Resolving the historical confusions about the meaning of software size and its use for project effort estimation

Charles Symons

(Retired, unaffiliated) Reigate, England. E-mail: cr.symons@btinternet.com

Abstract.
The software industry does not have a good track record of delivering systems on time and budget. In part this is due to weaknesses in software sizing and project effort estimating methods.
This paper discusses how some of these weaknesses have arisen historically, resulting in differing views on some basic underlying concepts such as the meaning of software size, types of software size and their relationships, the distinction between software 'size-drivers' and project 'effort-drivers', the meaning of the weights assigned to the components of software functional size measures, and the meanings of Functional User Requirements and of Non-Functional Requirements. Several conclusions are drawn, particularly that the current definitions of the latter two concepts are misleading in how they deal with quality requirements. Some recommendations are made on the need for 'coherent size/effort ecosystems' to help improve project effort estimation, and to revise some existing concept definitions.

Keywords:
Software measurement, project effort estimation, functional requirements, non-functional requirements

1. Introduction

75 years after the term 'software engineering' was first coined, there is still no common understanding of what we mean by the 'size' of a software item, and where to draw the line between methods for measuring software size, and methods for estimating the effort for a project to develop an item of software starting from an estimate of the software's size.

This paper aims to examine the origins from a historical viewpoint of some of the confusions that are common to all these methods where there is evidence that the confusions are still prevalent today, and to suggest some remedies to help eliminate the confusions.

Having been actively involved in many of the developments since Allan Albrecht first published his 'Function Point' (FP) method for sizing software requirements [1], and at last believing I can explain the origin of these confusions, one of my aims is to clear my own conscience. In what follows, where I criticise a development, I confess to being directly or indirectly involved in contributing to the historical confusion. Please forgive me. Along the way, I reach some new (to me!) conclusions that I wish I had realised years ago.

In writing this paper, I have assumed that the reader is familiar with the two most important uses of software sizes, namely a) for measuring the performance of a completed software project (e.g. project productivity = work output/work input = software size/project effort), and b) in 'top-down' methods for estimating the effort for a project to develop a new item of software. These effort-estimation methods require as input: an estimate of software size (usually the largest driver of effort); other system or project requirements; and past (or 'benchmark') performance data.

I also assume that the reader is familiar with common software size measures such as counts of Source Lines of Code (SLOC) and is familiar, at least in outline, with the most widely known Functional Size Measurement (FSM) methods, i.e. IFPUG [2], Nesma [3], MkII [4] and COSMIC [5], and with some effort estimation methods. All these methods have been analysed and criticised at length for their strengths and weaknesses and their suitability for their claimed purposes. Again, I assume the reader is aware of these discussions.

The methods of software sizing listed above have become less commonly used in recent years under multiple influences such as the increasing practice of assembling software from existing components and from the introduction of Agile methods. The latter, when first introduced, added to our list of confusions by using Story Points [6] to size short statements of software requirements. In practice a size

estimate in units of Story Points was typically also taken as an effort estimate. e.g. by assuming that one SP = one work-day. The result was that a measurement of 'velocity' (a more developer-friendly term than 'productivity'), i.e. estimated SP/work-day, was actually a measure of estimation accuracy rather than of performance. Nowadays, it seems to be acknowledged that use of Story Points is a 'bottom-up' means of effort estimation.

In spite of their less common use nowadays, methods for software sizing and project effort estimation, are still highly relevant to cases where a customer requires an estimate of software development project cost in order to make the business case for the investment, or to reach an agreement on costs with a software supplier. This is especially the case for the development of very large software systems. Software sizing methods are also important for quality control on software requirements, for managing project 'scope creep', for monitoring performance-improvement initiatives, and the like. In my opinion, it is therefore still important for future generations of software engineers to resolve some of the confusions that still persist today in this subject area.

2. So what do we mean by the 'size' of a software item?

Answering this question is important if we are to agree what parameters to include in our software size measures.

When Allan Albrecht first described his FP method [1], he defined his measurement as giving '*an effective relative measure of function value delivered to our customer'*. This statement initially caused much confusion. It was a nice marketing description, but the size of an item of software does not necessarily have any relation at all to its value[1] to the customer

Software size is a rather nebulous concept that is difficult to define. In ordinary discourse, we think of the size of a software item as a measure of *how big it is*, or as a measure of the *amount* of software *product*, i.e. the *work-output*, of a project. In ordinary conversation, we often talk about 'the' size of an item of software, only distinguishing the results of using different methods of measuring 'the' size. There is no one thing that one can call as *the* size of an item of software.

An item of software evolves over its life across different states, from an expression of requirements to an executing program. Figure 1 shows the life-cycle of a software item (the 'Entity') developed following a 'waterfall' approach, the artefacts[2] that exist in each state (the Entity's Attributes) and the Measures of a size of those attributes, using the EAM taxonomy [7]. Therefore a measurement of software size can only be properly understood when the following are all known: the state of the item's artefacts being measured, the specific method used for the measurement, and the unit of measurement. Some brief observations on Figure 1:

- 'Functional size' is generally accepted as meaning a measure of the size of 'what the software must do'. For a fuller discussion of Functional User Requirements and functional size, see section 4.
- Figure 1 shows a question-mark against sizing 'Non-Functional' Requirements (NFR) because there is confusion on how to distinguish functional from non-functional requirements and how to measure the latter, if at all. This topic is discussed in section 5.)
- Object Points are defined in Wikipedia as '… *an approach used in software development effort estimation* …. (they) *are a way of estimating effort size, similar to Source Lines Of Code (SLOC) or Function Points*.' (This is another perfect example of confusing software size and project effort.)

[1] Software certainly delivered value in the form of revenue to IBM (Albrecht's employer), likely in proportion to its size. But the value to the customer could range from negative if the software was never used, to many times its cost if it helped the customer achieve business efficiencies or increase revenue.

[2] These artefacts have their own life-cycle. For example outline requirements may be measured approximately by one method then, when the requirements are known in more detail, re-measured more precisely by another method.

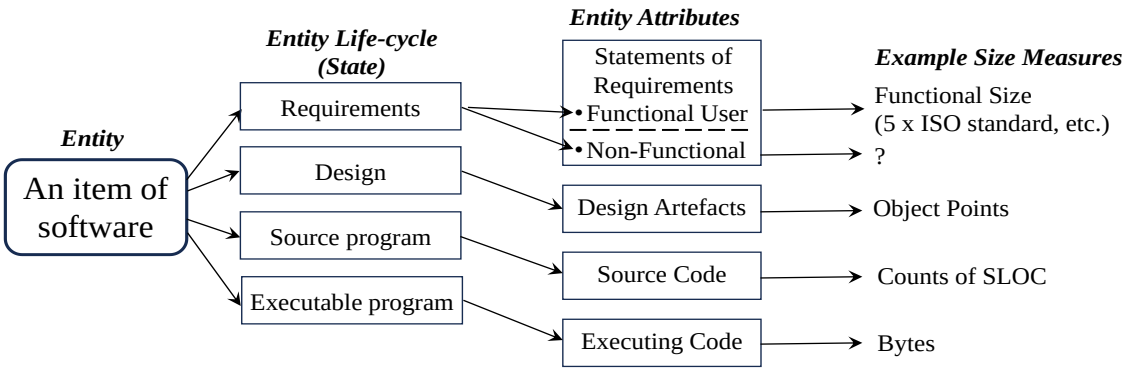

Figure 1. A software item (the Entity), its Attributes and size Measures over its life.

When we discuss a physical object, we usually talk of its size in terms of the measure of its major dimension, for example the height of an office block, or the tonnage of a ship. The nearest dimension of a physical object analogous to a measure of software size is its *length*. Examples of useful measures of the length of a software item include a count of the SLOC or of the number of bytes occupied in computer memory. The length of a software item may not, however, be an adequate measure of the *difficulty* of developing it.

Given the use of software size measures for effort estimation, an underlying aim of every designer of a software size measurement method has been that, for a given homogeneous set of projects, the method should measure software sizes that correlate reasonably with the effort of the projects to develop those sizes. A size measure that cannot be demonstrated to be reasonably correlated with effort is practically useless. Some software size measurement methods have therefore tried to account for other factors than length to reflect this extra difficulty, such as the 'complexity' of the software (a complex subject!)

After Albrecht first described his method, the IFPUG organization assumed responsibility for the method and refined its definition. A size in units of FP [2] was defined as the product of two numbers: the 'Unadjusted Function Points' (UFP) size, and the 'Value Adjustment Factor' (VAF) size.

Measuring a UFP size involves first identifying the occurrences of five 'Base Functional Component' (BFC[3]) types in the item's functional requirements. Next, each of the BFC types are classified into one of three sizes depending on their 'complexity', where the latter depends on two dimensions (e.g. counts of Data Element Types (DETs) and of File Type References (FTRs)). The 5 x 3 = 15 different possible BFC-type sizes are finally allocated a fixed number of FPs on a common, *one-dimensional* size scale. At this point, the foregoing two-dimensional classification parameters are 'forgotten'. This procedure is fundamentally no different from counting SLOC as a measure of the size of a software program. One SLOC BFC can be of several types (declarative, control, compute, etc.) and has a length in terms of the number of characters in the line which can vary enormously. Nevertheless, we ignore this detail and count each SLOC as one unit.

Given that we are accustomed to regarding a SLOC count as a measure of 'source code length', by the same logic we can regard a count of UFP as a measure of 'functional length', and a count of OO Points as a 'design length'.

IFPUG's VAF size, however is of a different nature. It accounted for 14 factors, mainly of various technical requirements that were judged to affect software size apart from its length. For example, in the early 1980's more work had to be done, i.e. it was more *difficult*, to develop a system to operate on-line than in batch mode. It therefore made sense to include various types of requirements in the FP size measure that accounted for such extra difficulties. One can criticise the way the VAF was designed, but Albrecht's intention in first designing the VAF was, in my opinion, perfectly understandable. I will refer to these other factors in his software size measure that reflect the goal of taking account of development *difficulty*, as 'size-drivers'.

Conclusion. There is no reason to expect that different methods for measuring the 'length' of software items at different states in their life-cycle will produce results that correlate well with

[3] A BFC is defined as an 'elementary unit of Functional User Requirements defined by and used by an FSM Method for measurement purposes'. [11]

each other, or with measurements of the same items that attempt to also account for the difficulty in developing the sizes, or that account for this difficulty in different ways. Yet many results have been published on the degree of convertibility between sizes of software in different states, measured by different methods, without acknowledging these factors or attempting to compensate for the inevitable differences that arise.

Although functional size is usually the main driver of the effort of a project to develop a new item of software, there are many other types of requirements for a software project besides functional size that must be taken into account when estimating the project effort. I will refer to the latter as **'**effort-drivers'. Confusion now arises because designers of different sizing and estimation methods have made different decisions about which of these requirements to consider as software size-drivers and which as project effort-drivers.[4]

Listed below are a few of the main groups of requirements that, historically, have been classified as *either* software size-drivers *or* as project effort-drivers.

a) Quality requirements such as for performance, reliability, etc.
b) The complexity of the software, particularly of mathematical algorithms and logic sequences.
c) The programming language used to develop the software
d) The number of users and/or implementations.
e) The extent of the software product that will re-use existing software components.

Some requirements are, however, 'true' project effort-drivers, meaning they are attributes only of the project to be developed. For example:

a) Project processes, risk, and governance.
b) Project constraints such as target delivery dates, budget limitations, inter-dependencies with other projects, etc.
c) The project staffing, taking account of the actual staff numbers available and their experience relative to the ideal needs.

Figure 2 illustrates the choices to be made for any process of estimating project effort starting from an estimate of software size (in this case assuming functional requirements as input), showing the other requirements that may be allocated as either software size-drivers or as project effort-drivers (from the first list above), and the 'true' project effort-drivers (from the second list).

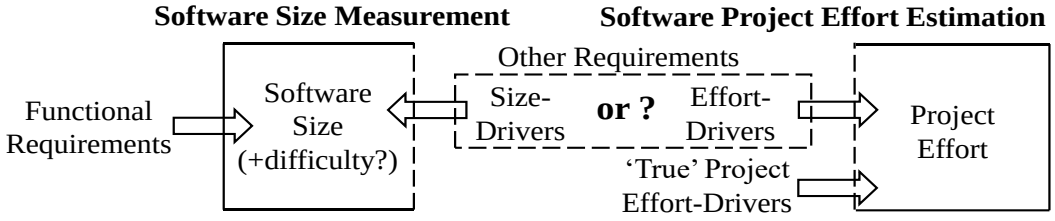

Figure 2. Which requirements to consider as 'Size-Drivers' and which as 'Effort-Drivers'?

Recommendation. Any method that aims to make a reasonably accurate estimate of project effort must ensure a coherent and consistent allocation of size- and effort-drivers between the software sizing step and the effort estimation step. Let us call the resulting system a **'Coherent Size/Effort Ecosystem'**.

As examples:

- The COCOMO estimation method [8] which takes counts of lines of code as the software size measure is a coherent eco-system (leaving aside whether the system has other possible limitations or deficiencies).
- The commonly-used procedure of estimating the size of the software functional requirements, then converting this size to counts of lines of code using external data, then entering this size into a black-box effort estimation tool, is certainly NOT a coherent size/effort eco-system and is most

[4] This paper focuses on the confusions arising in sizing and effort-estimation for a project to develop a software-item. The actual *processes* of a) estimating effort from the various types of requirements/effort-drivers, taking into account measures of past performance in delivering systems of the type to be developed (i.e. benchmarks) and then b) for converting estimated effort and other requirements into project *costs* are beyond the scope of this paper. This also explains why I avoid using the term 'cost-driver'.

unlikely to produce consistently reliable effort estimates. Santillo [9] pointed out how easily errors can propagate when using such a process.

- To establish a coherent *in-house* eco-system for estimating effort early in the life of a project, probably the best way (if one has the resources) is to use a standard method for sizing software functional requirements, to define a limited set of effort-drivers relevant to the in-house environment, and to collect sufficient size and effort data on past completed projects to establish in-house performance benchmarks.

3. On the 'weights' used by some early methods for sizing software requirements.

When I first used Albrecht's sizing method, I found some systems where the measurements did not seem to fully reflect the size of transactions that had to navigate through large, multi-level (i.e. 'complex') databases. My main aim in developing the 'MkII' FP Analysis method [4], therefore, was to improve on Albrecht's Unadjusted FP size component.

I chose to measure the size of a logical transaction by a count of the DETs as the measure of each of the input and output phases of the transaction respectively, and by a count of the number of entity-types referenced (ERs) as the measure of the processing phase of the transaction. But how to add two counts of DETs to a count of ERs? The three counts had to be weighted in some way. It seemed obvious to use weights proportional to the relative effort to implement the three types of counts. I therefore used a Delphi approach, asking the developers of over 60 projects to 'guestimate' the relative amount of effort needed for the three phases of the transactions. From these data I derived an 'industry-average' set of weights. The MkII UFP size of a transaction was then the weighted sum of the counts of the DET's and ERs.

At the time, I saw nothing inherently wrong with Albrecht's VAF, so I added a few more factors, e.g. requirements for security, interfaces to other applications, etc., re-named the result a 'Technical Complexity Adjustment' (TCA), and re-calibrated the weight of the sum of its components, using the same Delphi approach. Later, I received evidence[5] that Albrecht's weights for the components of his FP method were similarly derived from an IBM effort estimation method, i.e. Albrecht's weights were also derived from relative effort to develop the various components.

Shortly after publishing the MkII method, a software metrics expert for whom I have great respect, commented that MkII FPA was '*not a software sizing method, but an estimation method'*. So, is it true that these early Function Point methods, [2], [3], [4] and others are actually 'software sizing' methods? Or are they 'project estimation' methods?

Undeniably, from a mathematical viewpoint, given the weights applied to counts of BFCs were all derived from relative effort, the units of the methods must be proportional to effort. But the *Unadjusted* FP sizes they produce only take account of requirements for software functional 'length'; they do not take account of any of the other factors that may be considered as size-drivers or effort-drivers of the project being estimated, So these methods cannot, in my opinion, be considered as estimation methods in any practical sense; they are hybrids of software sizing and effort estimation methods.

Conclusion. The UFP components (BFCs) of these early FP methods actually measure a standard 'Relative Effort' for a software project. Describing these methods as measuring a standard 'Relative Effort' is more accurate than describing them as a standard 'functional length'. A Relative Effort size is a valid measure (or an 'index' if preferred) of the amount of work required to develop the functionality of an item of software, relative to an arbitrary standard work-size. Relative Effort size measurements are on a ratio scale. Their units of measurement have no meaning on any absolute scale (like a 'Dow Jones index' of software size, as Albrecht once commented.)

The concept of defining a 'standard effort' as a measure of the size of a task, against which actual effort could be compared was first introduced by Frederick Taylor [10]. His ideas on measuring work and on using them to measure and help improve productivity have been in use for over a century.

Taylor's ideas are applied to highly-repetitive work and his standard effort is measured in absolute units of time (e.g. minutes) for a specific process. In contrast, software development is non-repetitive

[5] I was given a paper copy of a set of Albrecht's hand-drawn OHP slides entitled ''Where Function Points (and weights) came from", dated February 2nd, 1986

work and Relative Effort is a measure that is independent of the development process. Nevertheless, the idea of comparing actual effort against a measure of standard effort in order to measure productivity is the same for both cases, and equally valuable. Similarly, the estimated Relative Effort of a software-item to be developed may be used as the primary software size input to a project effort estimation method.

Does it then follow that a Relative Effort size is also a valid measure of the *functional size* of a software-item, as the methods' protagonists claim?

4. The ISO/IEC standard 14143/1 on 'Functional Size Measurement: Definition of Concepts'

In about 1995, the International Organization for Standardization (ISO) established a Working Group (ISO/IEC/JTC1/ SC7/WG12). WG12 set out to define some principles for Functional Size Measurement. The resulting standard, ISO/IEC 14143/1 [11]. includes some important definitions.

"**Functional User Requirements** (abbreviated as 'FUR'): *sub-set of the User Requirements describing what the software does, in terms of tasks and services*."

"**Functional Size**: *size of the software derived by quantifying the Functional User Requirements.*"

[A first, maybe pedantic, comment about the definition of 'FUR' is to ask why are FUR a sub-set of the *User* requirements? Requirements may be specified by many actors, including the project sponsor and lawyers, who will never be users. What the term intends, I believe, is that these are the requirements for tasks and services that will be *provided to* the software users. Secondly, the term 'functional' (or function, or functionality) is not defined in 14143/1. The meaning is only implied by the phrase '*what the software does'*. These two interpretations of the term FUR turn out to be quite important - see Section 5. Another minor anomaly in the definition of FUR is that in reality Requirements specify what the software 'must or should do' *in the future*, when it is developed – not what the software 'does', *implying it already exists*.]

Recommendation. A better definition of FUR would be *'sub-set of the requirements describing what the software must or should do, in terms of tasks and services, for its users*.

One of the most important FSM principles defined in 14143/1 is (extracts):

"Functional Size shall have the following characteristics:

i. it is not derived from the effort required to develop or to support the software being measured;
ii. it is independent of the methods used to develop or to support the software being measured;
iii. it is independent of the physical or technological components of the software being measured."

Conclusion. The two definitions quoted above and the clause i) concerning FSM characteristics mean that **the 'Unadjusted' FP size components of early Function Point methods that rely on relative-effort-related weights (i.e. that measure Relative Effort sizes) *can* legitimately describe themselves as 'FSM Methods'.**

It follows, however, that the VAF component of Albrecht's method, the TCA component of the MkII FP method and their equivalents in other early FP methods do not comply with clauses ii) and iii) of the FSM characteristics. These components were therefore dropped for ISO FSM method standardization purposes and were, effectively, consigned to history.

5. Functional or Non-Functional Requirements?

The VAF and their equivalents in other FP methods had fulfilled a role as software size-drivers in the then-existing size/effort eco-systems (coherent or not). Now that their components no longer contributed to functional size, they inevitably had to join the list of effort-drivers and so they became entangled with the concept of 'Non-Functional Requirements'.

A web-search on 'Non-Functional Requirements' (NFR) reveals a plethora of different lists of example NFR and a variety of vague definitions, many published in recent years. How to define NFR, how to distinguish NFR from FUR, and whether it makes sense to measure a size of a set of NFR (as indicated in Figure 1), are still major sources of confusion in the software metrics community.

For many years the IEEE's SEVOCAB [12] gave a definition of NFR from the ISO/IEC/IEEE 24765:2010 standard as:

"A software requirement that describes not what the software will do but how the software will do it. Example: software performance requirements, software external interface requirements, software design constraints, and software quality attributes."

The 14143/1 definition of FUR has a (non-normative) Note, part of which effectively provides us with another definition of NFR. It states:

"User Requirements that are not Functional User Requirements include but are not limited to:

i. *quality constraints (for example usability, reliability, efficiency and portability);*
ii. *organizational constraints (for example locations for operation, target hardware and compliance to standards);*
iii. *environmental constraints (for example interoperability, security, privacy and safety);*
iv. *implementation constraints (for example development language, delivery schedule)."*

Notice the overlaps (but also the inconsistencies) between the components of a VAF or TCA (described earlier as 'size-drivers'), the examples of NFR given in the SEVOCAB definition, and the examples of requirements that are *not* FUR according to ISO/IEC 14143/1.

The early definition given in SEVOCAB is not very helpful. None of the examples it gives of NFR define 'how' the software will do what it must do. Moreover, some performance requirements typically apply at the *system* level e.g. requirements for response time or availability; these can therefore involve requirements for hardware as well as software.

Recognising the lack of a precise definition of NFR, the COSMIC and IFPUG organizations collaborated to produce a more refined definition of NFR and a glossary of 70 NFR terms [13]. Their definition of NFR[6], which now also appears in the SEVOCAB, is as follows.

"A*ny requirement for a software-intensive system or for a software product, including how it should be developed and maintained, and how it should perform in operation, except any functional user requirement for the software.*

NOTE: Non-functional requirements (NFR) concern:

- *the software system or software product quality;*
- *the environment in which the software system or software product must be implemented and which it must serve;*
- *the processes and technology to be used to develop and maintain the software system or software product, and the technology to be used for its execution."*

Taken together, the 14143/1 definition of FUR and this latest definition of NFR were intended to encompass all the possible requirements for a software-intensive system or software product, and to be mutually exclusive. In other words, any software system requirement must be *either* a FUR *or* a NFR. But with hindsight this is not true. In fact, making this hard distinction between FUR and NFR turns out to be highly misleading and has added to our confusions.

The problem arises with classifying *quality* requirements as non-functional according to the definitions of both FUR and of NFR. But to take a simple example, a quality requirement for security may be implemented either in software (hence arise from a FUR), or in hardware (hence arise from a NFR), or by a mixture of both software and hardware.

More importantly, many quality requirements may initially be expressed as NFR, but are likely implemented *entirely* in software and thus contribute to the size of the software. Examples are requirements for auditability, privacy, portability, maintainability, usability, etc. Such requirements could therefore equally be first expressed in statements of NFR or in statements of FUR.

Any requirement, whether initially expressed as a FUR or as a NFR, that is allocated to software must add to its size, which will show up in the software's size when measured by e.g. counts of SLOC or of bytes. However, FSM methods generally cannot account for every type of software requirement in their measures of functional size – requirements for mathematical algorithms being an obvious example. This inability of FSM methods to account for all types of requirements can lead to difficulties in their use as the primary input for estimating effort.

An example of the current confusion surrounding measurement of NFR exists in the form of the SNAP method [14]. The impact on software size and/or project effort of individual NFR can of course be measured or estimated. But the SNAP method aims to define how to measure a 'non-functional size'

[6] (which I personally initially drafted!)

for software which in turn, it claims[7], enables one to measure a standard *collective* size of the NFR for a software item. (The method assumes that NFR are defined as above by COSMIC/IFPUG but are limited to NFR for software). However, in my opinion, regardless of the validity of this claim, it was unwise to attempt to define a measure of the collective size of any set of NFR for a software item for several reasons.

- It is extremely difficult, if not impossible, to envisage a *meaningful* collective size of such a wide variety of *types* of NFR for software, which range across requirements for software quality, the environment it must serve (e.g. including the size of the user base), and the technology to be used for its development (e.g. including the programming language to be used, re-use of existing code, etc.)
- Not only is there a wide variety of *types* of NFR for software, but also *numbers* of possible NFR. For example, the COSMIC/IFPUG Glossary lists around 30 terms for different quality requirements for software, some of which overlap in meaning. This is no basis for a standard size measurement.
- The method defines 14 different BFCs, each of which can have 3 sizes. Applying expert-judgement to determine the 42 effort-related industry-standard weights for the various contributions is a challenge.
- As we have seen, quality requirements may be viewed as NFR in some cases and by others as FUR. (Indeed, at the time of writing, the IEEE/ISO/IEC Committee Draft version of the SNAP standard [14] acknowledges that '*The boundary between functional requirements and NFR does not have a universally-agreed definition'* and *'This document covers a subset of non-functional requirements'* - but does not state which sub-set. Again, this is no basis for a standard size measurement.
- It must be a serious challenge to design a coherent size/effort eco-system when having to incorporate both a functional size and a NFR size on different unrelated scales, to estimate the allocation of effort between work on the two sizes, then to work out how the functional and NFR sizes relate to existing size- and effort-drivers, and to existing benchmark data (probably based only on functional size).

Conclusion. We must recognise that *any* requirement for a software system project, however initially expressed, must ultimately be allocated

- ***either* to software functionality, and thus contribute to its size (though only counts of SLOC or bytes may be able to detect every size addition),**
- ***or* to project effort or other project costs. Such requirements can arise from non-software-related items such as for hardware, or for other activities which may consume project effort such as hardware installation or training, or the requirement must be a 'true' project effort-driver, i.e. a constraint on the project.**

In this context, the current ISO/IEC definition of FUR and the COSMIC/IFPUG definition of NFR create confusion because they classify quality requirements as non-functional, but we know that quality requirements can impact software and/or non-software. To resolve this issue and hence the confusion about the distinction between FUR and NFR requires changes to their respective definitions**.**

Recommendation. Remove clause i) in the NOTE to the definition of FUR in the ISO/IEC 14143/1 standard concerning 'quality constraints' and remove the corresponding Note in the 'COSMIC/IFPUG Glossary of terms for NFR (etc)' concerning 'the software system or software quality constraints'.

In both of these documents, replace the deleted clause by: *'NOTE. Quality constraints or requirements may be expressed as either functional or as non-functional requirements*.'

[7] The SNAP ('Software Non-Functional Assessment Process') method was apparently designed to measure a size of various types of requirements for software features that could not be accounted for by the IFPUG Function Point method. The size resulting from a SNAP measurement was called the 'Non-Functional Size' to distinguish it from the Functional Size. This decision was unwise: classifying a software feature as 'non-functional' simply because it cannot be measured by a FSM method is confusing. Next, 'requirements for the non-functional size of software' were taken to mean the same as 'the non-functional requirements for software', an entirely different concept. The claim that the SNAP method measures a size of the 'NFR for software' is therefore disputed [15].

A more radical, alternative option is to re-think the concept of NFR with a new definition that better reflects the name 'NFR':

Alternative Recommendation.

'*Any requirement for a software-intensive system or for a software product that does not add to software functionality*.

NOTE: Non-functional requirements concern organizational constraints (for example, numbers of implementations), the processes and technology used to develop and maintain the software system or software product, and the technical environment in which it is executed.'

In practice a consequence of the existing definition is that whereas quality requirements for, say, 'system availability' or for 'system performance' will be interpreted initially as a NFR, in the future, with either of the proposed options for the new definitions, it will not be clear initially whether such a requirement implies a FUR or is an NFR, or a mixture. If that forces earlier consideration of this question, that may be an advantage of the new definition.

Accepting the above Conclusion and the second Alternative Recommendation should mean that the model of Figure 2 can be rationalized making it easier to develop a Coherent Size/Effort Ecosystem, as shown in Figure 3.

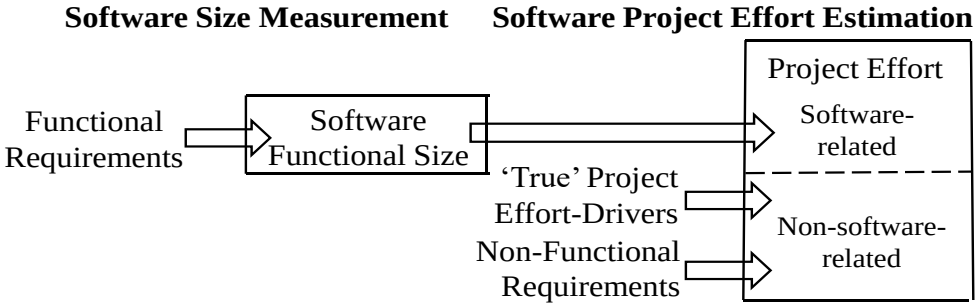

Figure 3. A simplified Coherent Size/Effort Ecosystem, assuming the recommended alternative revised definition of NFR

6. Overall Conclusions

The community of professionals interested in software size measurement and project effort estimation methods is relatively small and fragmented. This has perhaps led to the current inconsistent set of concepts and definitions. In turn, this has resulted in practitioners having to work with *incoherent* size/effort ecosystems, often producing poor effort estimates. The software industry's poor reputation for delivering systems to estimated time and budget seems likely to be due in part to the weaknesses of its software sizing and project effort estimation methods and products. These weaknesses then feed back into their limited use. A vicious spiral.

The various methods for measuring a size of a software item discussed in this paper all produce one-dimensional size measures, but they have different meanings with consequences for how they can be used to build coherent size/effort ecosystems.

- Sizing methods, the weights of whose BFCs were calibrated on project effort, with units expressed as 'Unadjusted' Points (e.g. early UFP sizes, Unadjusted Use Case Points, and the like) are really measures of the amount of work required to develop the functionality of a software item, relative to an arbitrary standard amount of work.
- The early UFP sizes qualify as measures of a 'functional size' of the FUR, according to the ISO/IEC 14143/1 definition. They can also be thought of as measures of 'functional length'.
- The COSMIC FP method produces functional sizes that are truly independent of effort (but that have been shown to correlate well with effort for several types of software at different levels of granularity). CFP sizes, with a single 'data movement' as a unit of measurement can also be thought of as a measure of 'functional length'.
- A count of the number of bytes of memory that a program occupies when it is executing is the software size measure that comes closest to our notion of a physical length.

- One line of SLOC can be thought of as a measure of 'source code length' but SLOC counts suffer weaknesses as a standard due to varying counting rules, dependence on programming language, programmer skill, and other factors.

As far as measurement practices are concerned, two priorities stand out for implementation from this discussion.

For the software sizing and estimation community generally, the greatest needs are to converge on a common understanding of the meaning of and relationships between FUR and NFR, and to define coherent size/effort eco-systems. This should help improve understanding and acceptance of the subject of software size measurement and improvement of effort estimation methods.

If the Agile community is serious about measuring its productivity and demonstrating improvement, then it needs to adopt an objective, standard measure of its work-output and to build a coherent 'top-down' size/effort ecosystem reconciled with a 'bottom-up' ecosystem for estimating effort at all levels of granularity, using the same FSM method. The COSMIC FP method can be used 'as is' for this purpose; other FSM methods can also be applied for use in Agile projects, albeit with adaptations of, or additions to, their standard rules.

More generally, software development practitioners would benefit from a) developing a better understanding of what are NFR, and b) giving greater priority to eliciting NFR and thinking about how they will be allocated (to software or to non-software) *early* in a project before starting design and coding, especially for NFR that apply across a whole system.

References

[1] Albrecht, A., 'Measuring Application Development Productivity', IBM Application Development Symposium, Monterey, CA, October 14-19, 1979

[2] The 'IFPUG Counting Practices Manual'. The International Function Point User Group. See www.ifpug.org. (Version 4.3 is published as ISO/IEC 20926.)

[3] The 'NESMA FPA Counting Practices Manual'. See https://nesma.org. (The 2008 version is published as ISO/IEC 24570.)

[4] The 'MkII FPA Counting Practices Manual'. (Version 1.3.1 of the MkII FPA method is published as ISO/IEC 20968.)

[5] The 'COSMIC Measurement Manual', See www.cosmic-sizing.org. (The 2011 version is published as ISO/IEC 19761.)

[6] See for example: What Are Story Points and Why Do We Use Them? (mountaingoatsoftware.com)

[7] The Entity – Attribute – Measure Taxonomy, Buglione L., Ebert C., Estimation, Encyclopaedia of. Software Engineering, Taylor & Francis Publisher, June. 2012, ISBN: 978-1-4200-5977-9)

[8] Boehm, B., *(1981). Software Engineering Economics. Prentice-Hall. ISBN 0-13-822122-7.*

[9] Santillo, L., 'Error Propagation in Software Measurement and Estimation', 16th International Workshop on Software Measurement, Potsdam, Germany, 2006

[10] Frederick Taylor, 'The Principles of Scientific Management', 1911. Re-published by W.W. Norton & Company, 1967, ISBN 0-393-00398-1.

[11] ISO/IEC 14143-1:2019 Information technology — Software measurement — Functional size measurement — Part 1: Definition of concepts

[12] ISO/IEC/IEEE 24765:2010 Systems and software engineering—Vocabulary.

[13] 'Glossary of terms for Non-Functional Requirements and Project Requirements used in software project performance measurement, benchmarking and estimating' version 1.0, September 2015. See www.cosmic-sizing.org or www.ifpug.org .

[14] 'Software engineering — Standard for software nonfunctional size measurements', ISO/IEC 32430, www.iso.org

[15] Abran, A., 'IEEE 2430 Non-Functional Sizing Measurements: A Numerical Placebo', IEEE Software , 2021 | Volume: 38, Issue: 3.