More FURST

Looking deeper

Paul Lynham MSc PGCE FIAP

Abstract

FURST was devised to summarise the simple ideas used in *Clean Code*[1] and *Code Complete*[2]. The principles of FURST were further decomposed, so for each principle a list of attributes were formed. These attributes may have further attributes. Attributes can be linked to tenets that apply to clean code and good practices in general. Acronyms were developed to summarise each principle.

Introduction

An overview of FURST has previously been given in "Professionalism FURST"[3]. However, further detail is required to make this concept of value. In particular, each principle of FURST needs to be examined to reveal its attributes and how these relate to the whole process.

Although a professional software developer can say that the software he or she produces should be **Fit For Purpose** (FFP), have **Unit Tests**, be **Reviewed** by one or more developers, comply with **Standards** and be produced in a **Timely** manner, these are general principles. As stated in "Professionalism FURST", asserting these principles shouldn't be difficult except for FFP.

Fit For Purpose?

In order to address the subjectiveness of FFP, it is good to establish what it is exactly. Here are several definitions that sum up the idea:

> *Something that is fit for purpose is good enough to do the job it was designed to do.*[4]
>
> *Appropriate, and of a necessary standard, for its intended use.*[5]
>
> *Fitness for purpose equates quality with the fulfilment of a specification or stated outcomes. Fitness for purpose has been a widely used approach by quality agencies. The notion derives from manufacturing industry that purportedly assesses a product against its stated purpose. The purpose may be that as determined by the manufacturer or, according to marketing departments, a purpose determined by the needs of customers.*[6]

The first two are good at summing up the concept in one short sentence. The latter definition equates quality with specification.

QUALITY

In an effort to firm up attributes of FFP, the acronym QUALITY could be used, making use of some of the underlined words in the FFP definitions:

Q **Quality**. It is important to have the correct balance of quality, although there are many aspects of quality, such as external (what the users see) and internal (the structure and form of the artefacts, such as source code). Too little quality is not acceptable, while excess quality (e.g. over engineering) may have an impact on cost and time.

U **Understandable**. The software should be easily understood, both from the point of view of customers using the software as well as coders reading the source code and other artefacts.

A **Acceptable**. The software must meet its stated outcomes and be acceptable to both the user and developers. This can be affirmed by acceptance criteria and code reviews.
Appropriate. The solution should be appropriate for the purpose i.e. neither too elaborate nor too brittle and inline within any organisation policies e.g. use of frameworks.
Attractive. Even though the software may be functional to the user, it is important that the design and code is attractive to developers.

L **Labelling**. To the user, the software uses familiar language, while for the developer this means ensuring that identifiers and methods etc. are well named within the source code.

I **Intention**. The software must work as intended and the code should be plain in its intention; there should be no doubt when reading the code of what it does when it runs.

T **Testable**. Coders should be able to explain and document how the features they have implemented can be tested by testers within the team and by the QA department.
Transparent. The aim should be to allow no place where bugs or malicious intent can hide.
Trustworthy. Being testable and transparent will go some way to making the software trustworthy. Any standards that apply e.g. Security standards may need verification of compliance to ensure that the software is trustworthy.

Y **YAGNI**. Only features that are required to meet the acceptance criteria should be developed i.e. it is good enough to do the job. Extra functionality may be a waste of effort, the code may never be exercised and it may be a source of problems, especially adding unwanted complexity and size. As stated in *The Pragmatic Programmer*[7] - *"Don't spoil a perfectly good program by over embellishment and over-refinement."*

The concepts within FFP can be laid out using the QUALITY mnemonic, showing how that concept can be fulfilled for both the user and the developer. This is shown in table 1.

Using these aspects, it may be possible to achieve an objective view of whether the software is FFP. Looking at the aspects, the ones where a binary decision could be made include *Understandable, Acceptable*, *Appropriate*, *Labelling*, *Intention*, *Testable*, *Transparent*, *Trustworthy* and *YAGNI*. Putting aside *Quality* for the time being, *Attractive* may be one where it may be slightly harder to quantify.

From the user's perspective, QA will decide if this passes. From the developer's perspective the code review could decide if the code is attractive, perhaps with the help of guidelines or check boxes. Thus the remaining aspect is the *Quality* balance.

Like some well-known mnemonics (e.g. GNU), *Quality* is recursive within the QUALITY mnemonic and this may flag up the question, can *Quality* ever be resolved? It could be argued that it can be resolved by 2 factors.

The first is the overall emphasis of the project. A project may have a mission statement or some other guiding principle which will determine how much effort is to be expended on it. Steve Maguire[8] recommends:

"Establish coding priorities and quality bars to guide the development team."

This is also echoed in *The Pragmatic Programmer*:

"The scope and quality of the system you produce should be specified as part of that system's requirements."

In mission critical software the quality factor is going to be high, whereas at the opposite end of the scale a throw away piece of software with a short lifespan may have a low quality factor. This is debatable, as some would say that quality must always be in the forefront while others may be more pragmatic, hence the subjectiveness.

Table 1. QUALITY aspects for users and developers

QUALITY	Concept	User Aspect	Developer Aspect
Q	Quality	External quality - . look and feel . performance . documentation	Internal quality - . architecture . design . code
U	Understandable	Easy to understand and use the software	It is a simple task to understand the design, the source code and other artefacts
A	Acceptable	It does exactly as required	. Passes acceptance criteria . Meets non functional and audit requirements . Code is well written and meets all mandatory standards . Passes QA
	Appropriate	. Features can be accessed and used in a logical manner . Information is presented in a meaningful way	. The design is an apt solution for the problem . Technoligies used are appropriate
	Attractive	The user interface is pleasing to use	The code is well written so any developer can understand and modify the code easily - . minimised duplication . no unused code . cohesive (self contained, independent, single purpose) . loose coupling etc.
L	Labelling	The software uses familiar language	Naming conventions . well named and self explanatory identifiers and methods . use names from the domain language
I	Intention	Should work as intended - . no surprises or unexpected behaviour . carry out tasks effectively	Code should be clear and plain in its intention - on the rare occassions where the code cannot be made self documenting, comments should be used for explanation
T	Testable	Can be relied upon to work correctly	The code is written so that it is testable Testing methods can be explained and documented
	Transparent		Code is clearly written so nothing is hidden
	Trustworthy	Can be relied upon to work correctly and securely	Code is written to prevent bugs, errors, unauthorised access and any other unwanted behaviour
Y	YAGNI	No superfluous or distracting features	Meet the requirements precisely

The second factor is the combined result of all the other aspects i.e. if U, A, L, I, T and Y have all been resolved positively, then it could be said that Q has been resolved!

There are further aspects of quality which may be of importance, depending on the focus of the project. These include Portability, Flexibility, Reusability, Size, Efficiency and Maintainability[9],

although this last aspect should be automatically covered if **QUALITY** is satisfied. An acronym **PARSED** can be used to represent these extra factors as follows:

P **Portability**. The ease of which a system can be modified to operate in an environment different from that which it was specifically designed for.

A **Accommodating**. The software is flexible to future changes i.e. the extent to which a system can be modified for use or modified for environments other than those for which it was specifically designed.

R **Reusability**. The extent to which and the ease with which, parts of the system can be reused in other systems.

S **Size**. There is a strong correlation between the size of an application and the faults found[10], therefore keeping the solution as small as possible will reduce defects. Size may also be a constraint in some environments.

E **Efficiency**. This is the minimal use of system resources including memory and execution time.

D **Defendable / Maintainable**. The ease of with which a software system can change or add capabilities, improve performance or correct defects.

An example where these aspects may be pertinent is with embedded systems. The software may need to be portable so it can run on different chipsets, it needs to be of a size so that it can reside on the chip and needs to be efficient with the limited resources.

For software that meets all attributes of FFP as well as the extra aspects, it can be said that it is **QUALITY PARSED**.

Unit Tested

At a high level, unit testing is the practice of testing routines and areas (units) of the code which demonstrates the ability to verify that these routines work as expected. For any routine and given a set of inputs, it can be determined if the routine is returning the correct values and will elegantly handle failures if invalid input is provided.

This helps to identify failures in the logic so as to improve the quality of the code that makes up a certain routine. As more tests are written, a suite of tests are created that can be run at any time during development to continually verify the quality of the code.

By taking this approach to development it means that code will be written that is easy to test. Since unit testing requires that code be easily testable, it means that the code must be written to support this. The consequence is that it is more likely to have a higher number of smaller, more focused routines that provide a single operation on a set of data rather than larger routines performing a number of different operations.

As a result of writing solid unit tests and well-tested code it means that future changes can be prevented from breaking functionality. Because the code is tested as the functionality is

implemented, a suite of test cases are developed that can be run each time the code is changed, thus when a failure happens, the coder knows where the error lies.

In the *Clean Code* book, unit tests should follow 5 rules. These are **Fast**, **Independent**, **Repeatable**, **Self-Validating** and **Timely** (FIRST)[11].

F **Fast**. Tests need to run fast so they can be run often. If they are not run often, the problems won't be found early enough to fix them easily.

I **Independent**. Tests should not depend on each other e.g. one test should not setup the conditions for subsequent tests. The test should be run independently and in any order.

R **Repeatable**. Tests should be repeatable in any environment. If the tests aren't repeatable in any environment, then there will always be an excuse for why they fail. It will also be found that tests are unable to be run when the environment isn't available.

S **Self-validating**. The tests should have a Boolean output. Either they pass or fail. There should be no need to read through a log file to tell whether the tests pass.

T **Timely**. The tests need to be written in a timely fashion. Unit tests should be written just before the production code that makes them pass. If you write tests after the production code, then you may find the production code to be hard to test.

Unit Tests are the minimum testing that is required. Other testing can be used for example behavioural testing frameworks such as *Cucumber* both tests the behaviour of the application as well as documenting how the application is supposed to work. These tests can be written just before the code is written (BDD). The point is that Unit Tests are the basic tests to confirm that the code works as expected at the lowest level and other tests written by developers are an added welcome bonus.

Reviewed

There are several ways of reviewing software including formal inspections and code readings, informal through pair programming or by use of code walkthroughs. There is even an IEEE standard.[12] A simple way is by peer review - other developers inspect the code as part of an automated quality procedure. As an example, when code is committed to the repository by a developer (in an agile process this may be on a story or task basis), one or more colleagues are notified. They must review the code (there are tools for this e.g. to compare differences and make comments) and the code must be accepted before the code can be merged into the current branch.

It is also makes sense that all other steps have been successfully carried out before the reviewer gets involved i.e. the code compiles and runs and the tests pass etc. so that time is not wasted at the review stage with code that won't make the product. The review could also be used to set a focus, such as compliance, code quality, efficiency or error detection etc.

There are lots of hard facts about reviews[13], so there are no doubts as to their benefits. Research has not only examined different review techniques, comparing each with error detection success, but has also found other insights such as the amount of time to devote to this effort to get the best pay back and has also provided insights as to how people review code and how likely they are to be good at finding errors. It has also found other facts that reinforce received wisdom, but can now put

quantities to these. As an example, it has long been accepted that routines that are smaller are easier to comprehend. However, it has been difficult to put a limit on the maximum number of lines in a routine (bearing in mind other rules such as 'it should do one thing only' and 'statements should all be at the same level of abstraction').

Many years ago, when code was read on paper more frequently, a rule of thumb was that a function should never exceed the length of an A4 piece of paper (around 60 lines of code). Examination of eye movements recorded during code reviews shows a large amount of time devoted to routine scanning. If a whole routine cannot be seen on the screen then the reviewer has to continually scroll up and down in order to reread parts of the routine. This hinders *chunking*[14], slowing down comprehension (and reviewing in general) as well as the flow[15] of thoughts. This research confirms the accepted wisdom of small routines are easier to comprehend, but can now put a limit on this – it is the maximum number of code lines that can be displayed in the IDE, code comparer or other tool that is used to review the code!

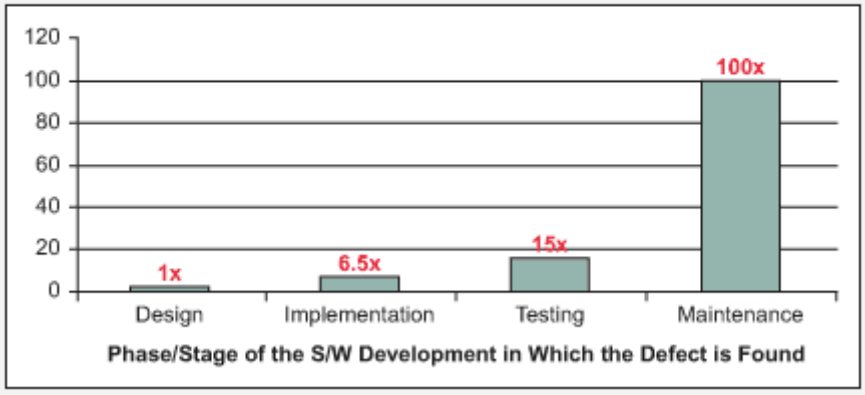
Taking error detection as one metric, it seems code reviews are at least 150%[16] more efficient at finding errors than QA. No sane organisation that produces software would ignore QA as a strategy, yet it is not uncommon to ignore reviews.

Reviewing software has other advantages besides error detection and if an organisation is not doing this, then they are missing a trick. The benefits of reviewing include:

1. Finding errors in the code.
2. Improving efficiency.
3. Increasing quality.
4. Enforcing compliance.
5. Disseminating best practice.
6. Teaching and mentoring.
7. Reduced load on Technical Support.
8. More customer satisfaction.

There has been a lot of research carried out that proves that the earlier defects are found, the easier and cheaper they are to correct, as shown in Figure 1. By finding defects at the code review stage, before the code is merged, defects are found ahead of going live and while the code is still fresh in the developers mind.

Figure 1: Relative Costs to Fix Software Defects (Source: IBM Systems Sciences Institute)



Coding errors also cause the majority of software vulnerabilities. As an example, 64% of the nearly 2,500 vulnerabilities in the US National Vulnerability Database in 2004 were caused by programming errors[17].

It would therefore seem blatantly obvious that reviews are an effective way to increase the quality of the software, saving money and increasing customer satisfaction.

Review Steps

The software review would cover the following steps:

- The code and tests can be checked to ensure that standards are being complied with.
- The tests can be checked to ensure that they are valid, the coverage is good and that they pass.
- The requirements and acceptance criteria can be checked against the implementation to ensure that the software does what it is intended to do. Any other tasks that need to be completed so that the *Definition Of Done* (DOD) is complete can also be checked.
- The code can be checked to ensure that it is understandable (comprehendible), it is compelling (it is persuasive - intention is plain) and it is attractive (appealing). A portmanteau has been created to combine the words *comprehendible*, *compelling* and *appealing* which is ***compealing***.
- Lastly, Linus' Law states that *"more eyes makes bugs shallow"* so the more people who review the code, the more likely that any unintentional behaviour, errors and omissions will be spotted.

Taking all the previous points into account, we can arrive at an acronym to summarise these, which is SAUCE.

S **Standards**. Whether internal or external, standards are checked for compliance.

A **Acceptable**. Any attributes of FFP that make the software acceptable and appropriate are checked, including only required features are present.

U **Unit Tested**. The code is confirmed to be suitably covered by valid tests that pass.

C **Compealing**. Any attributes of FFP that cover understandable, attractive, labelling and intention.

E **Errors**. The code contains no discernible errors, so FFP attributes trustworthy and transparent are confirmed.

When done correctly, code reviews actually save time in the long run[18]. The Review process is the backstop, ensuring that the product is FFP, Unit Tested and Standard.

Standards

What are standards? A commonly joke is *'standards are great as there are so many to choose from'*. However, joking aside, 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."*

Standards may be laid down by the organisation creating the software as well as external standards laid down by either best practice, external agencies or by the customer.

Standards can include:

- Coding and formatting standards.
- User interface standards.
- Security standards.
- Testing standards
- Technical standards e.g. protocols and data format
- Customer standards
- Legal standards e.g. Remote gambling and software technical standards[19]
- Other external standards such as those published by IEEE, IEC, ISO, ISOC or W3C.

Standards themselves may have standards! An example is the wording used in a standard so that consistency is kept in the meaning. The IEEE has standards for how its standards are laid out, the sections used and the words used to denote levels of compliance. These words are *Shall*, *Should*, *May* and *Can* and the definitions are taken from 2014 IEEE Standards Style Manual, which state:

- The word *shall* indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*).
- The word *should* indicates that among several possibilities, one is recommended as particularly suitable without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required (*should* equals *is recommended that*).
- The word *may* is used to indicate a course of action permissible within the limits of the standard (*may* equals *is permitted to*).
- The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals *is able to*).

The mnemonic **S**ei**SM**i**C** can be used in this case (similar to **M**o**SC**o**W** in prioritising requirements).

There are many motivations for following standards. The following is a list of ones that are applicable to a software project:

- Keep consistency across projects.
- Facilitate ease of maintenance.
- Ensure professional quality software.
- Guide staff and educate new developers.
- Follow best practice and received wisdom.
- Compliance with regulatory authorities.
- Mitigation in the case of legal challenges.

Coding standards

Taking coding and formatting standards as an example, the main benefit of adhering to this standard is efficiency. Such standards enable each member of a team to work anywhere within the source code without needing to recognise and adopt a different programming style. Just as importantly, any developer can look anywhere in the source repository with reliable expectations about how the code will be structured and how to locate what they need. In addition to this, maintenance, revision and shared use of code are simplified. Some pragmatists may say that it is better to follow a standard consistently rather than the details of a particular standard itself.

Coding and formatting standards may include:

- Comment conventions.
- Indent style conventions.
- Naming conventions.
- Programming practices.
- Programming principles.
- Programming rules of thumb.
- Programming style conventions.

Why they are useful

Conventions or practices can be general or can be language specific. An example of a naming convention could be:

"Constants should be in all uppercase with words separated by underscores" e.g.

```
MAX_STANDARD_HOURS_WORKED = 40
```

This is a common naming convention for constants and can be seen in the APIs provided by Windows. Whenever you read `MAX_STANDARD_HOURS_WORKED`, it is easy to say, ah, this is a constant. This may seem insignificant; however you have to consider the knowledge tied up in this fact. A constant in most languages is immutable. It can only be assigned to once and cannot be changed again. It is an excellent way of treating a common value that is spread across the codebase. It can be compared to looking at the 'national speed limit' road sign of the white circle with a black diagonal stripe. To a car driver in the UK this says *"if you are on a motorway or a dual carriage way and there is no other restriction, then the maximum speed you are allowed to travel at is 70mph otherwise it is 60mph"*. There is a lot of knowledge tied up in a seemingly insignificant symbol!

If however when reading the code, you read `max_standard_hours_worked`, are you sure this is a constant or is it a variable or something else? You would have to check and search for where this was declared or first used. This takes time and effort and in the meantime, you lose the flow and context! This proves that use of convention makes things easier to understand.

An example of a language specific convention could be:

"Use constants or literals on the left of the equality operator in C[20] and C++ where possible" e.g.

```
if (MAX_STANDARD_HOURS_WORKED == hoursWorked) {   //only enter this block if hoursWorked is 40
  . . .
}
```

The reason for this convention is because a common error for new users (or those that dip in and out of it) in C or C++ is to use = (assignment operator) instead of == (equality operator):

```
if (MAX_STANDARD_HOURS_WORKED = hoursWorked) {     //wont compile – saved by a convention!
  . . .
}
```

However, by putting the constant on the left it cannot be assigned to (because it is a constant) and the compiler will complain, whereas:

```
if (hoursWorked = MAX_STANDARD_HOURS_WORKED) {      //unintentional behaviour – can assign to a variable!
  . . .
}
```

In the latter case, the value of `MAX_STANDARD_HOURS_WORKED` will be assigned to the variable `hoursWorked` and because it is in a Boolean evaluation (`if`), it is greater than 0 and so will be interpreted as TRUE. In this case the block following the `if` will always be executed! If the convention is followed, it will cut out the mistake of using the incorrect operator. This is language specific as other languages such as Java and Delphi will not allow assignments within a Boolean evaluation (although some C/C++ compilers may have a compiler switch to stop this behaviour).

These examples may seem trivial but there are hundreds of examples where following a standard can prevent errors, keep consistency and make life easier for the developer and everyone else who reads the code, regardless of if it is a coding standard, security standard or any other standard.

Timely

To some, the **Timely** principle may seem out of place, as all the other principles in **FURST** are practices and some developers may think these are within their control. However they may consider time is not! This is not the case in terms of estimating. This is another skill that a professional developer should acquire and continually improve upon. Consider this statement from the *Pragmatic Programmer*:

> *"One of the cornerstones of the pragmatic philosophy is the idea of taking responsibility for yourself and your actions in terms of your career advancement, your project, and your day-to-day work."*

Estimating is something that developers are continually asked to do. In an agile environment, such when using Scrum, it is something that must be done in each sprint – items in the backlog are refined (grooming) and once in a state that they are clearly understood they can be estimated. Depending on the priorities, the stories can be moved from the backlog into the next sprint. Since the numbers of resources are known in advance, the stories that can be worked upon in the sprint will depend upon the estimate for each story, so the sprint is neither over nor under committed. In the sprint planning meeting, candidate stories are examined and depending on their priority and estimate are either pulled into the sprint or left in the backlog until the sprint is fully committed. Each member of the team contributes to the estimate of each story, so there is a perspective of BAs, testers, programmers, solution architects etc. Depending on the working culture, it is understood that an estimate is just that– an informed guess at how long it will take to complete the task.

A factor affecting estimates is a developer's efficiency factor. What percentage of a working day is expended on the direct work in realising the feature under development? This can be calculated by noting when non development tasks are being carried out e.g. stand up and other meetings, answering the telephone, reading email, administration tasks etc. Then a calculation can be made to

determine what percentage of the average day is actually expended on designing and implementation. This needs to be kept in mind when making estimates.

There are several techniques that can be used in arriving at estimations. An agile related one is Planning Poker, while another used for improving estimations is called the 3 point weighted average. To use this technique, think of the quickest time the task can be done in, then think about how long it could take if there were problems. Then think about what is the realistic length of time this can be done in. Then add these altogether with 3 more of the realistic ones and divide the total by 6. This is the 3 point weighted average (1 best + 4 realistic + 1 worst) / 6.

Stories can also be broken down into sub tasks. This does a few things. Firstly it makes the developer and others, think about the story and its tasks. When discussed with other members of the team including the product owner, it helps focus on exactly what is to be done. Communication is the key – it is easy to go down the wrong path by making assumptions when things are not crystal clear. Secondly, by discussing the tasks with other members of the team such as Testers, Business Analysts and Solution Architect etc., it is possible to get a much wider perspective on what is to be achieved. Lastly, it is easier to estimate a lot of small tasks than one big one – just estimate each task, sum them up and the estimate for the whole story is arrived at.

This processed can be summed up using an acronym which is BETCAP:

B **Breakdown** the story into sub tasks. This allows you and your team to understand the problem, communicate all perspectives and make it easier to estimate.

E **Estimate** each task. Once an estimate has been given, the developer must do their best to commit to it. Things can go wrong, so this must be factored into the estimate by using some type of technique and the efficiency rate. If it can't be estimated it should be spiked.

T **Track** the progress. If the estimate is 3 days, then at the end of the first day the developer must ask "am I a third of the way there"? If not what can be done to get back on track?

C **Communicate** the progress, so there are no surprises. The team should know on a daily basis if the work is ahead, behind or on time.

A **Analyse** the estimates when the task is complete and compare against the actual time taken. If the work was late, what were the reasons for this? What could be done better next time? If work was ahead, how did this happen? The developer needs to reflect upon the estimation process and use the conclusions to continually improve their estimating skills.

P **Pat** yourself on the back! If the developer completed the work before or on time he needs to reward himself and his team mates.

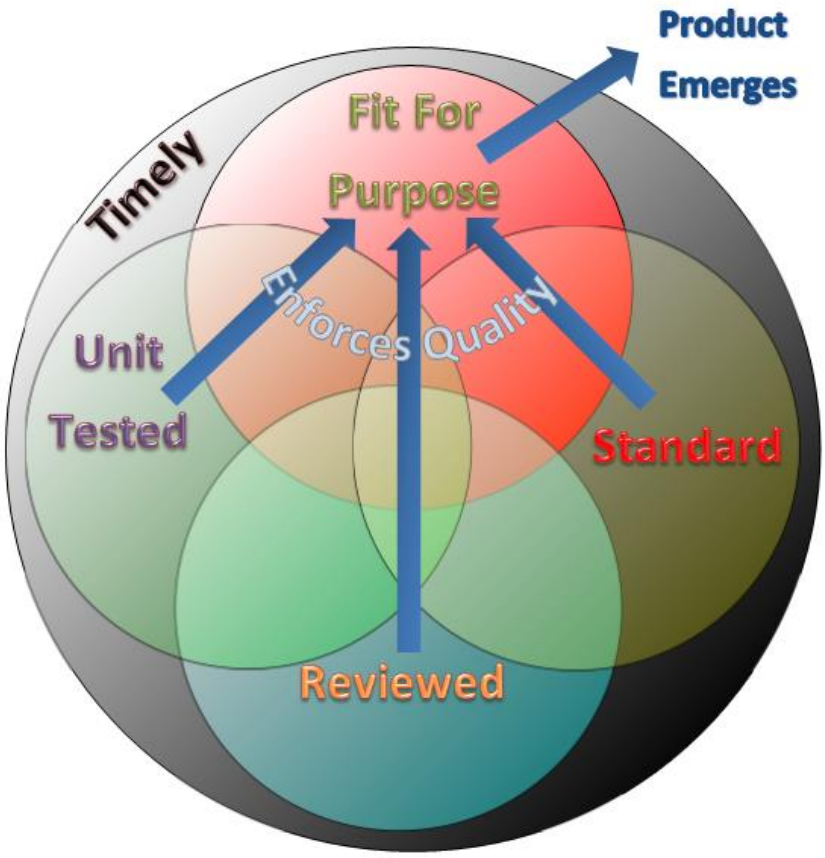
Estimating is not rocket science. It may start off as guessing, but as experience is gained, the skill can be honed.

> *"By learning to estimate, and by developing this skill to the point where you have an intuitive feel for the magnitudes of things, you will be able to show an apparent magical ability to determine their feasibility." – The Pragmatic Programmer.*

Conclusion

The 5 principles of FURST have been analysed and arguments made to support their inclusion. Of these principles, it is FFP where the product emerges from, as it is the artefacts such as source code, configuration files and images etc. that the final product is built from. Standards, Unit Tests and Reviewed are there to support FFP and all are bounded by Time. However without the support of these other principles, which in reality forces both the product and the artefacts to be FFP, the process and the product creation may be drawn out, taking longer to achieve, costing more and being harder to check, with the likelihood of a poorer quality product.

Figure 2. FURST Principles at work.



Besides helping FFP to be achieved easier, the other principles influence FFP in ways which naturally improve software quality. As an example, without Unit Tests it is harder to assert that the basic functions work as expected and the product works as expected after changes. But this is not the whole story. By using Unit Tests, the software is being built in a way that makes it testable and also confirms the requirements. If done in a TDD manner, the developer must think about the tests upfront, so must have the intention clear in his mind before the tests are written (otherwise how could it be tested), thus clarifying the purpose. If the purpose is not crystal clear, then it forces him to make enquiries until the purpose is crystal clear. Reviewing will also ensure that this perspective is shared by other developers, thus giving consensus and together with Standards will ensure consistency and conformity.

By decomposing the principles and forming acronyms for each, the key aspects can hopefully be remembered:

Table 2. FURST Acronyms

FURST	Principle	Acronym(s)
F	Fit For Purpose	QUALITY PARSED
U	Unit Tested	FIRST
R	Reviewed	SAUCE
S	Standard	SeiSMiC
T	Timely	BETCAP

From studying the principles of FURST you will see that there is a great undertaking when you decide to become a professional developer.

> *"Programming is a craft. At its simplest, it comes down to getting a computer to do what you want it to do (or what your user wants it to do). … You try to capture elusive requirements and find a way of expressing them so that a mere machine can do them justice…. What's more, you try to do all this against the relentless ticking of the project clock. You work small miracles every day." – The Pragmatic Programmer.*

[1] Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, 2008
[2] Code Complete: A Practical Handbook of Software Construction, First Edition, 1993, Steve McConnell
[3] Professionalism FURST, IAPetus Newsletter, November 2015, Paul Lynham
[4] http://www.macmillandictionary.com/us/dictionary/american/fit-for-purpose
[5] https://en.wiktionary.org/wiki/fit_for_purpose
[6] http://www.qualityresearchinternational.com/glossary/fitnessforpurpose.htm
[7] The Pragmatic Programmer: From Journeyman to Master, 1999, Andrew Hunt & David Thomas
[8] Debugging the Development Process, Steve Maguire, 1994, p. 19.
[9] Code Complete 2, Chapter 20, Page 464, Steve McConnell, 2004
[10] The Effects of Software Size on Development Effort and Software Quality, World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol:1, No:10, 2007, Zhizhong Jiang, Peter Naudé, and Binghua Jiang.
[11] Chapter 9 Unit Tests P.132.
[12] dis.unal.edu.co/~icasta/ggs/Documentos/Normas/1028-2008.pdf
[13] smartbear.com/lp/ebook/collaborator/secrets-of-peer-code-review/
[14] www.brunel.ac.uk/~hsstffg/papers/Chunking-TICS.pdf

[15] https://en.wikipedia.org/wiki/Flow_%28psychology%29
[16] NASA, Overview of the Software Engineering Laboratory, December 1994, Page 29, Figure 11
[17] Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security? *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS–04)*, Track 9, Volume 9, IEEE Computer Society, January 2004, J. Heffley and P. Meunier
[18] https://www.atlassian.com/agile/code-reviews
[19] http://www.gamblingcommission.gov.uk/Technical-standards.aspx
[20] Writing Solid Code, 1993, Page 216, Steve Maguire