An Introduction to Function Point Analysis

By Dr Paul Vickers

Northumbria University

School of Informatics
Pandon Building
Camden Street
Newcastle upon Tyne
NE2 1XE
UK

Tel: +44 (0)191 243-7614
Fax: + 44 (0)870 133-9127
paul.vickers@northumbria.ac.uk
www.paulvickers.com/northumbria

1. What's it all about, Alfie?

1.1. Why measure anything?

"When you can measure what you are speaking about and express it in numbers, you know that on which you are discoursing; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science" - Lord Kelvin

Lord Kelvin recognised that measurement is fundamental to any engineering discipline, and software engineering is, or should be, no exception. Over the last decade or so the software engineering community has started to take Kelvin's remarks seriously, although the move towards a metrics activity has been far from easy.

1.2. What are we measuring?

When talking about software metrics, we are not thinking of metre rules, kilogrammes or any other measure in the so-called metric system; rather we are concerned with measures, or metrics which give us information about software. In the same way that centimetres and grammes give us information about the size and mass of physical, real-world objects, so software metrics tell us something about the qualities of software. There is a very wide range of qualities which we could measure. With regard to project planning we are concerned with measures of productivity (the output of software development as a function of effort applied). If we want to estimate software development projects then we will want to measure the estimated size of the software product and the development time needed to produce it. If our interest is in quality assurance, then we will want a range of metrics which in some way measure the quality of software; perhaps in terms of mean time between failure, number of bugs per 1,000 lines-of-code, etc. Whatever our interest, we will need some way of measuring.

In the world of hardware engineering measurement is commonplace. There is usually an agreed set of standard measurements by which we can compare different projects or objects. For instance, the power output of an electric heater is measured in Watts, a standard measure of electrical power. Other qualities measured are weight, physical dimension, signal-to-noise ratios etc. However, within the field of software engineering measurement is not common and there is little agreement even on how and what to measure and how to evaluate those measurements. We shall briefly look at some of the different metrics found in use today.

1.3. Categories of metric

Just as physical measures can be categorised in two ways; direct measures (e.g., the length of a bolt) and indirect measures (e.g., the quality of the bolt, measured by counting rejects) so too can software metrics be categorised. Direct metrics include:

- Cost
- Effort
- Lines of Code (size)
- Speed
- Memory size
- Number of Errors.

Indirect metrics include:

- Function (size)
- Quality
- Complexity
- Efficiency
- Reliability
- Maintainability.

We can see that measures for software size occur in both classes of metric, that is, *function* and *lines of code*. We shall focus our attention on these two.

1.4. The Line of Code

The most commonly used metric to represent software size is the Line of Code (LOC). (This is more usually written as KLOC, or one thousand lines of code).

Most people involved in the writing of software programs have an intuitive understanding of the LOC measure; if we are told that Program A is 10 KLOC (or ten thousand lines of code) in size and that Program B is 100 LOC (or one hundred lines of code) then we would have a good feeling for the relative sizes of the two programs. From our own experience we like to believe that we have a good idea how long it would take us to write a 10 KLOC program and how long it would take us to write a 100 LOC program. The measure, at least on the face of it, seems quite self explanatory; but is it? What exactly is a line of code? Does our LOC measure include blank lines? Does it include commentary statements which have no function of their own but which provide information for the reader and the maintenance programmer? What about programmer styles? Two programmers may code a program to the same specification and provide the same function, but will their programs be the same size when measured in LOC?

This leads us to consider a further point; programming languages. Given a specification, using two different programming languages to implement the specification will result in two different programs of differing LOC size, yet delivering exactly the same in terms of function or utility. Therefore, it is possible that a program measuring 1 KLOC be functionally equivalent to another program measuring 200 LOC. Although the former appears to be five times larger than the latter, nevertheless it delivers no more useful function. By way of illustration, consider the following two program fragments. One is written in an assembly language, the other in COBOL. Both fragments perform the same task, that is add one number to another. But how many lines of code are needed for each?

Assembly Language

```
MOV AX, Total

ADD AX, 2

MOV Total, AX
```

COBOL

```
ADD 2 TO TOTAL.
```

Perhaps a very simple-minded example, but one which illustrates well one of the difficulties associated with using LOC as a universal measure. We can see that the LOC measure can really only be used to compare like with like.

1.5. The productivity problem

This problem is further exacerbated by the recent trend towards measuring productivity. Productivity is measured in terms of Output per Input (or effort). In the automotive industries, productivity may be expressed as Cars-manufactured-per-week, or Lorries-per-month. In these cases it is easy to measure the output; we can count the cars and lorries standing in the holding areas. But how do we measure the output of a software development project? Of course, we could always use the LOC measure, but how do we really know that Team A, delivering a productivity of 200 LOC per month is twice as productive as Team B, working at a productivity of 100 LOC per month? After all, Team B may be developing an application in C, while Team A is coding in an assembly language.

It becomes increasingly obvious that to talk about software size in terms of LOC is generally not appropriate. We have seen that what we are really interested in is the function of the software that we produce. If we can measure the function then we can produce productivity measures which can truly be compared with productivity measures from other development environments. Another way of looking at the

LOC measure is to compare it to the wrapping on a gift. A present is often in a gift wrapped box. This outer appearance tells us only how large is the packaging; it does not tell us how large, or even how useful is the gift inside. In summary, the main issues that lead to the LOC being an unsatisfactory measure are:

- **Comments** Some programmers will heavily comment their code; others may not. A less ambiguous measure often used to circumvent this problem is the NCSS measure - the number of Non Commentary Source Statements.

- **Style** Different programmers have their own style. Sometimes this may make little difference, but in other cases it may be very important. For example, a junior programmer may choose to code every iteration in his program with a WHILE . . . DO type construct whereas the more experienced programmer recognises that in this application 80% of his iterations are determinate loops and are thus more clearly rendered using the FOR construct. The first program will thus be substantially larger in terms of the number of lines of code used but will deliver the same utility as the second program (not to mention that the maintenance programmer will probably find it harder to understand.).

- **Functional equivalence** As illustrated above, two programs that are functionally equivalent can be very different in size when measured in terms of the number of lines of code. This difference in LOC size will be even greater if the two programs are written in different languages; it would be particularly apparent if one were to compare two functionally equivalent programs where one is written in assembly language and the other in a fourth generation language.

- **Environment/programmer/methodology dependent** It is almost possible to compare productivity of development activities which lie on different sides of these boundaries. In a very small, or well-controlled environment it might suffice, where only one language is used, coding style is uniform etc. But for any organisation with several development tasks productivity comparisons become meaningless: project A took one week to deliver 300 lines of a 4GL; project B took one week to deliver 100 lines of a 3GL. Can any sensible comparisons between these two projects be drawn?

- **Alternative solutions** For a given specification we will get different LOC measures when given to different developers to implement. The specification describes function to be delivered. The LOC measure does not measure delivered function; it is in many respects a meaningless figure. We might use our motor industry analogy again. We can liken the LOC measure to a car and then apply some comparisons. There are two manufacturing divisions in the AceyPacey Carriage Co. Ltd. Division A manufactures cars and so does Division B. Division A's productivity was recently measured to be 25 cars-per-week; however, Division B managed to produce 100 cars-per-week. At first, it would seem that Division B is significantly out-performing Division A. However, if we recognise that Division A is manufacturing luxury limousines with hand finished leather trim while Division B is building small, cheap and minimally equipped cars for general use the productivity figures are seen in a different light. Is the work product of the two divisions the same? Indeed it is not; if we analyse the cars produced in terms of delivered function it could be argued that the cars produced by Division A deliver more in terms of function and utility than those made by division B. Thus if we consider the functional aspects of the car (which could be abstracted out to such terms as Rolls Royce and Reliant Robin) we might even say that Division A is more productive than Division B.

  If we take the analogy back to our software development environment, the fact that one group is delivering software at the rate of 100 LOC per month and another at the rate of 1000 LOC per month is irrelevant if the former group is writing in a 4GL and the latter in FORTRAN. We know that 4GLs deliver more function per line of code than does FORTRAN.

➪ **Fitness for purpose/safety critical** A single line of code is harder to produce for a nuclear power plant than for a student record system due to the more rigorous constraints applied to the development task in the nuclear plant. The LOC measure cannot go any way to expressing the amount of effort involved in delivering a single line of program code.

1.6. IBM's solution

Charged with finding a way to measure productivity across sites within IBM in the late '70s, Allan Albrecht recognised that the LOC-based measures were insufficient for this purpose. IBM's justification for measuring productivity was [1]:

> "*A successful . . . [software development] project is one that satisfies the agreed-to user's requirements, on schedule, and within budget. However, since it is based on estimates and agreements the record of successful activities or projects can be misleading. Without a trend of their measured productivity, or a profile of measured productivity from other sites, it is difficult for a site to determine that the estimates are competitive. Management may not know how much more efficiently the activity or project might have been done.*"

In other words, how can an organisation know if its productivity is improving or declining unless it first finds some way to measure it? Albrecht identified five objectives that a successful software development measure should meet:

1) It must consistently determine the productivity and productivity trends of development, maintenance, or support activities, or projects, relative to other similar activities at the site, and other development sites.
2) It must promote actions or decisions that can improve the output of the development site.
3) It must demonstrate the results of the actions taken to improve the output of the development site.
4) It must support the estimating process at the development site.
5) It must support the management process at the development site.

It is the first objective with which we are primarily concerned. To assess productivity two basic measures are required. First we need to know what the output of our activity is and second, we must be able to measure the effort or cost to produce that output. For example, in the motor industry productivity may be measured in terms of cars manufactured-per-week. Here the output, or work-product, is cars and the cost is one week. Thus, work-product divided by work-effort we call productivity. Using the same measures we can derive other information such as unit cost (work-effort divided by work-product). In any environment it is desirable that the productivity increase while the unit cost's trend should be downward. Once we can establish a productivity measure we can start to compare software development projects with other software development projects. In so doing it is possible to see whether a particular project is hitting or missing a given productivity target. Such information can have an enormous impact on the organisation involved in development as it can then take steps to improve productivity. By measuring the software development process it will also be possible to start to assess quality and to estimate the process.

Albrecht's view was that it is the function of the system, or what it does, that is our primary interest. The number of actual lines of code taken to deliver this function is a secondary consideration. The measure which Albrecht developed is called the Function Point (FP). Function Point Analysis (FPA), or the method of sizing software in terms of its function and expressed in Function Points is now very widely used. It is interesting to note that FPA came about, not because a new measure of system size was requested, but because productivity was becoming increasingly important; it was out of the need to measure productivity that FPA was conceived.

Although originally used in the U.S.A., FPA has now found a wide user base in the U.K., Europe and other parts of the world. It has given the I.T. industry a standard against which productivity can be measured and compared across teams, divisions and organisations. Since its inception, many variations of FPA have appeared. The majority of these is concerned with applying FPA to very specialised work environments where Albrecht's original method was felt to be inappropriate. Of the many developments one, developed by British consultant Charles Symons, stands apart. Symons' method was not engineered to fill a specialist niche but to replace Albrecht's method. For this reason, the Symons method is known as Mk II FPA and Albrecht's as Mk I FPA. However, it should be noted that this nomenclature applied only within the U.K. and parts of Europe. Within the USA, Symons' method is not seen as the successor to Albrecht.

The underlying theory is substantially the same for the two methods; Symons' approach is more of a refinement which, it is claimed, provides more realistic and reliable results. MK II FPA is rapidly gaining in popularity in the U.K., although it has not successfully broken through into the American sphere. This is possibly due the very parochial nature of the American market. After all, Albrecht is American and Symons is not.

Users of FPA (both kinds) in this country include nearly all of the high street banks, IBM UK, BOC, the MoD, the Home Office, several major insurance companies and a host of other organisations.

2. Albrecht's FPA

2.1. Background

With regard to FPA, Albrecht [1] states that:

> *"The purpose...is to provide each AD/M* [Application Development and Maintenance, PV] *site a consistent way to measure, portray and demonstrate the productivity of their AD/M activities."*

FPA aims then, to provide a consistent measure of system size that:

- is independent of the development technology
- is simple to apply
- can be estimated (reasonably well) from the requirements specification
- is meaningful to the end user.

2.2. Work-product

We have stated that to measure productivity we need to know the size of the work-product. The method described below will allow us to arrive at a measure of the work-product of a software development activity, this measure being expressed in Function Points (FPs). The size of the work-product, or development task can be measured as the product of the following three factors:

- **The information processing size** This is a measure of the software itself (historically measured in LOC).
- **A technical complexity factor** This factor considers the size of the various technical and other issues involved in meeting the requirements of the proposed system.
- **Environmental factors** These are factors arising from the project's environment and involve things like staff skills, experience and motivation, the use of methods and tools and the programming languages used.

FPA aims to determine the size of a business software system based on the first two of these factors. Currently there is much debate over the inclusion of technical complexity factors; technical complexity means that the complexity of the task and not just the size of the software itself is addressed. Many organisations involved in FPA ignore the consideration of technical complexity.

2.3. The method

To determine the size of the information processing component, we identify the system components as seen by the end user (called *User Function Types* by IBM) and classify them as:

- external (or logical) inputs, or
- external outputs,
- external enquiries, or
- external interfaces to other systems, or
- the logical internal files.

In other words, values are given to inputs, outputs, logical user files, interchange files and enquiries. Each component is then further classified as being *simple*, *average* or *complex* depending on the number of data elements in each type and other factors. Each component is then assigned a points value on the basis of its type and complexity. The points values of all the components are then summed (see equation 2.1) to give a size for the system in unadjusted function points, or UFPs. Notice that each component type is multiplied by a weighting factor. Table 2.1 shows how to determine whether an individual input function is to be rated as simple, average or complex. Tables 2.2 to 2.6 show how to do this for the other component types. Table 2.7 shows how the total UFPs may be calculated and includes the values of the various weighting factors.

$$UFPs = \sum Input(w_i) + \sum Output(w_o) + \sum Files(w_f) + \sum Interface\ Files(w_{if}) + \sum Enquiries(w_e) \tag{2.1}$$

		Field Types		
		1 to 4	5 to 15	16 +
File	0 or 1	S	S	A
Types	2	S	A	C
Ref'd	3 +	A	C	C

Table 2.1 External Input

		Field Types		
		1 to 5	6 to 19	20 +
File	0 or 1	S	S	A
Types	2 or 3	S	A	C
Ref'd	4 +	A	C	C

Table 2.2 External Output

		Field Types		
		1 to 19	20 to 50	51 +
Record	1	S	S	A
Types	2 to 5	S	A	C
	6 +	A	C	C

Table 2.3 Internal File

		Field Types		
		1 to 19	20 to 50	51 +
Record	1	S	S	A
Types	2 to 5	S	A	C
	6 +	A	C	C

Table 2.4 External Interface File

		Field Types		
		1 to 4	5 to 15	16 +
File	0 or 1	S	S	A
Types	2	S	A	C
Ref'd	3 +	A	C	C

Table 2.5 External Enquiry (Input Part)

		Field Types		
		1 to 5	6 to 19	20 +
File	0 or 1	S	S	A
Types	2 or 3	S	A	C
Ref'd	4 +	A	C	C

Table 2.6 External Enquiry (Output Part)

	Simple	Average	Complex	Total
Input	$\times$ 3 =	$\times$ 4 =	$\times$ 6 =	=
Output	$\times$ 4 =	$\times$ 5 =	$\times$ 7 =	=
Files	$\times$ 7 =	$\times$ 10 =	$\times$ 15 =	=
Interface Files	$\times$ 5 =	$\times$ 7 =	$\times$ 10 =	=
Enquiries	$\times$ 3 =	$\times$ 4 =	$\times$ 6 =	=
		Total Unadjusted Function Points		=

Table 2.7 Unadjusted Function Points

The technical complexity factor is given by quantifying the effect of fourteen *General Application Characteristics* that affect the complexity of carrying out the design and implementation task. The General Application Characteristics are shown in figure 2.1.

1. Data communication	**8.** On-line update
2. Distributed Function	**9.** Complex processing
3. Performance	**10.** Usable in other applications
4. Heavily used configuration	**11.** Installation ease
5. Transaction rates	**12.** Operational ease
6. On-line data entry	**13.** Multiple sites
7. Design for end-user efficiency	**14.** Facilitate change

Figure 2.1: General Application Characteristics

The degree of influence of each of the characteristics can range from zero (meaning, not present, or has no effect) to five (meaning, a strong influence throughout). The sum of the fourteen characteristics, that is, the total degrees of influence (DI), given in equation 2.2 is then converted to the technical complexity factor (TCF) using the formula given as equation 2.3. This can also be done by using a form similar to table 2.8

Characteristic	DI	Characteristic	DI
Data Communications		**On-line Update**	
Distributed Functions		**Complex Processing**	
Performance		**Reusability**	
Heavily Used Configuration		**Installation Ease**	
Transaction Rate		**Operational Ease**	
On-line Data Entry		**Multiple Sites**	
End-User Efficiency		**Facilitate Change**	
		Total Degree of Influence	

DI Values:

Not present, or no influence	**= 0**	**Average Influence**	**= 3**
Insignificant influence	**= 1**	**Significant Influence**	**= 4**
Moderate Influence	**= 2**	**Strong Influence throughout**	**= 5**

Table 2.8 Technical Complexity

$$DI = \sum_{i=1}^{14} General\ Applications\ Characteristics[\,i\,] \qquad \textbf{(2.2)}$$

$$TCF = 0.65 + 0.01 \times DI \qquad \textbf{(2.3)}$$

We can see that each degree of influence is worth 1% of the TCF which can range from 0.65 to 1.35. The TCF is now used to modify the size of the system to give the overall size in function points by using equation 2.4.

$$FPs = UFP \times TCF \qquad \textbf{(2.4)}$$

From (2.4) above we can see that FPA provides a measure of system size incorporating both the intrinsic size of the software itself (measured in UFPs) and the complexity of the development activity (measured as the TCF). We can also see from (2.3) above that the TCF's range of effect is from lowering the original count by 35% to raising it by 35%. Thus, in Albrecht's terms, the difference in size between a task with the most benign technical constraints and a project with the most complex technical factors is, at most, 70%.

To illustrate the TCF's effect, consider a system that has already been given a size of 380 unadjusted function points (UFPs) using equation 2.1. If the general applications characteristics were scored such that equation 2.2 gives a result of 28 then the TCF would be (using 2.3):

$$TCF = 0.65 + 0.01 \times 28$$

which gives a value for TCF of 0.93. Applying equation 2.4 gives us an overall size for the system in adjusted function points thus:

$$FPs = 380 \times 0.93$$

which is 353 function points.

2.4. Where does all this lead?

From the above, Symons deduces that "*function points are . . . a dimensionless number on an arbitrary scale*"[2]. This is not meant as a criticism, but serves to show that the function point stands on its own as a measure and is not dependent on any other system of measurement. Also, it means that to try to compare function points with lines of code in anything more than an abstract and very rough manner is pointless and meaningless. If function points really do measure system utility free of the factors that influence a LOC count, then this is exactly the result we would expect. However, human nature being what it is, studies have been carried out to see if there is a link between LOC and function points. The most interesting of these studies was that carried out by Albrecht and Gaffney [3]. To quote them:

> *"The thesis of this work is that the amount of function to be provided by the application can be estimated from the itemisation of major components of data to be used or provided by it. Furthermore, this estimate of function should be correlated to both the amount [sic] of LOC to be developed and the development effort needed."*

Figure 2.2 shows the results of this study. It must be stressed that this only provides very rough estimates of the lines of code required to build one function point in various programming languages. Obviously, because of the factors discussed in chapter 1 these results can only be approximations; they serve merely as a comparison of the relative productivity of the various languages. Any attempt to read anything further into the study is dangerous.

Programming Language	*LOC/FP (Average)*
Assembly language	300
COBOL	100
FORTRAN	100
Pascal	90
Ada	70
PL/1	65
Object oriented languages	30
Fourth generation languages (4GL)	20
Code generators	15

Figure 2.2

Figure 2.2 gives some interesting insights into the differences in programming languages. Assuming that the figures are correct, it appears that COBOL provides about three times the utility per-line-of-code than does assembly language; a line of a 4GL delivers between three and five times the utility of a conventional programming language. Of course, one cannot conclude from this that a 4GL is in any way better than COBOL or Ada as such results tell us nothing about the maintainability of software written in these languages, nor do they suggest that a line of a 4GL is any easier or faster to write than a line of Pascal.

To reinforce the view that function points sensibly should not correlate well with lines of code, a study carried out by the Xerox Corporation [7] in 1985 tried to correlate Albrecht function points with lines of COBOL code. Figure 2.3 presents some of the results of this study.

Size in FPs	**Lines of COBOL (1,000s)**	**LOC/FP**
300	20 to 100	67 to 33
500	20 to 120	40 to 240
700	55	79

Average of all systems measured = 131 LOC/FP.

Figure 2.3

The overall average of 131 lines of code per function point is not very far removed from Albrecht and Gaffney's study. However, as the results of this study show, the average distribution has little meaning of any use. For example, in the systems measured to have a size of 300 function points, the number of lines of

COBOL used to implement them ranged from 20,000 to 100,000. A greater range of results can be seen in the 500 function point group. As the (function point) sizes increase, the range in LOC measures decreases. However, this is not because there is a higher correlation but because there were many fewer systems of the larger sizes; the majority of the systems measured lay in the 100 to 500 function point range.

The conclusion we must draw in the light of this and other studies which gave similar results, is that despite Albrecht's findings, there is little real correlation between function point size and the number of lines of code used. If, as claimed, the function point measures system size independently of any technology used, then these results are as we would expect given the limitations of the LOC measure previously stated. It is interesting to note that the Symons method of FPA (c.f.) gives a better correlation with LOC than that of Albrecht. It is interesting because

2.5. In summary

The main benefit of function point analysis is that it isolates the measure of system size from the environmental factors. This means it is free from the deficiencies of the LOC measure. In theory, two development projects could be sized using FPA and the resulting productivity measures compared regardless of the nature of the two teams, methods and tools used, programming languages, personnel skills and so on. Of course, this requires that the method be applied consistently in both cases.

The TCF that is used to moderate the raw system size measure means that the final count is based, not on the system size alone, but also on the complexity of the development of the software. To quantify this type of complexity necessitates a degree of subjectivity to be employed on the part of the FPA counter; despite the most rigorous guidelines, the assessor must still make a judgement on how much influence a particular factor has on a given project. It is quite possible that two assessors in two organisations will use different criteria in deciding how a particular characteristic should be scored.

Thus, there is a strong case for omitting the TCF from the sizing of software systems and using the raw, or unadjusted function point count as the true measure of system size. As previously mentioned, many organisations have adopted this approach. Current thinking, especially that of the U.K. Function Point User Group's Counting Practices Committee (UFPUG CPC), suggests that technical complexity adjustment will cease to be relevant. Indeed, studies such as the MERMAID project, have provided evidence that suggests that very few adjustment factors have a significant effect on productivity. Further, Kitchenham [4] states:

> ". . . *they cast doubt on the significance given to staff factors in cost estimation models . . . neither dataset showed any evidence that more experienced personnel improved project productivity.*"

FPA gives us an obvious advantage over the LOC-type measures when attempting to assess system size. However, as we shall see in the next chapter, Albrecht's work was not without its limitations.

3. Problems with FPA

3.1. Background

After its inception many organisations started to use FPA realising the benefits it has to offer. However, after applying and teaching the method for some time, U.K. consultant Charles Symons (working for Nolan, Norton and Co. Ltd) started to notice limitations.

3.2. Simple, average and complex

The first problem arose when assessing the size of a system in unadjusted function points (UFPs). The classification of all system component types as simple, average and complex is not sufficient for all needs. For example, table 2.1 shows that for an input transaction to be rated complex, it needs only to have sixteen field types. In 1979, sixteen input fields in a transaction may have been a great deal; however, by the mid- to late-1980s it was not uncommon for a transaction to reference a great many more than this. The problem was, that a transaction containing, say, fifty fields would be rated the same as one containing only sixteen. Apart from the obvious inadequacy, this also led to I.T. staff not trusting the method as they did not feel that their effort was being reflected sufficiently well in the final count.

3.3. Weights

The choice of weighting factors (see table 2.7) was justified by Albrecht [3] as reflecting "*the relative value of the function to the user/customer*" and "*were determined by debate and trial*". It is doubtful whether the weights will be appropriate for all users in all circumstances. We should remember that the weights were derived from study of projects *within* IBM; whatever its diversity, IBM is still only one organisation with its own culture. How can we be sure that the IBM-derived weights apply to non-IBM projects? Further analysis of the weights leads to some interesting anomalies. For example, if an enquiry is provided via a batch mode input/output combination it will score twice the number of function points as the enquiry will when provided in on-line mode. This is because the former case uses two components, an input and an output whereas the latter uses only one. This is some cause for concern as the consequence of this is that FPA is not truly independent of technology. We would hope that a function-oriented measure would score a particular piece of system utility the same regardless of its implementation.

3.4. Identifying logical files

To identify a logical internal files, Albrecht offers the following advice (emphasis his) [1]:

> *"Count each major logical group of a user* ***data*** *or* ***control*** *information in the application as a logical internal file type. Include each logical file, or within a data base, each logical group of data from the viewpoint of the user, that is* ***generated****,* ***used*** *and* ***maintained*** *by the application. Count logical files as described in the external design, not physical files."*

In modern system development it can be very difficult to identify a logical internal file on the basis if Albrecht's rules. The Counting Practices Committee of the International Function Point User Group (IFPUG) publishes a Counting Practices Manual which contains the definitive rules for applying Albrecht's FPA. The 1990 edition [5] defines a logical file as a *"user-identifiable group of logically related data or control information. . ."*; again, this is very open ended. In a relational-data base environment (very common today) such definitions are insufficient.

3.5. On-line transactions

Interpreting on-line interactive transactions also presents us with some difficulty when each input data element is followed by an output response on the same screen. Must we count the screen as an input, an output or both? Are the logical file references (we need to know these to determine complexity) made from the input, the output or both? Is a retrieve-before-update action the same as an enquiry? For each of these cases a counting practice must be established so that the method is applied consistently. Because of the ambiguity of the existing rules, any counting practice that a practitioner adopts may appear to be arbitrary. Furthermore, it is likely that practitioners in different organisations, in the absence of a clear, definitive set of rules will establish different procedures. As we have seen above, the IFPUG rules are not always clear. This means that the application of FPA will not be universally consistent, thus lowering the value of any results gained.

3.6. General application characteristics

In FPA, the technical complexity adjustment factor (TCF) is derived from the sum of the fourteen general application characteristics (see figure 2.1). It is reasonable to ask if these are the only factors that can ever be taken into account during the development of a system. In addition, they share the same weightings; that is, the maximum possible effect of any of the characteristics is five degrees of influence (remember that they are each scored from 1 to 5). Is this equality logical? Might not the maximum effect of data communication (no. 1) be greater in real terms than the maximum effect of operational ease (no. 12) in a particular environment? It is known that a system requiring implementation on multiple sites can easily require mores than 5% extra effort than a system to be installed on a single site only. Given that the TCF is meant to account for the environmental difficulties encountered in developing a system, it would appear that it is not flexible enough for the task.

Another problem with the TCF characteristics is that they can be very difficult to distinguish from one another. For example, what are the differences between characteristics 3, 4 and 5 (design for performance, use on a heavily-used configuration and design for a high transaction rate)? This problem of dimensionality has been the subject of formal study. Kitchenham, in reporting findings of the MERMAID project [4] states:

> *"The 6 largest principal components accounted for 85.5% of the variability of the data, and none of the remaining components accounted for more than 5% of the variability of the data. This indicates that the 14 technology factors could be represented by 6 new factors, and confirms that the original factors were not independent."*

3.7. Internal processing complexity

Consider two transactions A and B. Both produce as output the same number of data items and both take the same number of data items as input. Transaction A very simply transforms the input into the output and thus receives a complexity rating of *simple*. Transaction B requires many searches of a data base and needs to update data several times to produce its output; it is given a rating of *complex*. However, the counting rules (see table 2.7) mean that, at most, transaction B can score only twice the unadjusted function points of transaction A. Given the difference in internal complexity of the two processes, this range seems much too narrow. Symons [2]states that systems "*of high internal complexity . . . do not appear to have had their size adequately reflected by the FP method in the opinion of the designers of those systems.*"

3.8. Discrete and integrated systems

FPA, as devised by Albrecht, is biased towards discrete systems and against integrated systems. In other words, system utility that is provided by several discrete systems linked by interface transfers will receive a higher function point count than the same utility when provided by a single system. For example, if four systems, S1, S2, S3 and S4 each with its own transactions, but linked by interface transfers (see figure **Error! Bookmark not defined.**.1) are replaced by a single system, SX, with a data base that contains the original four files and the same overall transactions, or utility (see figure **Error! Bookmark not defined.**.2), then there would be no more need for the interface transfers.

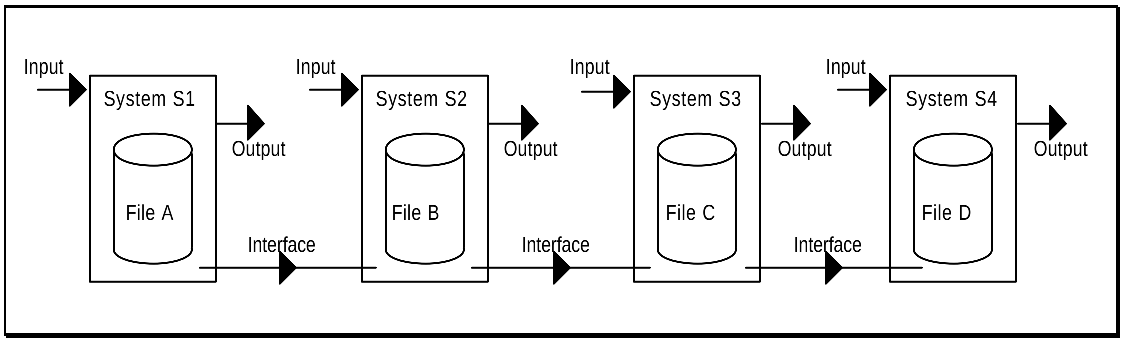

Figure** Error! Bookmark not defined..**1

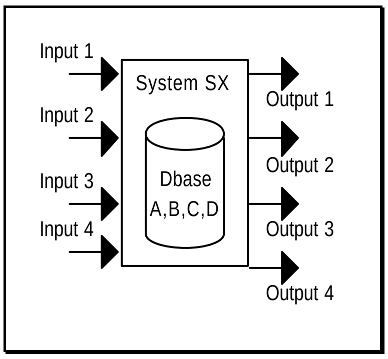

Figure** Error! Bookmark not defined..**2

It would be reasonable to expect that the integrated system would receive the same function point count as the four discrete systems added together. In practice this is not so. In FPA an interface would be credited to the sending and receiving systems. Also, three separate logical files would score more highly than a single data base.

3.9. Failure to measure large systems

In studying productivity, Albrecht [3] found that productivity falls off by a factor of three as the size of systems increase from 400 to 2,000 function points. All of the problems listed above tend to suggest that FPA under-weights large systems relative to small systems. If this is so then Albrecht's worrying productivity trend may not be as serious as it appears.

3.10. In conclusion

Despite the above problems, FPA still represents a big improvement over LOC as a measure of system size. The importance of FPA should not be overshadowed by its deficiencies. In the next chapter we shall look at Charles Symons' proposed solutions to the problems in FPA.

4. Mk II FPA

4.1. Why Mk II?

For the reasons discussed in chapter **Error! Bookmark not defined.** Symons was dissatisfied with Albrecht's FPA. Thus he devised his own method which he named Mk II FPA. Obviously intended to be the successor to FPA, Symons very much based his work on that of Albrecht. The basic concept of the function point remains the same. In Symons' words [6]:

> *"The Mk II Function Point Analysis Method was designed to achieve the same objectives as those of Allan Albrecht, and to follow his structure as far as possible . . ."*

All that Symons sought to do was to overcome the limitations of the method that he had identified. Mk II FPA retains the fundamental premise that the function point count obtained by the method is the product of the two components, Information Processing Size and the Technical Complexity Adjustment (TCA). The TCA component is largely unchanged from Albrecht's original, the most notable difference being the extension of the list of general application characteristics from fourteen to nineteen or more factors.

4.2. Logical transactions

Whereas Albrecht viewed a system as comprising the five component types, external inputs, external outputs, external interfaces, external enquiries and logical internal files, Mk II FPA sees a system as a collection of discrete logical transactions. Symons defines a logical transaction as "*a unique input/process/output combination triggered by a unique event of interest to the user, or a need to retrieve information*" [6]. Examples of logical transactions might be:

- Order an item,
- Display the aged-debtors list,
- Create a customer account etc.

Figure 3.1 shows a logical transaction pictorially.

Using this definition, we can view any system as merely a collection of logical transactions. Modern system design methods like SSADM make the task of identifying logical transactions relatively straightforward. Given that the method aims to provide a measure of system size from the requirements specification, modern methods and Mk II FPA work well together. Indeed, the latest version of SSADM includes the Mk II FPA Estimating Method to assist in the production of reliable project estimates.

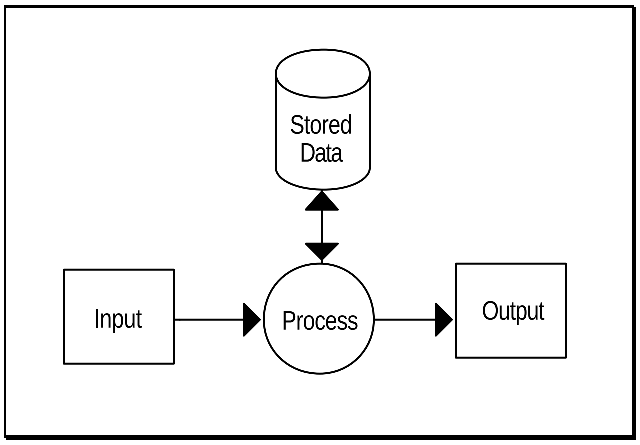

Figure 3.1

When sizing an installed system retrospectively the task of identifying logical transactions is harder. However, Symons claims that once the idea of the logical transaction is completely understood, even sizing installed systems is "*perfectly straightforward*" [6].

As with Albrecht, Mk II FPA determines the size of the information processing component of a system in unadjusted function points. Equation 3.1 shows how to calculate the size of a logical transaction, where *t* represents the size in unadjusted function points.

$$\begin{aligned} t &= W_i \times No.\ of\ input\ data\ element\ types \\ &+ W_e \times No.\ of\ entity\ types\ referenced \\ &+ W_o \times No.\ of\ output\ data\ element\ types \end{aligned} \qquad \textbf{(3.1)}$$

The size for a complete system is simply the sum of the sizes of all logical transactions, hence:

$$UFPs = \sum_{i=1}^{n} t[i] \qquad \textbf{(3.2)}$$

The weights W_i, W_e and W_o used in equation 3.1 are not unchangeable but can be subject to calibration. Nolan, Norton and Co. Ltd are responsible for maintenance of these *industry standard weights*. Changes to the weights are published, in the first instance, through the U.K. Function Point User Group (formerly known as the European Function Point User Group).

We notice several differences between Symons' approach and that of Albrecht. As the system is considered to be built of logical transactions we no longer cater explicitly for interfaces. If an input or output happens to come from or go to another application and that caused the size of the task to increase then it should be reflected in the technical complexity factor. Second, enquiries can now be viewed as any other input/process/output combination. Finally, the ambiguities caused by the inclusion of logical files has gone. Logical files are extremely hard to define unambiguously, and at this logical level, the notion of a file is inappropriate; files suggest physical implementation. Instead we view logical transactions as interacting with entities. Symons' definition of an entity is, "*anything (object, real or abstract) in the real world about which the system provides information*" [2].

The calculations used in Mk II FPA make several assumptions. First, we assume that the relative sizes of the input and output components of a logical transaction are proportional to the number of data elements referred to by those components. This is based on the premise that the work required to analyse, design, develop, test and implement any input or output is influenced by the number of data elements in that component. Symons admits that this assumption may need to be refined in future versions of FPA. What is clear is that Mk II FPA is much more sensitive than Albrecht's method to changes in the number of data elements; Albrecht allowed a difference in size of only a factor of two between the most simple and the most complex processes.

The second assumption is that it is possible to determine a set of weights that reflect the relative size of an input data element versus an entity reference versus an output data element. The term 'reference' means creating, updating, deleting or reading an entity.

These assumptions highlight another difference between Mk I and Mk II FPA. Albrecht's measure of system size represents the value of function delivered to the user. This can be very subjective and so Mk II takes the system size scale as related to the effort to analyse, design and develop the functions of the system.

4.3. Technical complexity adjustment

A criticism of Albrecht's approach was that the technical complexity adjustment used fourteen general application characteristics which were sometimes hard to differentiate. Furthermore, it was questionable if these fourteen were the only ones that might affect a system development. Thus Mk II modifies the adjustment in two ways:

Five additional characteristics are added to the list. They are interfaces to other applications, special security features, direct access requirements for third parties, special user training facilities and documentation requirements. Furthermore, the list is open ended so that more characteristics can be added as appropriate.

A coefficient, *C,* has been added to the calculation to allow the relative weights of the nineteen or more characteristics to be amended if desired. Equation 3.3 shows how the technical complexity adjustment factors is calculated under the Mk II method. *C* is obtained by calibration (see next section).

$$TCA = 0.65 + C \times DI \qquad \textbf{(3.3)}$$

The nature of the Mk II method allows for revision or calibration of the relative weightings of the individual components of the TCA. However, there is little incentive to carry out such a task. There are two main reasons for this. First, as we shall see in the next section, the impact of new technology is decreasing the utility and importance of technical complexity adjustment. Second, the observed range of sizes of TCA is narrow. 90% of TCA scores lie in the range 0.75 to 0.95, a factor of 1.27 in relative size. Given this, the importance of TCA in computing the relative size of individual projects is minor. Therefore, there is little to be gained in refining the TCA factor.

4.4. Weighting factors and calibration

The question arises over how the weighting factors were obtained. How should the relative weight of a data element on an input component, an entity reference in the processing component, a data element on the output component of a logical transaction, and the degree of influence of a general application characteristic of the TCA be defined? The answer lies in the underlying purpose of Mk II FPA. According to Mk II FPA the purpose of sizing is to help measure productivity and to help in estimating. Therefore, the relative weights should reflect the relative average effort needed to analyse, design and develop the software.

The process of calibration tries to determine the weights by discovering how much time was spent on these various components during the course of system development. The first step is to ask staff who had first-hand knowledge of the project to *estimate* the breakdown of the total project effort into the work-hours required to deal with the information processing component of the system (X) and those dealing with the factors associated with the technical complexity adjustment (Y). By doing this for several projects and by not relying on one member of staff alone, a good average can be developed for an organisation. By doing this for projects in different organisations, an *industry average* can be determined.

The time spent on the components X and Y can be found quite easily by asking the staff the question '*was the proportion of X to Y in the ratio of 70:30 or 80:20?*'. Most people who are familiar with the components of Mk II FPA can arrive quickly at a good estimate. Using this ratio of X to Y it is possible to deduce a value of the *actual* TCA using equation 3.4.

$$TCA\ (actual)\ =\ 0.65(1+Y/X) \qquad \textbf{(3.4)}$$

The constant 0.65 is used to make the actual TCA comparable with the TCA derived from adding up the degrees of influence. Once sufficient projects have been examined and enough data gathered a graph can be plotted showing, for each project, the actual TCA on the horizontal axis and the computed TCA (equation 3.3) on the vertical axis. For the computed TCA a coefficient is chosen to force a best fit between the actual and computed values. From data gathered by Nolan, Norton and Co., the value of the best fit coefficient (*C* in equation 3.3) turns out to be 0.005 instead of Albrecht's value of 0.01.

It is interesting to observe that the weight-per-degree of influence is now half that of the one given by Albrecht in 1975. The Mk II interpretation of this is that in the intervening fifteen years, the difficulty of attaining the various technical factors has been greatly reduced. For example, data backup is provided automatically on modern computer systems and so the software designer need not address this explicitly. Fifteen years ago, cross-system communication was much more complicated to achieve than it is today given the latest technology and telecommunications protocols. One can envisage that in another fifteen years, the effect of TCA will be even less still as technology continues to advance. Ultimately, the coefficient will fall to zero and TCA will be discarded.

The second step in calibrating is to break down the hours spent on the information processing component (X) into the relative time spent on the three sub-components of input, processing and output. As before, staff can be prompted to give an estimate by asking '*were the percentage proportions 33:33:33?*' The reaction may then be '*no, we spent much more time than that on the output part*'. Thus, by a process of careful reasoning, instinctive assessments and asking a number of key staff, a reasonable estimate can be arrived at. Figure 3.2 shows a sample project where the weights have been calculated from the estimates of time spent on the three components.

Project Alpha	**Input**	**Process**	**Output**	**Total**
Counts	1022	931	2552	-
Est. Effort%	30	50	20	100
Work Hours	6778	11287	4519	22584
Hours/Count	6.63	12.13	1.77	-
UFP Weights	0.81	1.47	0.22	2.50

Figure 3.2

Notice that the weights are re-scaled so that their sum comes to 2.5. Using the value of 2.5 it was found that unadjusted function point counts compared very closely to counts derived using Albrecht's method in the range 200 to 400 UFPs. During its development, Symons considered it to be important to be able to compare results gained with his Mk II method with those given by Albrecht. The fact that the constant has remained is merely a convenience. The development and practice of the Mk II method prior to its release into the public domain in 1990 led to the *Industry Average* set of weights. These average weights were the result of work done by Nolan, Norton and Co. on thirty-two systems covering a wide range of applications, environments and organisations. The values of these weights are given as Figure 3.3.

Industry Average Weights		
W_i	W_e	W_o
0.58	1.66	0.26

Figure 3.3

Examination of these weights suggests that it takes less than half the effort to analyse, design, program and test an output data element than it does to do the same for an input data element. As input includes everything up to and including validation and output entails mere formatting of data, this would seem to make sense. Further, the effort expended per entity reference is nearly three times that for an input data element.

Naturally, for the most accurate results the method should be calibrated for the environment in which it is to be used. If comparisons of productivity are to be made across the site or organisation boundary then it would make sense to use the industry average weights.

4.5. Criticisms of the approach

The list of general application characteristics in the technical complexity adjustment was extended by Symons to nineteen or more. The reasoning behind this sounds plausible, but we must not forget the conclusions of Kitchenham [4] which cast doubt on the existing fourteen characteristics. Much useful work could be done in investigating the whole area of technical complexity adjustment to see if it might not be improved.

In calculating the size of a logical transaction, we said that the work required to analyse, design, develop, test and implement any input or output is influenced by the number of data elements in that component. This is a good illustration of one of the apparent ambiguities in FPA. Function points aim to measure the size of a system in terms of utility or delivered function. However, as can be seen from the above, intrinsic to the measure is a recognition of the *effort* required to produce that function. This is seemingly a conflict; either FPA is a measure of the size of the system or it is a measure of the difficulty of delivering that system. It would appear that it is both of these. To consider the size of the task in addition to the size of the system results in the measure being, to some extent at least, technology dependent; it is reasonable to assume that technology will make it easier to deliver useful function to the user as time goes by. A good illustration of this is the appearance in recent years of tools like Visual Basic. Visual Basic allows relatively easy production of software to run on the Microsoft Windows platform. The difficulty of a task in Visual Basic is generally much less than the same task using a conventional programming language such as C or Pascal.

According to the assumptions of Symons, as technology progresses function point counts will still be reflecting the technological environment of the late 1980s. Of course, Symons admits that later versions of FPA may need to modify the underlying assumptions. Add to this the technical complexity adjustment and we can see that FPA is failing to meet one of its core objectives, that of providing a measure of system size independent of technological constraints. Indeed, Symons states [2],

> "*an important conclusion . . . is that function points are not a technology-independent measure of system size, which was one of Albrecht's stated aims. The technology dependence is implicit in the weights used for the UFP and TCF components.*"

Furthermore, this is not limited to Albrecht's method but "*applies equally for the Albrecht and Mark II approaches*". Indeed, the weights that an organisation uses in its function point measurements imply a baseline or "normative technology". Assuming that technology changes and a system is built under the new technology, FPA would still give it a size based on that normative technology; that is, the advantages of the new technology would not be reflected in the system size. Symons argues that this is useful for comparing productivity for different technological environments. As long as every organisation uses the same weights, then systems can be compared across organisational boundaries. For this reason, the weights supplied with Mk II FPA are said to be industry average. They have been compiled as an average figure from many organisations sampled by Nolan, Norton and Co. Ltd who are committed to updating them as necessary. This is analogous to the one hundred companies that form the Financial Times Share Index. The companies in the index change regularly to take into account a changing environment; however, the Index remains a baseline measure of economic performance. As long as the weights are updated regularly, then a function point count of 1993 will be comparable with a function point count of 2003.

Then again, perhaps, the size of a logical transaction does not, after all, have anything to do with the work required to analyse, design, test and implement it; perhaps Symons' assumption is simply wrong. It seems more reasonable to assume that a transaction does more or is simply more useful (delivers more function) the more data elements associated with it. Of course, this means that poor program and data base design work could give artificially high scores to relatively simple transactions, but then this could be the case anyway. However, given that Mk II counts are supposed to be taken on *logical* transactions, then the physical implementation should not affect the results. The restriction of the method to logical rather than physical aspects is very sensible.

Another apparent problem of FPA (Albrecht and Symons) is the way in which it attempts to measure productivity. The standard formula for calculating productivity is given as Equation 3.5.

$$P = \frac{O}{I} \tag{3.5}$$

Equation 3.5 shows productivity (*P*) is found by dividing the output of the task (*O*) by the input, or effort (*I*). In other words, we could measure *P* as being 10 cars-per-month, or 8000 LOC-per-quarter etc. However, one could argue that FPA, by its use of TCA, actually employs a formula like that given in Equation 3.6.

$$P = \frac{O}{I + i} \tag{3.6}$$

Equation 3.6 shows that the effort required to produce the output is accounted for twice as some of the work effort (*i*) is expressed in the TCA.

4.6. What next?

Given the above criticisms, it would seem that time will bring more advanced versions of FPA. If I were asked to predict trends, then I certainly see the technical complexity adjustment factors being discarded.

Perhaps there is some scope for a new version of FPA which completely disregards the effort required to deliver user function. The new method should be purist in its outlook and try to assess only those features that relate directly to the size of the utility delivered to the user.

It would seem desirable to have an absolute measure of system size in terms of delivered function without any recourse to accounting for technological constraints. Technology merely governs the complexity with which a particular piece of user-required function can be implemented. A true measure of system size should not be affected by such details.

There are two problems that I see that make this hard to do. First, function-oriented measures are, by definition, relative abstracts rather than concrete absolutes. To illustrate this, ask the question what is the fundamental unit of function? The nearest we can get to an answer in a computer system is the number of machine instructions carried out to meet the user's requirements. Of course, such a measure is affected by a number of factors, mostly efficiency-based, such as:

- compiler efficiency,
- source language coding efficiency,
- machine architecture differences.

We could argue that a good compiler would reduce any of the possible forms of expression of an algorithm to a single machine-code representation, thus circumventing the second factor. However, I feel this is wishful thinking. Weighting factors would have to be applied to the measure to normalise the different instruction sets of the different computer architecture. Naturally a RISC-based machine will require fewer instructions to execute the same task as an older machine.

Assuming that the machine-language-statement measure is even attainable, it would still not address the question of how useful a piece of function is to the user. Consider the Pascal program fragments in figures 3.4 and 3.5.

```
Readln (number) ;
WHILE number <> 9999 DO
```

```
   BEGIN
   Writeln (number * number ) ;
   Readln (number)
   END ;
```

Figure 3.4

```
Write ('Please enter a number (9999 to finish) : ') ;
Readln (number) ;
WHILE number <> 9999 DO
   BEGIN
   Writeln (number : 6, ' squared is ', number * number : 6 ) ;
   Write ('Please enter a number (9999 to finish) : ') ;
   Readln (number)
   END ;
```

Figure 3.5

Which of the two program fragments is more useful to the user? The answer is obvious, but it is not achievable simply by counting the number of machine instructions needed. Then again, should a function-based measure need to consider usefulness? The function point (as defined by Symons and not Albrecht) avoids consideration of function value as it is too subjective.

The other problem that arises when trying to separate utility from the effort required to deliver that utility is that it is simply hard to do. Once we move away from concrete, physical measures any discussion of size in terms of utility invariably includes the relative effort required to deliver that function. The car manufacturing example in chapter 1 took account of effort implicitly. We said that the luxury limousine delivered more user-function than the cheap run-about and, by extension, required more effort to produce. This seems like common sense; of course, something that does more will be more difficult to make. But is this any reason to include the complexity in our measures? Because increased effort seems to occur when functionally rich software is produced, then we take that effort as a measure of utility.

However, surely the effort is merely a pointer to the fact that the software may have more function than a system that was less difficult to produce. It seems that we are including technical complexity in our measures because we cannot, in fact, measure function. Technical complexity appears to be a symptom of function. In the absence of a definition of the fundamental unit of system function (or 'functionality' as the trendies would have us say) we are obliged to measure something else that is related but concrete.

To illustrate the difficulty of this problem, consider two nails. Both are two inches in length and both are made of steel. As they are nails, they can both be hammered into things as needed. At this level of concrete, physical measures the two nails are the same; we could say that they are functionally equivalent. When we learn that one nail is made from malleable, low-grade steel, and the other from strong, high-grade ice-tempered steel, what does this do to our perceptions? Are they still functionally equivalent? If the customer's requirement was merely for a nail with which to hang a picture, then they are both equal, even though the latter may have cost five times the price of the former. In terms of user requirements they are functionally equivalent. FPA would treat two such software systems in the same way; it would not take into account delivered function that was not specifically requested by the customer regardless of the cost or effort involved in producing it.

Given what we know about the two nails, can we infer anything about their function? In terms of picture hanging they are identical. What of holding a roof together? We can argue that the ice-tempered nail has more function because it can be used for more - the cheaper nail could not be put to the same range of uses. This is well and good, but how do we measure the functional difference? We can mention the attributes of the nails: one is a two-inch low-grade steel nail, the other is a two-inch high-grade, ice-tempered nail which possesses a higher load bearing capability and tensile strength. Fine. We have assessed the function of the nails in terms of measurable attributes; we can measure tensile strength and load bearing ability. What we also know is that the functionally rich nail is harder to make. At least it is until a new technology comes along that allows us to manufacture such nails as easily as the cheaper nails. However, despite the complexities involved in nail manufacture, there are measurable differences between the two in terms of what they are capable of, or what they can do.

There does not seem to be a problem in measuring function in terms of attributes when dealing with small and discrete items like nuts, bolts and nails. We see though that this is a problem of scale. When we consider two cars, how then do we measure functional differences?

A cheap run-about can do, in general terms, all that the limousine can do. That is, the two cars can drive on urban roads and motorways; they can both carry passengers; they can both carry luggage. In fact, the smaller cars can often carry more luggage than the larger cars. (Compare a Jaguar's boot with that of a Ford Escort to see this.) What is it that makes one more functionally rich than the other? We seem to know instinctively that one is functionally larger than the other and we rightly expect that one is harder to produce than the other. Again, effort and function are related. We could view a car in the same way as a software tool in that it has a multiplicity of uses any subset of which may be useful to the driver. Then we could measure each use directly by using physical measures such as load capacity in cubic feet, top speed, fuel economy etc. The problem remains with finding the equivalent set of measures that applies to software. There are all sorts of measures we could apply to measure automobile function, but in the final analysis, the best measure of delivered function is does the car meet the user's requirements and expectations?

As we moved from nails up to cars, the measures of function became harder to define. When we move away from manufacturing into software engineering, the difficulties become even greater. The attributes that apply to nails do not apply here. All we are left with is the (somewhat suspect) link between effort and function. That both methods of FPA (and their derivatives) take into account at some stage the effort involved in delivering function is testimony to the fact that we have no concrete way to assess utility.

It is clear that the function point is nothing more than a relative measure of somebody's idea of what system size is when attempting to assess it in terms of delivered function. It is not a concrete, direct measure and there is no scale against which to compare it. However, it is certainly perceived as being more useful than the line of code. It is a good first step on the way to finding the fundamental unit of software function. If one exists at all, that is.

To return to an earlier theme, we might judge the utility of a car by its ability to meet user requirements. If a customer does not find a car acceptable for his needs, no matter how functionally rich it is measured in whatever function-oriented metric we may choose, to that customer it is useless.

Perhaps the answer to our problem lies in measuring not the elusive utility, but customer satisfaction. Regardless of whether team A was more productive in producing 100 function points worth of software than team B, if team B's software meets customer satisfaction requirements and team A's does not, then team A will be looking for new jobs.

5. Applying Mk II FPA

Apart from the obvious conceptual difficulty in understanding the very nature of a function point, the ideas underlying the technique are quite simple. However, one should not become to emboldened and expect to become a skilled practitioner of the method overnight. Symons [6] warns that function point analysis '*is not as easy to carry out as it first appears, and certainly when first starting, it is advisable to have available someone with experience of the chosen method*'. Arguably Mk II FPA is easier to apply than Albrecht's given its focus on logical transactions and entities. Comparison of the counting practices manuals published by IFPUG and UFPUG which deal with Albrecht and Mk II respectively bear this assertion out.

Despite the advances made by the Mk II method, it is still important to bear in mind that it is only the second development in this line of measurement methods. As such, its application boundary is restricted. Mk II FPA, it is claimed by Symons, cannot be used to measure software tools. An illustration helps to explain this point. In one of Albrecht's early definitions of his method it was suggested that if a report-writing tool was provided as part of an application then the function points should be included for all possible reports that the tool could generate. It soon becomes clear that trying to give a count *for each* possible output of a report-writer is near to impossible; even with a very simple command vocabulary, the number of different reports that could be generated is extremely large, if not infinite. We can see then that other utilities like compilers, operating systems and expert systems are also inappropriate as a target for FPA. Perhaps a way could be found to count the underlying generic output of such software rather than the various individual cases as Mk II FPA demands.

The scope of Mk II FPA is given by Figure 3.6. The method is clearly geared towards business transaction application systems. Certain environments, such as those developing real-time systems, are developing specialised forms of Mk II FPA to cater for their particular characteristics.

The detailed rules for applying Mk II FPA are fully described in Symons [6] and so no attempt is made here to outline them.

Finally, does it really matter whether we use LOC, FPA or Mk II FPA in our system measurements? If very carefully defined and used, the line-of-code can provide a reasonable measure of system size that is suitable for assessing the productivity of the individual programmer. It can also be applied crudely to an entire project. Further, the line-of-code is the only measure of size available that works across the whole range of software regardless of type, development and implementation.

For business application systems that can be described well using the notion of the logical transaction (be they on-line or batch) the function point appears to be much more suitable for measuring productivity and assessing system size. It is obvious that Albrecht's method is better than using LOC for such purposes. However, we have seen that some of the ideas that underpin Albrecht's method are flawed and so Mk II FPA is the method that gives us the closest assessment of the true size of data-rich business applications. Almost certainly Mk III FPA will overcome some of the limitations of Symons' method; it only remains to develop it.

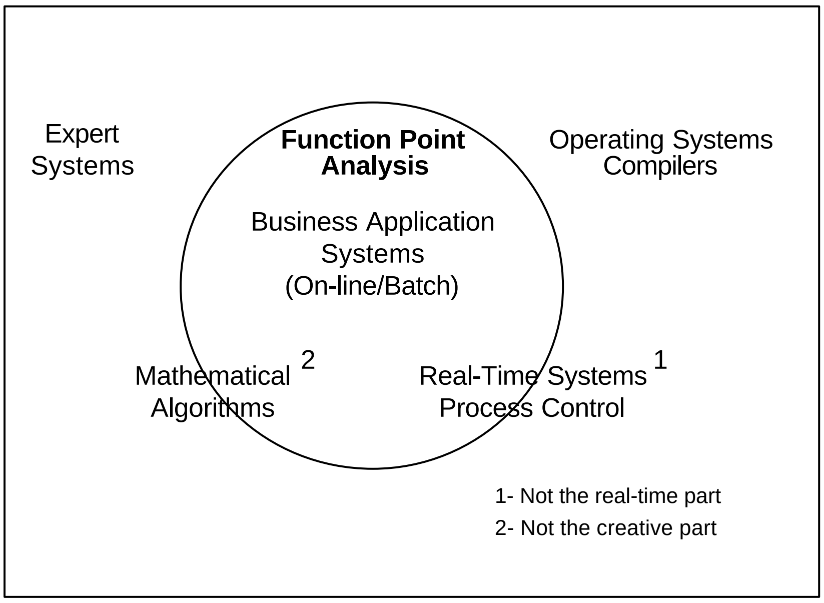

Figure 3.6: The Scope of Mk II FPA

6. References

[1] *AD/M Productivity Measurement and Estimate Validation*; Allan Albrecht, IBM Corporate Information Systems and Administration; 1985

[2] *Function Point Analysis: Difficulties and Improvements,* C. R. Symons, IEEE Transactions on Software Engineering, 1985.

[3] *Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation*; Albrecht A.J., & Gaffney J.E.; IEEE Transactions on Software Engineering, Vol SE-9, No. 6; November 1983.

[4] *Empirical Studies of the Assumptions Underlying Software Cost Estimation Models*; B. A. Kitchenham, NCC Ltd; 1991.

[5] *Function Point Counting Practices Manual: Release 3.0*; IFPUG Counting Practices Committee; 1990.

[6] *Software Sizing and Estimating: Mk II FPA;* Symons C.R.; John Wiley & Sons; 1991

[7] *A Cooperative Industry Study: Software Development/Maintenance Productivity*; Xerox Corporation; 1985