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.

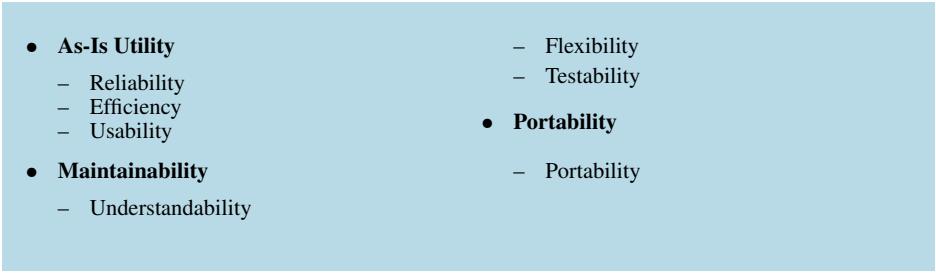

Fig. 2.3 Boehm's Q-Model

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.

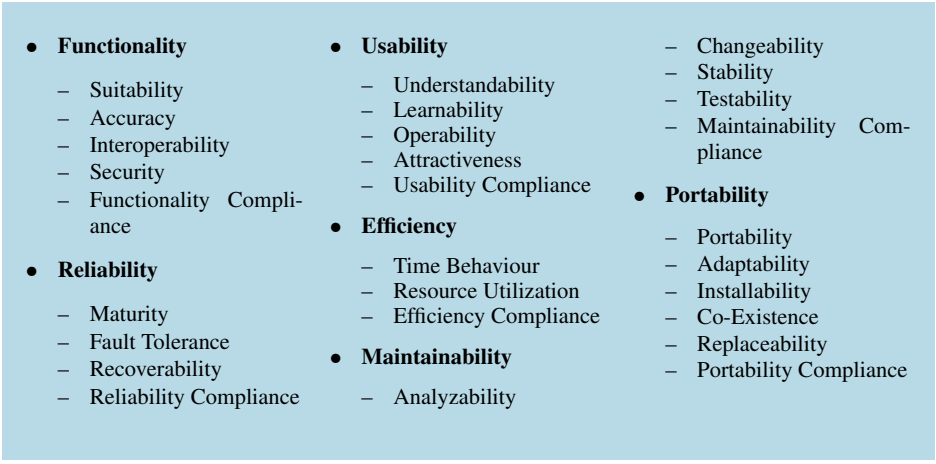

Fig. 2.4 ISO 9126

- **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).

http://www.springer.com/978-3-319-64821-7

Software Quality Assurance
Consistency in the Face of Complexity and Change
Walkinshaw, N.
2017, XI, 181 p. 63 illus., 41 illus. in color., Softcover
ISBN: 978-3-319-64821-7