Professionalism FURST

A summary

Paul Lynham FIAP

Introduction

If we needed a lesson in the disasters that arise out of poorly produced software, then we need to look no further than the Post Office scandal through the use of the Horizon system. Sub-postmasters were charged with false accounting and defrauding the Post Office, even though external reviews had shown the software had serious faults. In recent years this scandal has received a great deal of attention and an official enquiry has been set. Many of the convictions have already been overturned, but the vast number of criminal convictions resulted in false confessions, imprisonments, defamation, loss of livelihood and property, bankruptcy, divorce, and even suicide. The Post Office is likely to face lawsuits for malicious prosecutions, as they allegedly knew many of the victims were not to blame. Unfortunately, the Post Office may not have the money to pay the compensations that are likely to be awarded and so the taxpayer may be footing the bill!

Trustworthy Standards

How can we be assured that software is trustworthy and what can be done to prevent software from causing such disastrous problems? To be trustworthy, the problem domain and the software produced for it must be risk-assessed and managed. The facets of trustworthiness that need to be considered are safety, reliability, availability, resilience, and security.

The British Standards Institution (BSI) BS 10754-1 (which came out of work by Trustworthy Software Initiative) define the overall principles for effective trustworthiness but does not specify the detailed processes or actions that an organisation needs to follow. The IAP's John Ellis and Luc Poulin are working together within the Trustworthy Software Foundation[1] (TSF) with other members of the BSI to expand BS10754 to be more usable by businesses. TSF has a framework[2] that can be used to assess and determine the level of trustworthiness that needs to be applied to a project. In 2020, TSF became a subsidiary of the IAP.

FURST

To be of value, software must be produced on time, so it can properly address the issues it was designed for but it must also be compliant with standards and be open for modifications in the future. Of course, these abilities need to be verified.

A set of principles that can be used in producing trustworthy software is FURST. This stands for **F**it for purpose, **U**nit tested, **R**eviewed, **S**tandard, and **T**imely.

Fit for purpose (FFP) - does what it is supposed to do precisely

Unit Tested – it can be relied upon (doesn't exclude other testing such as BDD)

Reviewed - all artefacts checked e.g. code and designs are well-written and easy to understand

Standard - complies with all coding, UI, security, and other applicable standards

Timely - produced in time

As an analogy, the production of a screen used as a component in a computer, laptop, tablet, or mobile phone can be used in understanding these principles.

A screen needs to be built (and certified) within a clear time limit for it to be profitable for the manufacturer. It also has to meet certain requirements such as its fidelity in producing displays, colours, brightness, and contrasts, etc. as well as other non-visual abilities and standards to comply with, such as power consumption, safety and its ability to be recycled. The manufacturer will employ a quality engineer to review these requirements and will carry out a series of tests using a test bed and diagnostic equipment. The testbed and diagnostic equipment are never delivered to the end user, but without them, it would be uncertain that the product meets all the requirements precisely and without deficiencies.

Fit For Purpose

Software that does what it was intended to do precisely, without any unintended side effects can be said to be fit for purpose (FFP). This is not as simple as it first appears, as not only the functional requirements need to be met or exceeded but also the non-functional requirements. These latter abilities include its reliability, security, and how efficiently it performs.

FFP equates quality with the fulfilment of a specification. How accurately the requirements meet the specification is known as its fidelity. However, unless software is to die, it will inevitably need to change in the future (see **Unit Tested** in the next section for further details) and so its flexibility or openness for change becomes a part of its fitness. This openness to change depends on many factors, such as its design and structure and how easy the source code is to read and update.

This gives us two perspectives[3]. An external perspective, often of the users, looks at how well the software does what it is supposed to do (running software), while an internal perspective is concerned with the quality of the design and implementation and its ability to be updated (static view).

Many factors affect both the external and internal perspectives, with the author writing extensively about the latter. Further details can be gained from the IAP website, by navigating to 'Our Work' and then 'FURST' to see a list of articles on this subject.

United Tested

To ensure that software meets its requirements, it is tested at different levels. The whole application or system is tested as part of the quality process to ensure the running software meets all requirements with no faults or deficiencies (high-level testing). However, although this stage in the development cycle is always carried out to one degree or another, other testing strategies can be employed to good effect, and the more that are used, usually, the better the result.

Unit Testing is where individual units of source code are tested to see if they are fit for use. This often comes down to specific routines in the code i.e. a very low level of testing. These tests are often placed in their own units or modules, so they can be separated from the functional source code, thus permitting the test code not to be compiled into or delivered with the finished product.

This level of testing is important for at least two reasons. The first is quite obvious - it allows the developer to assert that the routine (or smallest unit) works as fully intended, with one or more tests for each routine. A routine can be run with different parameters to ensure the expected results are observed and so tests that check conditions such as unacceptable values or boundary conditions, can be used to see if they are properly handled.

In time, the requirements of the software will change. New standards arise, government policies, legal changes, or user expectations, all of which mean the software needs to be updated. With most applications, if it is not updated in the long term, it dies.

Therefore, the second reason why unit testing is important is that these tests can be run individually or as a suite to determine that the software still works with each change. This gives confidence and assurance to the developer to make the necessary changes to the source code without fear of breaking what is already there – like the Hippocratic oath 'First do no harm'.

Reviewed

Reviews can be carried out on artefacts at different stages in the development process including ones for requirements, architecture, design, implementation, and testing. For example, in testing the test plan can be reviewed as well as any test scripts and test data. However, in this brief discussion, we will look at code reviews.

Usually, more eyes on the code means fewer bugs and incentivizes higher quality – coders do not want to be picked up on silly mistakes or sloppy code. If they have a professional attitude, they want the reviewers to be impressed with their quality code i.e. code that is easy to understand, is clean, appealing, and efficient.

One of the biggest advantages of code reviews is the number of faults and deficiencies found before the code goes into testing or production – the earlier a bug is found the easier and the cheaper it is to fix[4]. A good review will find more bugs than system testing[5]. It also has many other benefits, with perhaps the learning opportunity it provides (for both the reviewer and author) being the next biggest benefit.

Software developed using pair programming or mob programming also benefits from being continually reviewed, so not only are requirements interpreted in an agreed manner (if not, the requirement needs reviewing) but the design, the code, and unit tests are reviewed as the code is produced. This type of review (although informal) is the closest to production, so is the quickest and cheapest way to debug software and ensure higher quality.

Standard

Standards are useful for many reasons, although there may be many to choose from! They may be laid down by the organisation producing the software as well as external ones that need to be followed, laid down by either best practice, external agencies, or the customer.

The ISO has defined standards as:

> *"documented agreements containing technical specifications or other precise criteria to be used consistently as rules, guidelines, or definitions of characteristics, to ensure that materials, products, processes, and services are fit for their purpose."*[6]

Some examples of standards used in software development are:

- Coding and formatting standards
- User and application programming interface standards
- Systems trustworthiness (BS 10754-1:2018)
- Security standards
- Testing standards
- Technical standards e.g. protocols and data format

Taking coding standards as an example, the use of standards maintains consistency throughout the code base, determining factors such as how code is laid out, how it is documented, naming conventions, and the use of exception handling, to name a few. When these standards are followed, 'blots on the landscape' are avoided and code becomes easier to read and understand. Since code is read many more times than it is written, this helps speed up the development process.

Timely

Time is of the essence – even if quality is high. It usually has a major bearing on the viability of the project and many projects have failed because they have overrun. It may affect the profitability of the organisation or its operational success.

Depending on the development process being used, it is sometimes possible to get a minimum viable product out (a product with just enough features to ensure it is useable) and then work on the remaining features, while gaining useful feedback from the users. Agile methodologies are supportive of this latter approach, but other software may need to be completely finished before it can be deployed.

The success of meeting time deadlines is dependent on several factors. These include the accuracy of the estimation of the effort, the recording of work completed, and the efficiency of the development team. The latter is dependent upon team leadership and the productivity of individual team members. Individuals' productivity is affected by their own standards and is also affected by their working environment and the culture of the organisation they are working for[7]. In reality, this last factor may have the greatest influence.

Therefore, techniques that aid time management and efficiency will have an effect on achieving time goals. One example that can be effective for developers is the use of the Pomodoro Technique[8] and another the Personal Software Process[9]. Depending on the development process, several techniques can be used to improve team dynamics[10]. However, changing the culture of the organisation is more difficult[11].

Bringing It Together

All these principles may seem daunting and you may wonder how they can be applied to a project? There are several concerns and these arise at different levels within the organisation, from the strategists, and highest managers, right down to members of the development team.

The commitment to produce trustworthy software at each level is important and it could be argued that it needs more than voicing but committed to words in statements of vision, missions, or goals, and demonstrated, so there is no doubt in anyone's mind that this is what is at the heart of what is being produced.

Compliance can be achieved using various tools and these can be linked into 'tool chains' so that one operation or process automatically kicks off another. Gates within the process check that procedures have been completed before the process can continue.

Over the years, the author has seen build tools evolve from simpler implementations that compiles (if necessary) and runs the application, perhaps running some tests and giving feedback, either via email or by a tray icon, etc. These tools have now been highly refined and cover the complete cycle, integrating various tools and development languages, and virtual machines.

As an example, a tool may be used for planning, say an agile Kanban-type tool where stories are curated, tasks pulled out onto a sprint, and assigned to a development team member. A programmer could select a task, with a feature branch in the repository being automatically created and, in some cases, the source code automatically downloaded onto the developer's machine. After the task has been completed the code is submitted, the Kanban app is updated with the current state of the task and reviewers are notified that the code for this task needs to be reviewed.

Some tasks may be automatically kicked off such as applying a standard format to the code, running an audit (such as checking no unauthorised libraries are being used), static analysis carried out, and the software run on a virtual machine, with tests run against it and the results posted. This latter step in itself can be quite complex, as the platform can be configured (hardware, OS, dependent software installed) and the machine automatically built and spun up, so any tests are consistent and are not dependent on any settings or files present on the developer's machine.

At this point, the developer can do no further work on this task until the review has been completed (or changes that come out of the review have been applied). When the review process has been passed the code can be merged (if required), and the application built and placed in an appropriate test environment. However, it can be seen that there are lots of tools strung together using scripts and triggers with applications, repositories, and databases updated and on and on.

Such tool chaining is great for automating compliance, assessing the software, and verifying that all required procedures have taken place. However, setting up and configuring such tools may not be straightforward and often requires a specialist DevOps member, rather than allow developers to spend an inordinate amount of time trying to configure such applications themselves.

With the advent of more sophisticated AI tools, quality can be improved while the software is being created e.g. an application running in the background while the programmer types the code could check for typos, errors or side effects, picking up things that the inbuilt development environment language interpreter or compiler may not pick up on.

Professionalism

Although many people from different walks of life write software, the key distinction is professional software is intended for others (rather than the developer themselves), and teams more often than individuals usually develop the software. In these circumstances, the software is maintained and updated throughout its life.

The attitude of the software developer is very important, both in terms of their willingness to learn new languages, tools and techniques, etc. but also their ethical outlook, so they treat the information they receive with the correct level of privacy and security. Their attitude and behaviour to both the organisation and their team are also important if they are to be highly productive. Of course, this responsibility goes both ways, so both other team members and the organisation need to encourage and provide developers with the opportunities to fulfil these ideals.

The IAP has a Code of Conduct[12] that all members must abide by and this sets out in clear terms what is expected of a professional software developer. There has also been debate about whether a Hippocratic Oath for Software Developers[13] should be created, based upon ones used in medicine.

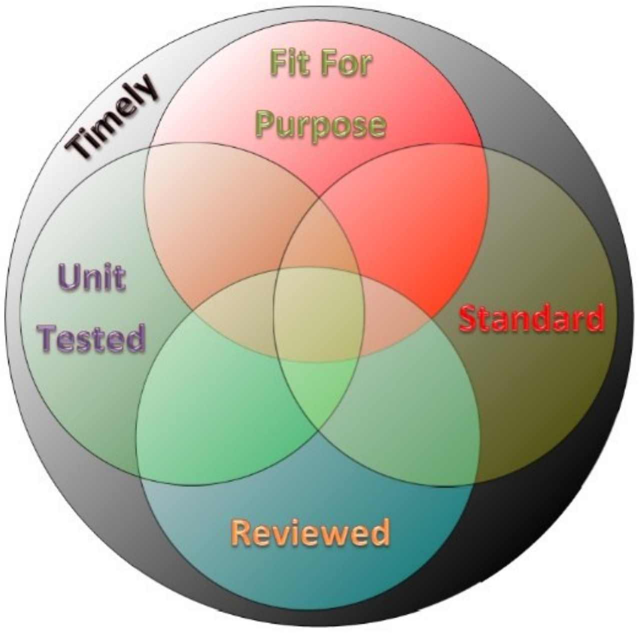
Conclusion

It can be seen that there are ways in which trustworthy software can be built – The TSF has a framework to help and BSI has the BS 10754-1 standard. However, at its most basic level, those involved in its development must be highly professional and highlight concerns about any issues that may impinge upon the trustworthiness of the software being produced.

There must be a commitment by the organisation producing the software. This needs more than a box-ticking exercise but an obligation to embrace this ethos and to clearly demonstrate this.

There are clear principles that can be followed that will allow trustworthy software to be produced and the FURST acronym has been briefly summarised to mark these out at a high level. Each of the principles within FURST is not difficult to comprehend and to many, they will make perfect sense.

It is true, that there is more to each principle than meets the eye, but this is nearly always true when something is useful. However, there is also a layering effect between these principles – they do not exist in pure isolation but feed off each other and help each other, as shown below.



As an example, **standards** help the software meet its obligation to be **fit for purpose**, they set out guidelines for **unit testing** and can be used for compliance within a **review**. Using existing standards, rather than trying to reinvent the wheel saves **time**, bringing familiarity and consistency.

By professionally utilising FURST, the trustworthiness of the software can only be enhanced, and those involved in its production can feel satisfied that they have done a good job.

[1] https://www.tsfdn.org/
[2] https://www.tsfdn.org/ts-framework/
[3] https://wiki.c2.com/?InternalAndExternalQuality
[4] https://www.functionize.com/blog/the-cost-of-finding-bugs-later-in-the-sdlc
[5] https://kevin.burke.dev/kevin/the-best-ways-to-find-bugs-in-your-code/
[6] https://www.iso.org/obp/ui/#iso:std:iso:tr:19300:ed-1:v1:en
[7] https://www.codurance.com/publications/why-is-culture-so-important-in-software-development
[8] https://francescocirillo.com/products/the-pomodoro-technique
[9] https://www.win.tue.nl/~wstomv/quotes/humphrey-psp.html
[10] https://www.agileconnection.com/article/7-ways-change-culture-devops-success#:
[11] https://cyclr.com/blog/change-organisational-culture-for-digital-transformation
[12] https://www.iap.org.uk/main/about/code-of-conduct-for-members/
[13] https://queue.acm.org/detail.cfm?id=1016991