "engineered" was not widely understood. Software development was seen as a simple, quasi-administrative process. The widespread view was that any reasonably able engineer (in the traditional sense of the word) could turn their hand to programming. This, after all, merely consisted of "building a flow-chart and turning it into computer code" (the phrasing comes from this charming 1962 film on "Programming" by Bell Labs[9]).

Clearly this attitude failed to appreciate just how complex software development can be. Nevertheless there were several notable successes. Figure 3.4 shows Margaret Hamilton, who was the software development lead for the Apollo 11 moon landing mission. The software was highly complex for its time (it comprised approximately 450,000 lines of code and was responsible for, amongst other things, managing the allocation of power to different parts of the spacecraft). The source code

[7] `https://en.wikipedia.org/wiki/Six_Sigma`

[8] `https://en.wikipedia.org/wiki/ISO_9000`

[9] `https://www.youtube.com/watch?v=dFZecokdHLo`

is now available online[10], and is worth a read, especially for the code comments, which have also been transcribed. These convey a good impression of how the developers struggled at times to fully understand some of the details of the code. A good example is the module LUNAR_LANDING_GUIDANCE_EQUATIONS.agc, where comments for variables include "TEMPORARY, I HOPE HOPE HOPE", and "Numero Mysterioso". Nevertheless, the mission succeeded and the software did its job. Considering that this took place before the emergence of software engineering as a discipline and before a widespread appreciation of software complexity, this was an astonishing feat.

Fig. 3.4 Pioneer of "Software Engineering" Margaret Hamilton standing next to print-outs of the source code for the Apollo 11 moon landing mission. Draper Laboratory 1969; reprinted with permission[12].

This somewhat dismissive attitude to software development gradually changed as the engineering challenge was exposed by an increasing prevalence of notable software-related problems. The complexity of software systems, coupled with the fact that they often required continuous development and maintenance, inevitably started to lead to problems. This realisation was perhaps best symbolised by the 1968 NATO conference on Software Engineering, where the term "software crisis" was coined. The concept was summarised in Edsger Dijkstra's 1972 Turing award lecture [43]:

[10] `https://github.com/chrislgarry/Apollo-11`

[12] Source: `https://www.mediawiki.org/wiki/File:Margaret_Hamilton.gif`

> "The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

The problem was subsequently elaborated by a collection of essays on the challenges of developing software from an industrial perspective (now hailed as landmarks in software engineering literature) by Fred Brooks [25]. He drew upon his experience of working with IBM on the development of their OS/360 operating system.

Exercise: *The most famous of Brooks' essays is entitled "No Silver Bullet - Essence and Accident in Software Engineering". Read this essay to obtain what remains a concise and highly relevant insight into the core software engineering challenges [and thus the challenges to quality assurance].*

This widespread recognition of a crisis gave rise to several efforts to systematise software development. There was a general recognition that good engineering practices ought to be developed to make software development more "rigorous". This resulted in a large number of methodologies which sought to build guarantees of quality into the development process. As we shall see, many of these were inspired by the manufacturing principles that were explored in Section 3.1.

3.2.1 The Waterfall Model

The canonical software development process is generally referred to as the Waterfall Model [114]. This envisages the software development process as an entirely sequential, staged process. Development starts off with the elicitation of requirements, moves on to design, etc., until the product is finalised and can be tested and deployed. The process is illustrated in Figure 3.5.

The waterfall model represents the first popular effort to provide a formal description of the software development process. Perhaps for the lack of a better alternative it became the de-facto development standard. It was promoted by the US Department of Defence throughout the 80s, and enshrined in standard DOD-STD-2167A [2], which stated that:

> ... the contractor shall implement a software development cycle that includes the following six phases: Preliminary Design, Detailed Design, Coding and Unit Testing, Integration, and Testing.

Exercise: *Consider the possible flaws of the Waterfall approach. What assumptions does the approach make about the software developer(s)? What situations*

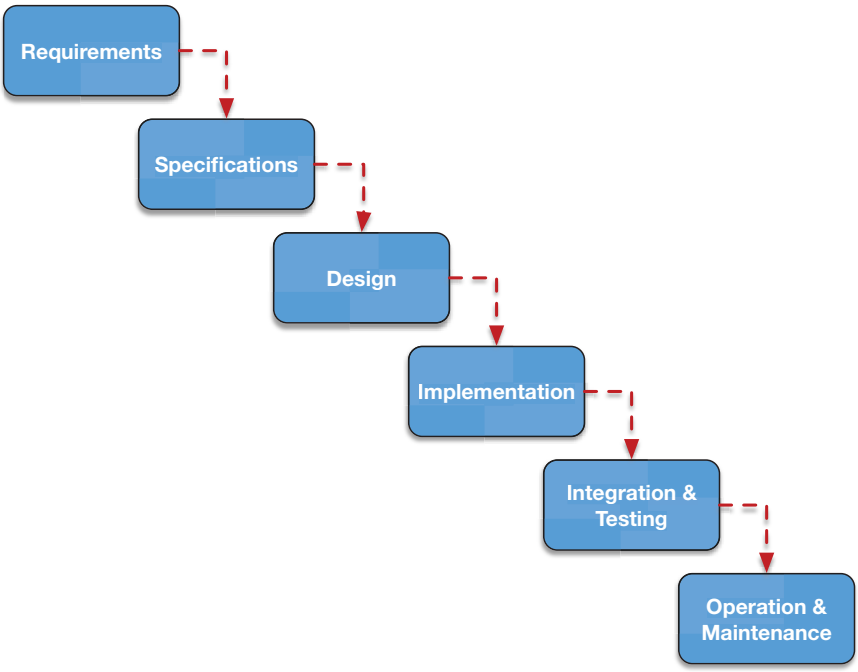

Fig. 3.5 Waterfall model.

does it suit or not suit? Looking at the model, consider the testing box. Consider the following question: What happens in this model if tests detect serious bugs at a requirement level?

Clearly, the Waterfall model leaves little room for error. The requirements must be correct the first time around, because once development has begun, there is no going back (at least not according to the model). As a consequence, implementation of the waterfall model often included a substantial emphasis on documentation, and on the various processes that were involved in making sure that the documents were correct.

It is worth highlighting that, although the Waterfall approach as depicted in Figure 3.5 represents the standard interpretation, it does not represent the approach as originally envisaged by Royce [89]. Royce's proposed process in fact included the suggestion of two iterations, where the first iteration focussed on the development of a prototype. However, this was discarded in what eventually became the predominant, more rigid, sequential interpretation.

The various problems that beset the Waterfall process became apparent throughout the course of the 80s and 90s. The need to nail down all requirements up-front, and for these requirements to be completely accurate, often turned out to be unreal-

istic. This made adoption of the waterfall model highly risky; if the requirements-elicitation stage failed, it could cost huge amounts to rectify. The stakeholders[13] were only involved at the beginning, and then didn't have any formal opportunity to "steer" the project as it developed.

3.2.2 Iterative and Incremental Software Development

As the problems with the Waterfall model became increasingly apparent, its popularity waned. Developers and organisations became increasingly receptive to alternative development processes. One such process came out of the domain of engineering and manufacturing, and was known as Iterative Incremental Design (IID) [89]. The key characteristics are as follows:

- **Iterative:** Instead of constituting a single flow from inception to completion, software development should consist of time-boxed cycles of refinement. Instead of producing everything in one go, the software product should be completed over several iterations.
- **Incremental:** The software product would be constructed in increments. Development would adopt a process of divide-and-conquer, choosing to build individual components or features one-at-a-time, instead of all at once.

Exercise: *How might these two principles address the key weaknesses of the Waterfall model? Are you familiar with any existing software development processes that spring to mind?*

IID was based on the fundamental notion that the development of products in iterations made it possible to regularly check on progress and the quality of the product, and to improve things if they went wrong. These ideas had been employed within NASA for the development of the experimental X-15 plane in the 1950s, which had been considered a major engineering success. Many of the engineers (and practices) from the X-15 project then moved on to more software-focussed departments within NASA and elsewhere, where the same notions of IID were carried over to software projects.

Throughout the seventies, IID became widely adopted within IBM, and was used for several large, safety critical systems for the Department of Defence, including the command and control systems for the US Trident submarines. Reasons for adoption tended to be two-fold. On the one-hand, IID made it easier to control the scheduling. The Trident system had to be delivered by a certain date, or the developers would face a fine of $100,000 per day overdue. This made it crucial to be able to time-box the development process.

[13] "Stakeholders" refers to anybody who has an interest in the final software product. So stakeholders could include purchasers, future users, customers, etc.

On the other hand there was an added flexibility that was notoriously absent from the waterfall model. This was noted by the team who developed NASA's space shuttle software from 1977-1980 (Larman and Basili [89] note their almost apologetic tone for not using the Waterfall-esque plan-driven approach):

Fig. 3.6 The X-15 jet was a pioneering, extremely impressive piece of engineering. Although it was built in the 50s, as of 2015 it still holds the world record for the highest speed ever recorded in a manned, powered aircraft (Mach 6.72). NASA, 1960, reprinted with permission[15].

> Due to the size, complexity, and evolutionary [changing requirements] nature of the program, it was recognized early that the ideal software devel- opment life cycle [the waterfall model] could not be strictly applied...However, an implementation approach (based on small incremental releases) was devised for STS-1 which met the objectives by applying the ideal cycle to small elements of the overall software package on an iterative basis.

Exercise: *What is the relationship, if any, between IID and PDCA.*

Although the Waterfall model was clearly dominant throughout the 80s (and is still used in several respects), IID gradually gained currency outside of the US aerospace and defence domains. This accelerated as the case against the Waterfall model became more apparent. One example of this movement is illustrated in a paper by Parnas and Clements [107], in which the authors enunciate a comprehensive critique of the Waterfall process, including the following (paraphrased by Larman and Basili):

- A system's users seldom know exactly what they want and cannot articulate all they know.

[15] Source: `https://www.mediawiki.org/wiki/File:North_American_X-15.jpg`

- Even if we could state all requirements, there are many details that we can only discover once we are well into implementation.
- Even if we knew all these details, as humans, we can master only so much complexity.
- Even if we could master all this complexity, external forces lead to changes in requirements, some of which may invalidate earlier decisions.

Although we present IID here as a 'process', to be more precise it refers to two basic design and development *principles*. There are many ways in which these can feed into a more concrete, well-defined software development process. In the 80s and 90s, several such processes emerged and became widely adopted within the industry.

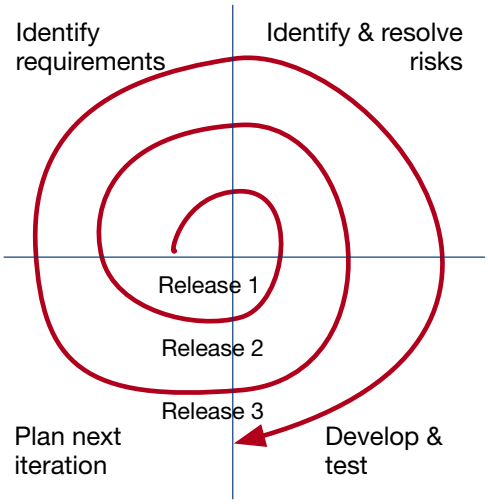

Fig. 3.7 Barry Boehm's Spiral model, which implements IID.

One example was Barry Boehm's Spiral model [21], shown in Figure 3.7. In this model, each iteration amounts to one circuit of the spiral. Each circuit consists of four phases: (1) generating requirements, (2) considering and addressing risks, (3) development and testing, and (4) planning the next iteration. The idea is that, for each circuit, multiple software artefacts are developed concurrently, according to their perceived priorities (we will come on to how this can be determined later on).

Since then several other IID-based approaches have been defined. The Rational Unified Process is another popular example, which was developed within IBM in the late 90s [73]. However, perhaps the biggest influence was the rise of Agile software development processes.

3.3 Agile Software Development

Since the end of the 90s, Agile software development principles have come to dominate software development. This section provides a light overview of some of the core ideas, and reviews these specifically in the context of quality assurance.

Exercise: *In the final section of this chapter we will reflect on agile software development in the context of some of the quality initiatives from the manufacturing industry discussed previously. As you read through this section, try to pre-empt this discussion by making these connections yourself.*

3.3.1 The Principles of Agile Software Development

The term "Agile Software Development" refers to software development techniques that are light-weight in nature, and within which developers can readily respond to changes in requirements. This was largely driven by a reaction to the documentation-heavy, planning-heavy Waterfall model. Such development processes were misaligned in certain sectors, because they struggled to deal with situations where the requirements were uncertain, where budgets and time-constraints were limited, and where neither the organisation nor the client had the time or appetite for lengthy contract negotiations.

To address this, a movement emerged towards the end of the 90s and the beginning of the 2000's. This was embodied in the 2001 *Manifesto for Agile Software Development* [55]. This set out the following twelve principles:

1. Customer satisfaction by early and continuous delivery of valuable software.
2. Welcome changing requirements, even in late development.
3. Working software is delivered frequently (weeks rather than months).
4. Close, daily cooperation between business people and developers.
5. Projects are built around motivated individuals, who should be trusted.
6. Face-to-face conversation is the best form of communication (co-location).
7. Working software is the principal measure of progress.
8. Sustainable development, able to maintain a constant pace.
9. Continuous attention to technical excellence and good design.
10. Simplicity – the art of "maximizing the amount of work not done" – is essential.
11. Best architectures, requirements, and designs emerge from self-organizing teams.
12. Regularly, the team reflects on how to become more effective, and adjusts accordingly.

With respect to previous software development processes, the gist of agile software development was to move the emphasis on the ingenuity of the developers, instead of relying on rigid processes. Nerur *et al.* [101] provide a good comparison

	Traditional	Agile
Fundamental Assumptions	Systems are fully specifiable, predictable, and can be built through meticulous and extensive planning.	High-quality, adaptive software can be developed by small teams using the principles of continuous design improvement and testing based on rapid feedback and change.
Control	Process centric	People centric
Management Style	Command-and-control	Leadership-and-collaboration
Knowledge Management	Explicit	Tacit
Role Assignment	Individual - favors specialization	Self-organizing teams - encourages role interchangeability
Communication	Formal	Informal
Project Cycle	Guided by tasks or activities	Guided by product features
Development Model	Life-cycle model	Evolutionary - delivery model

Table 3.1 Comparison of Traditional and Agile approaches

between "traditional" and agile approaches - which is reproduced in Table 3.1 (the original table by Nerur is larger; this version omits some rows).

These principles formed the basis for a large raft of structured development methodologies (Extreme Programming, Feature Driven Development, SCRUM, etc.).

3.3.2 An Example: SCRUM

So far we have merely discussed Agile software development in terms of its ideals and principles. There are many potential ways in which these can be applied in practice, which is why a large number of alternative agile development methodologies have arisen since then [45]. One of the most popular approaches is currently the SCRUM method [115]. The aim of this section is not to provide an in-depth treatment of SCRUM (you can find that elsewhere), but just to provide a flavour of what a typical agile development methodology looks like. In this description, we also involve some agile practices that are not specific to SCRUM, but nicely illustrate the quality-assurance tools that tend to be used.

3.3.2.1 Teams

The main roles in a typical SCRUM team are as follows:

- **Product owner:** Represents the stakeholders and is the voice of the customer. Responsible for providing feedback, customer-centric user stories (see below), and for prioritising the requirements.
- **SCRUM Master:** Represents the team-leader. Is the interface between the development team and the organisation management. Chairs the meetings, enforces protocols, and removes any impediments that might be hindering the team.
- **Development team:** A set of developers who are responsible for writing source code, tests, UI design, etc. SCRUM makes a point of *not* putting developers into specialised roles, but of having multi-functional teams.

Exercise: *On the subject of development teams - why does it make more sense to stress cross-disciplinary teams, instead of having developers specialise on certain areas such as testing?*

3.3.2.2 Work flow

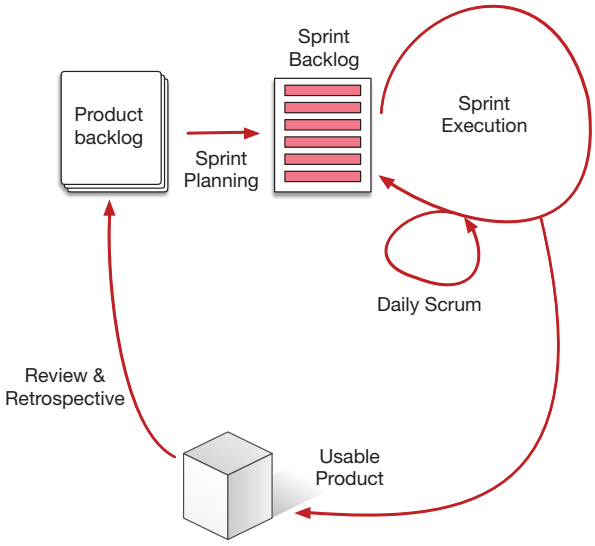

Fig. 3.8 SCRUM work-flow.

The essential SCRUM work-flow is shown in Figure 3.8. It consists of the following key components and procedures:

A product backlog of user stories

A list of desirable software features are compiled with the help of the stakeholders. The list is compiled in terms of *user stories*. A user story is a short-form natural language requirement, commonly used in agile software development. It commonly takes the following form:
"As a <role>, I want <goal / desire> so that <benefit> ".

User stories are commonly associated with a number of *story points* - a numerical estimation of how much effort they will require to implement. The purpose is to establish their *relative* complexity with respect to other stories (not to provide an absolute measure of the amount of effort required). An approach by which to link story points to the amount of time required will be covered in Section 5.2.6. Although the choice of a number of story-points is often entirely down to intuition, it can also be supported by activities such as Planning Poker, which we will discuss in Section 5.2.4.

The *product owner* is responsible for managing this list throughout the development process. The list is prioritised, ensuring that the highest-value, and riskiest items appear towards the top, and the low-value, low-risk items appear towards the bottom. The backlog can continuously change throughout the development of the product.

Exercise: *Consider why risk is a factor in the ordering of the product backlog.*

Exercise: *Consider a typical online banking system, and its users: bank employees and customers. Write down two user-stories for the bank employee, and five user stories for the customer, using the template given in the side-note.*

Sprints

A sprint is a time-boxed period of development, which should always yield a working version of the product. The end-date of a sprint is always firm, and the period tends to range from a week up to a month.

Sprint planning

A sprint-planning session involves agreeing upon a goal for the sprint, and choosing suitable elements for a sprint from the product-backlog. The session is usually limited to last no longer than eight hours. Half of the time involves the entire team, and the other half involves a development-team follow-up to agree upon a development strategy, discuss higher-level architectural questions, and to allocate tasks to individual members.

Daily Scrum

A daily stand-up meeting that lasts approximately 10 minutes. Every member of the development team answers three questions:

1. What have I done in the last day?
2. What am I planning to do today?
3. Are there any impediments?

Stand-up meetings are especially common in agile projects. In an environment where requirements can frequently change, it is necessary to change plans and discuss implementation implications, for which face-to-face discussions are taken to be the best form of communication. The challenge is to enable regular face to face meetings in such a way that the meetings do not end up taking too long and becoming counter-productive. The rationale of a standing meeting is that the discomfort of standing for too long will become uncomfortable for the participants, thus providing a natural incentive to keep the meeting brief.

Though popularised by agile software development, standing meetings have long been popular ways to keep meetings short. For example, the UK Privy Council is a formal body of governmental advisers to the king or queen (formed in 1708) who meet every month. Queen Victoria introduced the convention in the 19th century that the meeting should be held standing up to keep them as short as possible. This custom remains in place to this day[16]

Review and Retrospective

At the end of the sprint, the team should have developed a usable product. This is then reviewed with the product owner and, ideally, stakeholders. Any feedback is fed-in to the product backlog, and a new iteration begins.

Exercise: *Considering the work-flow and key SCRUM activities, can you relate these to the 12 principles set out in the agile manifesto (see Section 3.3.1)?*

3.3.3 Relation to Total Quality Management

Agile techniques and methods are different from traditional software development work-flows, because they shift the emphasis from planning and specification to development and refinement. Developers no longer rely upon a canonical specification and accompanying documentation as a reference for what they do. The specification

[16] `http://privycouncil.independent.gov.uk/work-of-the-privy-council-office/faqs/`.

is a 'living document', embodied in product backlogs, kanban boards with sticky-notes, and communicated by daily scrums.

These principles clearly owe a lot to their manufacturing predecessors. Let us revisit the Total Quality Management principles (see Section 3.1), and discuss them from an agile perspective:

- **Customer focus**: It is customer satisfaction that underpins the first agile principle of early and continuous delivery, and the second principle of embracing regular changes in requirements. Regular contact with the customer is encouraged. In SCRUM, the product-owner exists to ensure that the customer's views come centre-stage throughout the development process.
- **Process - reducing variations and achieving continuous improvement:** One of the key drivers behind agile development is the fact that the processes are lightweight – that they are easy to adopt and follow. This makes them straightforward to adopt, and there are often strong incentives from a quality standpoint. Finally, there are often roles in agile development teams (such as the scrum master) who's key task is to ensure that the processes are adhered to, and to remove any obstacles that prevent this.
- **Human side of quality:** The shift of responsibility from management to developers is a fundamental property of agile development. One of the principles of the agile manifesto states that "projects are built around motivated individuals who should be trusted", and another states that the team should regularly reflect on how to become more effective. Teams should self-organise. SCRUM meetings give individuals the opportunity to raise problems or obstacles, and to assist each other.
- **Measurement and analysis:** Agile development revolves around the use of techniques that convey progress, such as Kanban boards to keep track of development status, as well as burndown charts to show progress.

Exercise: *We have covered the relationship between Agile methods and TQM. Can you do a similar exercise for the Toyota Production System?*

Agile approaches capture much of the best-practice that pre-dates most of the traditional software development methodologies. This point can at times be missed; there is often a perception that many of the principles advocated by agile approaches do not have adequate foundations – for example that they are not adequately supported by sufficient experimental evidence. Although this may be true in the immediate empirical software-engineering sense, it is also often the case that the underlying rationale is very well established indeed, through decades of success in other areas of manufacturing.

3.3.4 Why Not Always Go Agile?

The principles espoused by agile software development are quite intuitive. They are directly or indirectly built upon decades of experience from the manufacturing industry. There have been many empirical studies [45], which have by and large indicated that software is of higher quality and that teams are more productive if the software is developed in an agile environment. Why then, would one not always opt for an agile approach instead of a top-heavy, centralised, documentation and specification-driven "traditional" approach?

There are some downsides. Removing the emphasis on architecture and design has the (unsurprising) consequence that agile software systems can end up with architectures and designs that are less intuitive [113]. There is also the perception that agile methods tend to favour smaller projects, but that larger projects are better off with techniques that emphasise central control and planning [33] (perhaps borne out by the following case study).

3.3.4.1 The UK Government Universal Credit Project

An interesting example of where agile failed can be found in a relatively recent large IT project commissioned by the UK government. Billed as the "world's biggest agile software project", the £2.4bn scheme was intended to showcase how agile software development could apply to large projects.

The project was complex. The idea was to provide an IT infrastructure that could support several significant changes to the British welfare system. These were to simplify the process of applying for state benefits, and to enable greater control of the conditions under which individuals could receive benefits, such as tracking their applications for jobs for work, as well as factoring in certain types of disabilities, whether they had dependants, etc.

Within two years, the project had run into severe difficulties. The government's Major Projects Authority reviewed the project and identified serious concerns, which are set out in a candid report [35]. Below are some quotes that, together, paint a picture of why the project failed:

The introduction of an agile methodology within an environment that lacked experience in agile development.

> In late 2010, the Department decided to use an 'agile' methodology to manage the programme. Agile approaches allow programmes to start technical work before requirements have been finalised, in contrast to traditional 'waterfall' approaches. . . . In 2010, the Department was unfamiliar with the agile methodology and no government programme of this size had used it.

The complexity of the system, and the fact that it interfaced with many systems that were not developed in an agile context.

> The Department recognised that the agile approach would raise risks for an organisation that was unfamiliar with this approach. In particular, the Department:
>
> - was managing a programme which grew to over 1,000 people using an approach that is often used in small collaborative teams;
> - had not defined how it would monitor progress or document decisions;
> - needed to integrate Universal Credit with existing systems, which use a waterfall approach to managing changes; and
> - was working within existing contract, governance and approval structures.
>
> To tackle concerns about programme management, the Department has repeatedly redefined its approach. The Department changed its approach to 'Agile 2.0' in January 2012. Agile 2.0 was an evolution of the former agile approach, designed to try to work better with existing waterfall approaches that the Department uses to make changes to old systems.
>
> The source of many problems has been the absence of a detailed view of how Universal Credit is meant to work.

From the above extracts, it is clear that many problems were at play here, and it would be entirely unfair to pin the roots of the problem on the adoption of agile software development alone. The last quote is especially revealing - it was seemingly difficult to get the stakeholders and ministers to accurately convey how the system was supposed to work.

In this case, the strengths of agile development seem to have turned out to be the roots of the problem. The ability to embrace changes in requirements is a strength. However, no amount of flexibility can make up for situations where the requirements are simply not known at all.

This weakness - the absence of an exact understanding on the part of the stakeholders - would probably have been exposed earlier on in a traditional software development context. Instead of starting development with a view to refining the requirements as they came in, the project would not even have begun in a waterfall context. And in this situation, that scenario would have probably been preferable.

3.4 Software Process Improvement - The Capability Maturity Model

Software development processes cannot simply be applied 'out of the box'. Processes invariably have to be tweaked to suit a particular business context, to suit an organisation, a client, or problem domain (this much is clear from the previous example of trying to apply SCRUM to the Universal Credit project). Throughout the 80s, this task of tailoring an improving development projects was an ad-hoc process, which varied from one organisation to the other.

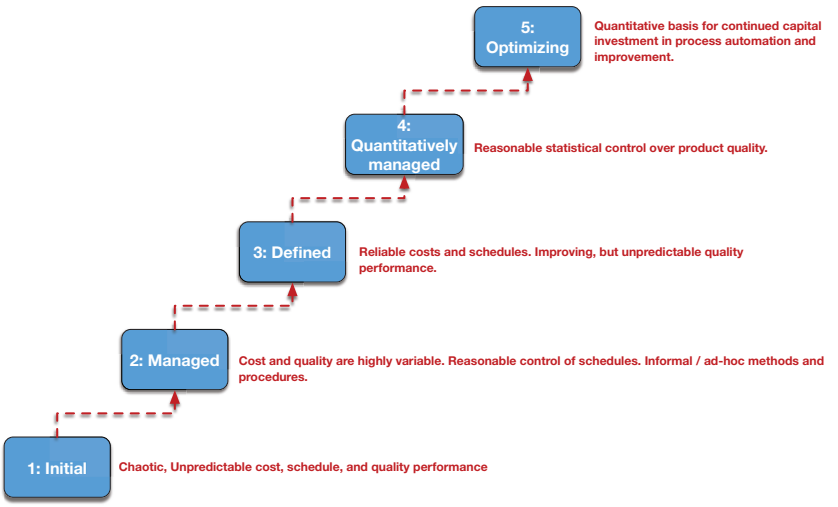

Fig. 3.9 CMMI 1.3 levels of maturity.

Towards the end of the 80s, practitioners and academics tried increasingly to draw upon the process improvement notions that had been so successful within the manufacturing sector. For one striking example in this regard, see this interview with Steve Jobs[17] about the impact that Joseph Juran had on him when he was asked to visit the NeXT corporation, of which Jobs was president at the time. In it, Jobs gives an excellent summary of the basic principles of Juran's work. Interestingly, Jobs also pays tribute to the rise of manufacturing in Japan (as discussed previously), and how he felt that the US had been usurped in this respect.

This led to the Capability Maturity Model (CMM) [69], which was developed at the Carnegie Mellon Software Engineering Institute. It was heavily inspired by the principles of Total Quality Management (see section 3.1.4). The CMM was intended as an instrument that could be used to assess the capabilities of software development organisations.

Over the past 20 years, the model has evolved through various different versions (different text books on software quality can often refer to different versions). The current version is CMMI 1.3 [127] (the I in CMMI stands for "Integration", since the new CMMI model is an integration of various previous versions and variants of CMM).

The current CMMI divides the broad practice of software development into 17 core "Process Areas" (with additional areas depending on the particular domain, e.g. for services, or acquisition). These are shown in Figure 3.10. In order to climb the CMMI ladder, organisations have to introduce increasingly stringent protocols that span all of these 17 areas.

[17] https://www.youtube.com/watch?v=XbkMcvnNq3g

- Causal Analysis and Resolution (CAR)
- Configuration Management (CM)
- Decision Analysis and Resolution (DAR)
- Integrated Work Management (IWM)
- Measurement and Analysis (MA)
- Organizational Process Definition (OPD)
- Organizational Process Focus (OPF)
- Organizational Performance Management (OPM)
- Organizational Process Performance (OPP)
- Organizational Training (OT)
- Project Monitoring and Control (PMC)
- Project Planning (PP)
- Process and Product Quality Assurance (PPQA)
- Quantitative Project Management (QPM)
- Requirements Management (REQM)
- Risk Management (RSKM)
- Supplier Agreement Management (SAM)

Fig. 3.10 17 Core Process Areas for CMMI 1.3.

The "maturity" of an organisation in each process area is measured according to five levels, which are shown in Figure 3.9. Each maturity level is associated with a selection of questions, of which some are designated as"key questions". To attain a given level, an organisation must affirmatively answer 80% of the questions for that level, including 90% of the key questions. The levels subsume each other. In order to obtain a given level, an organisation must also have attained the level below.

Thousands of companies around the world have been assessed according to the CMMI framework. Many organisations use CMMI certification as a basis for setting tender requirements, with one significant example being the US Department of Defence.

Exercise: *Adopting CMMI provides a structured framework within which to assess software development processes. What are the possible downsides?*

Although there are many strong drivers for organisations to certify themselves according to CMMI (and other software process improvement frameworks), many choose not to. On the face of it this counter-productive – in doing so an organisation can rule itself out of competing for potentially lucrative software development contracts. In order to understand why this was the case, Staples *et al.* [126] carried out a survey of 73 organisations that had specifically not certified themselves according to CMMI (this was in 2007, thus applying to versions of CMMI that predate the current version).

On the one hand, their findings were as one might expect; for 35% of organisations the primary reason was that it was too costly, and for 25% it was too time-consuming. CMMI certification certainly does require a lot of time and effort. Changing the business processes within an organisation can be very costly, and might not be considered worth-while (even if these changes are improvements and ought to lead to greater profitability in the long term). The overriding factor,

however, was the fact that CMMI is geared towards larger organisations. 43% of respondents suggested that, as a small organisation, the overheads required by certification were simply too high to absorb.

The CMMI framework is not the only approach within which to assess an organisation's software development capability. There are other frameworks and standards that accomplish similar aims. These include the ISO9001 standard, Motorola's Six Sigma framework, SPICE, and the Malcolm Baldrige Assessment (MBNQA). We cover only CMMI here. For a (perhaps somewhat dated) overview of the other approaches you can refer to Kan's book on Software Quality Engineering [80].

Ultimately however, all of these alternative approaches share the same roots in the various lessons to have emerged from the manufacturing industy. All promote the increased control (reduction of variance) of the development processes. They encourage the use of measurements and metrics to keep track of performance, the disciplined application of procedures, and the ability to adapt processes to address problems.

3.5 Key Points

- **The quality of a product, and the ability to control that quality, is dependent upon the choice of process that is used to produce it.**
- **The waterfall model dominated the formative years of software development.** It embodied the idea of extensive documentation, and development in a step-wise process. It was much maligned for being inflexible, and often led to cost-overruns.
- **Iterative and Incremental (IID) techniques differ from the waterfall model by framing product development as a continuous, iterative process.** They had shown some considerable successes, but were not widely adopted until the late 80s. IID techniques offer more flexibility and facilitate time-boxing. IID formed the basis for agile techniques.
- **Process improvement emerged from the drive to improve manufacturing processes.** These approaches were first recognised as a key driver of economic success in the US, with the emergence of advanced manufacturing (e.g. the Ford Assembly line).
- **Walter Shewhart first systematised the study and improvement of development processes with the PDCA cycle.** The PDCA cycle in its essence underpins current quality improvement techniques. Shewhart's lessons were further refined by Japanese companies during the Japanese "economic miracle". Toyota's TPS contained many of the pre-cursors to today's manufacturing and software development processes.
- **The success of process improvement in Japan inspired the development in the US and Europe of the notion of Total Quality Management (TQM).** TQM inspired several software-specific process improvement frameworks, of which CMMI is a leading example.

- **CMMI is a process improvement framework for software development.** It is widely used, but is often eschewed by smaller organisations because of the overheads involved in certification.
- **Agile software development refers to a family of iterative, lightweight software development processes.** They are 'lightweight' in the sense that they seek to shift responsibility for quality from the process itself to the developers. Many of the examples of 'good practice' espoused by agile techniques were inspired by advances in manufacturing process, such as the practices in the Toyota Production System.
- **SCRUM is one of the most popular agile methods.** It places a particular emphasis on face-to-face meetings, and time-boxed "sprints" within which to iterate versions of the software system.

Chapter 4
Managing Requirements and Code

Software development revolves around the ability to implement a set of requirements (whether explicit or implicit) as source code. Requirements can be diverse in nature, complex, and continuously subject to change. The same goes for the source code; it too is invariably complex, can constitute various libraries and languages, and is also subject to change, potentially by hundreds or even thousands of different, disparate developers. For a project to prosper and retain its quality, strategies and mechanisms need to be in place that can support management of both.

In this chapter we will examine some of the specific challenges that arise when it comes to managing requirements and source code, and some of techniques and tools that can be adopted to address them. For requirements, we will examine the challenges of requirements elicitation, security, traceability, and prioritisation. For coding, we will look at the challenge of handling concurrent contributions from multiple developers at the same time, and of trying to ensure consistency and good coding practice within the code base.

4.1 Managing Requirements

The term 'requirement' covers *any* feature or obligation that a stakeholder would the system to fulfil. Requirements are challenging to manage for a variety of reasons. They are ultimately the desires of a stakeholder, and can be difficult to express (and capture) clearly, and without any ambiguity. When circumstances change (or the stakeholder changes their mind) requirements have to change too, and these changes can have repercussions that can affect the rest of the system – something that can become especially challenging if large parts of the system have already been implemented. For any non-trivial software system there can be a large number of requirements, some of which invariably have a greater priority than others, which gives rise to the problem of how to sort out the most important ones from the rest. Finally, there is the task of providing some sort of oversight to the developer team, to give an intuition of progress, and what needs to be done.

N. Walkinshaw, *Software Quality Assurance*, Undergraduate
Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_4

Exercise: *There is one line of thought that suggests that requirements are the 'be all and end all' for software quality; that the quality of the final product can be definitively assessed in terms of its requirements alone. Who does this remind you of?*

4.1.1 What is a Requirement?

Although the above description of a requirement as "anything that a stakeholder expects the system to accomplish" is intuitive, it is perhaps a bit too simplistic. Software can have a huge range of different 'qualities' – its functional behaviour, the way in which it is constructed, the aesthetics of the user interface, its security, etc. Indeed, there tend to be too many possible considerations to be individually considered and written down by stakeholders.

In practice, requirements tend to be broken down into two high-level categories: *Functional* requirements that capture the functionality and *non-functional* requirements. This dichotomy is helpful because these two classes of requirements are captured and handled in different ways throughout the development life-cycle.

Functional requirements (often captured in the form of use-cases – see below) capture the core desired behavioural features of the software system. Tests of functional requirements tend to exercise and examine the input / output behaviour; an expected (or unexpected) input is executed, and the observed outputs are checked against the requirements.

Non-functional requirements are (to put it glibly) requirements that are not functional. Surprisingly, a more specific, unambiguous definition has proved to be elusive (Glinz's article on this matter nicely captures the various areas of confusion on this matter [58]). For the sake of simplicity (and accepting that this leaves scope for ambiguity), we will refer to a non-functional requirement as simply some characteristic that is *qualitative* in nature – it specifies *how* the behaviour or structure of the system should contribute to the fulfilment of some functional specification(s).

Exercise: *In practice, non-functional requirements cover a huge range of aspects that are relevant to software quality. Look at the software quality models in Chapter 2, and you will see that the majority of concerns in these models are ultimately non-functional. This should also serve to remove any preconception that non-functional requirements are in any way of secondary importance when it comes to software quality.*

4.1.2 Requirements Elicitation

One of the essential challenges of software development is referred to as *requirements elicitation*[1] – the process of taking abstract ideas and capturing in a concrete form that form the basis for subsequent phases of development.

4.1.2.1 The Challenge of Requirements Elicitation

A requirement should be detailed enough to be completely unambiguous to the developer. Every angle of intended software behaviour should be covered, and to a sufficiently detailed level. At the same time of course, requirements elicitation should be sufficiently lightweight so that they can be readily managed and updated throughout the software lifecycle, and to leave enough time for development.

Therein lies the tension. Producing a set of sufficiently detailed, unambiguous requirements can be enormously time consuming and tends to lead to large sets of documents that become hard to manage and maintain. On the other hand, requirements that are less detailed risk raising ambiguities, can form the basis for mistakes and bugs further down the line.

To make the point of how expensive requirements elicitation can be, we can look at some requirements projects at a somewhat extreme end of the spectrum – the use of 'Formal Methods' to construct software systems. Formal Methods is an umbrella term for techniques that use mathematical frameworks to reason about (amongst other things) software requirements, with the goal of providing mathematically justified guarantees about their correctness. In this context, the system in question is modelled (e.g. as a set of abstract functions, or as a state machine), and the resulting model is then analysed to ensure that it obeys (or conversely does not exhibit) particular properties.

One success story of such a technique is the CompCert compiler for the C language. Here, a complete compiler for the C language was modelled and "proven correct". However, this required an estimated two person years (of an expert in theorem proving) of effort, and 400,000 lines of proofs [91] (which embody the requirements) in the Coq theorem proving language. The downsides of such an involved approach are clear – they are time-consuming, and leave little resource for the actual development of the source code[2].

The alternative, however, of resorting to bullet points of natural language, has its own obvious problems. Complex functionalities can be difficult to capture in a way that they can be understood by developers, and are not ambiguous. The problem of elicitation was nicely expressed by Fred Brooks in his famous "no silver bullet" essay [25]:

[1] The term 'requirements elicitation' is used here out of convention. It would (in the author's opinion) be grammatically more appropriate to make 'requirements' singular.

[2] In the case of CompCert this was acceptable, because the code could largely be generated automatically from the model. However, this is not in general going to be practicable for broader families of software systems.

> "The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements . . . No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later."

4.1.2.2 Requirements Elicitation Steps

A host of techniques have been proposed to support requirements elicitation. As with most of the other topics covered in this book, there is no single approach that is best-suited for every situation. There is invariably a trade-off between the time / effort required, and the detail or reliability of the resulting requirements document. However, all approaches tend to follow a similar template. We will cover the high-level steps (as set out by Zowghi and Coulin [140]) in the remainder of this subsection.

Understanding the application domain

It is important to gain an understanding of the 'real-world' situation within which the software system will be deployed. Is the software safety-critical or business-critical? Who will be using the software? Will it be large numbers of users, using it in a variety of capacities, or will it be few users with a relatively fixed set of use-cases? Is the software ultimately going to be embedded onto a device? Or deployed as an app? Is the software subject to standards? Are these liable to change? Are there any pertinent political or cultural factors? How does the software align with the business goals of the organisation?

Identifying sources of requirements

Requirements come in various forms. Obvious sources include the stakeholders (see below) – the eventual users of the system, the people who will be responsible for its deployment and maintenance, etc. However, requirements can also come from other sources too. For example, if the software is intended to fit into an existing work-flow, or needs to interact with other established software systems, then these processes will need to be consulted. If the software is safety-critical, legislation will be need to be consulted (c.f. the UK law that the risk posed to a user should be reduced to the point that it is 'As Low As Reasonably Practicable' (ALARP)).

Analysing stakeholders

Stakeholders represent any people who have an interest in the system or are affected by its development in some way. These tend to include different user-groups (groups

of users who might use the system in different capacities, such as administrators, mobile app users, web-app users, etc.), people who are responsible for procuring the system, deploying it, maintaining it, etc. All groups of users will have a particular perspective or opinion on the system, and these could easily conflict with each other. Accordingly, it is important to identify who the stakeholders are, and to obtain all of their viewpoints so that they can be appropriately reconciled.

Selecting the elicitation technique

As mentioned previously, there are numerous different elicitation techniques. In their Requirements Engineering Roadmap Nuseibeh and Easterbrook [102] provide a nice overview:

- *Traditional techniques:* Questionnaires, surveys, interviews, analysis of existing documentation including organisational charts, process models, and user or other manuals of existing systems.
- *Group elicitation techniques:* Brainstorming, focus groups, and workshops. These are aimed to take advantage of group dynamics to build a richer picture of what is required.
- *Prototyping:* Developing a prototype to gain feedback from stakeholders, especially when the requirements are subject to a lot of uncertainty.
- *Model-driven techniques:* Provide a specific model of the type of information to be gathered, and use this to drive the elicitation process. These include goal-based methods such as KAOS [39] and I* [31].
- *Cognitive techniques:* A series of techniques originally developed for knowledge acquisition in knowledge-based systems. These include *protocol analysis*, where the developer thinks aloud whilst accomplishing a task and *card sorting*, where participants organise cards with domain concepts into groups, to highlight key functional areas within the system.
- *Contextual techniques:* A family of alternatives to traditional and cognitive techniques, in which ethnographic techniques are adopted to observe users. Instead of playing an active role in querying the participant, or in making the participant aware that they are being observed by getting them to speak aloud, these techniques are less obtrusive. The goal is to collect requirements by passively observing the participants, in the hope that they will be less self-conscious, and will indicate a more fine-grained set of requirements in the process.

Eliciting the requirements

Employing the selected technique(s) to elicit requirements from the identified sources and stakeholders. As a part of this process, it is necessary to identify the scope of the system – which requirements are essential, which ones are desirable, and which ones are irrelevant (we will explore some techniques to support this specific question below).

4.1.3 Requirements Documents

So far, we have discussed requirements in a purely conceptual manner. When it comes to actually writing them down, their format can vary depending on the development context - the choice of development procedure and the processes within an organisation. As we have seen, in a Waterfall context it is important to be as detailed and verbose as possible, whereas agile processes tend to keep requirements at a more lightweight level (i.e. single sentences on a post-it note).

4.1.3.1 Use Cases

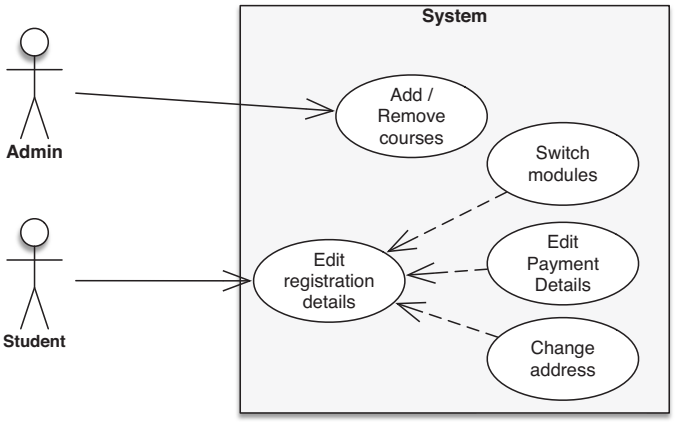

Fig. 4.1 Example of a Use-Case Diagram

Use-cases are light-weight requirements specifications that are designed to capture the various ways in which the system will be used. They were first conceived by Jacobson in 1986 as a means by which to capture functional requirements for Object-Oriented systems [72]. Use cases tend to have two components. Use-Case Diagrams indicate relationships between user-groups and use-cases, and link use-cases to each other. The textual component provides a description.

An example of a use-case diagram for a fictional university student registration system is shown in Figure 4.1. It shows how use case diagrams are quite intuitive; the main groups of users are shown by various actors. Their various interactions with the system are shown by (solid) arrows to the use-cases themselves, which are labelled ovals. These can then be linked to other use-cases by dashed 'extension' arrows, indicating that one use case is an extension of another. So here the specific actions of switching modules and changing address are extensions of the "Edit registration details" use case.

As far as the accompanying textual description is concerned, there is no prescribed template to be adopted. This will depend to an extent on the development context. User Stories (see Section 3.3.2.2, page 40) for example, are highly informal variants of use-cases that are well suited to an agile environment because they are particularly light weight, and can fit onto a postit-note.

1. **Title:** Edit Registration Details
2. **Description:** User is able to edit their details on the database.
   a. Logs in
   b. Selects "edit registration details"
   c. Selects option.
   d. ...
3. **Extensions**
   a. Switch modules
      i. User selects "Switch Modules"
      ii. User selects module they wish to switch.
      iii. ...
   b. ...

Fig. 4.2 Fowler's Proposed Use Case Structure, applied to "Edit registration details" use case in Figure 4.1.

One slightly more elaborate (yet still relatively simple) variant was proposed by Martin Fowler [53]. It consists of three sections: A title, a description (with key steps in a successful usage scenario), and any variants (extensions). An small illustration is shown in Figure 4.2.

4.1.3.2 Software Requirements Specifications

Use-cases and their variants are especially popular because they are intuitive and accessible. They provide a user-centric view of the system, and individual use cases can be readily altered and enhanced as development progresses. As such, they are especially popular for agile software development.

Their strength is, in some respects, also a weakness. Use cases are appropriate for functional requirements, and are good at presenting the user's concerns. They are however not so good at collecting non-functional requirements, and do not necessarily provide a sufficiently in-depth picture of the system requirements as a whole. For this, it is necessary to adopt a more comprehensive, and necessarily more verbose approach.

The Software Requirements Specification (SRS) [34] (IEEE Standard 830-1998) is a standard that sets out a typical document structure, and is shown in Figure 4.3. The structure makes it apparent that, as expected, requirements have to cover a lot of ground – from underlying assumptions and dependencies to detailing every possible facet of a given requirement. Producing a comprehensive requirement document for a non-trivial system is invariably a time-consuming task.

1. Introduction
   - Purpose
   - Scope
   - Definitions, acronyms, abbreviations
   - References
   - Overview
2. Overall description
   - Product perspective
     - System Interfaces
     - User Interfaces
     - Hardware interfaces
     - Software interfaces
     - Communication Interfaces
     - Memory Constraints
     - Operations
     - Site Adaptation Requirements
   - Product functions
   - User characteristics
   - Constraints
   - Assumptions and dependencies
   - Apportioning of requirements
3. Specific requirements
   - External interfaces
   - Functions
   - Performance
   - Logical database
   - Design constraints
   - Software System attributes
     - Reliability
     - Availability
     - Security
     - Maintainability
     - Portability
     - others

Fig. 4.3 Typical SRS structure.

Exercise: *Buried within the SRS specification highlighted in Figure 4.3, you will see a series of "Software System attributes". Which quality model (discussed in Chapter 2) do they indicate influenced the SRS structure?*

The level of detail to which requirements documents (SRS or other types) are populated in practice can vary substantially, and ultimately depends on the development context. In situations where there is a Waterfall-style development, or where the requirements are used as a basis for contractual obligations, there is invariably a drive to make each part as detailed as possible. If the development context is more relaxed or agile, it is crucial to keep the requirements as light-weight as possible.

When it comes down to the specific individual (non-)functional requirements, the SRS standard is relatively non-prescriptive as to how these should be captured. Regardless of the format (and the detail) of a requirement, it is generally important that a requirement captures the following:

- A unique identifier, including a version number.
- A clear description of the essential functionality (or property in the non-functional case) to be implemented.
- A measure of value or importance to the project.
- A success criterion - how should one be able to demonstrate that the requirement has been successfully implemented.

- Any dependencies upon other requirements (facilitated when requirements have unique ID's).

4.1.4 Security Requirements

From a requirements elicitation point of view, thinking about security might appear somewhat premature. After all, we only have a set of *requirements*; we have not yet decided the implementation details – the language, the frameworks we'll use, etc. Surely it makes more sense to wait until we have made some of the main engineering decisions, so that we can start to think about security in more concrete terms?

The problem with this is that "security" is not merely a technical concern. Although some aspects of security are without doubt low-level implementation concerns (e.g. input sanitisation), there are other security concerns that have to be considered from the get-go. Security-critical decisions at a requirement-level can have knock-on implications for other requirements in the system, and can require special treatment when it comes to testing and inspection activities.

As an example, we can return the module-registration use case diagram in Figure 4.1. The university in question might consider the use case "Edit payment details" to require special treatment; the system is storing sensitive bank account details that, if they fell into the wrong hands, could result in students falling victim to fraud. From a system design perspective, they might want to arrange separate, storage arrangements for bank details, or might want to consider different authentication mechanisms for students who wish to access their bank details on the system.

So where does one start when it comes to thinking about potential security requirements? One approach is to consider the potential threats to the system in line with an existing security model. A straightforward model is the C-I-A triad (Confidentiality, Integrity, Availability), embodying three principles that should hold for any kind of secure system:

- **Confidentiality:** The ability to hide data from unauthorised access. Can be enabled by the adoption of encryption methods.
- **Integrity:** Data should be incorruptible, should be possible to ascertain provenance of data.
- **Availability:** System should always be available to some degree of performance. Should not be vulnerable to DDoS attacks, etc.

A more granular approach is to consider the potential security vulnerabilities in terms of the existing system requirements. If there are use-cases there, one possibility is to consider the potential attacker as a user themselves. Instead of merely creating a set of use-cases geared towards conventional users, these should be accompanied by a set of *mis*use-cases [122], which could be employed by the nefarious attacker. The idea, first proposed by Sindre and Opdahl, is a powerful one because it can readily complement any existing requirements, by supporting the developer in adopting a more adversarial mindset.

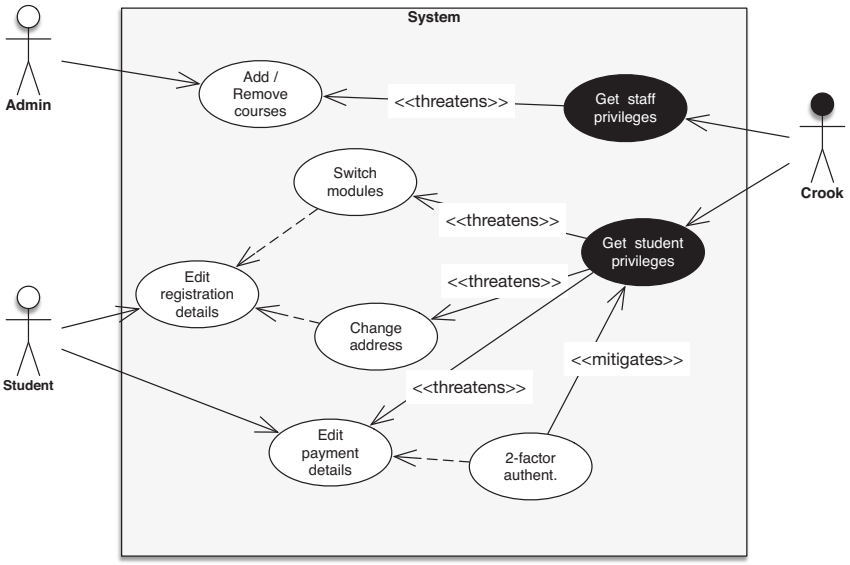

Fig. 4.4 Misuse case diagram example - an extension of the diagram shown in Figure 4.1.

Misuse cases are simply an extension of the use-cases we covered in Section 4.1.3.1. The diagrammatic part is extended with a special kind of actor (with a filled-in head) representing a 'misuser'. The misuse-cases are black ovals, which represent lines of attack that the misuser could conceivably take. These misuse-cases are linked to relevant normal use-cases with "threatens" arrows to indicate a threat. Normal use-cases can also be linked to misuse cases with "mitigates" arrows to indicate that a use-case could mitigate a particular line of attack.

The textual accompaniment for each use-case can be written in a similar structure to Fowler's use-case structure shown in Figure 4.2. Sindre and Opdahl [122] do provide a more detailed textual use-case template if desired.

4.1.5 Tracing Requirements

Requirements are bound to change, both during the development of the system, and once it has been deployed[3]. A customer can have a change of heart, the environment within which a system is to be used might change, testing might establish problems with the original requirements, feature requests will need to be addressed etc. This has to be borne in mind when capturing requirements. If a change in a requirement

[3] For a humorous illustration of how change can hamper and infuriate software developers, watch Tom Scott's video "The Problem with Time and Time Zones": `https://www.youtube.com/watch?v=-5wpm-gesOY`.

does occur, which other aspects of the software system will be affected? Which design artefacts, source code modules, test cases, etc. will need to be updated?

Keeping track of requirements (and their various changes) is a crucial activity when it comes to ensuring the quality of the final system. If the requirements of a stakeholder change, these changes must be reflected in changes to the requirements documentation. If the requirements documents change, these changes must be tracked so that they can be incorporated in to the system design, its source code implementation, and any tests that arise from it. The extent to which it is possible to "trace" changes in requirements down to their various low level components is known as "traceability" [61].

Traceability is not just important for accommodating change. In safety-critical software, it is vital that the implementations of any requirements can be closely scrutinised, and be subjected to testing etc. For this reason, traceability also plays a major role in most safety-critical software standards, notably DO178-B/C [1] and ISO26262 [4].

One standard approach to keeping track of requirements (and their subsequent changes) is via a *traceability matrix*. This is a table in which every row corresponds to a requirement, and every column corresponds to other software artefacts that are directly related to that requirement. An illustrative example of a traceability matrix for an imaginary software system that reads and writes files is given in Table 4.1 (we will return to this imaginary software system later on).

Requirement ID	**Design Artefacts**	**Source code**	**Tests**	**Deps.**
A.0.1: Develop storage format	Scenarios save file and load file	Package org.processor.xml	WriterTest.testSave, ReaderTest.testLoad	D 0.2
B.2.1: Develop file reader	load file scenario	Reader.java	ReaderTest.testLoad	A0.1
C.2.0: Develop file writer	save file scenario	Writer.java	WriterTest.testSave	A0.1
D.0.2: Develop core data structure	Scenarios save file and load file	Package org.processor.model	WriterTest.testSave, ReaderTest.testLoad	

Table 4.1 Example of a traceability matrix

Exercise: *Traceability matrices are clearly useful if they are complete and up to date. What are the potential weaknesses of using them?*

4.1.6 Prioritisation

Once requirements have been drawn up, the development team is commonly faced with the challenge that there are more requirements to be implemented than development time and resource will allow. As a result, it is necessary to somehow prioritise them, to figure out how much effort they will require, and how important they are with respect to the rest of the requirements. A large number of 'requirement prioritisation' techniques have been proposed. In their 2014 survey on the field, Achimugu *et al.* [7] identified 165 empirical studies on requirements prioritisation techniques. Their survey lists 49 different prioritisation techniques for requirements. In this book we focus on two specific, relatively popular techniques: The MoSCoW and Kano methods.

4.1.6.1 MoSCoW

One common approach by which to categorise requirements is to use the MoSCoW method [32]. This consists of categorising each requirements as follows:

- **Must have:** Requirements that are critical (either to the product as a whole or the current iteration). Omitting one of the requirements in this category will mean that the project (or the current iteration) will have failed.
- **Should have:** Requirements that are important but not critical for success of the current iteration or product.
- **Could have:** Requirements that are desirable, but not especially important. These provide value to the stakeholders with relatively little investment.
- **Won't have:** Requirements that are not important, which do not return sufficient value to the stakeholders to justify investment.

This ranking provides a simple basis upon which to proceed with development. The team starts with the "must haves", and proceeds down the scale as far as time and resource permits.

4.1.6.2 The Kano Model

The Kano model [81] takes a slightly different approach to weighing up requirements. Using this model, requirements are divided into three categories:

- **Baseline requirements:** Requirements that, if not fulfilled, will cause the user to be extremely dissatisfied. However, since the user takes the existence of these requirements for granted, their satisfaction will not be increased if they are present. These are the baseline requirements of the product.
- **One-dimensional requirements:** Requirements where the extent to which they are fulfilled has a direct bearing on the satisfaction of the user. If they are not

implemented or poorly implemented, the user will be dissatisfied. However (unlike the baseline requirements), if they *are* fulfilled, the customer will be very satisfied.

- **Attractive requirements:** These are neither explicitly expressed or expected by the customer, but can have a strong impact on the customer's satisfaction if they are implemented.

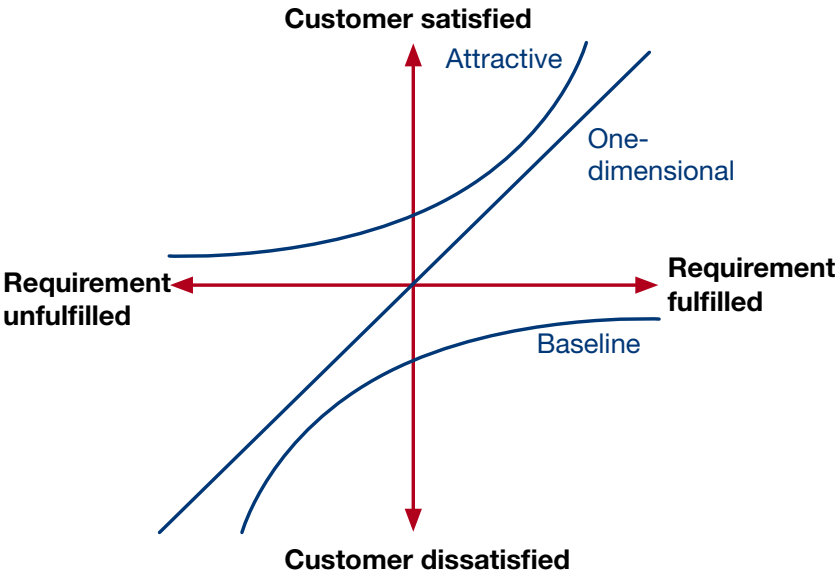

Fig. 4.5 The Kano relation between fulfilment of requirements and customer satisfaction.

From these definitions, Kano posited a non-linear relationship between the extent to which these requirements were implemented, and their effect on customer satisfaction. If one accepts this model of the customer (or user), the task of requirements prioritisation takes on an extra dimension; the relative pay-off from implementing a particular type of requirement depends on the relative progress of the other requirements.

Earlier on in the development cycle, it clearly makes most sense to focus on the baseline requirements. However, once the satisfaction in return for fulfilling these starts to level-off, there may be more to be gained from focussing on the one-dimensional requirements. And later on in the cycle again, there may be a greater return in terms of customer satisfaction to be gained from switching to focussing on the "Attractive" requirements.

Exercise: *Write down a set of 4-5 requirements for a fictional hotel room booking system. First, use MoSCOW to prioritise them. Put the MoSCOW ranking aside, and create a new list using the Kano approach (trying your best to ignore any considerations you took into account during the MoSCOW process). Are they the same? If not, why not? And which one do you consider to be more appropriate?*

4.1.7 Oversight with Kanban boards

There are usually large numbers of requirements. These are often subject to change (especially in an agile environment). So are their priorities. Some will take longer than expected to implement, others less. Some will stall, and will need to be revisited at a later stage in the development process. There is usually a constant "churn" in terms of which requirements need to be serviced, and this requires a continual dialogue with the development team, to establish a universal understanding of who is doing what, and what the various priorities are at any given moment.

This communication of the status of a project is often achieved with the help of Kanban boards, which are especially common in agile projects. The notion of "Kanban" is directly inspired by the Kanban practice used for the Toyota Production System. Kanban is a Japanese word for "signboard" or "billboard" from Japanese. The rationale is that the status of development should be visible to all of the developers, so that any problems or imminent problems can be addressed immediately.

In a software development context, Kanban boards are commonly maintained on a white-board, or electronically. They tend to follow a 'binned' system. The board is divided into a number of columns, representing the status of a job. Figure 4.6 shows a typical system, where the bins correspond to 'to do', 'doing', and 'done'. Sticky-notes are then used to represent individual tasks or user stories representing the ongoing work units, which can be transferred between bins.

For distributed development teams, several online variants have been developed. One popular example is the Trello app[4], which enables users to collaboratively edit and maintain kanban boards.

4.2 Writing Maintainable Source Code and Handling Change

Source code is the ultimate embodiment of any software system. This is what all of the other development processes culminate in; it is ultimately what is delivered to a client or to the general public. Much of the 'quality' of a software system is directly reflected by the quality of the source code itself.

[4] `https://trello.com/`

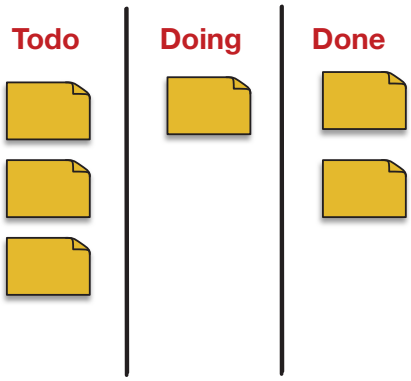

Fig. 4.6 Example of a typical three-bin Kanban.

Code quality can be encouraged in two ways. Firstly, developers can adopt particular design principles and coding practices to support a greater mutual understanding of the code base. Secondly, changes to the code can be controlled and recorded by mechanisms such as version repositories to minimise the likelihood of developers making conflicting changes to the code base, and to make it easier to compare against (or even revert to) older versions if bugs are introduced.

4.2.1 Coding Conventions and Design / Architecture Patterns

It is intuitive that certain styles of source code are easier to read, understand, and maintain than others. Source code that is written as a large, monolithic block of code, with lots of interdependencies and non-descriptive identifier names is going to be harder to understand and edit than source code that is split up into neat functions that are clearly labelled with comments and suitable naming conventions.

Perhaps the most renowned criticism of a particular coding-style was published by Dijkstra in 1968, in his essay "Go To Statement Considered Harmful" [42]. One key principle of Dijkstra's criticism was that the existence of 'go to' statements (which would represent a jump in control from one point in the source code to another) made it difficult for the developer to understand what the source code was doing.

He argued that such jumps should be avoided, and that more emphasis should be placed on the structure of the source code. The code should be constructed in such a way that individual functions can be understood without having to follow complex jumps of control throughout the system. This presaged the advent of a more structural approach to programming, championed by figures such as David Parnas [105], who emphasised the need for modularity (presenting functionality in terms

of well-defined interfaces whilst hiding internal data) and the ability to support code reuse.

Good coding practice nowadays spans every aspect of source code development, from design-level considerations such as information hiding and reuse, right down to syntactic considerations such as where to place the opening / closing brackets of a function, or the treatment of white-space in a code document.

4.2.1.1 Coding Style Guidelines and Conventions

The Google Style Guide[5] offers a comprehensive set of guidelines for open-source projects that originated at Google. The guide contains language-specific instructions for a variety of languages including Java, R, C, C++, and HTML. These include design choices (e.g. every class should be in a file of its own) down to formatting instructions and the use of braces.

There is not enough space here to cover the individual style guidelines. Figure 4.7 picks out some examples from the Google style guidec, to provide an idea of what guidance is on offer (some of which may surprise you). There are many resources online[6] that will explain terms such as the "egyptian brackets" mentioned in the guidelines, along with various other stylistic terminologies (including "Pokemon exception handling").

Guidelines can often be enforced with source code analysis tools such as the Lint tool [76], as will be elaborated when we discuss automated code inspections in Chapter 7. In Google's case, their style guide includes a Lint configuration file that encodes the various guidelines, and that can be used to flag up any violations automatically.

4.2.1.2 Design and Architecture Patterns

Coding conventions are useful for ensuring that the code is easier to understand at a low-level. However, good coding style can only go so far. Software systems also have to be easy to understand and maintain at the level of architecture. A non-trivial software system could have hundreds or thousands of classes (recall the admittedly somewhat extreme example from Chapter 2 that the Google code base contains approximately 9 million files of source code). For large scale systems, developers need to gain a shared understanding of the design of the system; which classes, or groups of classes, are responsible for which aspects of functionality.

It is immensely challenging to build upon the existing design of a system, to implement requirements in such a way that the overall design remains understandable and easy to maintain. Invariably, certain requirements cannot be implemented in isolation, but need to draw upon (and influence) design across the system in var-

[5] `https://github.com/google/styleguide`

[6] C.f. `https://blog.codinghorror.com/new-programming-jargon/`.

- **Source files**
  - Line terminators and the ASCII 'space' characters are the only white-space characters allowed in a file. Tabs are not allowed.
  - A source file consists of: (1) License information (if present), (2) Package statement, (3) import statements, (4) exactly one top-level class. Exactly one blank line separates each of these sections.
  - No wild-card imports.
  - No line-wrapping.
- **Formatting**
  - Braces are always used where optional.
  - Use of braces should follow the "egyptian style" - the opening brace should be in-line with the preceding declaration or predicate, and the closing brace trails the block on its own line.
  - Blocks should be indented with two spaces.
  - Empty blocks may be concise - i.e. {}
  - Only one statement per line.
  - Multiple consecutive blank lines are permitted, but never encouraged.
  - Optional grouping parameters are encouraged.
- **Naming**
  - Class names should be written in UpperCamelCase.
  - Method names should be written in lowerCamelCase.
  - Constant names should be written in CONSTANT_CASE.
  - Non-constants, parameter names, local variable names, should be written in lowerCamelCase.
- **Programming Practices**
  - Caught exceptions should never be ignored.
  - Do not use object finalizers.
- **JavaDocs**
  - At the minimum, Javadoc is present for every public class, and every public or protected member of such a class, unless the method is truly self explanatory (e.g. is a getter method), or overrides a method.

Fig. 4.7 A Selection of rules (paraphrased for the sake of conciseness) from the Google Style Guide for Java.

ious ways. For example, a developer might be implementing a feature that needs to be updated every time a variable in a different part of the system changes its state. This can, depending on the system, be deceptively difficult to achieve. There is the challenge of accessing the variable without needlessly exposing it to irrelevant parts of the system, and the need to monitor it in a way that does not impinge on the performance of the application.

One solution for such design challenges is similar to the solution for dealing with source code style: To develop and adopt standardised solutions to these problems. In the context of design, such standardised solutions are referred to as "design patterns". Design patterns have their roots (as is often the case) outside of software engineering. Christopher Alexander was an architect (in the buildings sense of the word), who was interested in developing generic architectural solutions to encourage good building and town design, which he published in 1977 in a book of "patterns"' [9].

In the mid-90s, four software engineers (Gamma, Helm, Johnson and Vlissides) adopted Alexander's idea of patterns and produced their own book of "Design Patterns" for software systems [57]. Essentially, a design pattern provides a template for a solution to a design problem that can be re-used throughout a system. Since their book was published, a large number patterns have been published, for a variety of programming paradigms, and addressing a large range of problems.

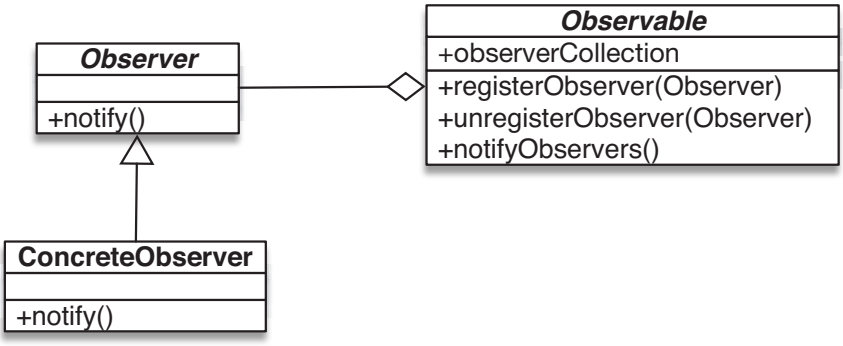

Fig. 4.8 The Observer Pattern template

To provide a more concrete example of a pattern, we can return to the above example; a developer is working on a piece of code that needs to be synchronised with the state of another part of the system every time its state changes. For situations like this, the "Observer pattern" [57] offers a potential solution. The solution is illustrated in Figure 4.8. The key lies in ensuring that any classes[7] that are to be updated implement the same interface (the Observer interface). Then, any class that is being observed by the observers can maintain a collection of Observer classes, and can notify them whenever it updates its state.

At an even higher level of abstraction, patterns can also apply at an architectural level [119]. As with the design patterns mentioned above, many patterns have been proposed; famous examples include Microservices[8] and Service-oriented architectures. Perhaps the most well-known pattern is the Model-View-Controller (MVC), which has its origins in the late 70s, and was first formally documented in the context of the Smalltalk-80 language [87].

The MVC pattern pattern, shown in Figure 4.9, can be used to construct systems where a user may wish to visualise and interact with data (a very large category of systems). The architecture provides a means by which to simplify implementation and maintenance by decomposing the system into three components that can be

[7] Patterns are traditionally discussed in object-oriented terms, but are certainly not exclusive to that paradigm. The Publish-Subscribe pattern is a similar solution for non object-oriented systems, for example.

[8] `http://microservices.io/patterns/microservices.html`

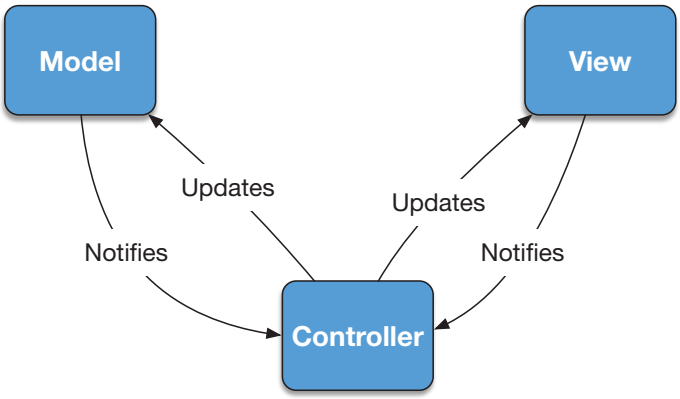

Fig. 4.9 The interactions in the Model View Controller architectural pattern.

broadly kept separate: The code for storing and manipulating the data (the Model), the code for visualising the data (the View), and the code for interacting with the data and view (the Controller).

Exercise: *What do architecture and design patterns, coding conventions, and process improvement have in common?*

4.2.2 Collaborative Development and Version Repositories

In the last two decades the process by which software is developed has changed substantially. The traditional presumption that individuals (or groups of individuals) are working from the same location has gradually been replaced by the widespread use of networked version repositories, where developers can be spread geographically, and across different time zones. This shift is especially apparent in Open Source Software (OSS) development.

This networked distributed software development has led to several innovations, to address the situation where source code is being changed by multiple developers, potentially at the same time in conflicting ways. A host of new tools have emerged to synchronise changes to the source code, and to enable developers to keep track of each others' work. These changes enable a much larger scale of development, potentially involving thousands of software developers. Developers can be geographically spread, operating from different time zones.

This change has however led to procedural challenges. One of the key problems of software development (which turned out to be the downfall of the Waterfall model) is the fact that development activities are highly interdependent; a change to

the code base can have knock-on effects on the wider design and the corresponding requirements. A development process needs to provide a framework within which these distributed, asynchronous changes are productive and do not gradually lead to the deterioration of the software system as a whole. In the remainder of this subsection we provide an overview of distributed version repositories - the predominant technology for distributed software development.

4.2.2.1 Version Repositories

Version repository systems have been a key enabler of distributed software development. A version repository provides a mechanism by which developers can contribute changes to a code base, whilst ensuring that their individual changes do not conflict with each other. They provide a log of what has changed, and provide a means by which to undo changes that might have been counter-productive.

Early version repositories followed what is known as a 'centralised' model. Systems such as CVS (Concurrent Versions System)[9] and SVN (Apache Subversion)[10] operated by hosting a central version of the source code on a server. Developers could then use client software to 'check out' a copy of the source code, to work on changes, and then to 'commit' their changes back to the central server.

Some of the key concepts of version repositories are shown in Figure 4.10. The current authoritative version is called the 'trunk' (this is version 1, 4, 6, 8, and 9). Developers can either make changes to the trunk directly (as happens through versions 6, 8, and 9). If they are embarking on a slightly more elaborate change, they can create a separate branch to be worked on separately whilst other developers carry on working on the trunk. Once they have finished working on their branch, they can combine their changes with the trunk again, an operation that is referred to as a 'merge'.

4.2.2.2 Merge Conflicts

Usually, different versions of source code files are merged together automatically. When a changed file is committed back to the repository, an algorithm can process it, finding changes in the newer version of the file, and replacing them. There are numerous variants of such merging algorithms [95], that manage to execute merges to varying degrees of fidelity.

Problems can arise when two developers have been editing the same file at the same time, because the same portions of the file could have been edited in different (conflicting) ways. If two developers check out the same version, edit the same portions of code in different ways, and then try to check these changes in, version

[9] `http://savannah.nongnu.org/projects/cvs`

[10] `https://subversion.apache.org/`

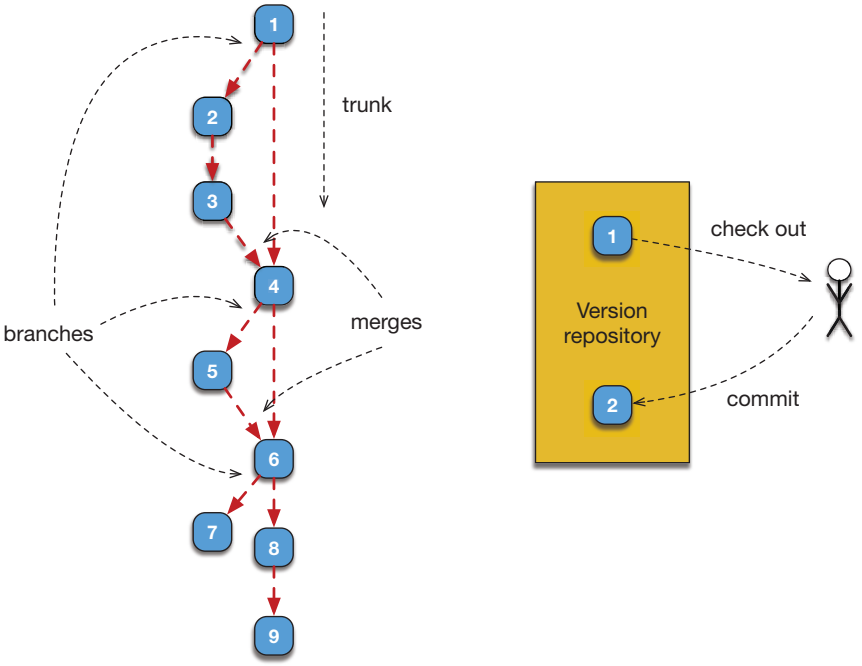

Fig. 4.10 Version Repository Concepts. The boxes represent versions, and the arrows represent the flow of changes to the system.

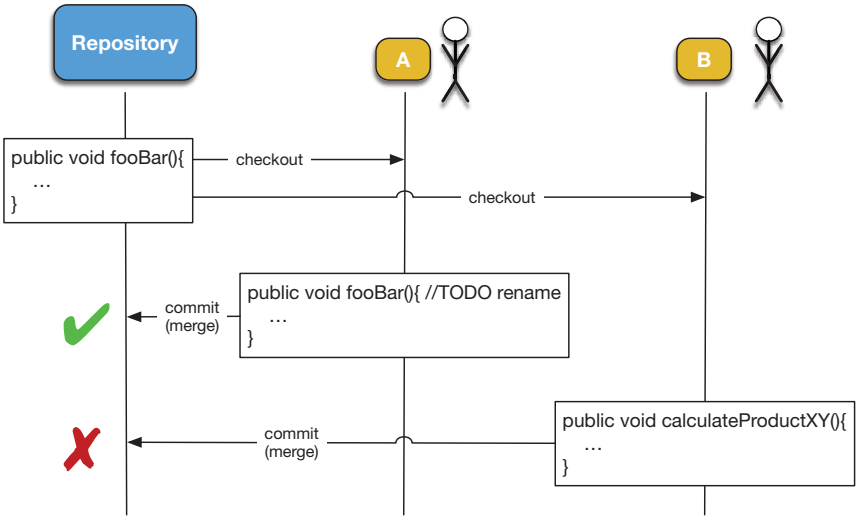

Fig. 4.11 Example of a merge conflict.

repositories have to flag these possibly conflicting changes, and enforce some form of resolution.

An example of this is shown in Figure 4.11 (it loosely follows the sequence diagram notation to give an idea of time, which runs from top to bottom). Developers A and B might check out version 1, which contains a poorly named method. Developer A might insert a comment highlighting the problem. Developer B might choose to rename the method. Developer A commits their change first, which is accepted. When developer B then tries to commit their change, the version repository identifies that the [line of] code that she is seeking to change has been updated since she last checked out the code, and that she could therefore be unwittingly overwriting the change already made by developer A.

At this point, developer B has to resolve the conflict by hand. This is conventionally handled by a process known as "three-way merging" [95]. The developer is shown three versions of the source code: Their own attempted commit, developer A's commit, and the common ancestor version in the repository. The difference between the three files is commonly highlighted with tools (e.g. as provided in the GNU DiffUtils package[11]). Developer B then has to edit the source code to produce a final version that accounts for developer A's changes, and can then commit a final merge.

Conflicts can understandably cause problems in projects, especially where many developers are frequently working on overlapping portions of source code. A conflict indicates that developers might have spent time attempting to address the same problem, which indicates a waste of resource and time. They can also lead to bugs, if merges fail to properly resolve the conflict [64]. To avoid this where possible, it is therefore important that developers follow a development process that enables them to communicate with each other to avoid overlap (e.g. to keep each other up to date with respect the source code files or aspects of functionality they intend to be working on). It is also considered to be good practice to encourage small, frequent commits (and ensure that people check out the latest version regularly), instead of few, larger commits, which intrinsically pose a greater risk of introducing conflicts.

4.2.2.3 Decentralised Version Repositories

In the last decade, centralised systems such as CVS and SVN have gradually been usurped by 'distributed' version repositories such as Git[12] and Mercurial[13]. Instead of operating from a single central repository, distributed version repositories are founded on the idea that every developer maintains a complete copy of the repository on their machine (as opposed to just the current version of the source code). There is one trusted central 'official' repository that developers can use as as a basis for

[11] `https://www.gnu.org/software/diffutils/`

[12] `https://git-scm.com/`

[13] `https://www.mercurial-scm.org/`

uploading and downloading their respective changes, but the core operations (code commits, reverts, merges, etc.) are carried out locally by the developers.

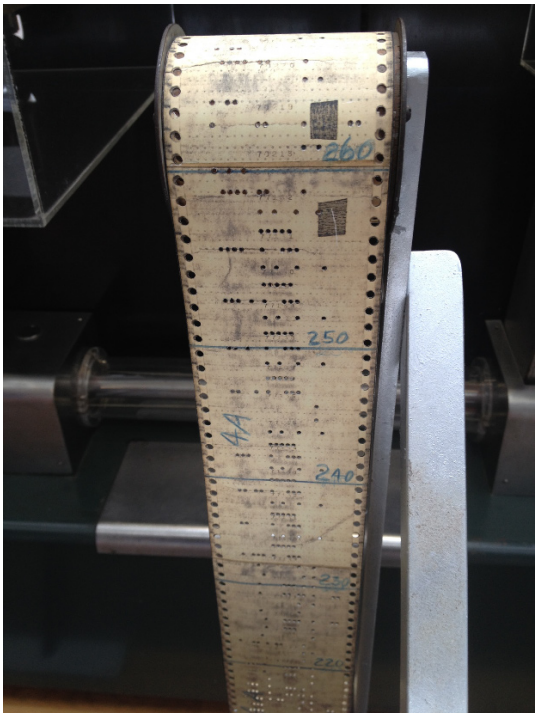

Fig. 4.12 The term ”patch” is often attributed to the physical patches that were applied to correct bugs in punch-hole programs (visible in this tape, being processed by a Harvard Mark I, for example). Arnold Reinhold, 2014, licensed under CC BY-SA 3.0[15].

Instead of operating in terms of ‘versions’ of a software system, decentralised repositories operate in terms of ‘patches’; specific changes to the code base. As an aside – Figure 4.12 shows a possible origin for the term ‘patch’; in paper-tape or punch-card systems, faulty segments of source code were commonly tweaked by patching up holes. Modern patches are the equivalent, but applied to two versions of a text file. A patch essentially contains the lines that need to be added or removed from one file to produce the other. Patches make it easier to keep track of changes to a software system by focussing on only the areas that have changed (instead of relying on a developer to pick them out). We will return to patch-inspection later on in Chapter 7.

[15] Source:
`https://www.mediawiki.org/wiki/File:Harvard_Mark_I_program_tape.agr.jpg`
Licence: `http://creativecommons.org/licenses/by-sa/3.0`

Exercise: *To appreciate how code repositories are used in practice, have a look at some of the open source code repositories on sites such as Github. If you look at a relatively active project, such as Apache Spark (`https://github.com/apache/spark/commits/master`), you will see that commits are made very frequently (by the hour), and that the current code base is the product of tens or even hundreds of thousands of commits.*

4.3 Key Points

- **Software development ultimately revolves around the ability to capture requirements, and to turn these into source code.** Generic software development processes tend not to be prescriptive about how these are accomplished, yet both are critical when it comes to ensuring the quality of the final system.
- **Requirements are intrinsically difficult to capture.** They can be difficult to express by stakeholders, and can be difficult to capture in a non-ambiguous, clear way by developers.
- **Use cases represent the most popular format within which to capture functional use cases.** Use case diagrams serve to provide an overview of who can interact with a system, and in what capacity. Each individual use case is captured individually in a structured textual description.
- **The IEEE standard for capturing requirements is in the form of the Software Requirements Specification (SRS) document.** This document contains (at least according to the specification) a multitude of sections and sub-sections, and goes to illustrate how tricky it can be to capture a requirement in a detailed way. It also illustrates the level of documentation that could be expected with top-heavy techniques such as the Waterfall method.
- **Security-specific requirements are particularly important to capture early-on, but can also be especially difficult to conceptualise.** Security-specific models such as C-I-A are useful prompters to guide developers towards salient security concerns. Misuse-cases can also be devised to complement existing use-cases.
- **Once requirements have been finalised, it is important to be able to trace them to their relevant development artefacts.** This problem is referred to as 'traceability'. It is common to use a 'traceability matrix' to link requirements to their various artefacts.
- **There are often more requirements than can be addressed within the available frame of time and resource, which means that they have to be prioritised.** There are many prioritisation techniques. In this chapter we have introduced the MoSCoW approach (splitting requirements into 'Musts', 'Shoulds', 'Coulds' and 'Won't's), and the Kano model.

- **Frequent changes in requirements and their priorities can be communicated to the development team with the help of a Kanban board.** This consists of a number of 'bins' or columns (e.g. "to be implemented", "currently being implemented", and "completed".). Individual requirements can then be moved between them, and can be listed in order of priority, so that the whole team is able to share the current state of the project.
- **Ensuring that all source code (and its design / architecture) is understandable and maintainable can be encouraged by trying to remove as much scope for variability as possible.** For source code, this is achieved by enforcing style guidelines to enforce a uniform approach to writing code. At a design level and architecture level, design and architectural patterns should be promoted to encourage the use of standard templates to solutions where possible.
- **Collaboratively writing the source code for a large system is a hugely complex endeavour, which relies on an commonly understood modus operandi amongst developers.** Organisations tend to support the adoption of coding conventions and style-guides in an attempt to maintain uniformity within a project.
- **Controlling changes to source code becomes particularly challenging when lots of developers are working on the same code base at the same time.** For this, version repositories are crucial. These come in various flavours, and recently distributed version repositories such as Git and Mercurial have become especially popular.

Chapter 5
Planning Activities and Predicting Costs

In Chapter 3 we saw that development processes can provide a broad work-flow to govern the development of a software product. In Chapter 4 we covered the various conventions and technologies that can help developers to successfully collaborate on source code. Even with those tools, there still needs to be an understanding amongst the developers of who will do what, and when. In a project with a tight time and resource constraints, how much time can be allocated to particular activities? Which activities are the best ones to prioritise? At the end of the day, how much is the whole endeavour going to cost, and how much time will it require?

Such questions are crucial to quality assurance. If the time and effort required for a project is not estimated properly, it can lead to huge cost overruns. A 2011 study of 1,1471 IT projects [52] established that 27% of IT projects are subject to cost-overruns. More worryingly, one in six projects was subject to cost-overruns of more than 200%, and schedule overruns of 70%. In financial terms, such overruns can be disastrous.

Although certain aspects of development processes can help to guard against such overruns (c.f. the use of time-boxed iterations by IID and SCRUM), these can only succeed if the developers are able to plan their work and predict the resources required. Over the years several approaches have emerged to support these activities.

This chapter presents these approaches in two parts. Section 5.1 presents approaches that support planning at a lower-level; determining when what activities should be undertaken. For this we present two general project-management techniques: PERT and Gantt charts. In section 5.2 we present techniques that are specifically geared towards predicting the cost of a project (or a portion thereof). These techniques are not concerned with the low-level sequencing of activities, but are more statistical in nature; examining previous experience and cost data to make a valid prediction of the overall cost of a project.

N. Walkinshaw, *Software Quality Assurance*, Undergraduate
Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_5

5.1 Planning

The cost and time required to develop a project are often intricately interlinked. Producing a high quality software system in a short amount of time can impose huge pressures on developers, taking them away from other projects or requiring the recruitment of new staff (and thus leading to higher costs). Allowing for a longer time-frame can relieve pressures on staff, can provide more scope for testing and other quality assurance activities, leading to higher quality software at a potentially lower cost.

Ultimately, in either case, success comes down to proper planning. A plan needs to set out the (anticipated) *resources* that will be required throughout the course of a project. It needs to set out when certain tasks need to be achieved. It needs to also, set of priorities amongst tasks, and set out which tasks potentially have some scope for leeway to allow for overruns.

5.1.1 Program Evaluation and Review Technique (PERT)

At this level, a software engineering project is akin to any other conventional engineering project. Accordingly, the most popular planning techniques were not specifically developed for software projects, but can be applied in the same manner. In this section we cover one of the most popular Program Evaluation and Review Technique (PERT).

The PERT technique was developed for the U.S. Navy in 1957 to support their development of the Polaris submarine-launched nuclear weapons system [92]. Its main selling point was the ability to explicitly incorporate uncertainty as far as the scheduling of individual tasks is concerned. This makes it especially appealing from a software engineering point of view where, as we shall see in Chapter 5, predicting the duration it takes to develop a software module is fraught with uncertainty.

The PERT technique starts off with a table that captures all of the essential information about the project. The project is split into its essential *activities*. Each activity is associated with the following attributes:

- **Predecessor:** Any other activities that must be completed in order for this activity to start.
- **Time estimates:**
  - **Optimistic (o):** The duration that this activity will take in an ideal setting where there are no hitches and the developers are able to make good progress.
  - **Normal (n):** The duration that this activity will take under 'normal' circumstances.
  - **Pessimistic (p):** The duration that this activity will take if problems are encountered along the way.

- **Expected time:** This is derived from the above time estimates as $\frac{o+4n+p}{6}$. In other words, it takes the average of o, n, and p, but gives n a weighting that is four times greater than the optimistic and pessimistic estimates[1].

The time-units used depend on the broader context of the project. For smaller projects with a more granular plan the unit could be person-hours. For larger projects it might be days or even weeks of developer-time. For our examples we assume days.

		Time Estimates			
Activity	**Predecessor**	**Optimistic**	**Normal**	**Pessimistic**	**Expected Time**
A: Develop storage format	D	2	3	4	3
B: Develop file reader	A	3	5	6	4.83
C: Develop file writer	A	3	4	7	4.33
D: Develop core data structure	-	3	5	10	5.5

Table 5.1 Example table of activities and time estimates, for a small component to load and save data.

An example table of activities is shown in Table 5.1. These activities are concerned with the activities that are involved in the development of a module for reading and writing data to the disk. There are some pertinent dependencies between the activities. The components for writing-to and reading from the file on disk (activities B and C) cannot be achieved until we know what the data format is that we are dealing with (which is developed in activity A). However, this cannot be achieved until we know what the underlying data structure(s) in the program are – which data elements will it be necessary for the program to be able to store and access in a persistent manner (activity D).

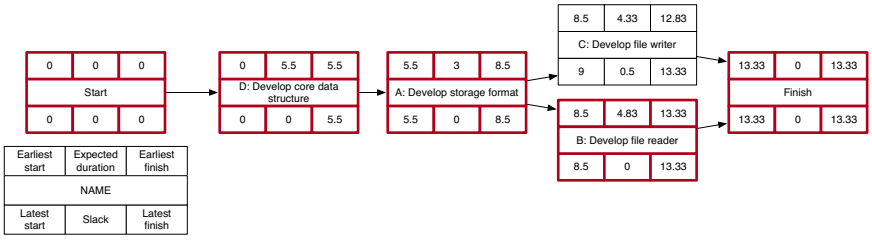

Fig. 5.1 PERT chart for activities in Table 5.1.

Once the activities have been entered into the table, they can be displayed as a PERT chart. This is created in three phases. Firstly, the 'network diagram' is created to highlight the sequential dependencies. This is achieved by creating a 'start' node, one node for each of the activities, and a finish node. The nodes are then connected

[1] It is not clear that this formula is clearly justified; i.e. *why* the weighting of four is chosen for the normal estimate.

according to the 'predecessor' activities set out in the table. The Start node is connected to the activity with no predecessors, and then the dependencies are traced out. Any activities with no successors are connected to the 'finish' node.

The next step is to identify the time-constraints that govern the activities. To facilitate this, we use the following structured labels to annotate the activities.

- Earliest start time and Latest start time
- Expected duration (as calculated in from the time estimates) and slack-time
- Earliest finish time and latest finish time

We populate the PERT chart with these values as follows:

1. In the 'start' node, set all of the values to zero.
2. For each of the nodes that follow on from the start node, set the earliest start time to zero, and set the earliest finish time to the expected duration of that activity.
3. Trace out the subsequent dependent activities, setting their earliest start time to the earliest finish time of the preceding activities, and calculating the earliest finish times by adding the expected duration to the earliest start time.
    - If you have the situation where you have a node with multiple preceding activities (such as the Finish node in our case) the earliest start time is the maximum of the "early finish" times of any of the preceding nodes.
4. In the 'Finish' node, set the latest finish time and latest start times to the earliest finish time.
5. Trace back to the preceding activities, and set each of their latest finish times to the latest start time of the finish node. Calculate their latest start time by subtracting the expected duration from the latest finish time.
    - In situations where several nodes are preceded by a single node, the latest finishing time of that node is set to the minimum latest start time of the succeeding nodes.
6. Calculate the slack for all of the nodes by subtracting the earliest finishing time from the latest finishing time.

Finally, we are able to calculate the Critical Path . This constitutes the path through the PERT chart where there is no slack. Where any delay to an activity will have immediate knock-on effects for the rest of the project and will result in a delay to the overall finishing time. On our PERT chart this is highlighted in red.

The critical path is particularly valuable when it comes to prioritisation. From a management perspective, if two activities are being worked on concurrently, where one is on the critical path and the other has some slack-time, it makes sense to prioritise the critical project. For example, by moving relevant staff from the from the less critical activity to the critical one.

5.1.2 Gantt Charts

PERT charts are a useful means by which to elicit the key dependencies and time constraints that will underpin a project. However, the PERT chart can be difficult to interpret. The chart does not visually convey the explicit *flow* or *duration* of the activities. One visual tool that can be used to better convey the timings of the activities is the Gantt chart. The Gantt chart is named after Henry Gantt, who devised them in the early 1900s (their specific origins are dates are not known [134]). They came to prominence through Gantt's work during the First World War, and became widespread project planning tools in the mid 1920s.

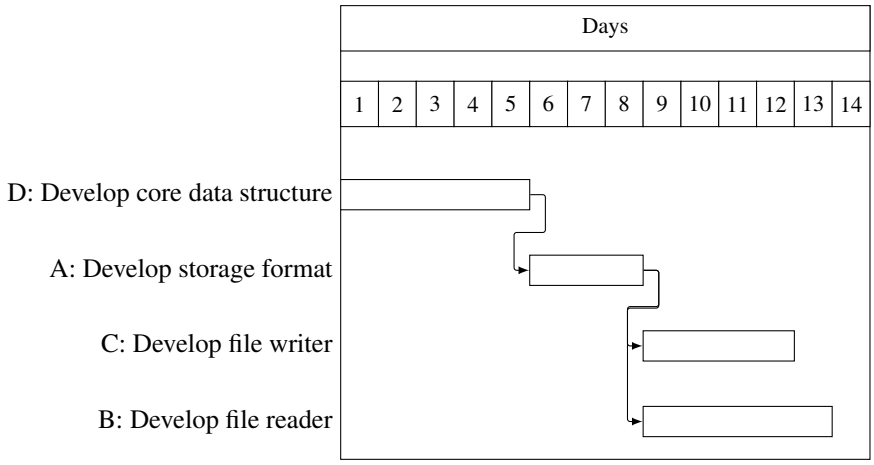

Fig. 5.2 Gantt chart corresponding to the activities in Table 5.1.

Figure 5.2 shows an example of a Gantt chart corresponding to the activities in Table 5.1 (using the expected durations). For the sake of illustration, decimal places in the durations are rounded to the nearest full day. The chart sets out the flow of activity (what happens when), and how long each activity is going to take. Gantt charts often include two different types of arrow: One type to represent the sequential flow between activities, and another to represent the dependency of one activity on another (in our example the flow and dependencies overlap anyway). It is trivially also possible to highlight the critical path on a Gantt chart (by simply highlighting the relevant activities on the path).

Even though they both represent the same information, Gantt and PERT charts are nevertheless complementary in nature. Whereas Gantt charts are purely visual, PERT charts also provide a basis upon which to calculate some of the data that might go into a Gantt chart; the durations, and essential dependencies (which in turn inform the possible flow of activities).

5.2 Predicting Costs

Effort[2] prediction is crucial when it comes to planning how a software system is going to be developed. It is vital to determine the resources that are required, and how this matches up ot the resources that are available. If the software is being paid for, it is necessary to agree upon a price for the development, and this can only be done in a reasonably reliable way if one can draw upon some prior experience.

The nature and fidelity of effort estimation depends on the type and amount of prior data that is available. At the simplest end of the spectrum, one might only have an idea of how large and complex the system ought to be. On the other hand, one might have both such an estimate, along side a wealth of cost data from previous projects.

5.2.1 Base Models

The simplest approach is to predict the cost in two steps. First of all, one derives an estimate of software complexity or size (S), by deriving a measure such as Albrecht's Function Points or an overall estimate of LOC from the requirements. The act of estimation is then reduced to estimating the cost of developing a single unit a (e.g. cost per LOC or cost per function point). So the cost E is computed as:

$$E = a * s \tag{5.1}$$

For example, if you estimate that your project will amount to 1,230 LOC, and you estimate the cost per LOC to be £25, then the total cost $E = 25 * 1250 = £30,750$.

Often, the cost of a model does not merely depend on "size". There can be a base-cost to a project before a single line of code has been written - with the preliminary requirements elicitation, the initial architecture, design, recruitment, etc. To account for this, we can add an additional parameter to the model, c, so that the estimate becomes:

$$E = a * s + c \tag{5.2}$$

In this case, if one were to plot the cost estimate as a line, c would be the intercept – the point at which the line crosses the y axis.

In reality, the relationship between size and effort is not linear. After all, every line of code (let alone entire function) that is added to a project presents the developer with an additional responsibility to maintain and test as well. To address this, an additional 'scale' coefficient b is added, resulting in:

$$E = a * s^b + c$$

[2] In this context, the terms "cost" and "effort" are used synonymously.

In this case $b > 1$ indicates a non-linear increase in cost per unit (i.e. per LOC). Conversely, $b < 1$ indicates a decrease, though this is highly unlikely to be the case in a typical project.

5.2.2 Parameter Fitting by Linear Regression

But where do the values for parameters such as a, b, and c come from? Unless you have some prior experience, it has to come out of pure intuition, and this is unlikely to be particularly reliable. However, if you do have experience – if you have recorded previous project costs and sizes, it becomes possible to make a more educated guess. This is the setting that broadly sets the scene for most of the techniques in software cost estimation.

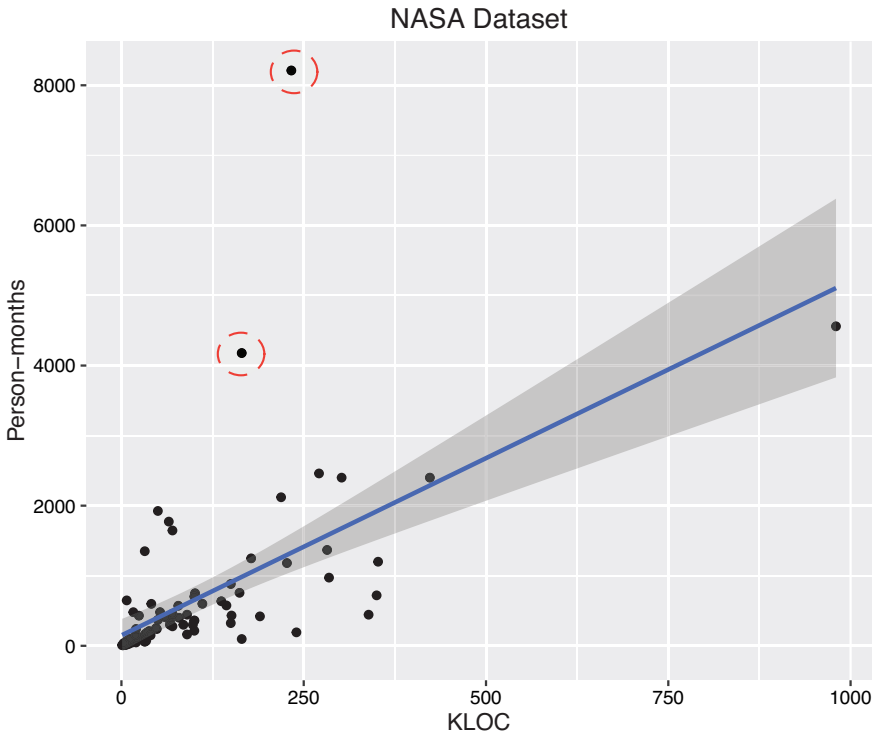

Fig. 5.3 Linear Regression on cost-effort data from the NASA'93 dataset. $a = 5.059$ and $c = 148.8$

The approach is best illustrated visually, and by example. To illustrate this we use a dataset collated by NASA in 1993, which maps the amount of effort in

person-months to KLOC (amongst other things) for 101 data-points, spanning eight projects[3]. The scatter-plot for this data is shown in Figure 5.3. Essentially cost estimation amounts to identifying a "model" that fits this data, and which can thus be used to predict the effort for projects of other sizes.

The task now is to identify a "model" that can explain this data, and that we can therefore use to predict the cost of our project. In the simplest case, we can assume that the model in question is a straightforward line (in this case we assume that $b = 1$).

The approach of finding the necessary values for a and c to best fit a set of data is known as *Linear Regression*. We will not cover the specific mechanics of Linear Regression in this book - there are plenty of references and tools that will do this for you. In our case, if we apply Linear Regression to the data in Figure 5.3, we end up the line that is plotted in the figure – which has the coefficients $a = 5.059$ and $c = 148.8$.

Exercise: *The model fits a lot of the data points quite well. However, it fares less well with some of the "outliers" highlighted in the plot. Why might the data points not fit a straight line? What would the implications have been, had NASA adopted this linear model?*

As shown above, the model of a simple straight line for modelling cost or effort can be somewhat coarse. For example, one simplifying assumption made by the straight-line cost model is that $b = 1$ – that the cost is directly relative to a fixed cost-per Line of Code (or whatever the unit of software size). In reality, it is likely that $b > 1$; that for every additional line of code there is an additional cost to bear, in terms of efforts involved in activities such as maintenance, testing, documentation, etc.

5.2.3 COCOMO

The linear regression can fit a straight line to existing data, and provides an rough basis upon which to begin approximating the costs of a project. However, a straight line does not allow us to account for other 'non-linear' factors, such as the scale factor b. Also, fitting the parameters by approaches such as Linear Regression can be tricky because one might not have access to the sort of historical data that is required to yield a useful prediction.

To address this, Boehm developed the Constructive Cost Model (COCOMO) effort prediction framework in 1981 [23]. Instead of requiring historical data, the model provides certain parameters that, collectively, should characterise the development task at hand. It comes in three levels:

[3] You can download this data yourself from the PROMISE software repository [6]

1. **Basic:** The project is in its very early stages and has been subject only to some very basic requirements capture.
2. **Intermediate:** The requirements have been collected to a reasonable degree of detail.
3. **Detailed:** The requirements have been collected and the system has been designed.

For each level, COCOMO provides additional parameters that can be populated by the user to provide a (hopefully) increasingly accurate prediction of the effort required. Here we present the Basic and Intermediate stages[4].

Over the years, COCOMO (and its successor COCOMO II) have become widely used within industry, especially in the Aerospace domain by organisations such as NASA.

5.2.3.1 Basic Model

The basic model for COCOMO provides three models, to predict three aspects of the effort that a project is going to require. The effort E is computed by the familiar equation we have covered previously:

$$E = a * s^b \tag{5.3}$$

This effort value contributes to the calculation of the development time, which is computed as follows:

$$D = c * E^d \tag{5.4}$$

Both of these values can then be used to calculate the number of people required:

$$P = E/D \tag{5.5}$$

To use these equations, the user has to provide s (the estimated size of the final system in KLOC), and has to select the values of four coefficients (a, b, c, and d). The COCOMO model provides suggested values for these parameters. These values were derived from data collected from 63 software development projects in the 70s[5]. These projects were largely from a single organisation, along with a selection of others from university courses and other organisations.

The selection of a suitable set of variables first of all depends on the "type" of project. This is decided as one of the following:

1. **Organic:** Small teams of experienced developers with flexible requirements.

[4] The detailed stage is a very Waterfall-model specific adaptation that refines the prediction in a way that is sensitive to the various development stages.

[5] The original COCOMO dataset can be downloaded here: `http://openscience.us/repo/effort/cocomo/coc81.html`

2. **Semi-detached:** Medium teams, mixed experience, working towards a mixture of tight and flexible requirements.
3. **Embedded:** The software is developed towards tight constraints.

Based on this categorisation, the values for a, b, c and d are shown in Table 5.2.

Development context	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.2	2.5	0.32

Table 5.2 COCOMO coefficients for the "Basic" model.

5.2.3.2 Intermediate COCOMO

The intermediate version of COCOMO incorporates some additional factors that can have a bearing on the time and effort required within a project, but which were not considered in the basic version. These "cost drivers" are shown in Figure 5.4.

- Product attributes
  - Required software reliability
  - Size of application database
  - Complexity of the product
- Hardware attributes
  - Run-time performance constraints
  - Memory constraints
  - Volatility of the virtual machine environment
  - Required turnabout time
- Personnel attributes
  - Analyst capability
  - Software engineering capability
  - Applications experience
  - Virtual machine experience
  - Programming language experience
- Project attributes
  - Use of software tools
  - Application of software engineering methods
  - Required development schedule

Fig. 5.4 Cost drivers for Intermediate COCOMO

For each of these cost drivers, the developer is required to rate them on an ordinal scale from 1 to 6, where 1 is "very low" and 6 is "extra high". Each combination of factor and rating is given a weighting, as shown in Table 5.3. The "Effort Adjustment Factor" (EAF) is calculated as the product of the weightings attributed to all of the cost drivers.

The Intermediate COCOMO uses this EAF to produce a more refined effort estimation:

Category	Cost drivers	Very low	Low	Nominal	High	Very High	Extra High
Product	Req. Reliability	0.75	0.88	1.0	1.15	1.4	
	Database size		0.94	1.0	0.08	1.16	
	Complexity	0.70	0.85	1.0	1.15	1.3	1.65
Hardware	RT Perf.			1.0	1.11	1.3	1.66
	Memory			1.0	1.06	1.21	1.56
	Volatility of VM		0.87	1.0	1.15	1.3	
	Turnabout time		0.87	1.0	1.07	1.15	
Personnel Attributes	Analyst	1.46	1.19	1.0	0.86	0.71	
	Apps.	1.29	1.13	1.0	0.91	0.82	
	Soft. Eng.	1.42	1.17	1.0	0.86	0.70	
	Virt. Machine	1.21	1.1	1.0	0.9		
	Prog. Lang.	1.14	1.07	1.0	0.95		
Project	SE Methods	1.24	1.1	1.0	0.91	0.82	
	Tools	1.24	1.1	1.0	0.91	1.83	
	Schedule	1.23	1.08	1.0	1.04	1.1	

Table 5.3 Weightings for Intermediate COCOMO "cost drivers".

$$E = a * s^b * EAF \tag{5.6}$$

The other calculations (for development time and personnel) are calculated as in the basic model. The coefficients for a, b, c and d are chosen from the same set of values (as shown in Table 5.2).

Exercise: *Compare and contrast Intermediate COCOMO with Albrecht's Function Point model. Can you identify any common weaknesses between them?*

5.2.3.3 COCOMO II

The application of COCOMO is problematic. Looking at the entire COCOMO model, it provides a set of fixed parameters for a, b, c, d, as well as the 15 cost drivers. It has derived values for these 19 parameters from a relatively small selection of projects (only 63). Furthermore, the projects in question that were used to derive these values were of a relatively specific nature; they were all from the 70s; most likely written in languages such as C and Pascal, with development methodologies that have since become largely outdated, such as the Waterfall model. Applying these values in a modern context, to an agile project that is being developed with modern tooling, by developers with different skill sets, adopting "new" language paradigms such as Object-Oriented development, risks producing results that are misleading.

To address this weakness, Boehm collated a new, larger set of data (amounting to 161 projects). He used this to fit variables for a new COCOMO II model [20], which is designed to be applied to software systems that are developed in a more modern

context. Their intention is to take into account factors such as reuse, automated programming, the integration of COTS components, etc., which were not explicitly accounted for by the original COCOMO model.

The equation underpinning the COCOMO II model is as follows:

$$E = a * \prod_i EM_i * s^{b+0.01*\Sigma_j SF_j} \tag{5.7}$$

Here, a, s, and b are the same as the variables that were used in the original COCOMO.

The *SF* represents a set of 'scale-factor' values that are added up to produce an overall scale factor. These are obtained by rating a set of factors on a scale from 1-6. These ratings are chosen according to Table 5.4.

Definition *(abbreviation)*	**Low (1,2)**	**Medium (3,4)**	**High (5,6)**
Development flexibility *(flex)*	Process rigorously defined	Some guidelines, can be relaxed	Only general goals
Process Maturity *(pmat)*	CMM Level 1	CMM Level 3	CMM Level 5
Precedentedness *(prec)*	Never build this kind of software before	Quite new	Thoroughly familiar
Architecture or risk resolution *(resl)*	Few interfaces defined or few risks eliminated	Most interfaces defined, most risks eliminated	All interfaces defined and all risks eliminated
Team cohesion *(team)*	Very difficult interactions	Basically cooperative	Seamless interactions

Table 5.4 Scale factors for COCOMO II

EM is a set of "effort multipliers". These are chosen on a similar basis to the scale factors, and are shown in Table 5.5. The actual coefficient values that are associated with each of these ratings for both *SF* and *EM* are shown in Table 5.6.

Once the *EM* and *SF* elements have been provided, the effort E can be calculated, which can feed into the calculation of developer time and number of people required as in equations 5.4 and 5.5. Boehm recognised that the given coefficients may not be appropriate to an arbitrary setting. It is unlikely that the data, even in its expanded form of 161 projects, would not adequately apply to every applied setting. Accordingly, it is customary to tune COCOMO. This can be achieved if an organisation has just a few examples, and is commonly accomplished by varying variables a and b whilst holding the other values constant.

Definition	Low (1,2)	Medium (3,4)	High (5,6)
Analyst capacity *(acap)*	Worst 35%	35% – 90%	best 10%
Applications experience *(aexp)*	2 months	1 year	6 years
Product complexity *(cplx)*	E.g. Simple read / write statements	E.g. Use of simple interface widgets	E.g. Performance critical embedded system
Database size (DB bytes / SLOC) *(data)*	10	100	1000
Documentation *(docu)*	Many life-cycle phases not documented		Extensive reporting for each life-cycle phase
Language and tool experience *(ltex)*	2 months	1 year	6 years
Programmer Capability *(pcap)*	Wost 15%	55%	best 10%
Personnel Continuity (% turnover per year.) *(pcon)*	48%	12%	3%
Platform experience *(plex)*	2 months	1 year	6 years
Platform volatility *(pvol)*	Major change every 12 months, minor change every month.	Major change every 6 months, minor change every 2 weeks.	Major change every 2 weeks, minor change 2 days.
Required reliability *(rely)*	Errors a slight inconvenience	Errors are easily recoverable	Errors can risk human life
Required reuse *(ruse)*	none	Multiple program	Multiple product lines
Dictated program schedule *(sced)*	Deadlines moved to 75% of original estimate	No change	Deadlines moved back to 160% of original estimate
Multi-site development *(site)*	Some - contact by phone and mail	some email	Interactive multimedia
Required % of available RAM *(stor)*	N/A	50%	95%
Required % of CPU *(time)*	N/A	50%	95%
Use of software tools *(tool)*	edit, encoding, debug		Integrated with life-cycle

Table 5.5 Effort Multipliers for COCOMO II

Category		Vey Low	Low	Norm	High	Very High	Extra High
Scale Factors	Prec	6.20	4.96	3.72	2.48	1.24	0.00
	Flex	5.07	4.05	3.04	2.03	1.01	0.00
	Resl	7.07	5.65	4.24	2.83	1.41	0.00
	Team	5.48	4.38	3.29	2.19	1.10	0.00
	Pmat	7.80	6.24	4.68	3.12	1.56	0.00
Effort Multipliers	Rely	0.82	0.92	1.00	1.10	1.26	–
	Data	–	0.90	1.00	1.14	1.28	–
	Cplx	0.73	0.87	1.00	1.17	1.34	1.74
	Ruse	–	0.95	1.00	1.07	1.15	1.24
	Docu	0.81	0.91	1.00	1.11	1.23	–
	Time	–	–	1.00	1.11	1.29	1.63
	Stor	–	–	1.00	1.05	1.17	1.46
	Pvol	–	0.87	1.00	1.15	1.30	–
	Pcon	1.29	1.12	1.00	0.90	0.81	–
	Acap	1.42	1.19	1.00	0.85	0.71	–
	Pcap	1.34	1.15	1.00	0.88	0.76	–
	Apex	1.22	1.10	1.00	0.88	0.81	–
	Plex	1.19	1.09	1.00	0.91	0.85	–
	Ltex	1.20	1.09	1.00	0.91	0.84	–
	Tool	1.17	1.09	1.00	0.90	0.78	–
	Site	1.22	1.09	1.00	0.93	0.86	0.80
	Sced	1.43	1.14	1.00	1.00	1.00	–

Table 5.6 COCOMO II Coefficients

5.2.4 Planning Poker

So far, all of the cost-prediction approaches considered have been 'model-based'. We are assuming that there is a hidden relationship at play, between certain properties of the software system such as its size and complexity, and the ultimate cost. However, not all cost-prediction approaches are based on this framework.

Planning poker is one cost-prediction approach that is *not* model-based. As the name suggests, Planning poker is based on a 'game' that several developers play amongst themselves. The approach is especially popular in agile-development contexts, and is an informal variant of the Wideband-Delphi approach proposed by Boehm in 1981 [23].

Planning poker provides a structured protocol that aims to elicit a consensus from the developers as to how much effort a piece of work is going to take. Each developer is given a set of numbered cards, where the numbers fit some non-linear scale[6], along with a wild-card "?". For a given user story, each developer picks a card that corresponds to their estimation of how much effort a story will take. Units here are commonly referred to as 'story points'. These are intended to measure the complexity or size of the task (but are not tied to a unit of time).

To play the game, each player places the card representing their estimate face-down on the table (to prevent biasing other players). Once the cards have been turned

[6] The scale often follows the Fibonacci sequence: 1, 3, 5, 8, 13, 20, ...

over, the players with the highest and lowest values are given the option of arguing why they believe they are right. Once they have made their cases, the entire process is repeated; players place their cards afresh until a consensus is reached.

There have been few studies into the accuracy of planning-poker, but those that have been carried out have been supportive. A study in 2008 by Moløkken *et al.* [98] on an industrial software development project indicates that planning poker leads to group-estimates that are more conservative than simply taking the averages of individual estimates, and that these consensus estimates were more accurate than individual estimates.

5.2.5 Uncertainty and Predictive Accuracy

One crucial factor in predicting software cost is the amount of reliable information that can be used about the software project in question. This amount of information increases as software development progresses. At the earliest stage, before requirements have been properly elicited, any estimates are bound to be less certain than they are in later stages.

This is one of the key sources of error in the use of, for example, the linear regression models above. In order to produce a cost estimate, it is still necessary to produce an estimate of the size of the final system. Without a proper grasp on the requirements, such a guess (and any cost estimate derived from it) has to be treated with extreme caution.

This problem was characterised by Barry Boehm as the "Cone of Uncertainty", shown in Figure 5.5. He suggests that predictions that are made in the very initial stages can be a quarter of the actual cost if overly optimistic, or four times the actual cost if overly pessimistic.

We have seen that cost estimation is one of the most important factors in quality assurance. Without the ability to anticipate costs properly, a project is doomed. Developers are not given the time and resource necessary to fulfil their obligations, the project becomes subject to overbearing time and cost pressures that, ultimately, undermine quality, and can even lead to projects being abandoned.

The "cone of uncertainty" goes some way towards explaining why effort estimation is difficult. Early on in a project, little is known, and the estimate is necessarily approximate. This can only become more reliable as the blanks are filled in once the project has had a chance to mature.

It also gives rise to a more fundamental question: How accurate is COCOMO? How does it compare to more recent proposed approaches, or more basic approaches such as the simple Lines of Code function in Equation 5.1?

These questions, and others, were posed by Menzies *et al.* (including COCOMO's originator Barry Boehm) in a recent paper [96]. To answer the above questions, they carried out a comprehensive study, comparing the performance of COCOMO and a raft of more recent techniques on a range of prediction tasks. Interestingly, their analysis showed that, despite its age and apparently biased coefficient

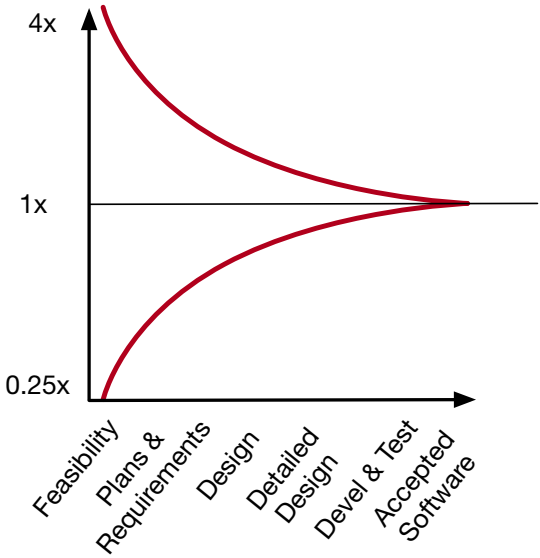

Fig. 5.5 Boehm's Cone of Uncertainty.

values, no technique outperformed COCOMO II. It was far more accurate than measures that are based upon Lines of Code alone. The key to accuracy, they found, is the choice of suitable data for calibrating the model, as opposed to the model itself.

5.2.6 Keeping Track of Progress

Once the cost of an activity has been estimated, it is important to monitor whether its actual cost reflects the prediction or not. If the prediction was wrong, it is better to realise this early on so that any miscalculations can be factored into subsequent planning iterations. For this we can refer to two useful notions that are particularly prevalent in agile software development: Team Velocity and Burndown Charts.

Team Velocity is the estimated speed at which a development team works. In an agile context it can be described as *"Number of story-points completed per sprint"*[7]. The velocity of a sprint is calculated by adding up the story point estimates for stories that were successfully completed in that sprint.

[7] We use the 'story point' unit here, but this could be replaced with whatever metric is being used to predict time or effort.

Exercise: *Having played a few rounds of Planning Poker, your team has established that the total number of story points for all of the stories amounts to 203 story points. Having completed a sprint, your team has implemented 5 stories, which amount to 25 story points. What is their velocity, and (assuming this velocity continues) how long will they take to complete the software?*

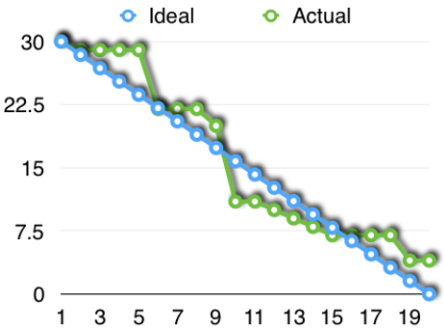

Fig. 5.6 Example burndown chart. Here, a sprint takes 20 days, and the total estimate of effort for the sprint amounts to 30 story points.

The velocity and the amount of work left to do can be visualised with Burndown charts. An example is shown in Figure 5.6. Burndown charts are quite flexible, and there is no prescriptive way in which they are to be used. They can be used to visualise progress throughout the lifetime of a project, or just for the day-by-day progress of a sprint.

The starting point is the total amount of effort that has been estimated. This can be achieved by totting up the total number of story-points estimated (or whatever other unit of effort is used). The length of the x-axis is determined by the amount of time planned for the given iteration or sprint (we refer to this number as x here), and is usually divided into days. The "ideal" line is simply a straight line from the total estimated value at day 1 to 0 effort at day x. The "actual" accomplishment is measured by the number of units that are deemed to have been completed after each day. The idea is that the burn-down chart is visible throughout the project, and is updated at regular intervals (e.g. every day).

5.3 Key Points

- **Coming in on time and on budget is a key consideration when it comes to the quality of a software system.** A huge portion of IT projects fail in this respect; a 2011 survey indicated that 27% of projects are subject to cost-overruns, and that one in six suffer cost-overruns of over 200%.
- **PERT charts were established in the late 50 so support project management activities.** They combine a simple form of cost-estimation with dependence analysis to compute dependency diagrams, where the estimated start and end-point of each activity can be predicted.
- **Gantt charts show the sequential activity within a project.** In doing so they complement PERT charts, which only show dependencies, but do not necessarily visualise the duration and actual order of activities.
- **There are several approaches to predicting the cost of a project. In most cases, techniques try to infer the cost of a project from the costs of previous projects.** The idea is that organisations collect data on the costs of previous projects, along with some basic metrics (e.g. their size in terms of lines of code). This then forms the basis for a relation between size and cost. Accordingly, if it is possible to predict the size of future projects, their costs can be extrapolated from the existing data. These approaches tend to require some parametrisation – e.g. the predicted size, and perhaps other parameters to capture the changeability of requirements or the experience of the team.
- **COCOMO is a cost prediction approach that provides pre-computed parameters.** The idea is that any organisation can calculate the predicted cost for a project by simply 'plugging in' a set of parameters that characterise the nature of the project and organisation within which it is developed.
- **Planning-poker is an example of a cost-prediction approach that does not take a 'model-based' view of cost prediction.** It is a variant of the Wideband-Delphi method, popularised by Barry Boehm. The idea is that a group of developers collaboratively estimate the expected cost of individual activities (drawing upon their own experience).
- **The accuracy of a prediction approach invariably depends on the amount of intelligence available.** At the very beginning of a project, there is little experience or knowledge to draw upon, which means that any prediction is necessarily accompanied by a great deal of uncertainty. This uncertainty diminishes as the project progresses. This was visualised by Boehm in the 'Cone of Uncertainty'.
- **It is important to keep track of progress, so that any deviation from predicted effort / cost can be detected.** This can be achieved with the use of Burndown Charts and Team Velocity.

Chapter 6
Testing

So far, we have discussed software quality purely in terms of what it is (Chapter 2), and in terms of the process frameworks (Chapter 3), and the good practice – coding, design, and planning (Chapters 4 and 5) – that can be used to ensure it. Ultimately, however, it becomes necessary to actually take stock – to step back and assess the quality of the system at hand. This chapter marks the point where we move from discussing 'good practice' to discussing families of techniques – testing, inspection, and measurement, that are specifically dedicated to providing an objective assessment of (a particular aspect of) the quality of a system.

As discussed in Chapter 2, two of the fundamental definitions of (software) quality are Joseph Juran's statement that it should be fit for purpose, and Crosby's statement that it should conform to its requirements. The most obvious way to establish quality in either of those respects is to actually execute the software. In doing so, one can assess whether or not the software is of some value, or whether it is fulfilling its various requirements. This, in a nutshell, is what testing is all about.

In practice, testing can be a highly potent means by which to expose faults. This has been recently shown in a study on testing by Yuan *et al.* [138]. They sampled 198 bug reports from five large (data-intensive) Java and C distributed computing systems (Cassandra, HBase, HDFS, MapReduce, and Redis). 48 of these faults were "catastrophic", i.e. would have prevented a majority of users from access to the system. Amongst their findings, they state for example that 58% of faults that led to "catastrophic" failures could have been found by testing the exception handling code. More significantly, they found that 77% of the production failures can be reproduced by a unit test. In other words, 77% of the failures could have been caught if the systems in question had been properly unit tested.

6.1 The Foundations of Software Testing

In superficial terms testing is straightforward:

N. Walkinshaw, *Software Quality Assurance*, Undergraduate
Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_6

A program is executed and the outputs are checked to determine whether the program is behaving correctly.

However, these notions – a program, its execution, checking its outputs, and determining 'correctness', can mean wildly different things. They can vary according to the characteristics of the software being tested, the expectations and abilities of the person doing the testing, and the prior knowledge or expectations about what the system is supposed to do in the first place. As a consequence, many different flavours and notions have emerged around testing; usability testing, integration testing, unit testing, model-based testing, etc.

It is beyond the scope of this book to provide a comprehensive introduction to testing that covers all of these various notions. There are plenty of good text books that do this already [110]. Here, we provide an overview of the core principles of software testing. Most importantly, we will look at how different types of testing techniques can be used to corroborate software quality, linking back to the various definitions of quality that were discussed in Chapter 2.

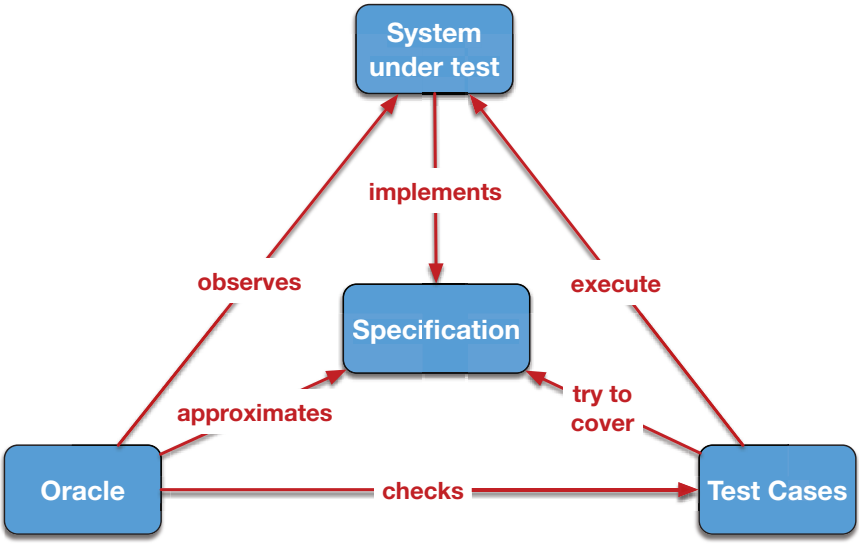

Fig. 6.1 Main elements and relationships that govern the testing process.

The key components of any testing process are laid out Figure 6.1, which was adapted from a paper by Staats *et al.* [125]. These are elaborated below.

System under Test

This is the system being tested, and is often abbreviated to SUT. It seeks to implement the specification (see below).

The SUT can either be a *white-box* system, where we have complete access to the source code and the run-time state (e.g. the call-stack), or a *black-box* system, where we only have access to the external interface or API (depending on the type of system). It can also be a mixture of the two; for example, library routines might be provided in the form of closed source components[1], whilst the source code for the main core of the system is available for analysis.

Although we use the term 'System', in our context this need not necessarily refer to the system in its entirety. A SUT might also just refer to just a single 'unit' in the system, for example a class or a function.

The system might be *reactive* where the input / output behaviour at one stage is affected by previous inputs (e.g. a GUI), or it might process inputs in a single batch and return to its initial state. This matters from a testing perspective, because in the reactive case, the test inputs have to be formulated as sequences.

The system might be *deterministic*, where it always returns the same answer for a given input. It might however also be *non-deterministic*, where the same input can elicit different outputs (perhaps because of randomised internal behaviour, or other factors beyond control such as thread-scheduling).

It is commonly important to ensure that the SUT is 'sandboxed' – isolated from the operational production system. The reasons for this were almost demonstrated to disastrous effect in the late 70s with the NORAD nuclear missile defence system we mentioned briefly in Chapter 2. Two of the false alarms raised by that system were inadvertently caused by tests. The following are extracts from the actual report [37] (the first bullet point must have been especially frightening!):

- On November 9, 1979, false indications of a mass raid were caused by inadvertent introduction of simulated data into NCS.
- On June 6, 1980, false attack indications were again caused by the faulty component during operational testing.

Specification

A specification represents the idealised behaviour of the system under test. Depending on the development context, this might be embodied as a comprehensive, rigorously maintained document (e.g. a set of UML diagrams or a Z specification). Alternatively, if developed in an agile context, it might be a partial intuitive description captured in a selection of user stories, test cases, and documented as comments in the source code.

The nature of the specification has an obvious bearing on testing. If a concrete, reliable specification document exists and there is a shared understanding of what

[1] Often referred to as Components Off The Shelf or COTS.

the system is supposed to do, this can be used as the basis for a systematic test-generation process. If this is not the case, then testing becomes a more ad-hoc and dependent upon the intuition and experience of the tester[2].

Test cases

The test cases correspond to the executions of the SUT. In practical terms a test case corresponds to an input (or a sequence of inputs) to the system.

Test cases should ideally cumulatively execute every distinctive facet of software behaviour. An ideal test set (collection of test cases) should be capable of exposing any deviation that the SUT makes from the specification. If it can be shown to do this, the test set is deemed to be *adequate* [125].

One of the fundamental problems of software testing is that there is no way of guaranteeing that a test set is adequate. This problem was best captured by Edsger Dijkstra, when he stated that "testing shows the presence, not the absence of bugs" [26]. He was in effect arguing that, since a program can accept an infinite number of possible inputs, testing can only represent a nominal sample of the program's possible behaviours, and cannot be used to make any claims about the absence of faults.

As a consequence, many approximations of test adequacy have arisen. These include notions such as code coverage and specification coverage, which we elaborate upon below. However, even when such goals do exist, there still remains the challenge of actually identifying the test cases that are able to achieve them. The reason for this is that the behaviour of a program (and thus the contribution to any measure of coverage that is made by an input) cannot generally be anticipated or computed in advance. This makes it impossible to determine which combinations of tests (if any) can achieve a given level of coverage. This problem is generally referred to as the *test generation* problem.

Test Oracle

Executing the test cases alone will not determine whether the SUT conforms to the specification or not. This decision – whether or not the output of a test is correct or not – is made by a test oracle. In practice, an oracle might be an assertion in the source code that is checked during the test execution, or it might be the human user, deciding whether or not the behaviour is acceptable.

Test oracles are notoriously difficult to produce [13]. There is in practice rarely an explicit, comprehensive, up to date specification that can be used as a reference. A successful software has usually been developed over the course of decades by a multitude of developers, which means that, ultimately, there is rarely a definitive record of how exactly the system should behave. What's more, there may be tens

[2] Not that this is necessarily bad; it just represents a shift in responsibility from process and documentation to trust in the individual developer

of thousands of test cases, each of which might produces complex outputs, which can make the task of manual validation of the outputs prohibitively time consuming. These issues are collectively referred to as the *oracle problem*.

6.2 White-Box Testing

Having covered the basic notions within software testing, we can now look at some specific scenarios and and the most popular approaches. As we will see, there is no "silver bullet" – no perfect testing technique that is guaranteed to highlight any bugs. Nonetheless, there are several approaches that are widely accepted and used.

White-box testing is concerned with the scenario where the tester has access to the source code and some aspects of the runtime state of the SUT. Source code presents perhaps the most popular basis for assessing *test adequacy* (as defined above); a test set can be deemed to be sufficiently complete if it exercises the source code in a sufficiently extensive manner.

6.2.1 Code coverage

Measuring coverage requires the ability to keep track of which source code elements are executed within the code. In many IDE's such facilities are built-in, or can be obtained via plug-ins[3]. Alternatively, it is often reasonably easy to construct a custom-made tool using a suitable source code analysis framework[4].

Many standards for certifying software quality mandate the use of code coverage to ensure the adequacy of test sets. One prominent example is the DO178-B/C standard for civilian aircraft software [1]. In this standard, software components are categorised according to their "criticality" - from components that would pose no threat if they failed, to components that are clearly safety-critical. On the non-critical end of the spectrum, it suffices to test components to achieve statement coverage, whereas for safety-critical components it is necessary to achieve the much more stringent MC/DC coverage (both detailed below).

Over the past 40-50 years, a huge variety of code coverage metrics have been devised [139]. Some of the most widely-used ones are as follows:

Statement coverage

The proportion of executable statements in the program that have been executed.

[3] IntelliJ IDEA ('urlhttps://www.jetbrains.com/idea/) has built-in coverage tracking. For Eclipse, there are plug-ins such as EclEmma (`http://eclemma.org/`)

[4] For Java, for example, there is the ASM framework `http://asm.ow2.org/`.

Branch coverage

The proportion of all of the logic-branches in the source code (e.g. outcomes of IF, WHILE, or FOR statements) to have been executed.

Def-Use or Dataflow coverage

The source code is analysed to extract the def-use relations, which relate statements at which a variable is defined (i.e. instantiated and given a value) to subsequent statements using that definition. The test-goal is to cover all of the possible def-use relations.

MC/DC (Modified Condition / Decision Coverage)

This is a more stringent extension of Branch coverage. The measure came to prominence because it was set as a standard against which to test safety-critical software (specifically critical aircraft software component that are to be certified to DO178-B/C [1]). MC/DC focusses on the especially rigorous testing of programs where there are decision-points (e.g. IF statements) that contain multiple conditions. For MC/DC, each individual condition must be shown to affect the outcome of a decision, independently from any other conditions.

We can consider an example (inspired by the NORAD missile-defence example discussed previously) in Figure 6.2.

```
public boolean raiseAlarm(boolean detected, boolean verified,
        boolean test){
    if(detected & verified & !test)
        return true;
    else
        return false;
}
```

Fig. 6.2 raiseAlarm code.

For typical branch coverage, the following inputs (for example) would suffice[5]:

```
raiseAlarm(true,true,false)
raiseAlarm(true,true,true)
```

However, from an MC/DC standpoint, this has only shown that the `test` condition independently affects the outcome of the if condition. To achieve MC/DC

[5] It is worth noting that this example is very simplistic. There is rarely a direct link from the input parameters to the conditions that one wishes to test, there is usually an additional challenge of finding the inputs that will guide the execution through the requisite branches / paths in the source code which, as we will see, is impossible to automate in a reliable way.

coverage, one would have to add additional tests that show the same for the other two variables:

```
raiseAlarm(false,true,false)
raiseAlarm(true,false,false)
```

The first test shows that `detected` will affect the outcome if it is switched to false. The second test shows that `verified` will affect the outcome if it is switched.

6.2.2 White Box Test Generation

Code coverage only addresses the question of how we measure test adequacy, but does not address the key problem of how to actually generate the test cases in order to achieve this adequacy. The subsequent task of finding a test set to achieve some degree of source code coverage is an interesting, and notoriously difficult, challenge. This comes down to the fundamental problem that test generation is **undecidable** [132]:

There is no approach or algorithm that can (in the general case – for all possible programs) determine whether or not a given piece of code within a program is executable.

This means that, even if we have full access to the source code, we cannot be guaranteed to be able to find an adequate test set for any arbitrary system. Despite this negative result (or perhaps because of it) numerous researchers have developed techniques that attempt to do their best anyway. If maximal coverage cannot be guaranteed (or even determined), the goal is to at least approximate this maximal coverage in some sense.

These approaches can be split into two families. The first family attempts to derive the necessary conditions on the inputs that are required to achieve coverage directly from the source code. The second family probes the source code by executing it with (quasi-random) inputs, and adjusts the inputs to gradually increase code coverage. These two approaches are introduced below.

6.2.2.1 Generating inputs by code analysis

This family of techniques attempts to figure out the relationship between each statement in the source code, and the input parameters that are required to reach it. The archetypical approach within this family is known as Symbolic Execution, and was proposed by King in 1976 [83].

Symbolic execution is based on the idea that the SUT can be "executed" in a special way, by substituting the actual inputs for symbols. Instead of running a single, conventional path through the source code, a *symbolic execution* traces all of

the possible paths through the program. For each path, it keeps track of the internal state of the program, and any conditions on the symbolic inputs that are required to reach that state.

To someone who is not familiar with the notion, this idea can be challenging to conceptualise from a mere description. Let us consider the piece of code in Figure 6.3, which calculates a person's Body Mass Index (BMI)[6].

```
1  public String bmi(double h,
2      double w){
3      double bmi = w / (h * h);
4      if(bmi < 16)
5          return "severely underweight";
6      else if (bmi < 18.5)
7          return "underweight";
8      else if (bmi < 25)
9          return "normal";
10     else if(bmi < 30)
11         return "overweight";
12     else return "obese";
13 }
```

Fig. 6.3 BMI code

A symbolic execution proceeds by associating every statement in the program under test with a 'path condition' – a condition on the inputs to the program that needs to be satisfied for the execution to reach that statement. This can be automatically generated by tracing the data-flow from the variables throughout the program. In our case, the `bmi` variable is formed as a computation on the `height` and `weight` variables. In order to execute line number 4, `bmi` must be less than 16. Accordingly the path condition for line 4 is:

$$w/(h*h) < 16$$

Consequently, the path condition for line 5 (the else-statement) is:

$$\neg(w/(h*h) < 16)$$

For line 6, the path condition is the conjunction of the above path condition and the condition that the computation is less than 18.5:

$$(\neg(w/(h*h) < 16)) \wedge (w/(h*h) < 18.5)$$

Exercise: *Try to figure out the path conditions for yourself! Pick an implementation of a method (c.f. a simple QuickSort implementation, such as:* `http://`

[6] `https://en.wikipedia.org/wiki/Body\_mass\_index`

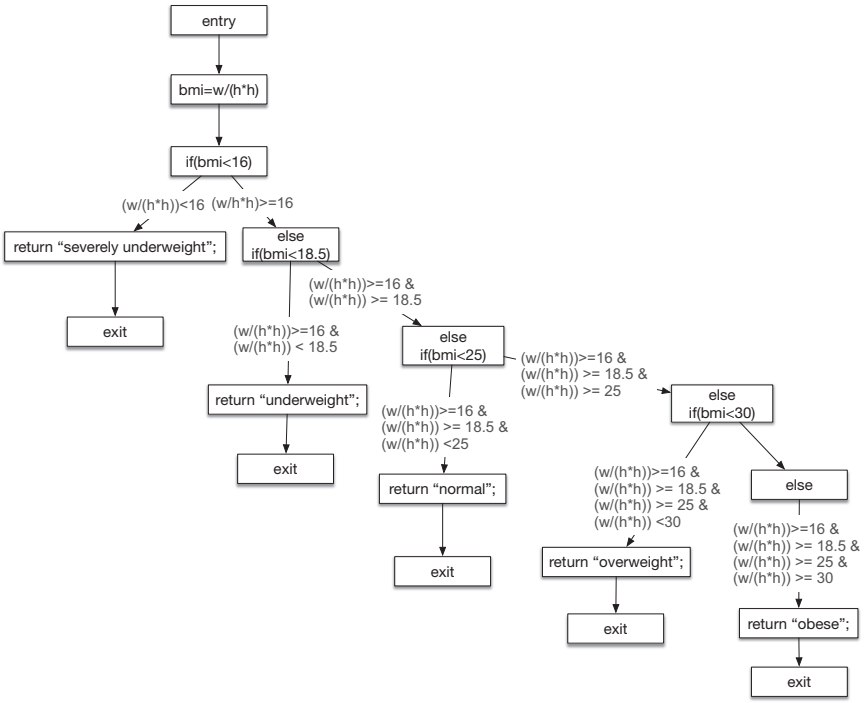

Fig. 6.4 Control Flow Graph of BMI example, with Path Conditions for different branches.

`www.algolist.net/Algorithms/Sorting/Quicksort`) and work out the path conditions that govern each branch.

The computation of the remaining path conditions continue in a similar fashion. The full control flow graph with path conditions is shown in Figure 6.4.

The computation of the path conditions is only the first step in symbolic execution. The identification of suitable inputs can only be achieved by solving them. Solving the path conditions can be achieved automatically with the help of automated constraint solvers.

Ultimately, the capabilities of a symbolic execution engine depend on the capabilities of its underlying solver. Commonly, solvers such as Microsoft's Z3 solver[7] are used, which can rapidly identify solutions for a many types of constraints.

Note the use of the phrase 'many types of' – there are several kinds of constraints for which there are no efficiently computable solutions. This means that symbolic execution cannot guarantee to determine the inputs for *every* statement in *every* program.

[7] `https://github.com/Z3Prover/z3`

Exercise: *This relates back to the problems of undecidability discussed above. How?*

In their basic form, symbolic execution techniques have been beset by problems. There are the limitations imposed by constraint-solving technology discussed above. However, there is also the problem that programs typically have loops; often lots of them. This is highly problematic when working out the path-conditions, because it is not always possible to determine when a loop will terminate (i.e. one can end up trying to calculate an infinite number of path conditions).

Work-arounds to these problems tend to require execution fo the program. Approaches such as *Concolic Testing* [117] combine symbolic execution with "concrete" execution. This makes it possible to plug-in actual values into symbolic variables whenever a condition is too hard to solve, or the limits of a loop are not determinable.

Such pragmatic solutions have recently made it possible to use symbolic execution in a more applied, industrial setting. To see symbolic execution in action, you can have a look at Microsoft's PEX For Fun site[8]. Microsoft have also employed symbolic execution as a part of their IntelliTest framework[9], which can be used to automatically generate tests for .NET programs as part of their VisualStudio IDE (from 2015 onwards).

6.2.2.2 Generating inputs by experimentation

An alternative approach to the analytical approaches espoused by symbolic or concolic execution is to ignore the contents of the conditions altogether, and to try to find the best set of test cases by an experimental process. This approach has broadly become to be known as 'Search Based Testing' [94]. The rationale is that, if for a given test execution it is possible to observe the extent to which the code has been covered, then the corresponding test inputs, paired with their coverage data, can be used as the basis for finding better inputs.

Perhaps, again, the simplest way to convey an intuition of how these approaches work is by example. Let us again consider the BMI code in Figure 6.3. Let us suppose that we know the input to bmi (that it consists of a pair of doubles), but we do not wish to / are unable to analyse the source code. All we can obtain is a number that corresponds to the number of statements (or branches, etc.) covered by a given test set.

Let us start with a completely arbitrary / random test set, shown in Table 6.1. This set of inputs would end up executing 8 of the statements in the program (as). One natural approach would be to simply carry on generating random test inputs. However, this could easily lead to a test set "explosion" – where the number of test

[8] `http://www.pexforfun.com/`

[9] `https://www.visualstudio.com/en-us/docs/test/developer-testing/intellitest-manual/input-generation`

h	w	Result	Statements covered
1	20	normal	1,2,3,4,6,8,9
2.4	100	underweight	1,2,3,4,6,7
7	75	severely underweight	1,2,3,4,6,7
2	119	normal	1,2,3,4,6,8,9
5	500	normal	1,2,3,4,6,8,9
		Total	1,2,3,4,6,7,8,9

Table 6.1 Initial, random test set. Covers 8 lines of code in total.

cases becomes too large to manage. Instead, we want to find a test set that is of approximately the same size, but with a better coverage. There are many strategies by which to achieve this. One of the simplest approaches is the 'hill-climbing' method. This operates by trying to "explore the neighbourhood" of the test set.

There are many possible ways in which to formulate the 'neighbourhood', and indeed many possible ways in which to formulate a hill-climbing testing algorithm. For example, one could try, for every value of every input to increase it, and then to decrease it by some amount. For the test set in Table 6.1 this would lead to 20 new tests (four new candidate tests for each of the five existing test cases). For every set, a hill-climbing search would keep track of the code covered, and then pick the test case for which the coverage had increased the most. This would then form the new test set, and the process would reiterate until no increase in statement coverage could be found.

To illustrate this process, Table 6.2 shows a search of the values around the input h=2 and w=119, by increasing and decreasing the height by 0.1, and increasing and decreasing the weight by 1. This ultimately ends up including test cases that cover the two cases that are not covered in the initial test cases (overweight and obese), which ultimately ends up covering the whole method. To avoid a rapid increase in the size of the test set, one might only retain new tests that execute new lines of code (i.e. ignoring the italicised test cases in Table 6.2).

h	w	result	Statements covered
1	20	normal	1,2,3,4,6,8,9
2.4	100	underweight	1,2,3,4,6,7
7	75	severely underweight	1,2,3,4,6,7
2	**119**	**normal**	1,2,3,4,6,8,9
5	500	normal	1,2,3,4,6,8,9
1.9	119	obese	1,2,3,4,6,8,10,12
2.1	119	overweight	1,2,3,4,6,810,11
2	118	normal	1,2,3,4,6,8,9
2	120	overweight	1,2,3,4,6,810,12
		Total	1,2,3,4,6,7,8,9,10,11,12

Table 6.2 "Searching" around the highlighted initial test set (h=2, weight=119). Three of the four candidate replacements improve the coverage of the initial test set by executing new branches.

This hill-climbing example is simply to show how a search-strategies (hill-climbing in this case) can be applied to the problem of test-generation. There are lots and lots of alternative search-strategies and heuristics, which have been applied to great effect [94]. There are also several tools that have emerged. The EvoSuite tool [56], for example, uses a search-framework known as 'Genetic Algorithms' [59] to maximise coverage of Java units.

6.2.3 The Case(s) Against Code Coverage

White-box testing is appealing because code coverage provides a seemingly direct measure of test-adequacy. Coverage can, it seems, be used to assess and improve test cases. As such, it is appealing for use as a tool; many software certification standards (such as the DO178 standard we have mentioned previously [1]) explicitly impose code coverage targets on software systems.

In reality, however, code coverage is not as useful as it appears and can, if anything, lead to a false sense of security. In other words (and we will come back to this term in Chapter 8) code coverage is not a *valid* measurement. Although a score of 100% adequacy is meant to guarantee that the full range of behaviour of a program has been executed (and thus that any bugs within the program are guaranteed to have been exposed) this is not necessarily the case.

```
1  public String bmi(double h,
2      double w){
3      double bmi = w / (h * h);
4      if(bmi < 16)
5          return "severely underweight";
6      else if(bmi < 18) //BUG! <- 18.5
7          return "underweight";
8      else if(bmi < 25)
9          return "normal";
10     else if(bmi < 30)
11         return "overweight";
12     else return "obese";
13 }
```

Fig. 6.5 Buggy BMI code

We can give a simple illustration of this by returning to our bmi example. This time, in Figure 6.5, we have inserted a bug by changing the threshold bmi value in line 6 from 18.5 to 18. A successful test case would produce a different output in this program from the original program in Figure 6.3. A *valid* adequacy criterion should always ensure that any 'adequate' test set would be guaranteed to include this test case.

h	w	result
100	4	severely underweight
70	2	underweight
80	2	normal
110	2	overweight
130	2	obese

Table 6.3 A test set that achieves statement (and branch) coverage, but does not expose the bug in Figure 6.5.

It is easy to show how, in this case, statement coverage is invalid. For example the test set in Table 6.3 provides a test set that will cover all of the statements and branches in the bmi code (i.e. they are *adequate* with respect to these criteria). However, when it comes to computing the "underweight" category, the value computed for the bmi is 17.5, and would lead to the same output in the unbuggy program.

Exercise: *Find the test input that would expose the bug!*

This criticism has been widely acknowledged for many decades. Goodenough and Gerhart originally discussed the problems of using code coverage to assess adequacy in 1975 [60] – it was their work that led to the proliferation of the various alternative metrics discussed above. However, ultimately, none of the existing source code metrics is impervious to these problems.

Their criticisms have recently been supported by a large amount of empirical evidence. For example, in their recent work on assessing code coverage, Inozemtseva *et al.* [70] studied 31,000 test suites from five open-source projects. They found that, although there was a weak correlation between test suite effectiveness and code coverage, there was a much higher correlation between the shear size of the test set.

Exercise: *Think about the experiment that Inozemtzeva* et al. *might have carried out to reach the conclusions that they did. If code-coverage is not a valid measure, they must have had some basis for asserting this. What could they have referred to or done to give them a 'true' measure of the effectiveness of a test case?*

Inozemtseva's findings (along with other similar findings) are unsettling. At the beginning of this section we discussed how the need to achieve code-coverage is a fundamental part of most well-established quality assurance standards (such as DO178-B [1]). The fact that such measures are in fact invalid means that achieving code coverage fails to provide the assurance that had been presumed. Although code coverage can be a useful starting point for identifying test sets that are effective at exposing faults, it is by no means sufficient.

6.2.4 Goto Fail: A Case For Code Coverage

Code coverage has been shown to be a poor basis for producing truly adequate test sets. Executing the code is merely a first-step to exposing a bug (which can also depend on more subtle underlying data constraints). And this 'first-step' part is important. If a piece of code is not executed, then any bugs therein will simply not be exposed. If a piece of code remains un-executed after testing, it still indicates that there is a problem with the test set, or with the program itself.

One appropriate example is the Apple 'Goto Fail' bug [10]. This fault arose in 2014, and featured in its iOS and OSX operating systems (i.e. spanning many of its devices, from mobile devices such as iPhones and iPads up to desktops, laptops and servers). The bug was contained in an important unit of code that is responsible for verifying SSL signatures. Any bugs in this code (such as this one) represent a potentially major security vulnerability that can give unauthorised users the ability to eavesdrop on communications to and from the device (for example bank details sent to an online banking or shopping website). The source code for the bug is shown in Figure 6.6.

```
1  static OSStatus
2  SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
3          SSLBuffer signedParams, uint8_t *signature, UInt16 signatureLen)
4  {
5      OSStatus        err;
6      ...
7
8      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
9          goto fail;
10     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
11         goto fail;
12         goto fail;
13     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
14         goto fail;
15     ...
16
17 fail:
18     SSLFreeBuffer(&signedHashes);
19     SSLFreeBuffer(&hashCtx);
20     return err;
21 }
```

Fig. 6.6 The code for the Goto-fail bug, provided by Adam Langley `https://www.imperialviolet.org/2014/02/22/applebug.html`

Exercise: *Can you spot the bug?*

The key lies in the two consecutive `goto fail;` statements. In the event that the `if` statement at line 10 evaluates to false, `err` will contain a successful value.

[10] CVE-2014-1266 `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266`

This is exploitable, because it is not being checked according to the condition in statement 13.

In other words, statement 13 is *not executable*. No matter what test-input you provide to the program, this statement will not be covered. In a testing context that places emphasis on code coverage, this problem would be flagged up straight away, because lines 13 and 14 *can never be executed*.

6.2.5 An Alternative: Mutation Testing

Let us remind ourselves of the definition of what code coverage was trying to measure: 'test adequacy' (as described in Section 6.1) is the extent to which a test set can be guaranteed to expose any bugs in the program [60]. If trying to maximise syntax-coverage is ineffective, then what are our alternative options?

Mutation testing [74] presents an alternative means to assessing adequacy by interpreting this definition in a very literal sense. The underlying idea is to produce lots of copies of the SUT, but where each copy is buggy in a subtly different way. In other words, the goal is to insert lots of bugs into the program, then to assess how good a test set is at exposing them. This (to answer the above exercise) is the technique that was used by Inozemtzeva *et al.* to assess their test sets.

In mutation testing terminology, a 'bug' that is introduced to a program is called a 'mutation'. This is because each mutation 'mutates' the existing source code, according to a set of given *mutation operators* . For example, one mutation operator might change a '+' in the source code into a '-'. It might change the definition of a String variable from `CustomerID` to `null`, etc. For every application of a mutation operator, a new, separate program is produced.

Figure 6.7 provides some examples of what mutants look like. You can imagine different versions of the program, but with one of the currently commented lines replacing the lines they precede. For example, one mutated program could replace line 5 with line 3.

Although mutation testing is intuitive, its validity can be called into question. Is the ability of a test set that is able to kill a large proportion of mutants necessarily indicative of its ability to expose "real" bugs? This, one would think, depends on the choice of mutation operators, and on the nature of the program.

There have however been several empirical studies that have examined the relationship between the two – between the ability to kill lots of mutants and the ability to expose real bugs. One notable example is a recent paper by Just *et al.* [79]. They collected a large set of real bugs from a variety of Java programs, and used this as a basis for assessing a range of test sets, and specifically comparing the relative mutation scores of the test sets to their relative capacity to expose real faults. Their results indicate that there is indeed a significant correlation; test sets with higher mutation scores do tend to expose more real faults.

```
public String bmi(double h,
    double w){
//    double bmi = w / (h / h);
//    double bmi = w / (h * w);
    double bmi = w / (h * h);
//  if(bmi < 0)
    if(bmi < 16)
        return "severely underweight";
    else if(bmi < 18.5)
//      return null;
        return "underweight";
//  else if(bmi > 25);
    else if(bmi < 25)
//      return "";
        return "normal";
    else if(bmi < 30)
        return "overweight";
    else return "obese";
}
```

Fig. 6.7 BMI code from Figure 6.3, with potential mutations added as comments. Note that only one of these would be activated for a given mutated copy of the program.

6.3 Black-Box Testing

Whereas white-box testing suggests that the tester has complete access to the internal workings of the system – source-code and runtime-state information – black-box testing is the opposite. It suggests that the tester *cannot* access the internals of the system; all that they see is its interface. In practice, the system under test might, for example, be a remote network protocol, or a closed-source executable binary.

This setting obviously rules out the various white-box strategies discussed previously. We cannot assess test set adequacy by coverage, because we cannot observe what parts of the internal structure have or have not been covered. This setting for testing was first described by Edward Moore in 1956[99], who added the following, somewhat colourful scenario (he assumed that the SUT was not restricted to being merely a software system):

> There is one special situation that can occur in such an experiment that is worthy of note. The device being experimented on [tested] may explode, particularly if it is a bomb, a mine, or some other infernal machine. Since the experimenter [tester] is presumably intelligent enough to have anticipated this possibility, he may be assumed to have conducted his experimentation by remote control from a safe distance.

We will assume, for the purposes of this chapter, that the SUT is something more pedestrian such as a closed-source Java unit or a web-service.

In this black-box scenario, there are essentially two ways by which to go about generating test sets. One approach is to draw upon any available specifications of the system and to use these as a basis for generating targeted test inputs. Alternatively,

if these do not exist, the only alternative is to resort to quasi-random test generation approaches. We cover each of these approaches in the following sub-sections.

6.3.1 Specification-Based Testing

Specification-based testing[11] is concerned with the situation where we have a document that, in a reasonably comprehensive manner, captures the expected behaviour of the system. The fundamental idea of specification-based testing is to use this specification as a basis for driving the selection of test inputs, to ensure that any tests focus on establishing the correctness of whatever has been specified.

The specific mechanisms that are used to generate tests depend on the type of specification. For certain types of specification, there are very well-established test-generation techniques. Ultimately, this also depends on the nature of the system under test. For systems that depend on highly sequential activities, such as GUI's or network protocols, a specification might take the form of a state machine. For systems that are more data-driven, a specification might take the form of a simple data-invariant (i.e. an assertion in the source code) or a Z-specification [136]. If the specification is less formal, it might take the form of a structured natural-language specification such as CUCUMBER [137].

To provide a broad-brush overview of black-box testing, we will not go into the details of how all of these different types of specifications can be tested. We will focus only on two (complementary) aspects of software behaviour: sequential behaviour, and data constraints. For the sequential behaviour, we will provide a brief overview of state-machine testing (state machines are most commonly used to model this aspect of behaviour). For the data behaviour, we will focus on natural-language requirements, and the use of 'categories' to formulate test cases.

6.3.1.1 Testing Sequential Behaviour with State Machines

State machines[12] capture requirements on sequential behaviour of a system. An example is shown in Figure 6.8. This shows what should (and conversely should not) happen when someone tries to log-in to a system (e.g. a web-service). The 'entry-point' is the dash-board, and the user is not logged-in. If they click on the 'login' button, this takes them to some dialogue box where they can enter their log-in details. If accepted, they return to the dash-board in a logged-in state. If the login

[11] For the purposes of this chapter, by 'specification-based testing' we also refer to 'Model-based Testing', or 'Property-based Testing'. In practice these all have slightly different connotations, but are all fundamentally based on the same principles that we discuss here.

[12] There are lots of different types of state machines; they can be hierarchical (e.g. UML state-charts), factor in data conditions, temporal properties, etc. In this chapter we simply consider the simplest type of state machine - labelled transition systems where a label corresponds to some 'action' in the SUT.

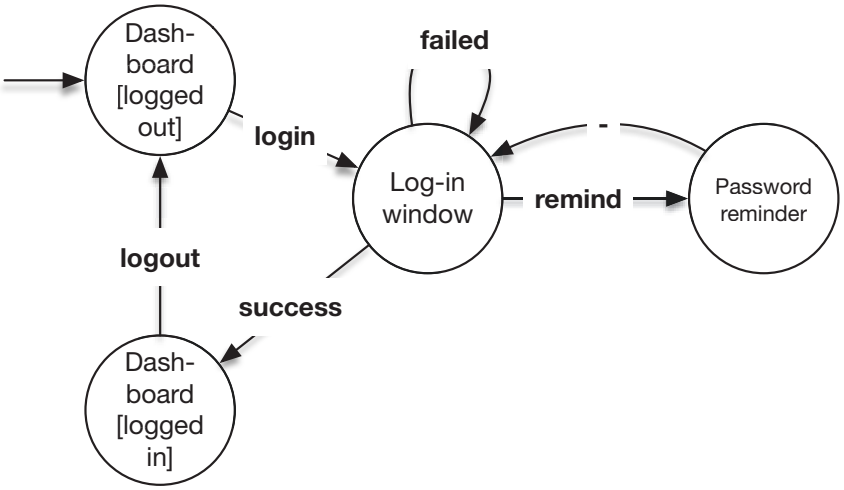

Fig. 6.8 Example state machine of a simple login requirement.

fails, they remain in the dialogue box. Alternatively, they can request a reminder of their login details.

Since Moore's early work on the subject [99], a large amount of research has been invested into suitable algorithms that can be used to test them [90]. The challenge is to determine the ideal set of 'program executions' – paths through the state machine – that will ensure that any deviations between the model and an implementation of it are exposed.

The approaches are broadly reminiscent of the various levels of code coverage discussed in Section 6.2.1. The simplest approach is 'state-coverage' - find a set of sequences in the state machine that 'cover' every state in the machine. In our simple example, it is possible to achieve state coverage with a single test:

$$< login, remind, -, success >$$

This however means that certain state transitions might go uncovered, so the next level up is 'transition coverage' (which is reminiscent of Branch coverage in the source code). Again, this can be achieved with a single test:

$$< login, failed, remind, -, success, logout >$$

State machine testing can be challenging because of the problem of 'equivalence'. Since the system under test is a 'black box', we cannot (usually) know for certain that the traversal of a state in the model amounts to the traversal of an identical state in the underlying system. As an example, the implementation might take the form of the faulty model shown in Figure 6.9, where all three inputs (`reminder`, –,

and `failed`) all lead to the same state, and no input leads to the password reminder state.

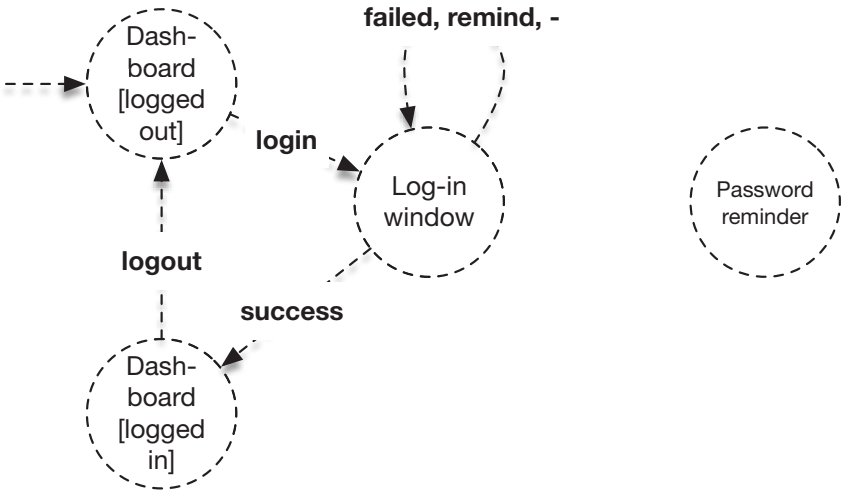

Fig. 6.9 A faulty implementation of the state machine in Figure 6.8.

Whether or not such a fault would be detected or not depends on the fidelity of our oracle. The most common assumption is that the oracle is "weak" – it is only able to check whether a sequence of events has executed or not. In this case, using either of the test cases listed above, it would not identify these faults. Of course, if the oracle is more capable, e.g. is able to verify that the SUT is in the correct state by checking the GUI, then this fault would be flagged up. The down-side of such an approach, however, is that this will tend to imply that the oracle is either human with plenty of time to spare, or that there is more than just the state-machine specification to rely upon, which is rarely the case in practice.

If we assume that the oracle is weak (i.e. that we are only able to check the sequencing of the inputs), then it is often necessary to carry out a more rigorous form of testing than merely achieving state or transition coverage. One approach (a simplification of what is known as the W-Method [30]) is to build a set of test cases as follows:

1. Let T be our test set (set of test sequences).
2. For each state s, find the shortest path p from the start state to s.
   - Construct a set A from the emphalphabet of the state machine (every possible input), and the empty event ε.
   - For each element $a \in A$, add $p.a$ to T

In other words, we find a path to each state, and try every possible input from each state. From this we can check whether an input is possible from a state that should not be, or vice versa.

Exercise: *Produce this more rigorous test set for the state machine in Figure 6.8.*

It is often insufficient to merely find sequences that cover all states or all transitions, because these will not necessarily guarantee that a defect will be found. To ensure this, once a state has been reached, it is also necessary to attempt sequences that can elicit responses from the system that can guarantee that the system is in the expected state. There are lots of ways to compute these sequences, and some of the main approaches are surveyed by Lee and Yannakakis [90].

6.3.1.2 Category Partition Method

State machines have historically been appealing to study because they form a graph, which offers an intuitive basis for assessing coverage (in a similar way to the control flow graph for white-box testing). What, then, about non-sequential black-box systems? What about programs that take as input some data argument, and process that input in a single step?

The Category Partition Method [104] provides a basis for testing such non-sequential systems. The fundamental idea is to use the specification to identify specific 'categories' of inputs, which are supposed to elicit a particular output characteristic. Test generation then consists of ensuring that every category (or combination of categories) is executed.

As an example, we can consider a program that calculates personal income tax-rates in the UK. As this is a black-box scenario, let us assume that the system we are trying to test is a web-service. To apply the Category Partition Method, we would start with the high-level requirements. The high-level rules for tax allowances are shown in Figure 6.10.

Looking at the rules, we can discern several obvious categories, pertaining to the straightforward tax bands: Personal allowance, basic rate, higher rate, and additional rate. We also know that the question of whether or not a claimant is married or blind might affect their tax returns. There is also a threshold value of £100,000 that could affect the tax rate.

From the requirements we can also surmise that there must be (at least) three inputs in the interface to the program: The salary / income (*pay*), and two boolean variables that indicate whether the individual is married or blind respectively (*married* and *blind*). The full set of categories is captured in Table 6.4.

Having identified the categories, the Category Partition Method continues by systematically combining these to formulate a comprehensive set of test cases. It is obvious, however, that many of these categories are mutually exclusive; for example

someone's income cannot be below £11,000 and above £150,000 at the same time. Nor can somebody be blind and not blind, etc.

Your tax-free Personal Allowance

The standard personal allowance is £11,000, which is the amount of income you don't have to pay tax on.

Your Personal Allowance may be bigger if you claim Marriage Allowance or Blind Person's Allowance. It's smaller if your income is over £100,000.

Income Tax rates and bands

The table shows the tax rates you pay in each band if you have a standard personal allowance of £11,000.

Band	**Taxable income**	**Tax rate**
Personal Allowance	Up to £11,000	0%
Basic rate	£11,000 to £43,000	20%
Higher rate	£43,001 to £150,000	40%
Additional rate	over £150,000	45%

You can also see the rates and bands without the Personal Allowance. You don't get a Personal Allowance on taxable income over £122,000.

Fig. 6.10 UK Tax rules for the year 2016[13].

Category	**Condition**	
A	In Personal Allowance	$pay \leq 11,000$
B	In Basic Rate	$11,000 < pay \leq 43,000$
C	In Higher Rate	$43,000 < pay \leq 150,000$
D	In Additional Rate	$pay > 150,000$
E	Above 100k, sub additional rate	$100,000 < pay < 150,000$
F	Blind	$blind = true$
G	Not blind	$blind = false$
H	Married	$married = true$
I	Not Married	$Married = false$

Table 6.4 Possible categories for the tax calculator.

One straightforward way to set out the possible and impossible category combinations is to set these out as a graph[14]. The relationships between the categories in Table 6.4 is shown in Figure 6.11. Categories A, B, C, D, and E all correspond to

[13] Source:
`https://www.gov.uk/income-tax-rates/current-rates-and-allowances`

[14] Although straightforward, this approach cannot always be applied as-is. If there are conditional constraints that hold between different categories (e.g. the example in Ostrand's paper [104]), a more involved approach is required.

values of a single variable (*pay*), so these cannot be combined with each other, and are put in a row without edges between them. The same goes for categories F and G (*blind*) and H and I (*married*). A test case thus corresponds to a path through this graph, from a node without incoming edges to a node without outgoing ones. In the graph in Figure 6.11 there are 20 such possible paths.

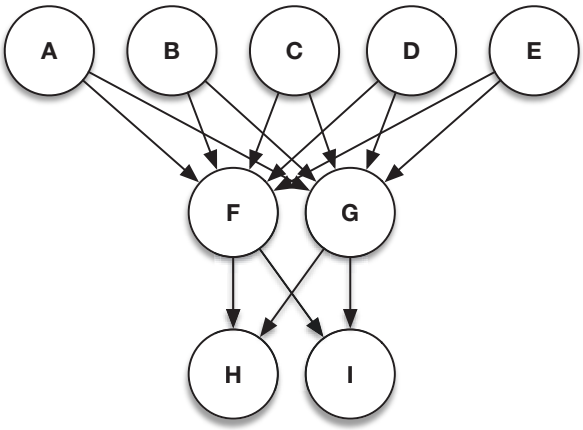

Fig. 6.11 Graph of possible category combinations, where category labels are defined in Table 6.4.

Exercise: Try applying the Category-Partition approach to producing test cases for a different system. You could, for example, attempt the `wc` command in Unix. Use the man-page (obtained in Unix / Linux by typing `man wc` in a terminal) to identify the categories. For inspiration, you can look at Ostrand and Balcer's original paper, which used a similar example (the find command).

6.3.2 Random Testing

So far, black-box techniques have depended upon some form of specification / requirements document to guide the selection of test cases. These have either been in the form of an explicit model (e.g. state machines), or have been extracted from informal documents (e.g. the categories in the Category Partition method). This reliance upon 'test-specifications' is in many circumstances unrealistic. Requirements documents are rarely kept up to date; there is rarely a firm, up-to-date document that concisely sets out what to expect from the behaviour of a system. The system is also black-box, which makes it impossible to scrutinise its internals to attempt to derive

its behaviour. In this context, the only apparent way to generate test cases is to do so at "random" [67].

Given the fact that it is not especially dependent upon the availability of specifications or models, random testing is popular in the industry. Random testing was used within NASA for their Mars exploration rovers [63]. Numerous testing companies exist that are able to apply their random testing tools to systems as diverse as mobile phones and cars (c.f. Quviq[15] and Vertizan[16]).

6.3.2.1 Defining the Input Space

At this point it helps to be a bit specific as to what is meant by the term "random". In simple terms, it is picking a possible outcome from a finite (but potentially very large) number of possibilities, where the probability that any given outcome will be selected is the same for all outcomes. You can envisage the selection of a random element as being similar to a lottery-machine. Each outcome can be one of the labelled balls in the machine, and the selection of one of the balls occurs at random.

The effectiveness of random testing rests on our ability to define and constrain the "input space" in such a way that maximises the likelihood that a randomly selected set of inputs will expose as many distinct facets of behaviour (including potentially faulty behaviour) as possible. Returning to our lottery analogy, the effectiveness of random testing relies on our ability to pick a suitable set of balls for the machine.

As an example, we can imagine a scenario where we are testing a black-box implementation of the BMI calculator that was discussed in Section 6.2[17]. To randomly test this, you would essentially choose a set of random values for height and weight. The input space would be defined by the upper and lower limits that you impose both parameters.

Thanks to the fact that we know what is inside the black box (and bearing in mind that you usually don't know this), we can visualise the input space as shown in Figure 6.12, and in doing so illustrate how critical it is to define it properly. In the plots height and weight are plotted along the x and y axes respectively. Colours correspond to different outputs. Both plots also contain 100 randomly scattered dots, corresponding to random test inputs.

The plots illustrate just how important it is to carefully pick value ranges for input variables[18]. In the left-hand plot, the range is selected so that more of the inputs fall within plausible height and weight ranges, and thus more often elicit varying BMI scores. In the left plot, the random test inputs cover more fo the outputs (and thus expose more of the program behaviour) more often than in the right hand one.

---

[15] `http://www.quviq.com/`

[16] `http://www.vertizan.com/`

[17] As a reminder, the inputs for this program were a person's height and weight, and the output was a categorisation of severely underweight, ..., obese.

[18] Although we used numerical inputs here for the sake of simplicity, the same principle of course also applies to other types of input.

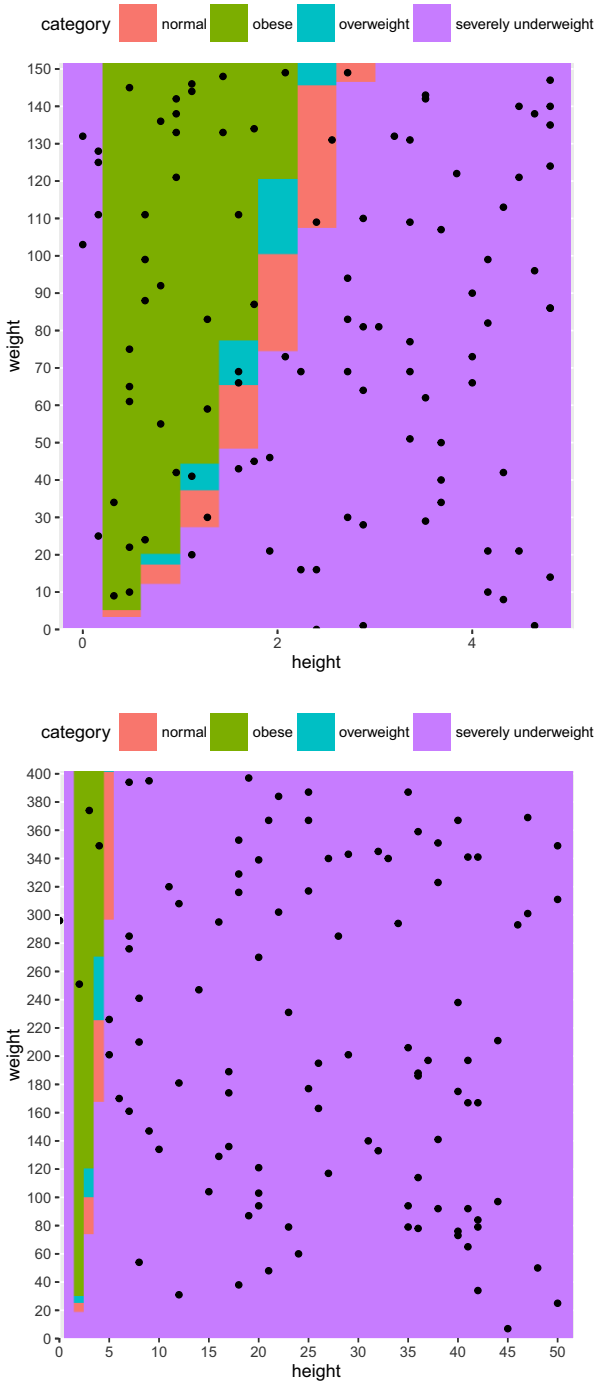

Fig. 6.12 Visualisation for input space for BMI program. The left plot shows the space for $0 < height < 5$ and $0 < height < 150$. The right plot shows the space for $0 < height < 50$ and $0 < height < 400$. The plots are scattered with 100 dots (within the specified limits) which correspond to random test inputs.

Complex inputs

One useful property of random testing is the following property: If the test generation approach is truly random is capable in theory of producing inputs that cover every distinctive facet of software behaviour, then it will eventually almost certainly[19] do so [71].

It is crucial to leave open the possibility that *any* feasible input to the program could be executed (regardless of how non-sensical it is). However this becomes difficult when considering larger programs, with less straightforward input types. One could, for example, consider a web-app, where input is provided via browser, and the modes of the GUI change according to which options are selected via widgets in a webpage. A truly random set of movements, clicks, and text entries would probably fail to (within a reasonable amount of time) meaningfully explore the app beyond the entry page. On the other hand, enforcing too many constraints on the input selection will remove the randomness that makes random testing so powerful.

The problem is reminiscent of the hypothesis that if you put a sufficiently large number of monkeys in a room for a sufficiently long time, that they would eventually end up typing up the works of Shakespeare. If you place no constraints whatsoever on the monkeys, you probably won't get them to type anything[20]. On the other hand, if you place several constraints on them, and ensure that they continuously type, then it becomes more plausible. This is what happened in a subsequent experiment [10] (which was carried out on virtual monkeys due to the obvious ethical implications).

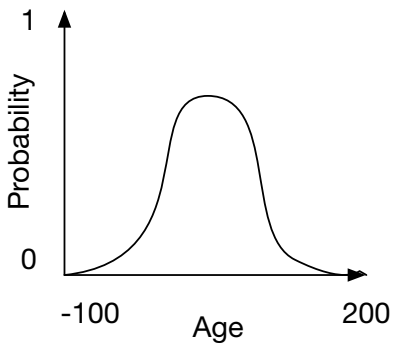

Fig. 6.13 A probability distribution over the "age" input for a fictional program.

In the context of testing, random test-generators tend to employ various tricks to restrict (most of) the inputs to those that are liable to trigger some interesting form of behaviour. If the inputs are numeric, it is possible to use probability distributions.

[19] The probability of doing so tends to 1.

[20] Confirmed in the only such experiment involving real animals: `http://news.bbc.co.uk/1/hi/3013959.stm`

An example of a probability distribution over an "age" input is shown in Figure 6.13. This seeks to ensure that, although some inputs might fall outwith the age of a typical human, that most will fall somewhere closer to a realistic age. This is in contrast to a "uniform" distribution, where the likelihood of a random generator choosing the value of 200 or -100 is identical to the probability of choosing the value 0, 10, or 30.

For more complex input types (e.g. a program that takes a date as input) custom-made data generators can be created. For example, Quick-Check[21] – one of the leading random test frameworks – revolves around such generators. To generate a date (as three numbers), a constrained generator might focus the selection of the number representing the month on the values -1 to 13 (for example), and might pick numbers representing the days to focus on particular combinations of months and days that could trip up an implementation.

Exercise: Devise your own random date generator for the sake of testing. Write a routine that can generate an unlimited number of dates, most of which are valid, but some of which are invalid.

6.3.2.2 Quantifying Reliability

Random testing is more than just a last resort to produce input data. It offers certain unique properties that do not arise with other non-random approaches. If we assume that the inputs are selected independently (that the selection of one cannot affect another) and that they are truly random, then it becomes possible to apply probabilistic reasoning to the question of how *reliable* it is.

The reasoning works as follows [67]. First, let us use θ to represent the probability that a system will fail (its 'failure-rate'). Of course, in reality this value is rarely definitively known in advance. Nonetheless, it can often be estimated (e.g. by counting the proportion of failures in previous executions).

This means that the probability that the probability that a given test will succeed is:

$$1 - \theta$$

For N independent tests, the probability that all tests will pass is:

$$(1 - \theta)^N$$

The probability e that at least one failure will be observed is

$$e = 1 - (1 - \theta)^N$$

[21] `http://www.cse.chalmers.se/~rjmh/QuickCheck/`

Exercise: Let us assume that in 2,900 executions, there have been 121 failures. How many executions would it take to observe at least one failure?

6.3.2.3 The Rule of Three

The traditional approach of quantifying reliability relies upon some way of estimating the probability that an individual test will succeed or fail – θ. This is fine if there are previous failures to draw upon; if the proportion of failed test executions can be measured. However, what if there have been no observed failures at all? In this case, there is no reliable basis for estimating reliability.

For this we can refer to a similar problem that is prevalent in the field of Medicine. Imagine that a new medical intervention is devised – e.g. a surgeon comes up with a new surgical procedure, or a new medical treatment is developed. At some point it becomes necessary to quantify how safe this is for patients. As is the case with testing, it can easily be the case that, even after several hundred operations, no patient has suffered ill effects. How then can the surgeon produce an estimate for the safety of their procedure?

One rule of thumb that medics have used to work around this problem is known as the "Rule of Three" [47]. The rule is as simple as it sounds. If you have N observations (where $N \geq 30$), and have not observed any "adverse events" (in our case test failures) the rule of three runs as follows. You can estimate with a confidence of 95% that the probability of a future failure is $\frac{3}{N}$.

Exercise: Let us assume that you have observed 37 software executions that have not failed. What is the probability (to a confidence of 95%) that a future execution will fail?

6.3.2.4 Improving upon Random

Random testing has an bad reputation – in practice the vast majority of its inputs end up being non-sensical, and thus only skirt a small, often trivial, subset of functionalities[22]. As a result, it is unlikely that random tests will yield a test set that is anywhere close to adequate within a reasonable number of tests. To address this issue, several approaches have been devised that seek to use some form of strategy to select inputs, in the hope that these will more rapidly converge upon an adequate set.

[22] This reputation is somewhat unfair in my opinion, because it probably arises from the fact that an insufficient amount of effort has been invested in the definition of the domain from which the inputs are selected, as discussed above.

One technique that has gained some traction (at least in the academic testing community) is Adaptive Random Testing (ART) [28]. This approach adds a step to conventional random testing. Instead of simply selecting a single test case and executing it proceeds as follows: It selects a set of possible test inputs (say 10) and, for each candidate, computes a 'distance' between it and the set of test cases that have already been executed. The nature of the distance measure depends entirely on the types of the inputs. In the simplest case, if the inputs are all numerical, it becomes possible to use one of the types of distance measures that arise between multi-dimensional numerical coordinates, such as the Euclidean distance. Having computed the distances between every proposed test input and all of the existing test cases, ART then chooses the test case with the biggest distance, on the rationale that this is most likely to elicit some form of behaviour that has not already been witnessed.

h	w	Result
1	20	normal
2.4	100	underweight
7	75	severely underweight
2	120	obese
5	500	normal
Proposed by ART		Minimum distance
1	12	*8.0*
7	3	*18.02*
2	200	***80.0***
3	100	*0.6*

Table 6.5 Illustration of ART with respect to the random BMI inputs produced in Table 6.1.

As an example, let us look at Table 6.5. In the upper section, there is a randomly generated set of tests for the BMI program (taken from table 6.1). Below are four random test inputs proposed by ART. For each test case, the (euclidean) distance was calculated with each of the other test cases (in the upper half of the table). The value to the right of the test cases represents the minimum distance that was recorded. Here we clearly see that the input $(2, 200)$ has the largest minimum distance, which makes it a prime candidate, as it is 'furthest away' from all of the other tests.

Exercise: The choice of distance measure for ART is clearly crucial. Looking a the above BMI example, what is an apparent problem with our use of the Euclidean measure?

ART is not without its problems. Firstly, the choice of distance measure is crucial. In our BMI example, height and weight are in different units; height is in meters, and weight is in kilograms. Euclidean distance is of course unaware of this. Accordingly, a unit difference in weight has the same effect on distance as a unit difference in weight.

Secondly, there is the question of time and performance – a point made by Arcuri *et al.* [11]. As the size of the tests set (or the number of input parameters) increases, there are more tests against which new candidate tests have to be measured. This incurs a performance penalty. Ultimately, if tests can be executed very rapidly anyway, it might make more sense to simply execute *all* of the candidate tests (i.e. to revert to conventional random testing) instead of using ART.

6.3.3 Exposing Security Flaws with Fuzz-Testing

Security is often an implicit factor when it comes to software testing. If we are able to highlight a bug - some unanticipated response by the software system - we are potentially highlighting security vulnerabilities in the process. When it comes to security-testing as a discipline, a large number of tools essentially apply the various white-box and black-box techniques described above (symbolic execution is especially popular in the security domain).

Fuzz testing is however a security-focussed technique that makes a subtle but interesting departure from what we have discussed so far. Fuzz testing does not merely have the goal of highlighting arbitrary bugs in a system; it aims to identify those bugs that are particularly pernicious – security loopholes where an input that might appear superficially valid can lead to some form of unauthorised access to the system.

Instead of generating test inputs from scratch, by analysing code or following some user-model, fuzz testing attempts to produce inputs that are "nearly valid". This can be accomplished a various ways, and depends on the material that is available, from which the inputs can be synthesised. For a black-box setting, there are two basic fuzz testing techniques: Mutation-based fuzzing, and input-structure based fuzzing. These are discussed in more detail below.

Mutation-based fuzzing

Mutation-based fuzzing operates on an existing test set. The idea is that there is some corpus of existing valid inputs to the system, such as a corpus of PDF files for a PDF-reader. The task for the fuzzer is to take these inputs, and to apply mutations to them. Mutations might vary from flipping a random bit, to being more advanced (e.g. trying to detect a data field within a data file and changing its value).

Exercise: Mutation-based fuzzers are appealing because they are very easy to create. Try to write your own mutation-based fuzzer to fuzz-test a file-processing program such as a PDF or image reader.

Input-structure fuzzing

Whereas mutation-based fuzzing does not assume prior knowledge of the input format, the availability of the format can be an enormous asset. If we consider the above example of testing a PDF reader, an awareness of the PDF file format can provide some valuable indicators of which areas within a PDF file might be the best areas to mutate. One particularly popular input-aware fuzzer is the Peach Fuzzer[23].

6.4 Key Points

- **Testing is a process that encompasses several components: The SUT, Specification, Test Set, and Oracle.** The SUT represents the system under test. The test set represents the set of inputs to the system that we wish to exercise. The specification represents an implicit or explicit capture of the idealised behaviour of the system under test. The oracle is a decision procedure (e.g. a programmatic assertion) that can determine whether the outcome of a test execution is correct or not.
- **Adequacy is the term attributed to the ability of a test set to 'cover' the behaviour of a system (and thus its capacity to expose potential faults).** There are no reliable ways by which to assess adequacy. The approach ultimately depends on what can be observed of the system under test, and what we know about its intended behaviour.
- **White box testing is concerned with testing systems where the source code and run-time dynamics of the SUT can be tracked.** In white box systems, test adequacy is commonly measured in terms of the executed source code. This can be measured in many ways, ranging from statement coverage down to mutation coverage and dataflow coverage.
- **Black box testing is the opposite of white-box testing – the internals of the system are not observable.** In this case, test generation tends to rely on the availability of an external document of how the system is supposed to behave (e.g. a model). Detailed models (e.g. state machine models) can be used to produce comprehensive test sets. Other models can take the form of broad constraints on inputs, which can be used by approaches such as the Category Partition method.
- **In the absence of any significant knowledge of what is to be expected from a system, random testing can offer a useful basis upon which to generate useful tests.** Random testing is (unsurprisingly) good at producing unexpected inputs. There are various random testing techniques that enable the tester to reason about the probability of failure within a system, and to reason probabilistically about the reliability of the system.
- **Fuzz testing is a special form of random testing, which is particularly popular for trying to highlight security flaws.** Fuzz testing works by starting from

[23] `www.peachfuzzer.com`

inputs (or input structures) that are known to be valid, and then proceeds to generate inputs that are ‘nearly valid’.

Chapter 7
Software Inspections, Code Reviews, and Safety Arguments

In the chapter on software testing, we have seen that there are numerous strategies by which to assess the quality of a software system by executing it. However, the effectiveness of testing is subject to a range of limitations; there is rarely a complete and reliable oracle, and there is no accepted means by which to generate adequate test sets (or even by which to measure adequacy). Furthermore, many aspects of software quality (such as the maintainability of the source code) cannot be established by testing.

Software inspections and reviews are concerned with the (usually manual) review of software artefacts. Whereas the goal with testing is relatively narrow (to establish correct behaviour), the goals with inspections can be broader; reviewers can consider whether the architecture is sensible, the code is maintainable, a code change is necessary, whether there are better potential solutions, etc. Inspections and code reviews are widely used within organisations, and in open source projects. The requirement for peer-review is an integral part of most development and quality assurance processes and tools.

In this chapter we will briefly cover the origins of software inspections and reviews. In their original form, software inspections were perceived to be too "heavyweight" for routine software development. This led to the development of new, lightweight "Modern Code Review" techniques, which we will cover in Section 7.2. The more heavyweight traditional inspections do however remain a key part of the development of safety-critical systems. We will thus look at inspections – specifically the safety-specific aspects of software inspections – in Section 7.4.

It is worth noting that the placement of sections on "conventional" software inspections alongside sections on safety assessments of software is somewhat unorthodox; in a traditional text book they would probably be in separate chapters. They are put together here because, conceptually, they are fundamentally related; they involve the manual scrutiny of software artefacts, with the aim of detecting problems and ensuring quality.

N. Walkinshaw, *Software Quality Assurance*, Undergraduate
Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_7

7.1 Formal Inspections

Software inspections are well-established. Michael Fagan originally developed the notion in 1976 [48]. He proposed what are now known as "formal" inspections, which were geared towards structured organisations. The idea was that inspections would take place in structured meetings, where different stakeholders would prepare for the meeting by carefully reading through the document in question (e.g. the source code), and would then discuss this during the meeting.

This then led to a very substantial amount of research throughout the 80s and 90s, which focussed on the specifics of reviews and inspection meetings, with a view to maximising their effectiveness. This culminated in various structured reading techniques, such as Active Design Reviews [108] and Perspective Based Reading [16], which were underpinned by extensive empirical studies indicating their efficacy.

Nevertheless, inspections never became particularly widespread in their formal incarnation. They were largely intended for structured work-flows such as the Waterfall model (see section 3.2.1), and failed to adapt to emergent agile processes. The need for synchronicity (for inspectors to meet and discuss a piece of code at the same time) was at odds with the tendency towards distributed software development teams, where different groups of individuals would work on different pieces of code at different times. There also emerged several studies that showed that they did not (in their meeting-centric form) tend to improve defect detection efforts, despite their high costs [75].

This eventually led to the development of more light-weight inspection techniques that could more easily be integrated into routine software development practices. For example, Pair Programming (see section 7.3.2.2), which became an integral part of Extreme Programming, is an apt example of a peer-review technique that does not revolve around costly meetings.

7.2 Modern Code Reviews - Reviewing Code During Development

In recent years, the term 'Modern Code Review'[12] (MCR) has emerged to refer to this new family of inspections. MCR does not rely on face-to-face meetings. On the contrary, it commonly presumes that developers are distributed, and are operating asynchronously. To enable this, MCR operates on tools and processes that are based upon the use of a version repository (as introduced in Chapter 3).

The specifics of MCR vary extensively from one development context to another, and often depend upon the nature of the development process (e.g. whether or not it is agile), the makeup of the development teams, and the work-flow required by the version-control system or the associated code review tools that are used. This section will set out some of the most common MCR features and practices.

7.2.1 Tool-Driven Code Review

One major factor that distinguishes MCR from traditional inspections is the fact that MCR focusses on what are usually small, manageable updates to an underlying system (such as the patches that form a commit to a version repository), as opposed to entire files or modules. To enable reviews, reviewing tools such as Gerrit[1] can easily be integrated with version repositories to act as 'gate keepers' to commits. Aside from Gerrit (which was developed by Google for the development of its Android operating system), other notable examples of reviewing tools include Microsoft's CodeFlow tool [12], and Facebook's Phabricator tool[2].

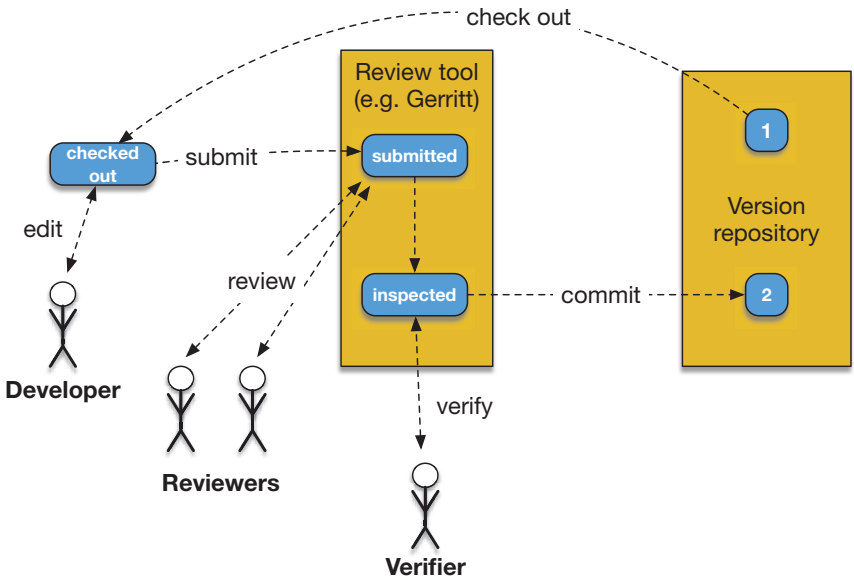

Fig. 7.1 Example of a typical MCR-flow, using a code reviewing tool.

A typical work-flow is illustrated in Figure 7.1. A developer will check out a version from the repository and make their change. This change is then submitted for review (this is often automatically initiated when the developer tries to push their change to the central version repository). Before the commit can be finalised, the patch submitted by the reviewer is subject to review by a pool of reviewers. If the patch is accepted by the reviewers, it is then passed to someone who 'verifies' it - e.g. by testing it, and finally commits the reviewed, verified code fragment to the version repository.

[1] `https://gerrit.googlesource.com/gerrit`

[2] `http://phabricator.org/`

Although MCR is commonly associated with code reviewing tools, these are not essential. Most versioning systems (see Section 4.2.2) provide plenty of mechanisms that can be employed for code review without the use of explicit tools. For example, one convention is to use branching and merging; developers use dedicated branches to make their changes, but these changes are only merged to the 'trunk' by a dedicated user, and only after a review. Reviewers can use plenty of tools (such as repository logs) to inspect the various changes that were made to the code under review. The challenge, in the absence of appropriate code review tools, is to ensure that developers ultimately bother to go through the various reviewing steps, which can easily be neglected if they are not enforced.

7.2.2 Pull-Based Development

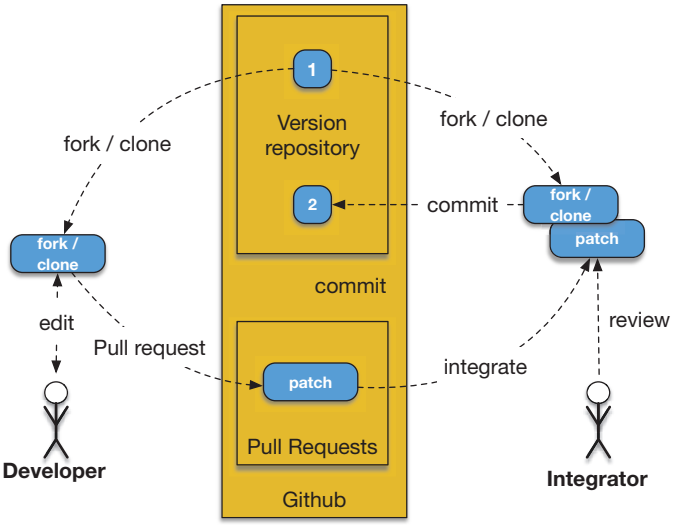

Fig. 7.2 Pull-Based Development

One 'tool-less' development approach that enforces MCR is known as 'Pull-Based Development' [62]. The approach revolves around a mechanism known as a 'Pull Request'. The flow is illustrated in Figure 7.2. A developer (who does not have the privileges to push their changes directly to the central repository) takes a clone of the code from the repository. They make their changes, which are typically small (of the order of a few dozen lines of code [62]). They then submit their proposed changes to the repository as a 'pull request'. A developer with appropriate privileges (often referred to as an 'integrator') then examines their proposed changes, integrates them with their own clone of the source code, and makes the commit. This

integration step can often be automated (if there have not been too many intervening changes).

Pull requests are becoming increasingly popular in open source systems because they enable *anybody* to at least attempt to make contributions to any open source system (that supports pull requests). Reviews and discussions of pull requests do not necessarily have to be restricted to a small band of privileged developers, but can be opened up to the wider community of developers. This can, at least in principle, open up software development to greater scrutiny, and a higher level of shared decision making.

It is instructive to skim some of the open-source projects on GitHub to witness how extensively MCR is embedded within large software development projects. We can take, for example, a relatively extensive change proposed as a pull request to the Erlang OTP libraries: `https://github.com/erlang/otp/pull/960`. The pull request includes changes to 52 files within the libraries, and is accompanied by a discussion of 121 (largely substantial and constructive) comments, eventually culminating in the acceptance of the proposed feature.

7.2.3 The Impact of MCR on Software Development and Quality

MCR has, as a practice, emerged out of necessity. Different organisations developed their own internal reviewing approaches. As a result, there is no single canonical definition of the "rules of MCR"; the term covers a relatively broad range of practices. This makes it difficult to analyse or discuss how it is used. Nevertheless, there have been some recent efforts to study the trends underlying MCR, and these are discussed here.

In their 2013 study, Rigby *et al.*[112] analysed the review practices for thirteen projects, including large open-source projects such as Android, Chromium OS, Apache, and Linux. They found the following:

- **Reviews tend to occur before changes have been committed (as per Figure 7.1), and occur frequently.** At AMD the median time for the completion of a review was 17.5 hours. At Microsoft the median review time for the Bing, SQL, and Office projects was 14.7, 19.8, and 18.9 hours respectively. Office had a median of 4384 reviews per month.
- **Change sizes (and thus reviewer loads) are small.** The median change size observed for most of the active open source and industrial projects was below 100 lines of code. For Android and AMD the average was 44 lines.
- **The number of reviewers that typically take part in a review is 2.** This is in line with findings from the era of formal code reviews that 2 reviewers are optimal for defect detection meetings [135].
- **Review is about more than just detecting defects.** Rigby and Bird noticed that the goal of reviews was rarely to record defects – in fact, current reviewing platforms rarely offer such a feature. Instead, the nature of discussions tends to be

'perfective'; reviewers and code authors share the goal of refining and improving a submitted piece of code to a point where it can be merged back into the main code base.

In a similar study, focussed specifically on the use of MCR at Microsoft (with the CodeFlow tool), Bacchelli and Bird[12] interviewed developers who used MCR and analysed their respective changes. Their interview responses emphasised the benefits aside from finding defects that can be brought about by code reviews. These include:

- **Code improvement.** One of the managers at Microsoft noted that the *"discipline of explaining your code to your peers drives a higher standard of coding. I think this process is even more important that the result."*. This sentiment echoes one of the key drivers behind pair programming.
- **Alternative solutions.** 17% of developers put this as their first motivation - the rationale being that different team members could have better ideas as to how to implement a particular solution.
- **Knowledge transfer.** Code reviews can be a key driver to disseminating knowledge about the system within a team, and especially to new members. Being privy to a code review encourages participants to familiarise themselves with aspects of the system that might be outside of their area of expertise.
- **Team awareness.** Code reviews serve as a useful basis for notifying the rest of the team about what is happening in different parts of the system.

Bacchelli and Bird followed up their interviews with a review of the various code changes that arose from the reviews. They found that the majority of code changes were concerned with code improvement. The second largest proportion of changes was concerned with defect finding.

Their findings were supported by a subsequent study by Beller *et al.*[18]. Their studies of two large open source systems also illustrated that 75% of code commits that resulted from code reviews were related to maintenance and improvement, whereas only 25% are related to functionality. These statistics nicely illustrate the complementary nature of software testing and inspection. Whereas testing is primarily concerned with functional software correctness, inspections can support the assessment of a much broader range of concerns.

7.3 Code Reviewing Techniques

Code quality is one of the key factors in whether a software system as a whole is *maintainable* or not. Poor code quality hinders the ability of developers to understand software. This in turn raises the likelihood of accidentally introducing faults, and can lead to further degradation as the source code evolves.

These 'patterns' of poor code quality are often referred to as 'code smells'. The term originated from Martin Fowler [54], to describe the sorts of things that developers should routinely be keeping an eye out for when trying to improve their code.

These can range from highly localised problems (e.g. function or variable names that are non-descriptive, or the existence of a data class in an object-oriented system), to problems that span the whole system, such as the existence of duplicate code.

In this section the focus primarily on the task of detecting such smells. The task of fixing them once they have been detected is somewhat beyond the scope of this book. However, good starting points (at least when dealing with object-oriented software) would be Fowler's refactoring book (which was geared towards addressing code smells) and, for some larger architectural problems, Demeyer *et al.*'s book on software reengineering [41].

There are two families of approaches to inspecting code quality: (1) automated code analysis techniques that can pick out problematic patterns within the source code, and (2) reviews by the developer themselves to pick out problems. In this section we provide a brief overview of the key approaches, and cover what they look for.

7.3.1 Tool-Driven Code Review

Automated code reviews are ultimately driven by source code analysis. These tools go back to the Lint tool for C [76], which first emerged in 1977. Since then, code checkers have become heavily used, and are often routinely built in to IDEs such as Eclipse and IntelliJ. For example, if a variable is declared but never used, or used without being initialised this is commonly flagged up without the developer even having to explicitly invoke a tool.

The nature of the code problems that are (or can be) identified varies from one tool to another, and depends to an extent on the nature of the underlying language. For example, Strongly typed languages such as Java provide more information through which to discern potential problems than dynamically typed languages. Automated tools are especially good at identifying problems that are relatively localised (e.g. are localised to a single method or class). For tasks that encompass larger parts of a system.

Within Java, which is more relatively amenable to static analysis, perhaps the most popular tool (which is not already built in to an IDE) is Findbugs[3]. This tool analyses the byte-code of the compiled program and checks for over 400 problems, from unwritten object fields to non-terminating loops.

One problem that besets tools such as Findbugs is their propensity to overload the developer with warnings. Not every "bug" that a tool checks for is a genuine bug; they can often be false alarms. Nevertheless, when faced with large numbers of fault reports, the developer invariably has to spend a large amount of their time reading through them to separate the genuine problems from the false ones.

[3] `http://findbugs.sourceforge.net/`

Exercise: *Open Eclipse or IntelliJ with the Findbugs plugin, and run it on a software system (preferably your own!).*

7.3.2 Developer-driven Code Reviews

Some properties of code quality are very difficult, if not impossible, to automatically assess. Questions of good design, for example, are often subjective. Some aspects of good design are measurable by algorithms (as we will see in Chapter 8), but others (such as whether the files and modules are intuitively aligned with the problem domain) require a degree of insight that requires a degree of human reasoning.

Developer-driven code review is usually a two-step process [124]. The developer first has to first *understand* the program so that they can form an opinion of its implementation and design. Only then can they appraise the program in properly.

When it comes to the appraisal itself, the specific goals depend on several factors. There is the paradigm of the language that was used to write the system (Object-oriented systems are understood according to a different mental model from functional or imperative programs, for example). There are also the priorities – for example, developers that are focussed on security might look for different potential problems than developers who are focussed on issues such as maintainability.

7.3.2.1 Understanding the Code

How can a developer (or a group of developers) aggregate the knowledge knowledge about a code-base, and use it to form a coherent mental model? It is rarely tractable to try to read through all of the source code to systematically form a complete understanding of code behaviour (imagine being confronted with a truly large-scale system, comprising thousands of files, and millions of lines of code).

The question of code comprehension is a challenging one. Figuring out how to best understand a code-base ultimately requires a sound understanding of human psychology. Although a large body of research has hypothesised how humans understand code, confirming these hypotheses is difficult. Experiments rely on large numbers of human participants, and can be difficult to design. They have to convincingly address all manner of confounding factors (such as the prior experience of participants or the paradigm of the programming language), and also have to somehow measure the knowledge gained during a comprehension exercise, which can be difficult in its own right.

In practice, software comprehension amounts to a mixture of (a) taking stock of what you already know about a program, and (b) exploring the program to find out more about what you do not already know (but need to know) [129]. Elements of a program that you already know about can be indicated by ‘beacons’ – method names

that are sufficiently descriptive to render their functionality obvious, or classes that contain sufficiently detailed comments, or obey design patterns [57] (recall Section 4.2.1) with which you're already familiar.

This is why good programming practice is so important. Sensible names and succinct comments remove the need for other developers to invest precious time into comprehension, and enable them to spend it more productively. This can also be fostered by carrying out light-weight changes to the source code of a program throughout its development [41][4]; if a developer learns something about a program whilst trying to understand it, this understanding can be embedded into the source code as a comment to assist other developers further down the line.

7.3.2.2 Pair Programming

Pair programming is an alternative approach to code review that takes place *during* coding (as opposed to afterwards). The idea is that software development is carried out by pairs of developers, so that each developer is in effect continuously scrutinising the others' work. Pair programming tends to be promoted with the following arguments:

- Code quality is higher, because it is continuously being reviewed.
- Good practice is shared between the programmers.
- There is greater shared knowledge of the code base, which leads to better solutions.

Most of the claims that advocate the use of pair programming have been the subject of empirical studies. In 2009, Hannay *et al.* helpfully collated them in a meta-analysis [68] of 18 relevant articles. This analysis indicated that: (1) pair-programming is faster than solo-programming when the complexity of the task is low, and (2) pair-programming produces results that are of a higher quality than solo programming when the tasks are complex. On the down-side, the higher-quality for complex task comes at a significant cost, requiring much more effort, whilst the quicker completion time for lower-complexity tasks comes at a cost of significantly lower quality

The ability to pair program makes some obvious assumptions about the development context, which often might not apply. For example, it assumes that the development team is collocated, and that the team is sufficiently large. In recent years, the enhanced role of collaborative software development environments and distributed version repositories has perhaps diminished the prevalence of pair programming[5].

[4] See the 'Tie Code and Questions' pattern.

[5] I am not aware of any data on the prevalence of pair-programming, so this is based on intuition, not evidence.

7.4 Safety Arguments and Inspections of Safety Requirements

For the majority of routine software development projects, MCR alone, coupled with some testing, can suffice to provide a sufficient confidence in software quality. For certain systems, however, more evidence is required, which cannot be obtained during routine software development. In the domain of safety-critical systems, for example, software has to be certified before it can be deployed, and certification can often only be achieved by examining a single, fixed version of the system (not one that is continuously in flux).

A profusion of software safety certification standards exist, which are often tailored for particular domains. For civilian aircraft software, there is DO178-B/C[1], which we have already encountered. There is ISO26262 [4] for automotive software, IEC 60880 [3] for software in the nuclear domain, etc. These various standards all share the commonality that they place an onus on the organisation that is responsible for developing the software to collect evidence that demonstrates that the various safety requirements have been met.

Establishing that a software product meets a particular set of safety standards is challenging for the following reasons [100]:

1. The document describing the standards can run into hundreds of pages of natural language text that is subject to interpretation.
2. The evidence that is required to establish a particular safety requirement can be difficult to collect and can, depending on the requirement, at best corroborate but not prove that a requirement has been fulfilled.

The amount and diversity of evidence that can be required can be truly bewildering. In their taxonomy of evidence types, having studied over 200 papers on safety certification, Nair *et al.* present 49 basic evidence types. Although their complete taxonomy is too large to present here, we can look at the high-level categories in their taxonomy, which are shown in Figure 7.3. The figure is instructive because it shows just how far-reaching a safety certification must be.

There are several approaches to collecting and presenting safety evidence. The top two approaches (at least according to their prevalence in research publications [100]) are checklists and what Nair *et al.* refer to as 'qualitative argumentation'. We examine both of these techniques in more detail below.

7.4.1 Checklists

Checklists set out requirements against which a software system should be inspected as a 'to-do' list of guided questions that need to be answered. Ultimately, the rationale of a checklist is to make the inspection repeatable – to ensure that, regardless of the expertise of the individual inspector, they would be prompted to highlight any potential faults.

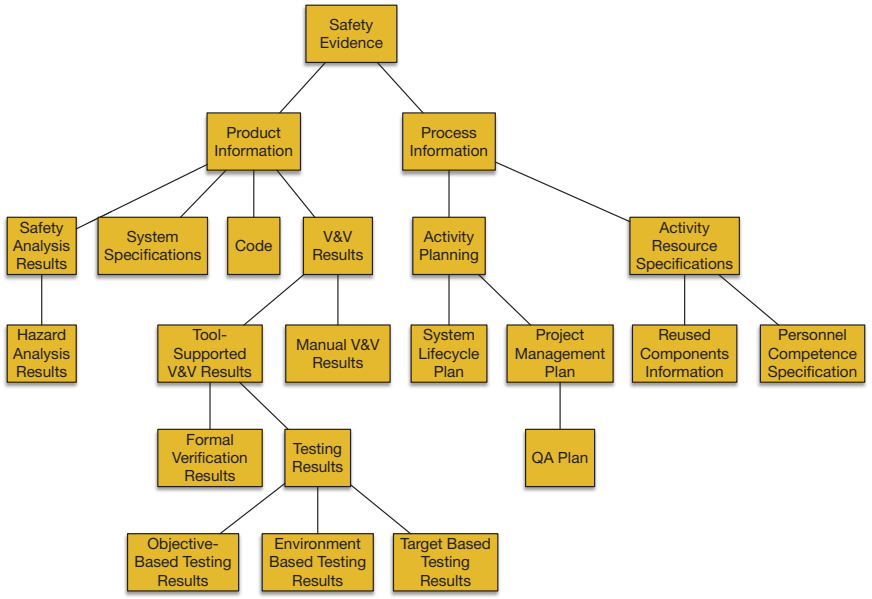

Fig. 7.3 Categories from Nair *et al.*'s safety evidence taxonomy [100].

Exercise: *Checklists are intended to support repeatability. Think back to Chapter 3 - Process-Based Quality Assurance. Which aspects of process-based quality assurance does this remind you of.*

Constructing an effective checklist is an art. If the list is too focussed and granular, the inspector can become too focussed on following the checklist objectives, but can miss out on specific faults that are not in the list. If it is too abstract, the inspector is left to their own devices and the performance of the checklist will vary more from one inspector to the other. The key is to (1) cover as many areas that are of importance from a quality perspective, (2) to ensure that the inspector does a thorough job – goes to the effort to fully understand the item under inspection, and (3) to not require a prohibitive amount of time and effort.

Two potential approaches by which to strike this balance are as follows[6]:

1. Write the questions from the perspectives of different stakeholders[16]. Ideally, task different individuals to adopt the perspectives of, for example, the designer, the tester, the coder, etc., and have them write a set of questions. The rationale is that these will force the inspector to consider the broadest possible range of concerns.

[6] These are techniques that are inspired by two 'formal inspection' approaches that appeared in the 90s – we will not go into their associated methodological details in this book, but their essential lessons can be readily applied to simple checklists.

2. Write the questions in such a way that the inspector is forced to engage with the document under inspection (i.e. by completing some task)[108]. E.g. instead of asking a relatively subjective question that is easy to answer in a hurry *"Are the test sets sufficient?"*, one could write *"Is the branch coverage achieved by all tests greater than 80%?"* – thus forcing the inspector to execute the tests and to monitor their coverage.

7.4.2 Safety Argumentation and the Goal Structure Notation

Although checklists can offer valuable guidance and are easy to use, they also have their weaknesses. If some aspect of safety is not properly captured by way of the questions in a checklist, it is unlikely to be inspected. They only illustrate *what* has been inspected (what has been ticked-off), but not *why*. There is no guidance as to how specific items might help to substantiate a higher-level safety property.

Argumentation-based approaches add an additional dimension to the traditional checklists. The underlying rationale with argumentation-based approaches is that, instead of checking a box, the inspector has to build an explicit argument that explicitly links the evidence to the conclusion that a given safety requirement has been satisfied. For example, instead of merely ticking a box to indicate that test coverage was satisfactory, a safety argument might indicate how the coverage data had been collected, and what coverage criteria had been used, etc.

Safety case argumentation can come in various forms [100]. They can be simply consist of structured natural language arguments, or be graphical in nature. Goal Structure Notation (GSN)[82] is a popular example of such a notation that sets out an argument in a hierarchical form – high level safety goals are broken down into sub-goals and arguments.

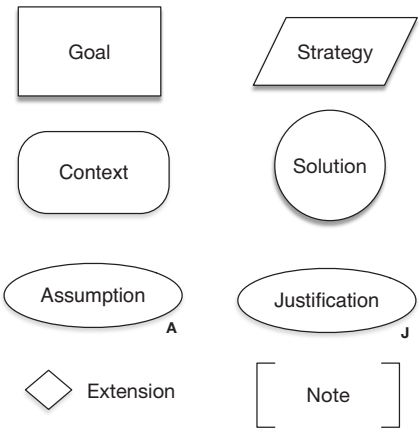

Fig. 7.4 Key GSN Symbols

The main GSN symbols are shown in Figure 7.4. Their purposes are briefly summarised below:

- **Goal:** A goal represents the safety property that we seek to build an argument around.
- **Strategy:** A description delineating *how* a goal is established.
- **Context:** Supporting information required to understand what is meant by a goal.
- **Solution:** A piece of evidence that can contribute towards establishing a goal.
- **Assumption:** Conditions under which a piece of evidence can validly be held to establish a goal.
- **Justification:** Reasons justifying the use of a piece of evidence.
- **Extension:** An indicator that further argumentation and support is required to support a particular (sub-)goal.
- **Note:** Additional textual annotation.

When put together, arguments can be constructed, forming hierarchical structures[7], where high-level safety goals can be broken down into sub-goals, and linked to the sources of evidence that would be required to establish them.

A nice, comprehensive example of a safety case is the safety case that was published in 2004 for the Panavia Tornado combat aircraft [128] (which can be downloaded in its entirety from the Gov.UK website). This was constructed, in part, to comply with legal obligations stating, for example, that it was necessary to show that the risk to its stakeholders was ALARP (As Low as Reasonably Practicable).

A small example GSN fragment is shown in Figure 7.5. This shows the high-level goal at the top, which is broken down into sub-goals further down. Where relevant, goals are linked to contexts, to explain what they mean, of justifications, to contribute to the argument of why they exist. At the lowest level, goals are tied to solutions – techniques that are to be used ot fulfil the goals.

7.5 Key Points

- **Software inspections were formally proposed by Fagan in 1976.** These 'Formal Inspections' remain widespread in certain sectors, but tend to be considered 'heavyweight', involving a lot of time and effort from the development team.
- **In recent years, the notion of a 'Modern Code Review' has become widespread.** These are characterised by being more lightweight, tool-based, and suited to distributed, asynchronous development. Reviews are often conducted on a per-commit basis to a version repository. These can be facilitated by tools such as Gerritt. An alternative is to adopt pull-based development, where proposed changes to the code base are sent to the development team in the form of a pull-request.

[7] Specifically, these are directed acyclic graphs.

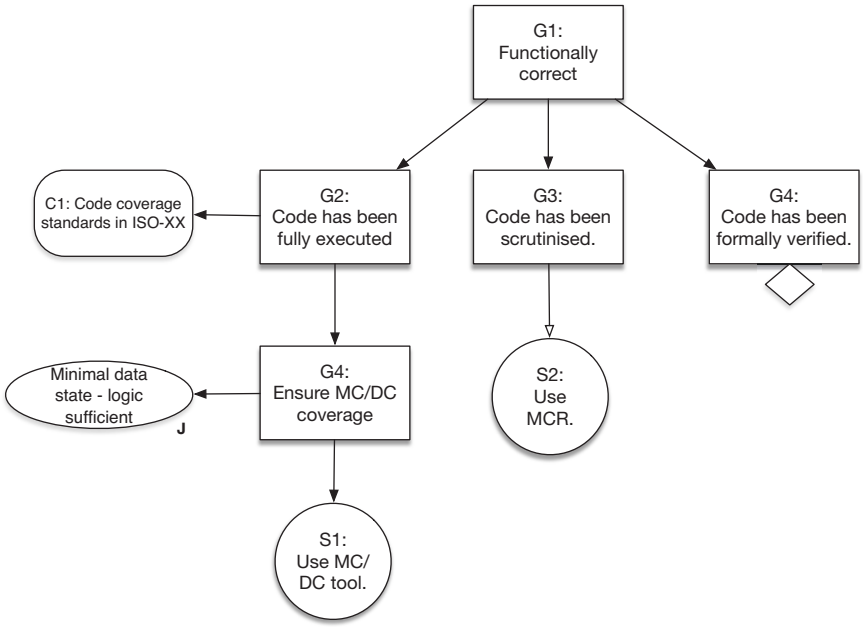

Fig. 7.5 GSN example.

- **A series of empirical studies have emerged to show several positive impacts of code reviews.** A series of empirical studies have emerged to show several positive impacts of code reviews. Results indicate that the code quality increases, along with team awareness of the code. They do not merely serve the narrow purpose of detecting bugs, but also spread knowledge about the system within the team, and lead to code improvement.
- **Code reviews necessarily start with a phase of 'program comprehension' – understanding the structure and functionality of the source code.** The activity is facilitated when there are areas of the source code ('beacons') that can be used as a basis for orientation.
- **Pair-programming is a means by which to review code *during* coding, as opposed to afterwards.** It was popularised with the emergence of agile software development. Instead of development being undertaken by individuals, pair programming sees pairs of developers sitting together to write code. Whilst one developer does the typing, the other can assist and can help to guard against errors.
- **Safety inspections constitute a special type of software review.** This tends to be carried out by independent teams of inspectors, who seek to inspect a system with respect to particular safety properties. These safety properties are often specified in the form of check-lists or graphically in the form of Goal-Structure Notation.

Chapter 8
Measurement

Chapter 3 showed us how software is the result of a process, and showed how the quality of a product and the process used to create it are intricately linked. We have seen how successful product development processes tend to be highly iterative and feedback-driven (remember Shewhart's Plan-Do-Check-Act cycle). And we have seen how, with the emergence of Agile software development, these principles have gradually filtered through to conventional software development. Ultimately, successful software project management depends upon the ability to continuously keep track of progress and quality. This is perhaps best captured by the following quote[1]:

> "You cannot control what you cannot measure." – Tom DeMarco, 1982

In software development, this is commonly facilitated by various measurements of the system and the surrounding process – commonly referred to as *Metrics*. These examine factors such as the progress made on a particular task, the time and resources consumed, alongside product-specific metrics such as the complexity (e.g. number of lines of code), or the extent to which it has been tested (test coverage).

Software measurement is fraught with difficulties. There are hundreds of possible metrics that can be applied to a software system or process. However, these rarely provide an accurate, complete picture with respect to what the developer (or some other stakeholder) wants to know. Accordingly, metrics often have to be used an interpreted with great care, with a full understanding of their limits and potential threats to validity.

The use of metrics in the Toyota Unintended Acceleration trials

Metrics can provide valuable insights into the state of a software system. One useful example arises when we consider the problems surrounding the software for the Toyota engine controller unit, which was suspected of being responsible for the Unintended Acceleration faults that had apparently led to numerous deaths (see Section

[1] Tom DeMarco was a leading figure in structured software development in the 1970s.

N. Walkinshaw, *Software Quality Assurance*, Undergraduate
Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_8

2.1). In the US, this led to several law suits in 2013-14, amounting to over one billion dollars. The problem that confronted experts who were called in to support the families of the victims was that they were not given access to the source code. All that they could rely upon was information about the software that was already in the public domain, by referring to previous trials, and an openly available report produced by NASA [85] who had been initially tasked with investigating the software (they had failed to find the specific cause of the bug, but had flagged up numerous instances of bad practice).

One notable source of information was a set of metrics pertaining to the Electronic Throttle Control System (ETCS), which was suspected of causing these unintended accelerations. In particular, although they could not look at the code itself, they could look at metrics that quantified its complexity (the Cyclomatic Complexity measure, which is something we will encounter in detail later on in this chapter). In short - Cyclomatic complexity tries to gauge the number of possible paths through the source code. An "acceptable" value that indicates that the function is reasonably comprehensible lies somewhere betwee 5 and 15. Over 50, a function is considered "untestable" – it is impossible to account for every possible path through the function. In the Toyota ETCS code, there were 67 functions with a complexity > 50. One function – the "throttle angle" function – had a complexity of 146, was over 1300 lines long and had no unit-test plan!

As if the complexity metrics were not damning enough, there was another set of metrics that furthermore painted a picture of a software system that was impossibly complex to manage. The number of global variables (variables that can be written and read by and from anywhere in the source code) should ideally be zero, though in practice it can be quite common to have a moderate number of 'constant' values and configuration variables. However, in the ETCS case, there were between 9,273 and 11,528 global variables (including variables that 'commanded the throttle angle' and reported engine speed).

Ultimately, it was this evidence (along with plenty of other examples of bad practice highlighted by Koopman[86]) that led the jury to decide that ECTS defects had caused a death.

8.1 Measurement Basics

Although metrics can be useful (indeed vital) for software development, this is only the case if they are properly understood. Metrics that are misunderstood or misinterpreted can become highly misleading, and can in turn lead to poor decision making when it comes to managing a project. In order to prevent such mistakes from being made, it is necessary to acquaint ourselves with some of the fundamental notions that underpin measurement. Many of the definitions in this section closely follow those by Fenton in his overview of software measurement [50].

There are many theoretical frameworks within which to understand measurement. In this section we adopt the one that is most frequently associated with soft-

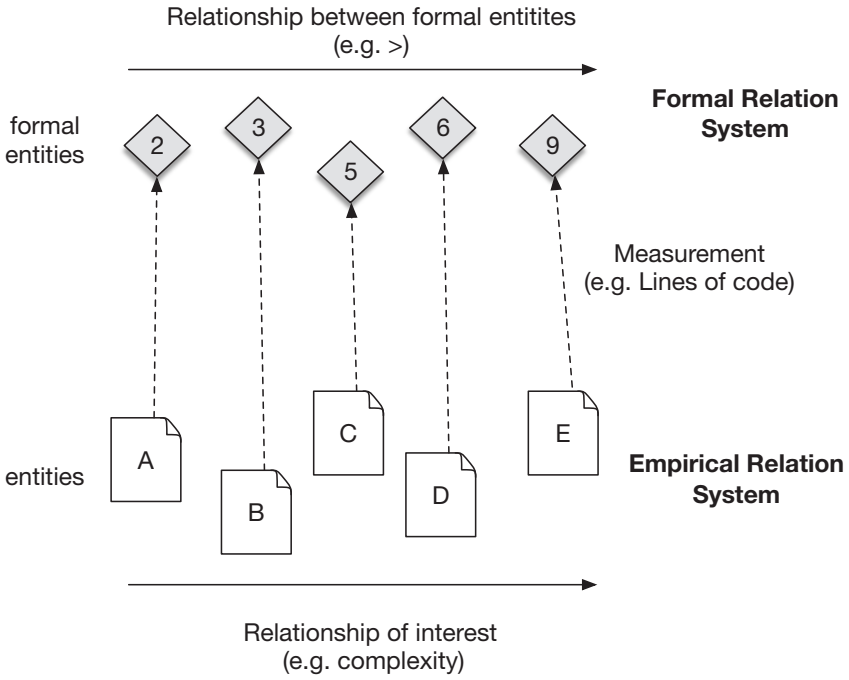

Fig. 8.1 Illustration of the framework for the Representational Theory of Measurement.

ware measurement – the Representational Theory of Measurement. The essential notions involved in the framework are shown in Figure 8.1. Empirical entities represent the subjects of interest. There is some qualitative relationship between them that needs to be formalised by a measurement. The measurement provides a means by which to associate quantitative values with the entities, so that they can be ordered and compared in a more formal way. The details of this framework are provided below.

Entities.

In abstract terms, the subject that we wish to measure is referred to as an 'entity'. In our case, this might be a software class or an entire software development project.

Empirical Relation System.

An empirical[2] relation system describes the essential aspects of the real system that we wish to measure. Specifically, it tells us what the objects (entities) are that are of interest to us, what the relationships between these entities are that are of importance, and different ways in which we can potentially combine these entities (which will be of value to us to ensure that our measurements are ultimately consistent). In formal terms, we define an empirical relations system as $E = (A, R)$. The components of E are elaborated as follows:

- A is the set of entities that we wish to measure and compare to each other. For example, a set of classes in an object-oriented system.
- R is the set of empirical relations between attributes of the entities that we are interested in. For example, one relation comparing the complexity of different entities might be "is more complex than". Another relation comparing size might be "is larger than", etc.

It is worth noting that, in the empirical relation system, all of the relations are *qualitative*, not *quantitative*. They are intuitive relations that we wish to establish, but for which it is not yet necessarily clear how we would assign numbers to the entities by which to measure them.

Formal Relation System

A formal relation system provides a 'formal' counterpart to the empirical relation system described above. Whereas the empirical relation system is grounded in entities and qualitative relations, the formal relation system is grounded in formal objects – primarily numbers, and quantitative relationships. In formal terms, we define a formal relations system as $F = (B, S)$. The components of F are elaborated as follows:

- B is a set of formal objects - such as numbers, sets, or vectors.
- S is a set of relations that can hold between elements in B, such as $\leq, <, >, \ldots$

Measurement

Measurement is the process of attributing a number or a symbol to an attribute of an entity. In formal terms, M is a measure with respect to an empirical relation system $E = (A, \{R_1, \ldots, R_n\})$ is valid when:

1. $M : A \rightarrow B$
   There is a mapping from every entity to a number or a symbol.

[2] The term 'empirical' refers to the use of evidence to support a proposition.

2. $\forall(a,b) \in R, \exists(M(a),M(b)) \in S$.
   There is an equivalence between empirical relations and formal relations. This second condition is referred to as the *representation condition*.

Scales and Admissible Transformations

Depending on the purpose of a measurement, it can lead to different types of values. We may, for example, seek to classify source code files according to the language in which they were written. We may want to rate developers according to their involvement in a project (e.g. according to five levels from "not involved at all" to "a key developer"). We may want to measure the size of a source code file in terms of the number of lines of code within it.

These three types of measurement produce values (formal objects) that lie on different scales (Nominal, Ordinal and Ratio scales respectively). These scales can be characterised by referring to the relationships that they impose on the formal objects (i.e. the results of any measurements). This characterisation is founded upon the notion of *admissible transformations*.

An admissible transformation is a transformation that can be applied to the results of a measurement, whilst maintaining its validity. In other words, a transformation is admissible if it retains the relationship between the empirical and formal relation systems.

Formally, let us suppose that we have some measure M, some set of entities E, a set of formal entities B and F – a transformation (mapping) $M(E)$ to some numeral N, so $F : M(E) \rightarrow N$. T is an admissible transformation if and only if $F(M(E))$ is a valid measure.

The following scales are some of the most common[3]. The structure of the descriptions is based upon the structure used by Jørgensen in his paper on quality measurement [77]:

- **Nominal:** As shown in Figure 8.2, a measurement is on a nominal scale if the admissible transformations retain equality between the measured entities; a pair of entities either belong ot the same category, or they do not. For example, a metric that divides files into those that have been inspected and those that have not been inspected would be on a nominal scale.
- **Ordinal:** As shown in Figure 8.3, admissible transformations are valid as long as they retain any equality and order amongst the measured entities. For example, a metric that rates the quality of a class according to "very poor quality", "poor quality", "good quality" and "excellent quality" would be on an ordinal scale.
- **Ratio:** As shown in Figure 8.4, admissible transformations are of the form $T(x) = ax, a > 0$. The empirical relations possible are related to equality, order, difference, and relative difference. Ratio scales are distinguished by the fact that

[3] There are other scales that are common outside of Software Engineering, such as the Interval scale, but we do not consider this here because it is rarely (if ever) used in the context of software development.

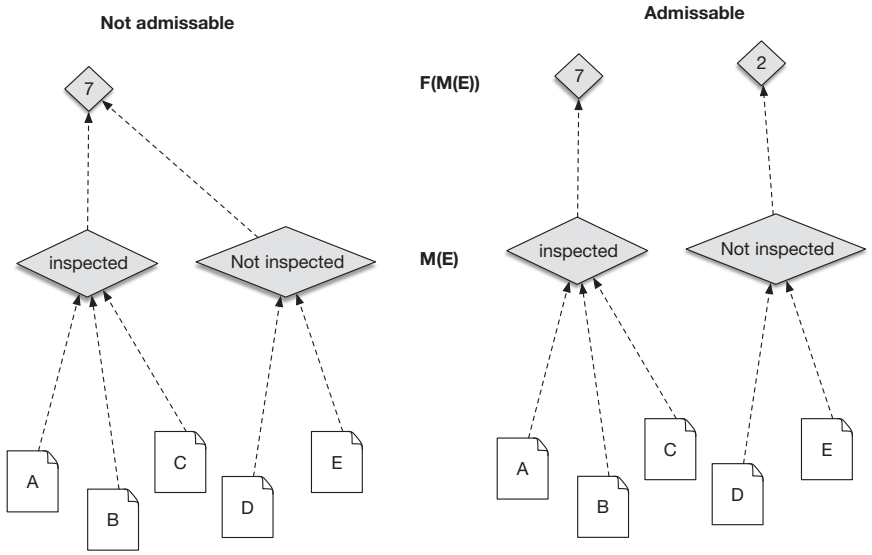

Fig. 8.2 Inadmissible and admissible transformations for the nominal scale. Any transformation is valid as long as it retains the equality relation with respect to $M(E)$.

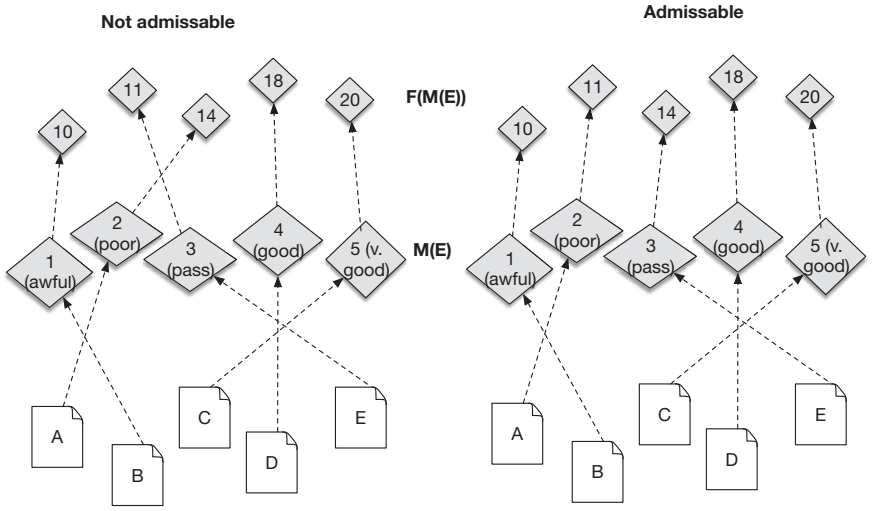

Fig. 8.3 Inadmissible and admissible transformations for the ordinal scale. Any transformation is valid as long as it retains the equality relation and the ordering of the entities with respect to $M(E)$.

they are based upon a definitive notion of 'zero'. For example, the use of Lines of Code (see below) to measure the length of a program would be on a ratio scale, as would the use of the number of person-hours to measure the resources spent on a project (again, see below).

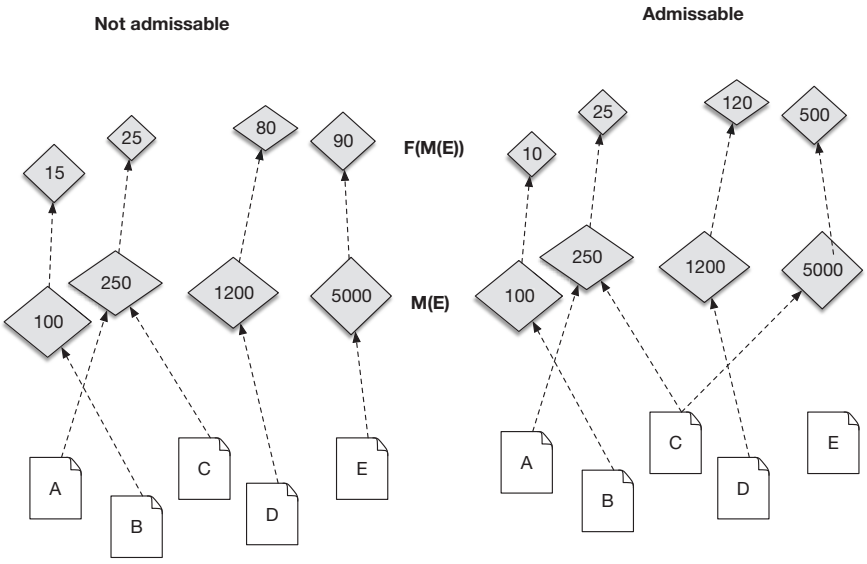

Fig. 8.4 Inadmissible and admissible transformations for the ordinal scale. Any transformation is valid as long as it retains not only any equality and order, but also the ratio between elements in $M(E)$.

8.2 Metrics

A software metric is, in the loosest possible terms, a measurement where the entity that is the subject of the measurement is related to a software system. Metrics can be concerned with the actual structure of the software system itself – for example the complexity of individual classes, or the modularity of the system. Metrics can also be used to measure qualities of the software development process – the amount of time and effort required, the ability throughout the process to prevent bugs, or the prevalence of bugs in different parts of the system. In this section we provide a relatively brief overview of these different types of metrics, picking out a couple of the most popular ones (at least from a quality assurance perspective).

In the introduction to this chapter, we discussed the Toyota case study, where code metrics were used (amongst other data) to argue that the software in question

was sub-standard. These metrics are calculated by analysing the underlying software (usually the source code or bytecode). Metrics can focus on various levels of a system – from individual functions to classes or modules (depending on the programming paradigm), all the way up to individual metrics that apply to the system in its entirety. The rest of this section provides an overview of some of the most popular metrics.

8.2.1 Size and Complexity

A multitude (perhaps even the majority) of metrics seek to quantify how "large" or "complex" a software system is. Such metrics have various applications; if derived from the source code, they can be used to estimate how much effort has been invested in its development. If derived from the design, they can be used to estimate how much effort might be required for future implementation (in a similar vein to what is achieved by Planning Poker in agile contexts - as described in section 5.2.4). This subsection will provide a brief overview of some of the key metrics, spanning both code and design metrics.

8.2.1.1 Lines of Code

The number of Lines of Code (LOC) is frequently used to gauge the size and complexity of a software system. Its main strength is that it is easy to compute – it requires no ability to parse source code.

There are plenty of obvious objections to the use of LOC. Firstly, it is not necessarly clear what a "line" is. Is a comment a line? What about empty lines in the source code that have been added to facilitate readability? What if every file in the system is accompanied by a lengthy copyright proforma? To address these queries, various specific notions of LOC have emerged, including NCLOC (Non-Commented Lines of Code), SLOC (Source Lines of Code — referring to executable statements), etc.

Exercise: *Open up the Bash Shell on a Linux or Mac computer (or use Cygwin on Windows). Locate a directory containing some source code files. Figure out how to use the built-in Bash commands to count all of the lines in all of the files in the source directory (including sub-directories).*

Despite being widely derided, the LOC measurement (along with its various variants) remains the most widely used metric. As an *approximate* measure of size, it is often as good a measure as any. In other words, a system of the order of hundreds of thousands of lines of code is liable to be "larger" and "more complex" than a system of the order hundreds of lines of code.

8.2.1.2 Measuring Complexity with McCabe's Cyclomatic Metric

McCabe's Cyclomstic Complexity [93] seeks to provide a more insightful measurement than LOC by accounting for branching structure within the source code. The aim of Cyclomatic Complexity is to count the number of "linearly independent paths" through the code (every path that arises from a unique combination of branch-decisions). The metric can either be computed for individual functions / methods, or for entire files or modules.

The metric is often computed by characterising the system in question as a *Control Flow Graph* – a directed graph where individual statements are nodes, and were edges between nodes denote the possible flow of control from one statement to the next. Within such a graph, the Cyclomatic Complexity (*CC*) can be computes as $CC = E - N + 2P$, where E is the number of edges, N is the number of nodes, and P is the number of separate procedures or functions within the graph[4].

```
1  void iqsort0(int *a, int n)
2  {
3      int i, j;
4      if (n <= 1)
5          return;
6      for (i = 1, j = 0; i < n; i++)
7          if (a[i] < a[0])
8              swap(++j, i, a);
9      swap(0, j, a);
10     iqsort0(a, j);
11     iqsort0(a+j+1, n-j-1);
12 }
```

Fig. 8.5 Toy implementation of the QuickSort algorithm, from Bentley and McIlroy [19]

Exercise: ***Do not read anything below this Exercise box!*** *Looking at the source code in Figure 8.5, try to draw out the Control Flow Graph by hand. Once you have done this, you can compare what you obtained with Figure 8.6. Using your CFG (or the corrected version), calculate the Cyclomatic Complexity.*

To provide an illustration of how to calculate Cyclomatic Complexity, we draw upon an example of a very simple sorting function, provided by Bentley and McIlroy [19], shown in Figure 8.5. The corresponding Control Flow Graph is shown in Figure 8.6.

Applying the formula for Cyclomatic Complexity to the Control Flow Graph in Figure 8.6, we obtain the following. There are 11 nodes, and 13 edges. We are only

[4] So, if you have a large C file with 11 functions, the graph would be disconnected (you would have multiple connected sets of nodes) and P would be 11.

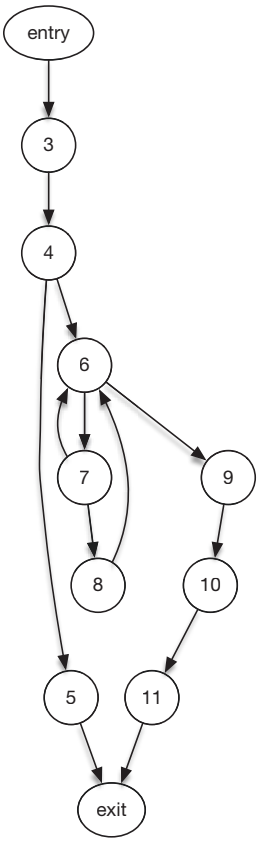

Fig. 8.6 Control Flow Graph for sort function in Figure 8.5, where nodes are numbered according to the line numbers in the code.

dealing with a single procedure, so $P = 1$. This leads to the result: $CC = 13 - 11 + 2 = 4$.

8.2.1.3 Halstead Complexity Metrics

The Halstead complexity metric suite [66] is a set of metrics that are markedly different to LOC and Cyclomatic Complexity. Instead of focussing on mere line counts or logic structures, they focus entirely on the actual textual content of the source code – the names of variables and operators within the source code. They are rarely used as stand-alone metrics, but they are widely implemented; this is primarily because they form a basis for the higher-level "Maintainability Index" metric, which remains popular in the industry, and which we shall come to later on.

The Halstead metrics are essentially founded on the idea that source code text can be split into two categories: *operators* and *operands*. Operators amount to source code tokens that refer to elements of "control" in the source code, such as `if` or `while` instructions, and function declarations. Operands on the other hand correspond to data, such as variable and function names, and variable values.

The metric suite is calculated by dividing the source code in question into these two categories – producing the set of operators and operands. These are then used to produce four values:

- $n1$ is the number of distinct operators.
- $N1$ is the total number of operator occurrences in the code.
- $n2$ is the number of distinct operands.
- $N2$ is the total number of operand occurrences in the code.

From these values, the actual metrics are calculated as follows:

- Program Length $N = N1 + N2$
- Program Vocabulary= $n = n1 + n2$
- Volume = $N * (log_2 n)$
- Difficulty = $\frac{n1}{2} * \frac{N2}{n2}$
- Effort = $Difficulty * Volume$

Exercise: *Pick a (small) source code file, written in your favourite language. Compute the Halstead metrics for it.*

If you attempted the above exercise, one of Halstead's weaknesses will rapidly become clear: It is not always obvious what constitutes an operator, and what constitutes an operand. However, the boundaries between these two concepts are absolutely crucial to the metric values. Whilst there are obvious control and data constructs (such as those listed above), others can be more problematic. What, for example, about syntax such as parentheses or semi-colons? What about text that is commented out? This raises serious questions of *validity*.

Exercise: *In terms of the definition of validity in the Theory of Representational Measurement, why does the fact that some of the metrics are difficult to justify threaten their validity?*

8.2.1.4 Function Points

Measurements of size can transcend source code alone. Often, when reasoning about the size or complexity of a software system, the source code may not yet be available (e.g. when trying to predict the cost of a system that has yet to be developed). There

are (perhaps surprisingly) very few alternatives to source code-based metrics when it comes to providing a measure of software size.

One of the most popular, longstanding measures is the Albrecht Function Point. The measure was introduced by Allan Albrecht in 1979 at IBM [8]. Since then it has become a standard measure of size, incorporated into several ISO standards. Function points seek to capture the complexity of a function from an "external" perspective, irrespective of the underlying logical complexity.

Function points are calculated by categorising every "interaction" between the system and its environment as follows:

1. **External Inputs:** Inputs are provided by a user.
2. **External Outputs:** Outputs are presented externally (e.g. via a GUI to the user).
3. **External Inquiries:** Inputs are solicited from a user (e.g. via dialogue boxes).
4. **Internal files:** Data is stored to files.
5. **External files:** Data is stored via external mechanisms (e.g. data bases).

Each interaction is also graded according to three difficulty levels – Simple, Average, and Difficult. For each combination of category and difficulty, there a "difficulty weighting" (w) is provided (this is a fixed value). The resulting combinations of interaction counts and weightings can be illustrated in a tabular format, as shown in Table 8.1.

Category	Simple	Average	Difficult
1: External inputs	3×**4**	4×**3**	6×**1**
2: External outputs	4×**0**	5×**2**	7×**1**
3: External inquiries	3×**7**	4×**0**	6×**0**
4: Internal files	7×**3**	10×**0**	15×**0**
5: External files	5×**0**	7×**5**	10×**2**

Table 8.1 Example of a set of external interactions, categorised according to difficulty. In each cell, fixed weights are shown in a regular font, and the numbers provided for our imaginary software system are shown in bold.

Once the interactions have been categorised and scored according to difficulty, the *Unadjusted Function point Count (UFC)* can be computed. This is computed by, for each category of interactions, multiplying the number of interactions by the weighting of the given difficulty, and summing all of the values up, then adding all of these summed difficulties together. More formally:

$$UFC = \Sigma_{i=1}^{5}\Sigma_{j=1}^{3} w_{ij} \times x_{ij}$$

where x_{ij} is the number of interactions of type i and complexity j, and w_{ij} is a weighting factor.

Exercise: *Calculate the UFC for the values in Table 8.1.*

Once the UFC is calculated, a further score called the Technical Complexity Factor (TCF) is computed. This is computed by first of all rating 14 'technical factors' on an ordinal scale from 0 to 5 (where 0 means 'no influence' and 5 means 'critically influential'). The factors are shown in Figure 8.7.

1. Requires reliable backup and recovery?
2. Requires data communications?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require input over multiple screens?
8. Are the master files updated online?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple platforms / organizations?
14. Is the application to facilitate ease of use by the user?

Fig. 8.7 Technical Complexity Factors

This enables us to compute the TCF as follows:

$$TCF = 0.65 + 0.01 * \Sigma_{i=1}^{14} F_i$$

where F_i refers to the score given for the technical factor number i above. Finally, the Function Point Count is computed as:

$$FPC = UFC \times TCF$$

There are plenty of potential pitfalls to function points. For one, the weightings that are attributed to different types of interactions need to be estimated, and it is not clear how to do so. This is especially problematic because the wrong choice of weightings (or categorisations of interactions for that matter) can lead to invalid results. As is also quite evident, the process of computing the function point count can be tedious, and has been criticised as being unnecessarily complicated for purposes such as resource estimation [51].

8.2.2 Modularity Metrics

Software systems are easier to understand if their various functionalities are neatly packaged into well-defined areas of the system. For example, if we consider the implementation of a word-processor, it would be easier to understand if the code that is responsible for loading and saving files is not mixed in with the same code

that is responsible for text editing and formatting. This criterion is what is loosely referred to as "modularity".

The issue came to prominence in the late 60s, when the general issue of software quality first came under the spotlight. At the time, computer programs were generally envisaged as flow-charts; a program was conceived as a sequence of instructions telling the computer what should happen when. This was supported by the mainstream languages of the time – COBOL and FORTRAN. They would offer high-level syntax for a multitude of functions, and it was essentially up to the programmer to put these instructions in the right order. To enable complex control constructs, they would offer commands such as `GO TO`, which would enable the programmer to redirect the flow-of control to arbitrary points in the program if they wanted to bring about loops etc.

The problem with this mode of programming was the fact that this could easily lead to very labyrinthine programs, where the flow of control became very difficult to disentangle. Such programs were commonly referred to as 'spaghetti code' (where the various possible flows of control resembled a plate of spaghetti). This rapidly gave rise to programs that were very difficult (and thus very expensive) to maintain, and could easily contain lots of hidden bugs. These issues were brought to wide-spread attention in a now famous essay by Edsger Dijkstra, called "Go To Statement Considered Harmful" [42].

Exercise: *Read Dijkstra's essay.*

Dijkstra's observations led to several developments in programming language design that sought to encourage modular software development. This led to approaches to 'modularisation', pioneered by figures such as David Parnas. In his 1972 essay on the decomposition of software into modules [105], he developed the notion of "information hiding" – that different modules (concerned with their own aspects of functionality) should not need to be aware of each other's decisions and data. Such ideas led to many of the notions (information hiding, interfaces, inheritance, etc.) that form the basis of modern programming languages and paradigms.

8.2.2.1 Coupling and Cohesion

"Coupling" and "Cohesion" are terms that have been around for as long as the notion of modularity itself. They are terms that can be used to discuss the extent to which a given system is "modular". Intuitive definitions are as follows:

- **Coupling** characterises the extent to which two modules 'are related' with each other (see discussion below).
- **Cohesion** pertains to a single module, and characterises the extent to which the data and the functions are interlinked. In a cohesive module, variables and functions will often be strongly interdependent. If this is not the case, it will often be the case that several groupings of variables and functions within a module will

be independent from each other (and could conceivably be split into separate modules).

The notion of a "relation" in coupling requires some more discussion, because it is rarely well defined. In strict, code-analytical terms, two modules can be related if there is a concrete relationship in the source code: one module calls a function in the other module, it reads from / writes to a variable in the other module, or (in an Object-Oriented context) inherits from the other module.

However, there are many other possible ways in which a pair of modules can be related. For example, Beck and Diehl [17] refer to several other, more latent relationships:

- They might have similar relationships to other classes / libraries in the system.
- They might have been changed at similar points in time (revisions in the version repository) during development.
- They might contain several code fragments that are similar or identical to each other.
- They might share common authors.
- They might be textually similar (contain similar key-words, function names, etc.).

Exercise: *There are many possible ways in which a relation between a pair of modules can be characterised. How could this potentially be problematic from a measurement perspective?*

For now, we stick with the simplest, structural definition of a relationship. To provide an illustration of the basic notions of coupling and cohesion, we first consider the illustration in Figure 8.8. Here we see that there is an extensive amount of communication between the two modules; many of the methods in the left module call methods in the right, and both sets of methods frequently access data members in the other module.

High coupling can lead to a multitude of serious problems. If, for example, a programmer wishes to understand what precisely is happening in module A, they will need to trace along any dependencies to module B (and to any dependencies that B has on other modules in the system). If a change is to be made, the profusion of dependencies means that there is a greater risk that it can have unintended side-effects. If we have a large, highly coupled system, individual modules also become harder to disentangle and replace with alternative implementations, etc.

Cohesion is illustrated in Figure 8.9. Module A on the left demonstrates high cohesion, functions share a lot of the variables. Module B on the other hand represents a function where none of the variables are shared. In Object Oriented systems, data classes can often adopt this pattern. Each variable will have a getter and a setter, but these will rarely read from or write to other variables.

Although Coupling and Cohesion are often used in tandem, they are measured by separate metrics. The main Coupling and Cohesion metrics to be used today

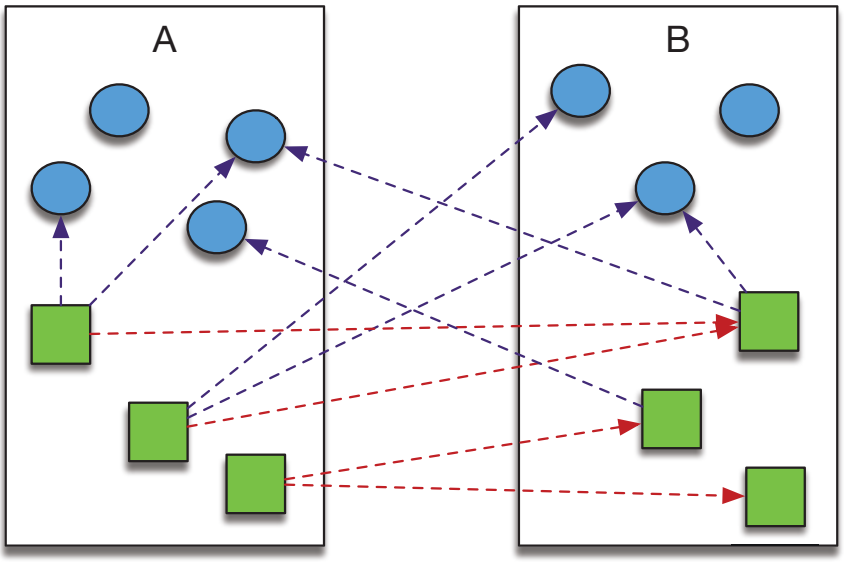

Fig. 8.8 Illustration of high coupling (and low cohesion). The two rectangles represent modules (e.g. classes in an Object-Oriented system). The circles represent data variables (e.g. class attributes), and the squares represent functions (e.g. class methods). Arrows between a pair functions represent function calls, and arrows between functions and data attributes represent reads or writes.

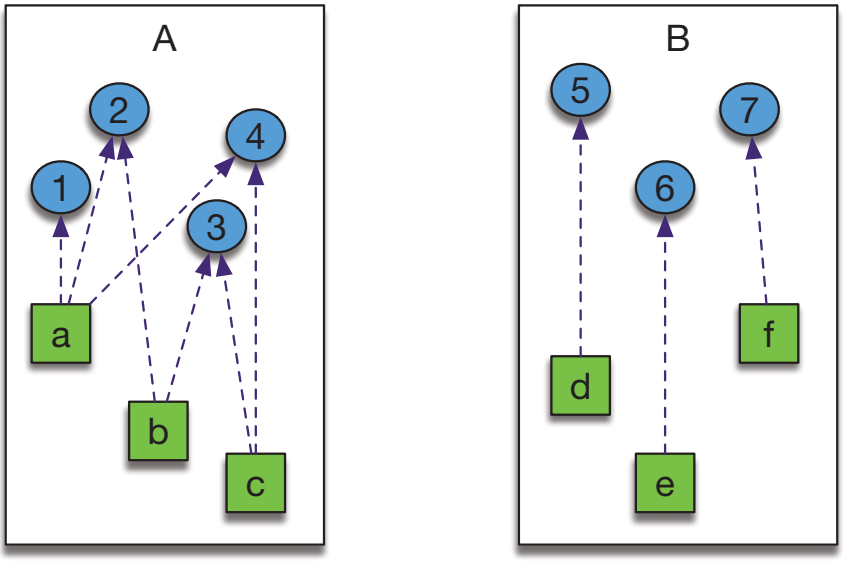

Fig. 8.9 Illustration of high cohesion (in module A) and low cohesion (in module B).

emerged from the suite of Object-Oriented metrics devised by Chidamber and Kemerer in 1994 [29]. Coupling was measured by the Coupling Between Objects metric, and Cohesion was measured by the Lack of Cohesion metric, both described below.

8.2.2.2 Coupling Between Objects (CBO)

Coupling Between Objects (CBO) calculates a value for every class in the system, representing the number of other classes to which it is coupled. The relationships considered are the standard ones: reading from / writing to a variable, or invoking a method. If a method call is polymorphic (there are multiple implementations of a method that could be invoked by a given call) then all of the possible calls are counted.

Exercise: *In Figure 8.8, assume that you are looking at two classes. Calculate the CBO for class A.*

8.2.2.3 Lack of Cohesion between Methods (LCOM)

Cohesion (or more specifically, lack thereof) is commonly measured by the LCOM metric. In short, it is computed as the number of pairs of methods without access to shared class attributes minus the number of pairs of member functions with access to shared class attributes. This metric has changed over the years, having been subject to a great amount of criticism [15].

A		**B**	
Methods	Variables shared	Methods	Variables shared
a,b	{2}	d,e	∅
a,c	{4}	d,f	∅
b,c	{3}	e,f	∅

Table 8.2 Basis for calculation of LCOM for classes A and B in Figure 8.9

Table 8.2 shows how LCOM is computed for the examples in Figure 8.9. Each row corresponds to one pair of methods, and the 'Variables' column shows how many variables they share. For class A, all three pairs of methods share variables (which means that are no pairs of methods that do not share variables), so $LCOM_A = 0 - 3 = -3$, which is re-written as zero; in other words there is no lack of cohesion. For Class B there are again three pairs of variables, but none of them share access to a variable, so for class B $LCOM_B = 3 - 0 = 3$.

Exercise: *Referring to the various scale-types we learned about in Section 8.1, why might the LCOM metric be problematic?*

The major criticism of LCOM (in answer to the above exercise) is the fact that it is fundamentally measured on an interval-scale, but is somewhat clumsily coerced into a ratio scale by rewriting any negative values as zero. The problem is that this conversion does not quite work, because there is no canonical definition of what amounts to a zero-value; modules that differ wildly in terms of how (un)cohesive they are can all produce a zero LCOM value.

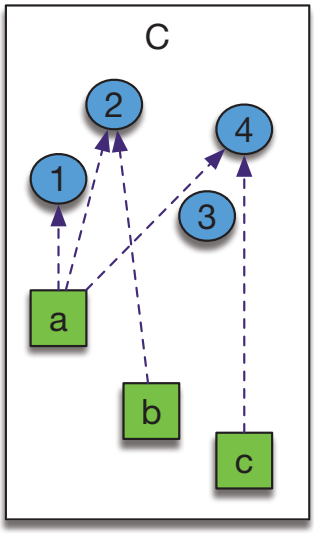

Fig. 8.10 Class that has a different cohesion to class A in Figure 8.9, but produces the same LCOM value.

As an example, we can look at class C in Figure 8.10. Looking at this class, one pair of methods does not share access to variables (b and c), whereas two do. This means that $LCOM_C = 1 - 2$, which is also re-written to zero. So although classes A and C clearly have different levels of cohesion, they share the same value.

8.2.3 Maintainability Metrics and the Maintainability Index

As we have seen in Chapter 2, "Maintainability" forms a key component of most mainstream quality models. If a software system is not maintainable, it cannot be readily adapted to changes in its environment and changes in requirements from

users. This renders it increasingly expensive to operate in the long run, because it is harder for developers to correctly make any necessary changes.

The Maintainability Index (MI), which was devised by Oman and Hagemeister in 1992 [103], is perhaps the most popular metric for measuring maintainability. Unlike the metrics discussed previously, it is not 'atomic', but is instead a combination of multiple other metrics. Specifically, it combines Halstead's volume (HV - see section 8.2.1.3), Cyclomatic Complexity (CC - see section 8.2.1.2), Lines of Code (LOC - see section 8.2.1.1), and an additional measure that we haven't covered in this book, which is simply the percentage of the LOC that are comments (COM). The combination is computed by the following formula:

$$171 - 5.2ln(HV) - 0.23CC - 16.2ln(LOC) + 50sin\sqrt{2.46 * COM}$$

To produce this formula, Oman and Hagemeister started with a number of systems developed by Hewlett-Packard, written in C and Pascal, between 1000 and 10,000 LOC in size. For each system, they asked the engineers to rate it in terms of its "maintainability" on a scale from 1 to 100. They then calculated 40 different metrics for each system and applied a statistical regression, which resulted in the above formula.

This metric subsequently garnered a lot of attention. Its use was widely promoted in the 90s and beyond. It has been included as part of the metrics suite in Microsoft Visual Studio since 2007, as well as a host of other metrics tools.

Exercise: Let us suppose that you are assessing a C# project. You load it into Visual Studio and go about computing the MI to gain an idea of its maintainability. Why might you hesitate to trust the resulting metrics? Write down three reasons.

8.3 Validity and the Use of Goal Question Metric

This section considers the problems of validity that tend to dog software engineering metrics. It then presents a general technique - Goal Question Metric (GQM), which can be used to address some of these issues.

8.3.1 Problems of Validity

As we have seen, metrics are fundamentally based on the representation condition: that the relationships between the elements that are being measured (e.g. source code files) are maintained within the measurements. Note that this means that the validity of a simple metric (e.g. LOC) cannot be assessed in isolation; its validity is intrinsically dependent upon the property that it purports to measure.

This relationship has certain strong implications. In order to use a metric to measure a property, it is necessary to (a) have an unambiguous definition of the property that one seeks to measure, and (b) be able to explain why a given metric will adequately represent that property. If either of these points is not fulfilled, there is no basis upon which to attest the validity of the metric. In this case, the question of whether or not the metric actually fulfils the representation condition ultimately comes down to luck. There are very few applications of metrics where one or both of these points cannot be called into question. There are countless published criticisms of most popular metrics, and most tend to focus on situations where the representation condition is violated and the metric becomes invalid.

In software, metrics tend to be valid when their purpose is specific. If, for example, one seeks to measure how easy a function will be to test with respect to Branch Coverage, then Cyclomatic Complexity is probably a reasonably valid metric. However, if one seeks to measure more abstract properties, such as development effort or reliability, the Cyclomatic Complexity is probably not a valid measure (at least in the general case) [120].

Metrics such as the Halstead Metrics and the MI are especially vulnerable to these criticisms. They purport to measure things that are not particularly well defined ("Effort", "Maintainability"). The formulae they use are also hard to justify. In the case of MI, these problems are almost pathological[5]. Maintainability is a very abstract concept with lots of different interpretations that are hard to pin down. On top of that, the MI formula is impossible to understand fully because it was produced automatically (e.g. why take the Sine of the square root of 2.45 times the number of commented lines of code?).

The problems with MI (and other metrics) are reinforced by more recent studies on maintainability. In 2012, Sjoberg *et al.* carried out a small empirical study into maintainability metrics [123], from which we can derive some useful insights. They compared a raft of maintainability metrics, including the MI, on four large systems, and factored in the actual maintenance effort that had been attributed to them. The key outcomes were as follows:

- None of the customised maintainability metrics (such as MI) were consistent with each other.
- The only metrics that were consistent with the actual maintenance *effort* required were LOC, and LCOM (see Secion 8.2.2.3).

8.3.2 Goal Question Metric

Without a thorough empirical study to back up the validity of a metric, the decision of whether a metric can be trusted comes down to one's subjective intuition. With-

[5] With respect to MI, some of these criticisms were eloquently summarised by Arie van Duersen in a blog post on the matter `https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/`

out any empirical evidence, one is essentially left asking the question *"Is there a plausible explanation as to why metric X will provide me with a reliable, valid measurement of what I am trying to assess?"*. One of van Duersen's criticisms of the MI was that this explanation is certainly not forthcoming; there is no obvious explanation why the various logs and square roots of various metrics should obviously produce a valid indicator for maintainability.

The Goal Question Metric framework[129] provides a useful basis for *explaining* choices of metrics. The concept is very straightforward. The challenge of measuring a particular property in a system is achieved on three levels:

1. **Conceptual:** This captures the high level *goals* of the measurement exercise. Which ineffable properties of the system do we need to establish?
2. **Operational:** This captures the *questions* that need to be answered in order to establish the goals. Which questions need to be answered in order to capture enough information from which to provide a reliable assessment of the goals?
3. **Quantitative:** What data that can be obtained to answer the questions?

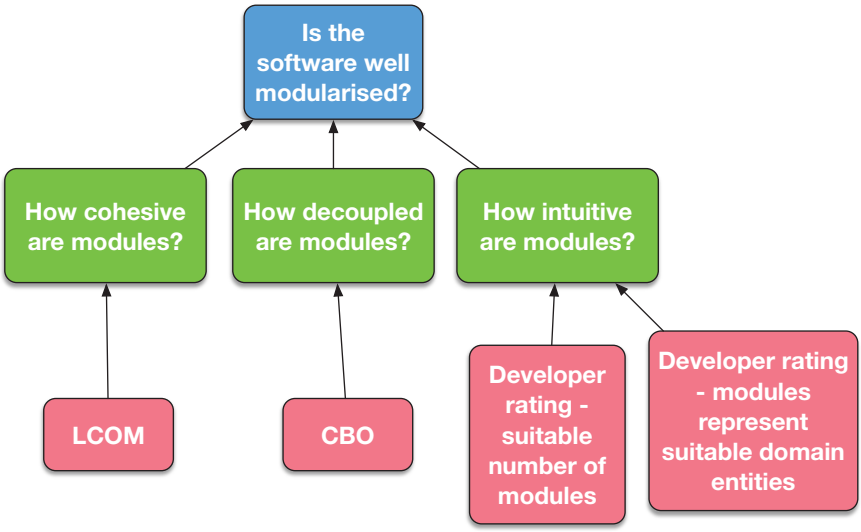

Fig. 8.11 Example of GQM applied to the assessment of software modularity. Each level represents conceptual, operational and quantitative levels respectively.

This will produce a tree-shaped structure, with high-level goals at the top level, each of which is linked to a set of questions, and where each question is linked to a set of metrics. An example is provided in Figure 8.11. Note that GQM makes no presumptions about whether or not the metrics are automated or not. For our metric, it modularity is not merely assessed by Coupling and Cohesion. These metrics are blind to other, more subjective questions (e.g. whether the modules correspond to

suitable entities in the problem domain [65]). Thus, for the third question, this decomposition takes into account a domain expert's opinion of whether the modules are suitable in number (i.e. there isn't just one large module for the entire system), and whether the modules represent intuitive entities within the domain.

At this point it is worth looking back to previous sections in the book. There are several point at which hierarchies have been advocated to facilitate some form of measurement.

Exercise: Can you remember what they are? Do not read further until you want the answer!

In Chapter 2 we encountered Barry Boehm's Q-Model [22]. This was a hierarchical means by which to capture software quality attributes, but placed an emphasis on the fact that the lowest "leaves" in the tree should be measurable. Later on, we encountered GSN [82], a tree structured notation by which to make safety arguments for software systems more measurable. In both cases (as with GQM) a complex task is made possibly by dividing and conquering, by splitting complex concepts into simpler, more measurable ones.

8.4 Key Points

- **Metrics are a lynch-pin for any efforts by which to ensure software quality.** As DeMarco's quote goes – you cannot control what you cannot measure, and quality assurance is ultimately about controlling what you can to maximise software quality. It also follows that metrics can also shine a harsh light on systems where quality assurance appears to have failed, as is testified by the use of metrics in the Toyota unintended acceleration case.
- **Metrics have to be interpreted with care; there has to be an understanding of what the numbers mean (what is a good or a bad value, can they be compared, and if so how, etc.).** The Representative Theory of Measurement provides a framework within which to do so. It provides a formal basis upon which to understand different scales (in the form of Admissible Transformations).
- **The size and complexity of a program can be measured in various ways.** The source code can be analysed to count lines of code, to measure Cyclomatic Complexity and Halstead Complexity. The stated functionality can be analysed to measure the Albrecht Function Point count.
- **Modularity can be measured by assessing a system in terms of its Coupling and Cohesion.** Doing so can be deceptively difficult in an Object-Oriented context, because it is not necessarily obvious how to distinguish between 'internal' and 'external' dependencies (e.g. in the presence of inheritance relationships between classes).
- **The Maintainability Index is a 'composite' metric by which to assess the maintainability of a system.** It is based on cyclomatic complexity, Halstead's

volume, and LOC. It is relatively popular, despite the fact that it is difficult to justify from the perspective of validity.

- **The Goal Question Metric (GQM) approach presents a means by which to quantitatively answer high level questions about a software system.** Goals can be hierarchically decomposed into questions, which can be answered by sets of specific metrics.

Chapter 9
Conclusions

This book has sought to cover the most salient aspects of software quality assurance. In doing so we have discussed what software quality is, why it is important, and how it is defined. We have examined how the activities of quality assurance are closely linked to the choice of software development process. We have covered agile software development, testing, inspections, safey reviews, metrics, and cost estimation.

9.1 Topical and Emerging Quality Concerns

With the rapid emergence of new technology, and rapid changes in the way technology is used, the landscape of software quality assurance is constantly shifting. In the rest of this chapter, we look at what could be considered to be some of the key quality assurance challenges to have emerged in recent years.

9.1.1 Autonomy in Socio-Technical Systems

Let us consider two relatively recent incidents. In 2009, an Airbus A330 Air France flight 447 on its way from Rio de Janeiro to Paris crashed into the Atlantic, killing all 228 passengers on board. A subsequent crash investigation indicated the following sequence of events [116]:

- The plane, flying on autopilot, had encountered an adverse weather system, which had caused the Pitot tubes[1] to freeze.
- The autopilot disconnected and the control of the plane fell to manual control.
- The pilots were unprepared for the sudden disengagement, and were confused as to why this had happened.

[1] Small external tubes that are used to measure air-speed on aircraft.

N. Walkinshaw, *Software Quality Assurance*, Undergraduate
Topics in Computer Science, DOI 10.1007/978-3-319-64822-4_9

- In the mean time the plane had entered into a stall and descended rapidly (11,000 ft per minute) towards the ocean and crashed.

For the second incident, in May 2016 a Tesla Model S was driving along a highway at 74mph. The driver had engaged the "autopilot" mode in the car and was not concentrating on what was happening. A tractor-trailer crossed the path of the car, but its sensors failed to detect it. The car crashed under the side of the trailer, ripping off the roof of the car, and subsequently crashed into a pole at the side of the road, killing the driver.

Although the vehicles involved and the scales of the tragedies are different, both share two links. Firstly, and most obviously, the Airbus and the Tesla had been entirely *autonomous* (controlled by their own software without human intervention) in the run-up to the crash (in the case of the Airbus) and during the crash (in the case of the Tesla). Secondly, and perhaps more surprisingly, in both cases the software was *not* held to be responsible for the tragedies, which were ultimately blamed on human error.

Why *human* error? In the case of the Air France crash, the pilots were deemed to have lacked 'situational awareness '[46]; they should have immediately been able to determine why the autopilot had disengaged, and have thus reacted more speedily, in a more appropriate way. In the case of the Tesla, the driver was deemed again to lack situational awareness. For that car model, the system was only designed to keep the car in its lane and to avoid crashes with other cars (the scenario of trailer crossings was beyond the scope of the system).

This is understandable on the one level; the software systems when viewed in isolation performed exactly as they were designed to. However, when one takes a step back, and considers the broader context within which the systems were used, the culpability of the users is perhaps not so clear-cut. These systems are invariably complex, comprising a multitude of components and operators, where it is not necessarily possible for a single unit (human or technological) to maintain a coherent, macroscopic overview of the state of the system at any given time. In the case of the Air France disaster this problem has been demonstrated [116].

However, even in the case of the Tesla accident one could also argue that, though contrary to the system specifications and instructions, it is only to be expected that some drivers will be seduced by the idea of "driverless cars" and be willing to test its capabilities to the limit. Indeed, for typical road users, who are unaware of the detailed sensor configurations and limitations on the underlying control algorithms and Machine Learnt models, it can be argued that it is ultimately impossible for the driver to maintain a sufficient degree of "situational awareness" to be able to truly account for the behaviour of their car when it is under its own control.

So, although lack of situational awareness is to blame, the question of whether it is the human operator's *fault* is another question. The problems caused by the fuzzy boundaries between an inscrutably complex system (or system of systems) and a human operator are not new, and are not even specific to digital systems. In his book *"Normal Accidents: Living with High Risk Technologies"* [109], Perrow highlights how similar incidents – misuse of technology, rooted in misunderstanding and a lack of situational awareness, have contributed to some of the great disasters of our

time, including the disasters at the Three-Mile Island and Chernobyl nuclear power plants.

Although the problem in its essence is not new, there is a strong argument to be made that the increased pervasiveness of software-driven technology is greatly exacerbating it. Software is taking over activities that have traditionally been entirely manual, and the rules by which people interact with technology are constantly being re-written. With these changes, users (or drivers or pilots) are bound to be uncertain about the boundary between their areas of responsibility, and the 'system's' set of responsibilities.

From a quality assurance perspective, this blurring of responsibilities between the operator and the system put a new spin on long-standing questions. What is the scope of a 'system'? What use and context should the system design take into account? What should be the 'contract' between the user and the system, and how should this be communicated to users, to prevent the sorts of misunderstandings that we have described above?

9.1.2 Data-Intensive, Untestable Systems

Machine Learning algorithms used to play a relatively confined role when it came to software systems, finding their uses for relatively 'niche' activities such as detecting junk emails or credit card fraud. However, as the prevalence of data (especially data that pertains to individuals) has grown, and Machine Learning algorithms have become more versatile and powerful, their role in the functionality of software has greatly increased. The driverless cars discussed above represent one particular area where Machine Learning has become a central component. However, it has become prominent in almost any area that involves large volumes of data, from detecting user web-browsing patterns, to intrusion detection in networks and detecting suitable trades in financial systems. These systems become problematic when the algorithms, which are developed and trained to react to a vast range of scenarios, encounter one example of a scenario that they have not been prepared for.

As an example we refer to another example of a driverless car crash, also in 2016, but this time one that did not cause any fatalities. In May 2016 in Mountain View, a Google driverless car pulled out from a parked position and crashed into the side of a bus that was overtaking it[2]. Here, the fault *did* lie with the software system. This was not an isolated event; according to a document filed by Google with the California Department for Motor Vehicles, its driverless car software experienced 272 failures and would have crashed 13 times had it not been for human intervention[3]. The various problems that are thrown up in driverless cars by Machine Learning

[2] https://www.theguardian.com/technology/2016/feb/29/google-self-driving-car-accident-california

[3] https://www.theguardian.com/technology/2016/jan/12/google-self-driving-cars-mistakes-data-reports

algorithms are discussed in more detail by Wagner and Koopman [130] (Koopman had also investigated the Toyota Unintended Acceleration fault - see Chapter 2).

Of course, driverless cars are far from the only area within which data-intensive algorithms have become increasingly prevalent. Another area is in the finance sector, and specifically in the form of *High Frequency Trading (HFT)* algorithms. HFT algorithms analyse data from a variety of sources; they monitor real-time stock-market data, often alongside large streams of news-information, and use this data to automatically trade on the stock market. They use this data to predict future stock values, often by the use of intricate statistical data processing algorithms, often using the results to execute thousands of trades per second.

When these systems malfunction, they can have potentially disastrous effects, not just on individual businesses, but entire economies. There are plenty of examples [84], and we pick out a few notable ones here. In August 2012, Knight Capital introduced a faulty trading algorithm to the market. As soon as they activated it, it lost approximately £6.4m per minute, ultimately losing £281m before it could be switched off. Aside from such individual failures, the more frightening problems arise when HFT algorithms interact with each other to produce "flash crashes". In 2010, such an event led to the loss of 998 points (approximately 9%) off the Dow Jones Industrial Average, only to rise to its previous level after about 15 minutes. More recently, in the aftermath of the Brexit vote, the British pound slumped 6% against the dollar in a similar flash crash, only to regain its value again after a couple of minutes.

What are the similarities that link HFT systems to driverless cars? They both involve the use of Machine Learning and other statistical data processing algorithms to process large volumes of data. Both are either safety or business critical; when the underlying software is faulty, the consequences can be disastrous.

Exercise: *Before reading on, think back to the chapter on software testing. Using testing terminology, what is the problem with statistical data processing / Machine Learning algorithms?*

Statistical data processing and Machine Learning algorithms pose interesting problems from a quality assurance perspective because their "correct" behaviour is difficult to anticipate and express. They exist to mine and discover things about data that are not necessary known or even knowable a-priori, which means that a lot of their outputs will by necessity be unpredictable.

As a result, such systems are intrinsically difficult to specify. Though straightforward in the abstract (a car should not collide with other cars, a trading algorithm should minimise losses and maximise gains) tying these requirements to implementation details and verification or validation activities is extremely difficult. "Other cars" could correspond to a vast range of possible sensor signal patterns and could, depending on the sensor configurations, vary according to light / road conditions, speed, etc. The question of whether a trading algorithm is successful at minimising losses and maximising gains depends on a host of external factors (the combination

of current events, competing trading algorithms, current stock market movements, etc.), where it is practically impossible to determine "the" ideal behaviour.

The standard approaches to quality assurance and testing would struggle (and probably fail) to provide compelling answers to these questions. Although the problem is not new [133], the prevalence of such systems certainly makes the task of finding a compelling solution all the more urgent.

9.2 Concluding Remarks

Software development does not take place in a vacuum, and is therefore inevitably an involved, at times messy, process. Users are capricious and liable to change their minds about what they want. Technology is evolving at a fast pace, demanding continuous changes to the way in which software operates. The time and effort required for various activities can often be grossly underestimated (and occasionally overestimated).

Ultimately, the challenge of trying to produce a successful software system in the face of these various forces lies with a band of fallible human software developers. They probably have varying abilities, are subject to different time pressures, may be geographically distributed, perhaps don't work under the auspices of the same organisation, and possibly don't even know each other. Often it is these same developers that are also responsible for managing the quality assurance of the product.

It is unsurprising that even the most safety-critical or business-critical systems can end up with quality problems. There are plenty of techniques that can improve quality assurance, however, no technique is bullet proof. All approaches tend to require a degree of intuition and experience, and rely on a level of discipline, time, and effort that is rarely practical in a realistic software engineering context.

This is what makes quality assurance especially interesting: Far from being an after-thought to be 'done' if there is time, it is integral throughout – the most vital, and most interesting angle of software development. The dynamics of software development are relentless, subject to so many human and technological factors. And in the face of all of this, you are given only an imperfect armoury of tools, and very little time to ensure the quality of a product where failure to do so can have significant (potentially devastating) consequences.

As it stands, it is impossible to guarantee that a software system will be bug free, will be readily maintainable, and be delivered on schedule. It is even impossible to reliably *measure* the quality of a software system. The big challenges of software quality assurance have not yet been solved. As time passes, technology and the way in which it is used will continue to evolve at breakneck speed. Software systems will inevitably become increasingly complex, larger in scale, and increasingly safety- and business-critical.

One aim of this book has been to present an overview of the key problems, principles, and techniques within the remit of quality assurance. Given the breadth of

the area, this has by necessity been selective. For any of the topics covered there exist enormous volumes of in-depth text books and research publications.

And this is where your pursuit of software quality assurance can begin in earnest! If you have found yourself looking up references, alternative approaches, or querying whether one of the techniques described in this book really is the most appropriate solution to a problem, then the book has fulfilled its ultimate goal – to pique your interest.

References

[1] (1982) DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (RTCA)

[2] (1988) DOD-STD-2167A, Defense Systems Software Development. Department of Defence

[3] (2006) IEC 60880: Nuclear power plants - Instrumentation and control systems important to safety - Software aspects for computer-based systems performing category A functions. International Electrotechnical Commission

[4] (2011) ISO 26262-1:2011: Road Vehicles - Functional Safety. International Standards Organisation

[5] (2011) ISO/IEC 25010:2011: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. International Standards Organisation

[6] (2015) The promise repository of empirical software engineering data

[7] Achimugu P, Selamat A, Ibrahim R, Mahrin MN (2014) A systematic literature review of software requirements prioritization research. Information and software technology 56(6):568–585

[8] Albrecht AJ (1979) Measuring application development productivity. In: Proceedings of the joint SHARE/GUIDE/IBM application development symposium, vol 10, pp 83–92

[9] Alexander C (1977) A pattern language: towns, buildings, construction. Oxford University Press

[10] Anderson J (2011) A million monkeys and shakespeare. Significance 8(4):190–192

[11] Arcuri A, Briand L (2011) Adaptive random testing: An illusion of effectiveness? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ACM, pp 265–275

[12] Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, pp 712–721

N. Walkinshaw, *Software Quality Assurance*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-64822-4

[13] Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S (2015) The oracle problem in software testing: A survey. Software Engineering, IEEE Transactions on 41(5):507–525
[14] Barrett SR, Speth RL, Eastham SD, Dedoussi IC, Ashok A, Malina R, Keith DW (2015) Impact of the volkswagen emissions control defeat device on us public health. Environmental Research Letters 10(11):114,005
[15] Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Transactions on software engineering 22(10):751–761
[16] Basili VR, Green S, Laitenberger O, Lanubile F, Shull F, Sørumgård S, Zelkowitz MV (1996) The empirical investigation of perspective-based reading. Empirical Software Engineering 1(2):133–164
[17] Beck F, Diehl S (2011) On the congruence of modularity and code coupling. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, pp 354–364
[18] Beller M, Bacchelli A, Zaidman A, Juergens E (2014) Modern code reviews in open-source projects: which problems do they fix? In: Proceedings of the 11th working conference on mining software repositories, ACM, pp 202–211
[19] Bentley JL, McIlroy MD (1993) Engineering a sort function. Software: Practice and Experience 23(11):1249–1265
[20] Boehm B, Clark B, Horowitz E, Westland C, Madachy R, Selby R (1995) Cost models for future software life cycle processes: Cocomo 2.0. Annals of software engineering 1(1):57–94
[21] Boehm BW (1988) A spiral model of software development and enhancement. Computer 21(5):61–72
[22] Boehm BW, Brown JR, Kaspar H (1978) Characteristics of software quality
[23] Boehm BW, et al (1981) Software engineering economics, vol 197. Prentice-hall Englewood Cliffs (NJ)
[24] Borning A (1987) Computer system reliability and nuclear war. Communications of the ACM 30(2):112–131
[25] Brooks F (1975) The Mythical Man Month. Information Systems Programs, General Electric Company
[26] Buxton JN, Randell B (1970) Software engineering techniques: report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969. NATO Science Committee
[27] Challenger PCOSS, Rogers W (1986) Report of the presidential commission on the space shuttle challenger accident
[28] Chen TY, Leung H, Mak I (2004) Adaptive random testing. In: Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making, Springer, pp 320–329
[29] Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Transactions on software engineering 20(6):476–493
[30] Chow TS (1978) Testing software design modeled by finite-state machines. IEEE transactions on software engineering 4(3):178
[31] Chung L, Nixon BA, Yu E, Mylopoulos J (2012) Non-functional requirements in software engineering, vol 5. Springer Science & Business Media

[32] Clegg D, Barker R (1994) Case method fast-track: a RAD approach. Addison-Wesley Longman Publishing Co., Inc.
[33] Cohen D, Lindvall M, Costa P (2004) An introduction to agile methods. Advances in computers 62:1–66
[34] Committee ICSSES, Board ISS (1998) Ieee recommended practice for software requirements specifications. Institute of Electrical and Electronics Engineers
[35] of Commons Public Accounts Select Committee H, et al (2013) Universal credit: early progress. URL https://www.nao.org.uk/wp-content/uploads/2013/09/10132-001-Universal-credit.pdf
[36] Company GE, McCall JA, Richards PK, Walters GF (1977) Factors in software quality: Final report. Information Systems Programs, General Electric Company
[37] Comptroller General (1981) Norad's missile warning system: What went wrong? URL http://www.gao.gov/assets/140/133240.pdf
[38] Crosby PB (1980) Quality is free: The art of making quality certain. Signet
[39] Darimont R, Delor E, Massonet P, van Lamsweerde A (1997) Grail/kaos: an environment for goal-driven requirements engineering. In: Proceedings of the 19th international conference on Software engineering, ACM, pp 612–613
[40] De Neufville R (1994) The baggage system at denver: prospects and lessons. Journal of Air Transport Management 1(4):229–236
[41] Demeyer S, Ducasse S, Nierstrasz O (2002) Object-oriented reengineering patterns. Elsevier
[42] Dijkstra EW (1968) Letters to the editor: go to statement considered harmful. Communications of the ACM 11(3):147–148
[43] Dijkstra EW (1972) The humble programmer. Communications of the ACM 15(10):859–866
[44] Domke F, Lange D (2015) The exhaust emissions scandal ("dieselgate"). URL https://events.ccc.de/congress/2015/Fahrplan/system/event_attachments/attachments/000/002/812/original/32C3_-_Dieselgate_FINAL_slides.pdf
[45] Dybå T, Dingsøyr T (2008) Empirical studies of agile software development: A systematic review. Information and software technology 50(9):833–859
[46] Endsley MR (1995) Toward a theory of situation awareness in dynamic systems. Human Factors: The Journal of the Human Factors and Ergonomics Society 37(1):32–64
[47] Eypasch E, Lefering R, Kum C, Troidl H (1995) Probability of adverse events that have not yet occurred: a statistical reminder. BMJ: British Medical Journal 311(7005):619
[48] Fagan M (1976) Design and code inspections to reduce errors in program development. IBM Journal of Research and Development 15(3):182
[49] Federal Aviation Administration (2015) Docket no. faa-2015-0936; directorate identifier 2015-nm-058-ad; amendment 39-18153; ad 2015-09-07. URL https://s3.amazonaws.com/public-inspection.federalregister.gov/2015-10066.pdf

[50] Fenton N (1994) Software measurement: A necessary scientific basis. IEEE Transactions on software engineering 20(3):199–206
[51] Fenton NE, Whitty RW, Iizuka Y (1995) Software Quality Assurance and Measurement: A Worldwide Perspective. Itp-Media
[52] Flyvbjerg B, Budzier A (2011) Why your it project may be riskier than you think. Harvard Business Review 89(9):601–603
[53] Fowler M (2004) UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional
[54] Fowler M, Beck K (1999) Refactoring: improving the design of existing code. Addison-Wesley Professional
[55] Fowler M, Highsmith J (2001) The agile manifesto. Software Development 9(8):28–35
[56] Fraser G, Arcuri A (2011) EvoSuite: Automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE'11, pp 416–419
[57] Gamma E, Helm R, Johnson R, Vlissides J (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley
[58] Glinz M (2007) On non-functional requirements. In: Requirements Engineering Conference, 2007. RE'07. 15th IEEE International, IEEE, pp 21–26
[59] Goldberg DE (2006) Genetic algorithms. Pearson Education India
[60] Goodenough JB, Gerhart SL (1975) Toward a theory of test data selection. IEEE Transactions on Software Engineering 1(2):156–173
[61] Gotel OC, Finkelstein C (1994) An analysis of the requirements traceability problem. In: Requirements Engineering, 1994., Proceedings of the First International Conference on, IEEE, pp 94–101
[62] Gousios G, Pinzger M, Deursen Av (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering, ACM, pp 345–355
[63] Groce A, Holzmann G, Joshi R (2007) Randomized differential testing as a prelude to formal verification. In: 29th International Conference on Software Engineering (ICSE'07), IEEE, pp 621–631
[64] Guimarães ML, Silva AR (2012) Improving early detection of software merge conflicts. In: Software Engineering (ICSE), 2012 34th International Conference on, IEEE, pp 342–352
[65] Hall M, Walkinshaw N, McMinn P (2012) Supervised software modularisation. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, pp 472–481
[66] Halstead M (1977) Elements of Software Science. North-Holland
[67] Hamlet R (1994) Random testing. Encyclopedia of software Engineering
[68] Hannay JE, Dybå T, Arisholm E, Sjøberg DI (2009) The effectiveness of pair programming: A meta-analysis. Information and Software Technology 51(7):1110–1122
[69] Humphrey W (1989) Managing the Software Process. Addison Wesley

[70] Inozemtseva L, Holmes R (2014) Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering, ACM, pp 435–445
[71] Isaac R (2013) The pleasures of probability. Springer Science & Business Media
[72] Jacobson I (1992) Object-Oriented Software Engineering. A Use Case Driven Approach. Addison-Wesley
[73] Jacobson I, Booch G, Rumbaugh J, Rumbaugh J, Booch G (1999) The unified software development process, vol 1. Addison-Wesley Reading
[74] Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5):649–678
[75] Johnson PM, Tjahjono D (1998) Does every inspection really need a meeting? Empirical Software Engineering 3(1):9–35
[76] Johnson SC (1977) Lint, a C program checker. Bell Telephone Laboratories Murray Hill
[77] Jørgensen M (1999) Software quality measurement. Advances in engineering software 30(12):907–912
[78] Juran J (1970) Quality planning and analysis: from product development through usage
[79] Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014) Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, pp 654–665
[80] Kan SH (2002) Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc.
[81] Kano N, Seraku N, Takahashi F, Tsuji S (1984) Attractive quality and must-be quality. Journal of the Japanese Society for Quality Control 14
[82] Kelly T, Weaver R (2004) The goal structuring notation–a safety argument notation. In: Proceedings of the dependable systems and networks 2004 workshop on assurance cases
[83] King JC (1976) Symbolic execution and program testing. Communications of the ACM 19(7):385–394
[84] Kirilenko AA, Lo AW (2013) Moore's law versus murphy's law: Algorithmic trading and its discontents. The Journal of Economic Perspectives 27(2):51–72
[85] Kirsch M (2014) Technical support to the national highway traffic safety administration (nhtsa) on the reported toyota motor corporation (tmc) unintended acceleration (ua) investigation. URL `http://www.nasa.gov/topics/nasalife/features/nesc-toyota-study.html`
[86] Koopman P (2014) A case study of toyota unintended acceleration and software safety. URL `betterembsw.blogspot.co.uk/2014/09/a-case-study-of-toyota-unintended.html`
[87] Kransner G, Pope S (1988) Cookbook for using the model-view-controller user interface paradigm. Object Oriented Programming pp 26–49

[88] Langner R (2011) Stuxnet: Dissecting a cyberwarfare weapon. IEEE Security & Privacy 9(3):49–51
[89] Larman C, Basili VR (2003) Iterative and incremental development: A brief history. Computer (6):47–56
[90] Lee D, Yannakakis M (1996) Principles and Methods of Testing Finite State Machines - A Survey. In: Proceedings of the IEEE, vol 84, pp 1090–1126
[91] Leroy X (2007) Formal verification of an optimizing compiler. Lecture Notes in Computer Science 4533:1
[92] Malcolm DG, Roseboom JH, Clark CE, Fazar W (1959) Application of a technique for research and development program evaluation. Operations research 7(5):646–669
[93] McCabe TJ (1976) A complexity measure. IEEE Transactions on software Engineering (4):308–320
[94] McMinn P (2004) Search-based software test data generation: A survey. Software Testing Verification and Reliability 14(2):105–156
[95] Mens T (2002) A state-of-the-art survey on software merging. IEEE transactions on software engineering 28(5):449–462
[96] Menzies T, Yang Y, Mathew G, Boehm B, Hihn J (2016) Negative results for software effort estimation. arXiv preprint arXiv:160905563
[97] Miguel JP, Mauricio D, Rodriguez G (2014) A review of software quality models for the evaluation of software products. CoRR abs/1412.2977
[98] Moløkken-Østvold K, Haugen NC, Benestad HC (2008) Using planning poker for combining expert estimates in software projects. Journal of Systems and Software 81(12):2106–2117
[99] Moore EF (1956) Gedanken–experiments on sequential machines. In: Shannon CE, McCarthy J (eds) Annals of Mathematics Studies (34), Automata Studies, Princeton University Press, Princeton, NJ, pp 129–153
[100] Nair S, De La Vara JL, Sabetzadeh M, Briand L (2014) An extended systematic literature review on provision of evidence for safety certification. Information and Software Technology 56(7):689–717
[101] Nerur S, Mahapatra R, Mangalaraj G (2005) Challenges of migrating to agile methodologies. Communications of the ACM 48(5):72–78
[102] Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering, ACM, pp 35–46
[103] Oman P, Hagemeister J (1992) Metrics for assessing a software system's maintainability. In: Software Maintenance, 1992. Proceerdings., Conference on, IEEE, pp 337–344
[104] Ostrand TJ, Balcer MJ (1988) The category-partition method for specifying and generating fuctional tests. Communications of the ACM 31(6):676–686
[105] Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12):1053–1058
[106] Parnas DL (1985) Software aspects of strategic defense systems. Communications of the ACM 28(12):1326–1335

[107] Parnas DL, Clements PC (1986) A rational design process: How and why to fake it. Software Engineering, IEEE Transactions on (2):251–257
[108] Parnas DL, Weiss DM (1985) Active design reviews: principles and practices. In: Proceedings of the 8th international conference on Software engineering, IEEE Computer Society Press, pp 132–136
[109] Perrow C (1984) Normal Accidents: Living with High Risk Technologies. Princeton University Press
[110] Pezzè M, Young M (2007) Software testing and analysis - process, principles and techniques. Wiley
[111] Potvin R, Levenberg J (2016) Why google stores billions of lines of code in a single repository. Communications of the ACM 59(7):78–87
[112] Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, pp 202–212
[113] Rosenberg D, Stephens M (2003) Extreme programming refactored: the case against XP. Apress
[114] Royce WW (1970) Managing the development of large software systems. In: proceedings of IEEE WESCON, Los Angeles, vol 26, pp 1–9
[115] Rubin KS (2012) Essential Scrum: A practical guide to the most popular Agile process. Addison-Wesley
[116] Salmon PM, Walker GH, Stanton NA (2016) Pilot error versus sociotechnical systems failure: a distributed situation awareness analysis of air france 447. Theoretical Issues in Ergonomics Science 17(1):64–79
[117] Sen K, Marinov D, Agha G (2005) CUTE: a concolic unit testing engine for C, vol 30. ACM
[118] Series NWP (1980) If Japan Can, Why Can't We? URL `https://www.youtube.com/watch?v=vcG\_Pmt\_Ny4`
[119] Shaw M, Garlan D (1996) Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall
[120] Shepperd M (1988) A critique of cyclomatic complexity as a software metric. Software Engineering Journal 3(2):30–36
[121] Shewhart WA (1931) Economic control of quality of manufactured product. ASQ Quality Press
[122] Sindre G, Opdahl AL (2005) Eliciting security requirements with misuse cases. Requirements engineering 10(1):34–44
[123] Sjøberg DI, Anda B, Mockus A (2012) Questioning software maintenance metrics: a comparative case study. In: Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, ACM, pp 107–110
[124] Spinellis D (2003) Code reading: the open source perspective. Addison-Wesley Professional
[125] Staats M, Whalen MW, Heimdahl MP (2011) Programs, tests, and oracles: the foundations of testing revisited. In: Software Engineering (ICSE), 2011 33rd International Conference on, IEEE, pp 391–400

[126] Staples M, Niazi M, Jeffery R, Abrahams A, Byatt P, Murphy R (2007) An exploratory study of why organizations do not adopt cmmi. Journal of systems and software 80(6):883–895

[127] Team CP (2010) CMMI for Services Version 1.3. Lulu. com

[128] UK MOD (2004) Tornado safety case report. URL https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/475938/20040616_Annex_C_-_Tornado_SC__v1_U_0__-_Baseline.pdf

[129] Van Solingen R, Berghout E (1999) The Goal/Question/Metric Method: a practical guide for quality improvement of software development. McGraw-Hill

[130] Wagner M, Koopman P (2015) A philosophy for developing trust in self-driving cars. In: Road Vehicle Automation 2, Springer, pp 163–171

[131] Wagner S, Lochmann K, Heinemann L, Kläs M, Trendowicz A, Plösch R, Seidl A, Goeb A, Streit J (2012) The quamoco product quality modelling and assessment approach. In: Proceedings of the 34th international conference on software engineering, IEEE Press, pp 1133–1142

[132] Weyuker EJ (1979) Translatability and decidability questions for restricted classes of program schemas. SIAM Journal on Computing 8(4):587–598

[133] Weyuker EJ (1982) On testing non-testable programs. The Computer Journal 25(4):465–470

[134] Wilson JM (2003) Gantt charts: A centenary appreciation. European Journal of Operational Research 149(2):430–437

[135] Wood M, Roper M, Brooks A, Miller J (1997) Comparing and combining software defect detection techniques: a replicated empirical study. In: ACM SIGSOFT Software Engineering Notes, Springer-Verlag New York, Inc., vol 22, pp 262–277

[136] Woodcock J, Davies J (1996) Using Z: specification, refinement, and proof, vol 39. Prentice Hall Englewood Cliffs

[137] Wynne M, Hellesoy A (2012) The cucumber book: behaviour-driven development for testers and developers. Pragmatic Bookshelf

[138] Yuan D, Luo Y, Zhuang X, Rodrigues GR, Zhao X, Zhang Y, Jain PU, Stumm M (2014) Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp 249–265

[139] Zhu H, Hall PA, May JH (1997) Software unit test coverage and adequacy. Acm computing surveys (csur) 29(4):366–427

[140] Zowghi D, Coulin C (2005) Requirements elicitation: A survey of techniques, approaches, and tools. In: Engineering and managing software requirements, Springer, pp 19–46

Index

N. Walkinshaw, *Software Quality Assurance*, Undergraduate
Topics in Computer Science, DOI 10.1007/978-3-319-64822-4