# Professionalism FURST

*Creating a new mnemonic for professional coding*

Paul Lynham MSc PGCE FIAP

## Abstract

In considering how the concept of *Clean Code*[1] could be summarised in a few words, the acronym SAD (**S**imple, **A**ttractive, **D**ependable) was formed to suggest that code should be simple, well written and reliable. SAD isn't really specific as with some acronyms e.g. DRY (Don't Repeat Yourself), as you can look for repeated literals or code within a project and categorically say if it is WET (Write Everything Twice) or DRY. Like beauty, SAD is in the eye of the beholder. However, another acronym FURST was devised to fill this void to ensure that code is **F**it for purpose, **U**nit tested, **R**eviewed, **S**tandardised and **T**imely. This goes further than SAD in that it gives some more tangible expressions which can be checked or measured. FURST is not a complete end to end process, but sets out what needs to be done rather than how to do it (in this regard like Scrum). However, from it, check lists can be devised to be used at certain points within the development cycle and could be automated to some extent using tool chains.

## Introduction

Over the last decade several ideas have had a significant impact on software productivity. Some of these just simplify things, making them easier to understand and use. One such example is REST[2] which allows messages and data to be transferred in a simpler, lighter format (usually JSON) which is easier to read for both machines and humans and can lead to a higher-performing and more maintainable architecture. *Clean Code* is another example of a concept which encapsulates numerous simple ideas, many of which on their own have been around for a long time[3], but when applied collectively and routinely, produces code that is easier to read and understand and results in long term positive consequences, mainly because it is more reliable and easier to maintain.

## Clean Code

Many of the *Clean Code* ideas are considered best practices, but every developer will have their own idea of what sums up *Clean Code*. One such summary is[4]:

- Easily accessible to others (straightforward, clear intent, good abstractions, no surprises, good names) – this is absolutely the most mentioned point.
- Is made for the real-world i.e. has a clear error-handling strategy.
- The author clearly cares for the software and other developers (which implies both readability and maintainability).
- Is minimal (does one thing and has minimal dependencies).
- Is good at what it does.

Another summary states[5]:

- Easy to understand.
- Easy to modify.
- Easy to test.
- Works correctly.

There is no conclusive definition but it has been summarised in one sentence as[6]:

*"Clean Code is code that is easy to understand and easy to change."*

On top of this is the attitude of a professional craftsman, being proud of the code that you craft and taking responsibility for it[7]. This includes demonstrating how it works and how it can be verified.

## Names and acronyms

Quite often principles have been given acronyms so they can easily be conveyed without spelling out the full concept, similar to names being given to design patterns and refactoring strategies. A few acronyms that spring to mind are:

DRY Don't Repeat Yourself - every piece of knowledge must have a single, unambiguous, authoritative representation within a system e.g. don't have repeated literals, statements or routines that are exactly the same.

WET Write Everything Twice – this is the antithesis of DRY and should be avoided. In this context, literals, statements or routines appear twice or more.

KISS Keep It Simple Stupid - most systems work best if they are kept simple rather than made complicated.

YAGNI You Aren't Going to Need It – don't add extra functionality until it is necessary.

Others used in the OO paradigm include SOLID and GRASP and in Design Patterns MVC, MVP and MVA. However, it is not always easy to convert a concept into an acronym that works. As an example, try converting the idea of decoupling code using abstractions so that techniques such as dependency injection[8] can be used:

*"Code to an interface rather than a concretion[9]".*

## Some new acronyms

On postulating how the concepts of both *Clean Code* and the attitude of professional craftsmanship could be summarised, a few solutions came to mind. The first is the smallest number of words that could be used to sum up the essence of the concepts.

## SAD

SAD stands for Simple, Attractive and Dependable.

**Simple** – clear cut, easy to understand, clean, humble

**Attractive** – appealing, pleasing, fascinating, sweet

**Dependable** – reliable, trustworthy, responsible

When code is **simple**, it is easy to read, easy to understand and easy to spot any flaws. If it is well written, using naming conventions and words from the domain language and it is descriptive so that its intention is plain, then it is **attractive** and pleasing code, the kind of code that others will admire. It should also be **dependable**, proven with unit tests, so that it will be easy to change without fear of breaking it, as any bad changes will be made visible by failing tests. It will also handle errors

gracefully. *Clean Code* will also demonstrate that it is trustworthy, by virtue of its transparent intention, being unable to conceal any malicious intent in esoteric code.

However, although this distils the concepts to a bare minimum, the acronym as a word has a derogative meaning and to call someone's code SAD doesn't really inspire them! Compare this in contrast to SOLID, which conveys the meaning of strong and reliable! Another problem with SAD is the acronym seems to return "Single Administrative Document" when queried by Google and sometimes, "System Analysis and Design". Thus it is sad, but SAD is unlikely to stick.

## FURST

An improved set of principles that will help in producing good 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** - check that it is well written compliant code that is easy to understand.

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

**Timely** - produced in time

## A simple analogy

Although producing a piece of software may not be the same as producing a more tangible product, for the purpose of a better understanding of FURST, the production of a car can be used as an analogy.

A car needs to be built within a certain time limit in order for the manufacturer to make a profit. The car is produced to certain requirements such as engine size, number of doors, top speed and utility such as the ability to fold down the back seats, to meet the customers' needs. Besides being attractive, reliable and easy to drive, the car also needs to comply with government or agency standards so that it is safe to be driven on public roads. The manufacturer will employ a quality engineer to carry out or review a series of tests to ensure that the car meets both its own requirements and any standards laid down, such as health and safety and road worthiness. The quality engineer may use a 'rolling road' to test the car. The rolling road can be used for basic testing such as to ensure the brakes work efficiently and reliably and the exhaust emissions meet their own requirements and exceed the government standards. However, the rolling road cannot determine how the car handles in general, such as if it is a soft or hard ride, or how it handles corners. The rolling road is not part of the car and is never delivered to the customer, but without it there would be great difficulties in asserting that different sub systems or components within the car work to the prescribed limits.

A software product has to be built and delivered within a certain **time** for the venture to be of value. The software will have a set of requirements that have to be met for the product to be useful to the customer (**Fit For Purpose**). These may include normal features, utility features and it will need to perform within certain limits. The software will also need to be reliable, easy to use and comply with certain **standards** such as security, safety, and look and feel amongst others. **Unit tests** are written to test the basic functions of the software to assert that it meets or exceeds the requirements and any compulsory standards. A **reviewer** will look at the product to ensure that the quality and other

standards have been met and to check that the tests are valid and pass. However, the unit tests don't test the behaviour of the whole system. The unit tests are not delivered to the customer, but without them it would be difficult to prove the basic functions work as expected.

## Professionalism

A non-professional programmer may think, as long as the requirements are met, then I have done my part! Why bother with unit tests, following standards and having someone else reviewing what I have done? It will get completed when I finish, so why worry about time?

The whole purpose of software is to produce value for the customer[10]. For this to be true, it must also be reliable and trustworthy, while for the organisation developing the software it must be easy to maintain and enhance.

Keeping code clean can help in optimising development, as the source code is always kept in a fit state, thus allowing flexibility for a change in priorities or requirements. Code is read many more times than it is written, so it must be easy to understand and change and it must look like all the other code i.e. standard and coherent. If it is changed, you have to be sure you haven't broken anything and unit tests can help with this. Having other developers review all these points will ensure that these ideas are adhered to. All this must be done in a timely manner. Thus a professional attitude brings a lot more responsibility, but also produces rewards in creating a better quality product.

The concept of FURST could be combined with the idea of professionalism to produce "Professionalism FURST".

There are reasons for this combination:

1. To reinforce the idea that a professional developer cares about his/her work (professional pride). Following FURST will help in this regard.
2. Robert C Martin in his follow up book to "Clean Code", "The Clean Coder", highlights what it means to be a professional programmer.
3. The IAP wants to promote the idea of professionalism in software development

Another suggestion is "FURST Principles"[11].

## Use of FURST

Mnemonics are useful in this context as something to keep in mind when writing code. Initially it may help to reinforce good practice, but hopefully would become second nature after a while. It may be especially useful for trainee developers, reminding them that their code will be reviewed to ensure that it is well written, contains no obvious bugs, complies with standards, meets acceptance criteria, so it is fit for purpose and tests are in place. As a professional programmer you must also acknowledge the business circumstances and equate time with money. There is no point in producing a fantastic piece of code if it takes too long or is not produced in time to be used!

FURST is not only more descriptive than SAD but is also less subjective. As an example you can check if there are **U**nit Tests in place, if the code has been **R**eviewed and if it has been completed within the current sprint (**T**imely). **S**tandards are ones that an organisation demands to be followed, so the review can ascertain this. However, **F**it for purpose may be more of a problem!

You can show that the code meets requirements and acceptance criteria and this would go a long way towards FFP. In an agile process this would be the minimum for any 'Definition of Done'. However, the code could be a *big ball of mud*[12] (because of legacy code) and at this point in time it works. Is this fit for purpose? It may not be reliable as a small change may take a long time to get working and its intent is not transparent i.e. it is not easy to read, understand or change and like beauty, it may not be pleasing. Therefore FFP, unlike the others is more subjective. You can say definitely if it is not FFP when the application doesn't meet requirements or acceptance criteria, but the converse is not necessarily true.

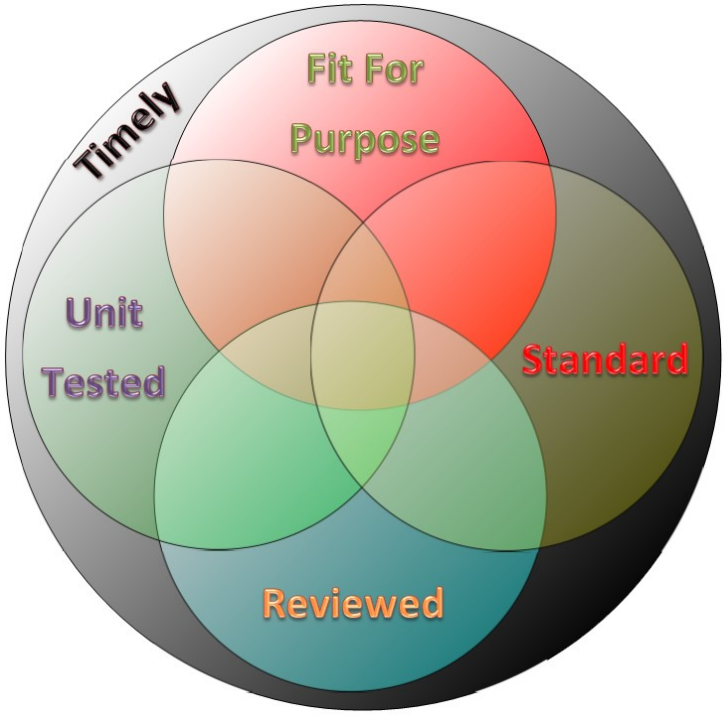


**Figure 1. FURST principles**

## FURST Principles

The principles of FURST do not exist in isolation; rather they overlap as shown in Figure 1. As an example, to be **FFP**, **Standards** must be complied with, while **Review** will ensure **Standards** have been complied with and it has been **Unit Tested**. All are bounded by time - **Timely**.

Each principle of FURST has its own attributes. In the Clean Code book, unit tests should follow 5 rules. These are **Fast**, **Independent**, **Repeatable**, **Self-Validating** and **Timely** (FIRST)[13]. FFP have attributes that include **Quality**, **Understandable**, **Acceptable**, **Appropriate**, **Attractive**, **Labelling**, **Intention**, **Testable**, **Transparent**, **Trustworthy** and **YAGNI**, thus forming QUALITY. Some of these attributes will have further attributes or tenets, thus for code to be **Attractive** it must be **DRY** and contain no **Magic Numbers** amongst other properties. There is also some overlap between attributes as DRY could also be a requirement of the Coding Standards.

## SAD, FURST and Clean Code

SAD is a low level summary of the concepts, while FURST expands these a little more, so that although not every tenet of Clean Code is named, the intention can be related to the corresponding tenets. Table 1 may be helpful to picture the relationship to some of the Clean Code tenets.

| SAD | FURST | Sample Clean Code Tenets |
|---|---|---|
| Simple | Fit for purpose<br>Standard<br>Timely | KISS<br>One responsibility<br>Max 2 arguments<br>Boy Scout Rule (continuous refactoring)<br>Minimise duplication (DRY)<br>No feature envy (smell)<br>Avoid Magic Numbers |
| Attractive | Reviewed<br>Standard | Be precise<br>Use naming conventions<br>Use explanatory variable names<br>Delete unused and commented out code<br>Describe the intention<br>Encapsulate primitives<br>Formatting (team rules) |
| Dependable | Unit tested<br>Standard | Be precise<br>Use names from the domain language<br>Describe the intention<br>Use appropriate types<br>Handles errors<br>Unit Tests |

**Table 1. SAD, FURST and Clean Code comparisons**

## FURST versus SMART

SMART is an acronym often associated with project management. This is used to judge objectives and a common description is:

**Specific** - e.g. target a specific area, process or metric.

**Measurable** - one or more indicators of progress or success.

**Achievable** - can it be accomplished with available resources?

**Relevant** / **Realistic** - does it align with our goals, timeline and constraints?

**Time bound** - can it be achieved by the target date?

Looking at how SMART can be applied to a software project in terms of implementing features in code to meet requirements, through acceptance criteria, the following alignment can be made:

**Specific**- this is covered by **Fit for purpose (FFP)** i.e. it is good enough to do the job it was designed to do. As Albert Einstein said:

> *"Everything should be made as simple as possible, but not simpler".*

By making something simple, it is specific. A simple hand held screwdriver is used for driving screws into some material. Little can go wrong with a screwdriver, so it is reliable and has few dependencies. A more complicated tool is an electric drill, which reliant on the attachments (dependencies) can also be used for driving screws, but in addition can be used for drilling holes and sanding and polishing materials amongst many other abilities.

**Measurable** – this can be achieved by some degree by **Reviewed**, as other coders will assess the code against metrics. These may be a simple YES/NO points on a check sheet (**Standards**) or may involve statistics such as those produced by tools to give percentage of code **Unit Tested**, code complexity and other static code analysis etc.

**Achievable** – this ties in with both **FFP** (if it is not achievable how can it be fit for purpose?) and **Timely** (if it can't be done in time it is not achievable). In agile methodologies, if it is uncertain whether a feature can be implemented, it can be *spiked* (such as experiments, prototypes or further investigations carried out for a limited time) to ascertain if it is achievable.

**Relevant / Realistic** – this again ties in with both **FFP** and **Timely** as the requirements will determine if something is relevant (assuming the requirements themselves are correct) and it can be achieved in a realistic timeframe with the given resources.

**Time bound** – aligns directly with **Timely**. If a feature cannot be implemented in a timely manner then there is no point in writing it! If you don't know if it can be implemented then spike it!

## What next?

The concept of FURST is still at a basic stage and more work is required to get the concept honed and established. Each principle of FURST needs further decomposition and perhaps linking to a catalogue of tenets. It may be possible to produce checklists or build processes that reinforce FURST. This would be an ideal task for IAP members as even the most experienced developer is only going to have a limited view, while together the membership should cover all bases. For checklists, the points could be quite specific, while processes could be geared to towards particular development environments.

As an example it is possible with some tool chains to automatically carry out these processes:

1. Code can be implemented as part of a feature branch. This can be initiated from the story or task within the current sprint.
2. When the code is committed, it can either be passed through a source formatter to ensure layout standards, or some utilities can go a long way in determining whether coding standards have been complied with.
3. Static analysis is carried out and the results saved, emailed or published.
4. The code may be compiled (if necessary) and then unit tests are then run and any failures are flagged up. The developer(s) concerned are alerted.
5. The code is held at a gate until it has been reviewed. The review may be light or heavy or anything between. It may look at the code, the requirements, and the acceptance criteria and verify that these are aligned. It may check that standards have been complied with. If changes are required as a result of the review then the cycle starts at point 2.
6. Once successfully reviewed the code can now be merged with the current branch.

## Conclusion

The IAP[14] may want to use this concept as an initiative in the 10 E's policy. In particular, **Exchange**, **Educate** and **Engender** are points that are applicable, allowing members to *exchange* views in developing this idea, *educating* trainee programmers and *engendering* the professional attitude of our membership. There is also a tie in with the Trustworthy Software Initiative[15], so linking with

**Enlighten** – (we will enlighten our profession, shaping its development by liaising more with government, academia and employers).

[1] Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, 2008
[2] https://en.wikipedia.org/wiki/Representational_state_transfer
[3] For example many of these simple and good practices are documented in:
Code Complete: A Practical Handbook of Software Construction, First Edition, 1993, Steve McConnell
[4] https://dzone.com/articles/what-clean-code-%E2%80%93-quotes
[5] http://stackoverflow.com/questions/954570/definition-of-clean-code
[6] http://cvuorinen.net/2014/04/what-is-clean-code-and-why-should-you-care/
[7] The Clean Coder: A Code of Conduct for Professional Programmers, Robert C. Martin, 2011
[8] http://www.martinfowler.com/articles/injection.html
[9] ConTAIneRTAC is the best I can come up with!
[10] http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment
[11] Personal communication with Martin O'Brien.
[12] https://en.wikipedia.org/wiki/Big_ball_of_mud
[13] Chapter 9 Unit Tests P.132.
[14] The Institution of Analysts and Programmers, http://www.iap.org.uk/main/
[15] http://www.uk-tsi.org/