Week 1
Introduction to Software Quality

In this chapter, we shall discuss the following topics:

- What is Software Quality?
- What is Software Quality Assurance (SQA)? - it is not finding and fixing bugs.
- What kind of techniques can be used to apply software quality assurance practicing? It is not just testing.
- What can and can't be achieved with SQA?

This is mainly scene setting, but it is an important scene setting. You need to understand the context of software quality assurance for the techniques we are going to discuss (and do) in the rest of the unit to make any sense.

When software quality assurance goes wrong?

Case1: Queensland health Payroll

In 2009 and 2010, Queensland Health, a state government department responsible for running the state of Queensland's public hospital system, attempted to replace its payroll system. The process did not go well. In fact, the results were seen to have contributed to the defeat of the incumbent government in the next election.

A commission of enquiry was established by the government, to investigate why the project went wrong. In its lengthy report, it summarized the consequences of the botched payroll system:

> *The system cost, in terms of payments to IBM (the principal contractor for the system) alone, over four times more than the contract price. It took three times longer to deliver than originally scheduled. When it went live it was seriously deficient, causing very many QH staff not to be paid, or to be paid inaccurately...*

So why did the project go so spectacularly wrong? While the causes of the problems were numerous, complex, and best understood by reading the report itself, there were some key problems:

- Queensland Health did not provide the contractor, IBM, with enough detail about requirements for IBM to write a functional payroll system.

- Queensland Health accepted the Scope Document which documented the requirements. IBM went ahead and built the system according to the Scope Document, despite both parties realizing that the Requirements written in the Scope Document were not good enough to build a system to "pay staff and pay them correctly" .
- User acceptance testing by an external contractor revealed literally thousands of defects, including many critical ones. Despite this, Queensland Health decided to go live with the system.

This was, in part, a failure of management, who ignored the results of competently performed user acceptance testing, a key quality assurance activity. But it was also, in part, a failure of software quality assurance.

Some of you might be sceptical - if the acceptance testing was done and revealed the defects, then the QA was fine, wasn't it? In short, no. The acceptance tester did their job competently, and the managers of the project should have listened (the reasons why they did not are fascinating, relevant to software engineering students, but too lengthy to go into here). But SQA is not just about testing.

Case 2: Malaysian banks

Three Malaysian banks were reportedly hacked into by a Latin American gang which made off with over RM3 million why? Because the banks refused to upgrade their bank systems to Windows 7. All banks were running Windows XP, and they were asked to upgrade their system because Microsoft announced that it will stop supporting Windows XP from 2014. The upgrade will cost banks millions of Ringgits (Malaysian currency). Malaysian Crime Investigation Department revealed that the suspects used a computer malware known as "ulssm.exe" to hack into the ATMs. This is also related to change in technological requirements - Windows no longer provide update to Windows XP but the software developers did not bother to test their software against their vulnerability of this new malware.

One of the other ways we can view these scenarios as a failure of risk management. In short, the SQA is a **risk management strategy.**

What is Software Quality?

In context of Software Engineering, Software Quality is mainly related to two notions:

- Software functional quality: how well a software conforms to the given functional requirements. In other words, it is the degree to which the correct software was produced.
- Software non-functional quality: It supports the delivery of functionality requirements such as maintenance, robustness etc.

SQA as a risk mitigation strategy.

Risk, and risk management, is a concept that will come up repeatedly in software engineering units. Software quality assurance is, as we've said, a tool to mitigate risks related to software projects. So, it's important to understand what risk is!

This is how the IEEE SWEBOK [1] introduces the concept of risk:

> *Risk and uncertainty are related but distinct concepts. Uncertainty results from lack of information. Risk is characterized by the probability of an event that will result in a negative impact plus a characterization of the negative impact on a project. Risk is often the result of uncertainty. The converse of risk is opportunity, which is characterized by the probability that an event having a positive outcome might occur. Risk management entails identification of risk factors and analysis of the probability and potential impact of each risk factor, prioritization of risk factors, and development of risk mitigation strategies to reduce the probability and minimize the negative impact if a risk factor becomes a problem. Risk assessment methods (for example, expert judgment, historical data, decision trees, and process simulations) can sometimes be used in order to identify and evaluate risk factors.*

In English, risk factors (or, informally but more commonly "risks") are things that *might* happen to a software project (or, more broadly, any project) that would be bad. In risk management, you figure out what those things are, and estimate:

- How likely they are.
- How bad the results are.

Based on this analysis, you can then decide what to do about each risk. Depending on the analysis, the right thing to do can be everything from *"note it and do nothing"* to *"have a plan of what to do if the risk becomes a problem"*, to *"spend a great deal of time and money to reduce the probability that the risk occurs to zero"*.

There is a great deal more to say about risk and its management, but that is a topic for FIT2101 and your project units in future years.

Software quality assurance is a way of reducing some of the risks associated with software development. To discuss this, however, we need to define some terminology before moving further- *artifacts* and deliverables.

Artifacts & Deliverables

In a software development project, the tangible (or at least identifiable) outputs are called artifacts. Informally, anything you can save in a file and put in a project repository - or gets auto-saved on your online tool - it's an artifact. Artifacts are not restricted to source code. There are a huge variety of non-source code artifacts, for example:

- Requirement documents
- Design documents
- Process document reports
- Meeting minutes
- Even chat logs can be viewed as an artifact - though it is unlikely you'd edit them afterwards as SQA!

Some artifacts are delivered directly to a customer[1], and these are called deliverables. Executable software is a deliverable, but it is not the only one. User documentation, for instance, is deliverable. Requirements documentation, if it is viewed by a customer, is also a deliverable. Even process artifacts like project schedules are deliverables if a customer will see them.
At a very high level, software quality assurance is designed to mitigate risks that artifacts are not of sufficient quality. We have not defined *"quality"* yet - believe me we will - but for now you can think of "insufficient quality" as "not good enough".

In 1983, during a particularly tense period in the nuclear-armed standoff between the Soviet Union and the United States, a Soviet warning system misinterpreted light reflecting off clouds as an American missile launch [2]. Luckily, the Soviet duty officer had a cool head, and waited for additional confirmation of the launch before retaliating, which never came. But a buggy system for interpreting satellite data came extremely close to causing a nuclear war.

Software Quality - A formal definition

In the preceding section, we presented a working definition of "*software quality*" as being *"good enough"*. But whether a piece of software is *"good enough"* is actually quite a subtle and complex concept, because the software can be *"good"* or *"bad"* in a number of non-obvious ways.

The IEEE standard for software quality assurance [3] defines software quality as:

> *The degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations.*

[1] A "customer" in this context may be a paying external customer, a product manager within your company - pretty much anyone outside the software development team and particularly if they are not IT professionals.

It is both unfortunate and ironic that the definition of software quality in a standard descripting software quality assurance is of such dire need of quality assurance being applied to it.

What the writers of the standard are trying to *actually* say is something like this:

- Quality software is software that does what the customer needs.
- However, the fullness of what the customer actually needs, and the requirements that actually get written down, may be two different things.

So quality is about meeting both ***stated*** *and* ***implied*** customer needs.

Part of the reason why we must consider both stated and implied needs is that experience has shown us that it is extremely hard to state up front, everything that a software product will ultimately be required to do! Even well managed projects will have many requirements changes. Therefore, while quality is about the behaviour of software on initial delivery, it also encompasses how that software is able to change into the future.

Quality in use - or why the same system can be both high and low quality

A Digression - Robert's Impreza WRX

When Robert turned 25, he bought himself a new car:

It had a turbocharged engine, a spoiler on the back, fat tyres (by the standards of the day), a manual gearbox for maximum performance, and constant four-wheel drive. Oh, and it looked (something) like Possum Bourne's[2] rally car. As far as I was concerned, it was an extremely high-quality car.

[2] https://en.wikipedia.org/wiki/Possum_Bourne

Then a friend asked me to teach them to drive.

My car had a touchy accelerator, a heavy gearbox which was difficult to get into first gear, and a clutch pedal which required a lot of force and went from disengaged to fully engaged in a couple of millimetres.

As far as my friend was concerned, the quality of this particular car was awful.

As the digression above illustrates, **different users can have very different views of the quality of the exact same product**.

In the context of a software product, be aware that "users" can include people other than "end-users". It may include system administrators, or even the developers responsible for maintaining the system - and they may have a *very* different viewpoint on the quality of a system compared to end users.

This idea is formalised in the notion of "quality in use", defined in the ISO/IEC 25010 [4] standard based on work by Bevan [5].

> *Quality in use is the degree to which a product or system can be used by specific users to meet their needs to achieve specific goals with effectiveness, efficiency, freedom from risk and satisfaction in specific contexts of use.*

There's a lot more information about these in the standard, but in short, quality in use is measured by whether users can do what they want to do with the software, whether it uses a reasonably low amount of the resources they care about to do so, whether it puts them at risk in some way (be it physical, emotional, or financial). It also addresses their emotional reaction to the system - is it *pleasurable*? Is it *trustworthy*?

We cannot directly address whether quality of use is met for individual users. Instead, as developers, we work with groups of users/stakeholders to develop *product quality* goals, design, build, and perform quality assurance to check that the product meets them acceptably well. If we can do that, hopefully, enough of our users will find the quality in use sufficient. There is no guarantee - if users decide to use our spreadsheet as a database, it is going to be poor quality in that context - but it is the best that we can do.

Software Quality Attributes/Factors/Characteristics

A quality factor represents a behavioural characteristic of a system. Customers, software developers, and quality assurance engineers are interested in different quality factors to a different extent. For example, customers may want an efficient and reliable software with less concern for portability. The developers strive to meet customer needs by making their system efficient and reliable, at the same time making the product portable and reusable to reduce the cost of software development. The software quality assurance team is more interested in the testability of a system so that some other factors, such as correctness, reliability, and efficiency, can be easily verified through testing.

The product quality model from the ISO/IEC 25010/2 standard has a hierarchical set of quality properties that are designed for just this purpose.

Functional suitability is a degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. Following sub attributes come under functional suitability.

- Functional completeness: How complete is the software with respect to the functionality?
- Functional correctness: How correct is the software with respect to the functionality i.e. the agreement of program code with specifications?
- Functional appropriateness: Does the functionality look appropriate to the requirements provided?

If a software fulfils all three sub attributes it will consider fully functionally suitable.

Performance efficiency is a performance relative to the amount of resources used under stated conditions

- Time behaviour
- Resource utilization
- Capacity

Compatibility is a degree to which a product, system or component can exchange information with other products, systems, or components, and/or perform its required functions, while sharing the same hardware or software environment. A product can

- Co-existence, or
- Interoperable: Interoperability of one system to another should be easy for the product to exchange data or services with other systems. Different system modules should work on different operating system platforms, different databases, and protocol conditions.

Usability is a degree to which a product or system can be used by specified users to achieve specific goals with effectiveness, efficiency, and satisfaction in a specified context of use. A system should possess following properties:

- Appropriateness recognizability
- Learnability
- Operability
- User error protection
- User interface aesthetics
- Accessibility

This can be measured in terms of ease of use. The application should be user-friendly, easy to learn, simple navigability etc.

Reliability is a degree to which a system, product or component performs specific functions under specified conditions for a specified period of time that is under different working environments and conditions. We look at following properties while dealing with the reliability:

- Maturity
- Availability
- Fault tolerance
- Recoverability

In other words, if the product is reliable enough to sustain in any condition.

Security is a degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. Following properties lie under the security attribute:

- Confidentiality
- Integrity
- Non-repudiation
- Accountability
- Authenticity

System integrity should be sufficient to prevent unauthorized access to system functions, preventing information loss, ensure that the software is protected from virus infection, and protecting the privacy of data entered the system.

Maintainability is a degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers. For maintenance, following properties must be assured.

- Modularity: A software should have modules or divided into small components.
- Reusability: these components should be reusable across different products
- Analysability
- Modifiability: software can be customised easily as per the changes in requirements.
- Testability: suitability for allowing the programmer to follow program execution (runtime behaviour under given conditions) and for debugging.
- Extensibility: could be extended in adding more modules or into separate components.

Modular, well-structured programs prove more suitable for systematic, stepwise testing than monolithic, unstructured programs. Testing tools and the possibility of formulating consistency conditions (assertions) in the source code reduce the testing effort and provide important prerequisites for the extensive, systematic testing of all system components. Different versions of the product should be easy to maintain. For development it should be easy to add code to the existing system, should be easy to upgrade for new features and new technologies from time to time.

Portability is a degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another. Following properties must be conformed:

- Adaptability
- Installability
- Replaceability

The ease with which a software system can be adapted to run on computers other than the one for which it was designed. A software system can be said to be portable if the effort required for porting it proves significantly less than the effort necessary for a new implementation.

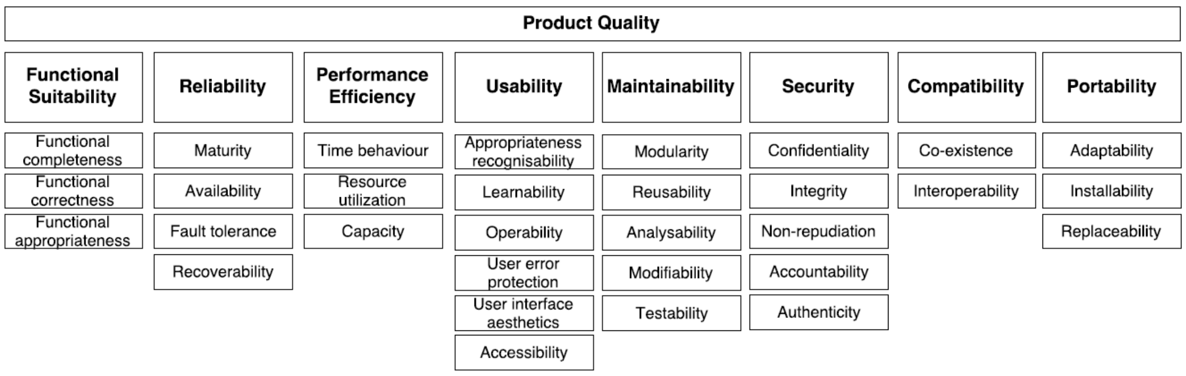


Figure 1: Quality Attributes as per the ISO 25010 standard

Requirements documents very rarely explicitly cover all of these in detail. Maintainability particularly, is rarely obvious to the customer, is very difficult for a non-technical customer to even specify accurately and is rarely obvious at the initial delivery of the system. But nevertheless, it is extremely important over the life of most software projects.

Not all of these properties are equally important in all software projects. In some projects some will not be relevant at all. But in most projects all of them apply to some extent, and for a project to be of sufficient quality it needs to be of sufficient quality on all the relevant properties, whether they have been fully and explicitly specified or not.

Incidentally, one of the quality properties of software requirements documentation is whether they specify in sufficient detail the quality requirements in the relevant categories listed above.

During the life cycle of a project, but particularly early on, this quality model is a good way to frame consideration of what quality properties are important for your project. The details of how you organize that discussion will depend on your process model, but you will need to determine (and document - see the next chapter!) what you care about for your project, and how you will assess your project as it proceeds to determine whether those quality goals are achieved.

Assess software quality to ensure goals are met? That sounds an awful lot like…
Now you know how to measure the quality of software using attributes – let's move to the Assurance.

Software Quality Assurance

Informally, software quality assurance is how you measure your software project to ensure that the quality is up to snuff. More formally, here is the IEEE SQA [6] standard to the rescue with another definition:

> *A set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes. A key attribute of SQA is the objectivity of the SQA function with respect to the project. The SQA function may also be organizationally independent of the project; that is, free from technical, managerial, and financial pressures from the project.*

Again, it is not going to win the Plain English awards! The key point you should take out of this definition is that it is not just about ensuring that the *product* is of sufficient quality, it's about ensuring that the *software processes* are appropriate.

We have another term to define if you have not already come across it - *software process*. In units like FIT1003, FIT2101, FIT3065 and FIT4002 we cover the concept of software process in a great deal of detail, but for now a software process is for software what a recipe is for cooking. It is how we get from our inputs to our outputs. There are lots of different software processes, and they can be specified at many different levels of detail. A process can be a high-level overview of how the software is to be developed, or cover just some aspects - for instance, how we convert design documentation into code. A high-level overview process is also sometimes called a software development life cycle (SDLC).

The IEEE SQA standard goes on to identify three main parts of SQA:

a) *SQA Process Implementation – A strategy for conducting software quality assurance is developed. Software quality assurance activities are planned and executed. Evidence of software quality assurance is produced and maintained.*
b) *Product Assurance – Adherence of products to the established requirements is evaluated. Problems and non-conformance are identified and recorded.*

c) Process Assurance – Adherence of processes and activities to the applicable standards and procedures is verified. Effectiveness of processes is evaluated, and improvements are suggested. Problems and non-conformance are identified and recorded.

So, quality assurance ensures checking that the:
- *product* is of sufficient quality.
- *process* by which the product has been built is of sufficient quality.
- that the process of checking the first two things have been done is sufficiently comprehensive and sufficiently documented.

Performing quality assurance on your quality assurance process brings up an interesting issue - do you then need quality assurance on the QA of your QA? And QA on your QA of your QA of your QA? At some point, you have to call a halt. Most projects do not bother with extensive QA process on their QA, and even the largest, most safety critical projects would only do so to one level!

Projects using Agile methodologies tend to be less considerate about process QA, because one of the key tenets of Agile software development methodologies is that software is delivered to clients as soon and as regularly as possible. When you have an evolving, working product to QA, process QA is of secondary importance.

Covering *everything about SQA* is impossible in one unit, so we are not going to try. We are mostly going to concentrate on **artifact quality assurance**. Artifacts can be about the process as well as the product, of course! We will leave most aspects of process QA to other units.

Software Verification & Validation

Another pair of frequently used terms in the SQA literature are "Verification & Validation", often abbreviated as V&V.

In English, Barry Boehm described the difference as follows:

Validation: Are we building the right product?
Verification: Are we building the product right?

That is, validation is primarily about whether the system as a whole is fit for its purpose - which may or may not mean that it meets the documented requirements. Verification is about whether the outputs of some part of the software development process meet the requirements specified by that part. For instance, if the software development process we are using has us write a detailed design document covering each class - verification of the code implementing that class would involve checking whether the code actually does what the detailed design document says it should do.

So - what artifacts can we verify and how?

In short, *any* artifact can be verified, though the methods can and will vary. One way to look at it is to divide artifacts into *informal, formal,* and *executable*. Please note that this terminology is something I have coined; this is not something from a standards document.

Anyway, ***Executable artifacts*** are source code: that is, they can be run, and they do something. These can be *tested* - that is, they can be run and their behaviour observed. This is unique to executable artifacts. Executable artifacts are a subset of formal artifacts.

Formal artifacts are those that have a rigorous, unambiguous definition of their syntax (the notation used to make them) and their semantics (what that notation means). Formal artifacts include source code, but also includes non-executable artifacts such as formal specifications. You can learn about these in more detail in FIT3013. Formal artifacts can be verified, in part, using mathematical techniques such as *theorem proving* or model checking. For instance, in some cases you might be able to prove formally that regardless of the input to a method, the method will always return in a finite time.

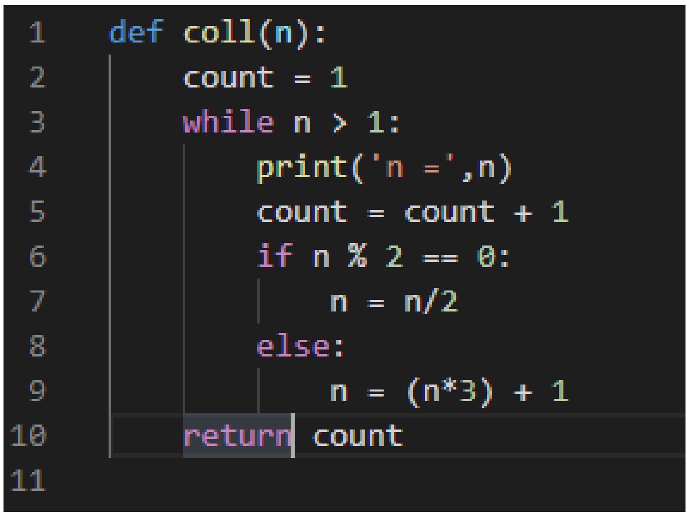
Informal artifacts include everything else. These cannot be tested by running them, and they cannot be formally proved correct. In these cases, the process of verification is manual. *People* must check that they are satisfactory. Over the years, systematic techniques for organizing these checks - reviews, walkthroughs, and inspections, have been developed, and can be and are applied to all types of artifacts, not just informal ones.

Formal verification is a topic for another unit (FIT5170). In FIT2107, we will spend a little time on reviews and walkthroughs, and most of our time on executable testing.

Limits of SQA

No SQA technique can remove all risk.

Example 1: Consider this Python method that prints the Collatz Sequence for an integer n:



```python
def coll(n):
    count = 1
    while n > 1:
        print('n =',n)
        count = count + 1
        if n % 2 == 0:
            n = n/2
        else:
            n = (n*3) + 1
    return count
```

The detailed design documentation for `coll(n)` says the following:

> *...the method should always return in a finite period of time...*

Unfortunately, no SQA method on Earth can guarantee us this.

Informal review does not tell us anything useful. Furthermore, formal mathematics has not helped much either. Mathematicians have examined this problem (the Collatz conjecture) in great detail. While for every number we have tried, it returns in a finite time, but we cannot prove it returns in a finite time for all integers. We have not been able to prove the alternative - that there exists a number for which it will not return in a finite time - either.

But what about testing? Could we test with all possible values? Well, in Python 3 you cannot. There is no fixed maximum integer size in Python 3 - it is determined by the maximum available storage.

While this is a slightly contrived example, it serves to illustrate a more broadly applicable point -SQA cannot give absolute guarantees.

Furthermore, the level of confidence we can give is dependent on the amount of time and effort we can put into SQA. For instance, the Sel4 project at NICTA[3] proved that an operating system kernel, written in C, implemented a specification written in formal notation. This isn't an absolute guarantee that the system does the "right" thing, as the specification itself may not reflect what the users actually wanted the system to do, but it's a better approximation than just about any other piece of software. It was a major achievement for fans of formal verification. However, the difficulty of the task shows the limits of this kind of mathematical SQA.

The kernel was 8,700 lines of C. The manually written proof was 200,000 lines long and took 30 person-years to complete - that translates to producing 1 line of verified code per programmer per day! While we do not do risk quantification in this unit, it should not be too hard to understand that that level of SQA is unaffordable and unjustified for most software projects. However, there are software projects where this kind of SQA effort is justified. For example, where the consequences of a bug are a plane crash, you'd want to make sure your software was highly reliable!

Given that we cannot provide absolute certainty, all we can aim for is to use the right SQA techniques, in the right amounts, to reduce the risk of poor-quality artifacts to "acceptable" levels for a given project.

[3] NICTA is the old name for what is now Data61, part of CSIRO that does IT-related research. The website for the Sel4 project is at https://sel4.systems/

To achieve this, first we need to analyse the nature and consequences of the risks for the particular project we are working on. Then we need to devise and document a quality plan that achieves the best balance between risk mitigation and cost. We can then apply the techniques in that quality plan.

It would be nice if we had a magic formula into which you could put information about a project, and have it told you exactly what type and how much SQA you should apply. Software engineering, unfortunately, is not sufficiently mature as a discipline that we have the evidence base to come up with such a formula. However, over time, research and experience has taught us some things about how to go about performing SQA, and we will try to pass them on in this unit.

Who does SQA?

Everybody involved in a software project has a responsibility for the quality of that software. But, particularly as a project gets larger and the work gets more specialized, different people, often with different job titles, will take on different parts of the SQA.

One view, expressed in the classic 1970s textbook The Art of Software Testing by Glenford Myers [8], was that programmers should never test their own code. In this world SQA is performed either by another programmer on the team, or a separate SQA team.

In more recent times that view has been rejected, particularly in the agile development community where programmers are encouraged to write their own tests as part of the design process for code. Test-driven development takes this to the extreme, where programming is done by writing a test, ensuring the current code fails the test, and then writing the code to make the test pass, and repeating until the module is completed.

While most developers have come around to the view that testing one's own code is a good idea, there is still a job for dedicated testers or SQA professionals. Testers in modern projects tend to concentrate on testing the entire system at a user-facing level, and SQA specialists keep an eye on the entire project to ensure that QA is being carried out according to plan, as well as assisting in SQA activities.

Summary

1. SQA is about ensuring the quality of both the process and the product.
2. Quality is multifaceted, not just about delivering functionality or "not crashing".
3. Lack of quality in artifacts is a risk, so SQA is a risk mitigation tool.
4. SQA techniques can be applied to all artifacts in a software development project.
5. Software testing is one family of SQA techniques applied to executable artifacts.
6. The scope and methods of SQA depend on the project and should be based on analysis of risk.

7. SQA is everyone's responsibility, but different people will have different SQA tasks depending on their part of the project.
8. Error, Fault and Defect are the terms which are used in Software Quality and testing with minor differences that are an error is a human mistake, defect is an undesired behaviour and fault is caused by a defect.
9. Verification is about building the product right and the validation is that we are building the right product?

References

1. P. Bourque and R.E. Fairley, eds., Guide to the Software Engineering Body of Knowledge, Version 3.0, IEEE Computer Society, 2014; www.swebok.org.
2. https://web.archive.org/web/20060819033034/http://hnn.us/articles/1709.html (NB: the article, entitled The Nuclear War that Almost Happened in 1983, is about half way down the page).
3. IEEE Standard for Software Quality Assurance Processes," in IEEE Std 730-2014 (Revision of IEEE Std 730-2002), vol., no., pp.1-138, June 13 2014.
4. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en
5. Bevan, Nigel. "Quality in use: Meeting user needs for quality." Journal of systems and software 49.1 (1999): 89-96.
6. 730-2014 - IEEE Standard for Software Quality Assurance Processes. 10.1109/IEEESTD.2014.6835311
7. Software Quality Engineering Testing, Quality Assurance, and Quantifiable Improvement by Jeff Tian.
8. The Art of Software Testing, Myers, G.J. and Sandler, C. and Badgett, T. 2011