Function Points

Christopher J. Lokan

School of Information Technology and Electrical Engineering,
UNSW@ADFA,
Northcott Drive,
Canberra ACT 2600,
Australia

Email: c.lokan@adfa.edu.au
Phone: +61 2 6268 8060

Abstract

Functional size measurement — measuring the functionality delivered by a software application to its users — is vital to software project managers. Its uses include estimation, managing scope in project planning and tracking, and measuring productivity.

There is no direct scale for measuring software functionality. One must instead decide which aspects of software to measure, that are considered to capture functionality; how to measure those aspects; and how to combine the measurements into an overall measure of application size.

Many methods for how to do this have been proposed, beginning with Allan Albrecht's invention of Function Point Analysis in 1979. The methods vary in what things are measured and how the measuring is done; and also in usefulness, applicability, and industry acceptance. Two methods have achieved significant industry acceptance, and a third is on its way.

In this chapter we trace the evolution of functional size measurement, from Albrecht's original ideas through to the recent development of ISO standards. We describe Albrecht's method, and what has been learned about it through extensive experience and empirical research. We present some interesting variants on Albrecht's method that did not achieve widespread success, as well as the two other methods that can now be described as mainstream. We describe research concerning function points, and conclude with thoughts on the current status of functional size measurement and some expectations for future work.

Contents

1 Introduction **3**
- 1.1 Length vs. Functionality 3
- 1.2 Outline of this chapter 5

2 Albrecht/IFPUG Function Points **5**

2.1 The 1979 Definition 6
2.2 The 1984 Revision 6
2.2.1 Unadjusted Function Points 7
2.2.2 Adjusted Function Points 9
2.3 IFPUG 10

3 Experience with IFPUG Function Points 11
3.1 Success of Function Points 11
3.2 Theoretical problems of construction 12
3.3 Weighting the components 13
3.4 What is actually measured? 14
3.5 Subjectivity 14
3.6 What is missed out? 15
3.7 The General Systems Characteristics and VAF 15
3.7.1 General Systems Characteristics 15
3.7.2 Value Adjustment Factor 16
3.8 Relationships between FP component types 17
3.8.1 Function point breakdown 17
3.8.2 Correlations between component types 18
3.9 Backfiring 19
3.10 Summary 20

4 Mark II Function Points 21
4.1 Symons' criticisms 21
4.2 Mark II Function Points 22
4.3 Experience with Mark II Function Points 23

5 Some other early variations 24
5.1 DeMarco's Function Weight 24
5.2 3D Function Points 25
5.3 Feature Points 26
5.3.1 How to count algorithms? 27
5.4 Summary 27

6 COSMIC 27
6.1 Full Function Points 28
6.2 COSMIC Full Function Points 29
6.3 Experience with COSMIC 32

7 Function points for Object-Oriented software 32
7.1 Mapping OO concepts to Function Points 33
7.2 Function Point-like measures for OO 34
7.2.1 Object Oriented Function Points 35
7.2.2 Use Case Points 36
7.3 Summary 37

8 Function Point Standards 38

9 Conclusions 39

1 Introduction

Throughout the history of software measurement, one attribute of computer software has always been seen as fundamental: its size.

Software is written to solve problems. Size measurement can focus on either the problem, or the program:

- The problem is expressed in the Software Requirements Specification. This document states the requirements that the delivered software must satisfy. It defines the *functionality* to be delivered by the software to the user.

  Problem size can be interpreted as the amount of functionality delivered by the software. Intuitively, software that lets the user do more things has greater functionality, and solves a larger problem.

- The program that solves the problem has a physical size. Various measures of program *length* capture this aspect of size.

Length and functionality are different aspects of size (although they tend to increase together). Programs with the same functionality can differ widely in length, and programs of the same length can deliver very different functionality.

Measuring length is easy (though there are issues to be careful of). Measuring functionality is harder. This chapter aims to give the reader an understanding of how software functionality is measured. We begin with a brief comparison of length and functionality measures, to demonstrate the need for measures of functionality.

1.1 Length vs. Functionality

The oldest measure of software size is simply the number of lines of code ("LOC").

Which lines to count (all lines? executable lines only? should comments be included?) is a matter of argument. The definition most commonly adopted is

> any line of program text that is not a blank line or comment, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, and executable and non-executable statements [18].

This is simply the physical length of the program, as a compiler sees it.

Physical LOC is determined by program layout, which can be arbitrary in many languages. Other measures have been proposed that appear less arbitrary. The main ones are the number of tokens that make up the program text, and the number of statements, or logical lines, in the program. In principle, these are much better measures of length: they measure the semantic units in the program, and are less prone to the vagaries of individual programming styles. In practice, these measures are all highly correlated with each other, and with physical LOC. The theoretical deficiencies of physical LOC are not such a problem in practice, with respect to other length measures.

Length measures are valuable to developers. There is a long history of using them to predict things of interest about a software development project (like number of errors, development effort, maintenance effort, *etc.*). They are genuine measures of packaging requirements for finished software [33], and the

storage required for source code [25]. Finally, they are easy to measure: many tools exist to count program length.

Length measures have been widely criticized (see for example Jones [58]). Criticisms include:

- sensitivity to individual programming style;
- lack of standard definitions;
- language dependent;
- difficult to apply to 4GL's, spreadsheets, application generators, *etc.*;
- difficulty of dealing with multiple languages;
- inability to measure specifications;
- not available until late in the life cycle, and difficult to estimate earlier;
- not relevant to users.

These criticisms are all well founded, to differing degrees. The first turns out not to matter so much in practice: in programming teams, individual variations tend to average out, and coding standards and pretty-printing software can also reduce variation. The next three can be largely overcome by applying some common sense. But the last three represent a real problem, corresponding to a shift in perspective from that of the programmer to that of the user. Program length means nothing to users.

The alternative is to try to define "functional measures" that measure the functionality of software, from a user's point of view. This involves identifying things in the problem specification that represent functionality, and somehow measuring those things.

If successful in their aims, functional measures bring several benefits [58]:

- understandable to users, and based on things that users care about;
- available early, at the specification stage;
- independent of programming language and development technology;
- reflective of real economic productivity.

Effective functional measures provide a more meaningful basis than length measures for things like measuring or benchmarking software development productivity, and quantifying an organization's software portfolio. They are also used effectively in software acquisition contracts, enabling the scope of a requirement — or change to a requirement — to be properly assessed.

Numerous functional size measures have been proposed, varying in what things are measured and how they are measured. The rest of this chapter describes the most important of those proposals.

1.2 Outline of this chapter

When people talk about functional size measures, they usually mean "function points" — more specifically, "IFPUG function points" ("IFPUG" stands for International Function Point Users Group). Section 2 describes IFPUG function points and their evolution.

Section 3 describes what has been learned, through experience and empirical research, about IFPUG function points. This section identifies the strengths, weaknesses, and limitations of IFPUG function points. It sets the scene for following sections that describe other proposals for functional size measurement.

Section 4 describes Mark II Function Points — until recently, the only alternative to IFPUG function points with significant industry acceptance. Section 5 presents a few other proposals, which featured important ideas but for one reason or another never achieved wide usage. Section 6 describes COSMIC Full Function Points, emerging now as a significant alternative.

Object orientation is important in today's software development world. Applying function points to object oriented software is an area of current research interest. Section 7 describes the main activity in this area.

Section 8 looks briefly at recently-approved ISO standards for functional size measurement in general, and some function point approaches in particular.

Section 9 draws everything together and presents some thoughts on future directions.

2 Albrecht/IFPUG Function Points

Function points were introduced by Allan Albrecht, of IBM.

Albrecht's aim was to be able to measure application development productivity in IBM's DP Services organization. His landmark paper from 1979 [8] is really all about measuring productivity. He begins with a description of the application development process used by the DP Services organization. He comments on why to measure productivity, and some things to be careful about when measuring productivity. He describes the measures he uses for product and cost. He presents and interprets trends in productivity in IBM's environment.

It is a good paper about measuring productivity. But what it is remembered for is one of the measures he used. He states:

> To measure productivity we had to define and measure a product and a cost. The product that was analyzed was function value delivered ... The cost used was the work-hours contributed.

For the product measure, Albrecht wanted to avoid measures "such as lines of code that can have widely differing values depending on the technology used". His objective was

> to develop a relative measure of function value delivered to the user that was independent of the particular technology or approach used.

He explained how his measure of function value was constructed, and gave it a name: *function points*.

2.1 The 1979 Definition

Albrecht defined a two-stage process for calculating function points.

1. Count certain factors that are the outward manifestations of an application, and weight those counts by numbers designed to reflect the function value to the customer.
2. Adjust that result for the effect of other factors relating to the technical complexity of the application.

The result is labelled as "function points".

The keys to this process are: which factors are considered to represent the outward manifestations of the application; and what weights to give to each factor.

Albrecht listed four factors, or types of component: external user inputs, outputs, inquiries, and master files. These four component types were identified from five years of experience with projects in IBM's systems development environment.

The relative weights given to inputs, outputs, inquiries, and master files were 4, 5, 4, and 10 respectively. These weights were determined "by debate and trial".

For the technical complexity adjustment phase, ten factors were identified. These were the extent to which the design or implementation of the application involved: reliable backup, data communications, distributed processing functions, performance considerations, a heavily used operational configuration, online data entry, data entry involving complex transactions, online update of master files, complex data and transaction components, and complex internal processing. Each of these ten factors could affect the function point value by 5%. The maximum possible adjustment was 50%, expressed as plus or minus 25% so that average technical complexity resulted in no change to the original function point value.

Validation of function points took the form of showing empirical relationships between function points and other measures of interest. Relationships were demonstrated between function points and effort, and between function points and LOC for three languages [8, 10].

Some essential features, common to all function point approaches since, are visible here. The focus is on externally-visible aspects of an application, that a user can see. A measure is sought that is independent of the technology or approach used for implementation. The structure involves identifying, counting, and weighting important elements.

Note also that this definition of function points is oriented towards file-oriented data processing; and that there are several subjective aspects involved in the calculation.

2.2 The 1984 Revision

Function points did not last long in the form that Albrecht initially defined them. Albrecht's 1983 paper with Gaffney [10] described a revised structure, which was documented fully in an IBM report published in November 1984 [9].

This revised formulation of function points is much more elaborate than the 1979 definition. It still has subjective aspects, though guidelines were provided

to try to reduce this. The method is still geared towards file-oriented data processing.

There are three main changes from 1979:

- There are five component types, not four, as "Master files" are separated into internal and external files.
- Instead of a single overall variation of plus or minus 2.5% to recognize simple or complex files and transactions, each component of the application is classified individually as having low, average or high functional complexity. The classification is based on numbers of file types, record types, and data element types involved.
- The technical complexity adjustment factor, previously plus or minus 25%, became plus or minus 35% based on a different set of adjustment factors (three were removed, and seven added). Guidelines were defined to reduce the subjectivity in scoring the adjustment factors.

Since this definition is still at the core of "Albrecht/IFPUG function points", the most widely-used function point approach today, it is worth presenting it now in detail.

2.2.1 Unadjusted Function Points

The first (and most important) phase of Function Point Analysis ("FPA") involves identifying certain components of the system that provide functionality to the user. Functionality is defined with reference to the "system boundary": the interface between the external user and the application .

Albrecht defined the five component types as follows [9]:

- *external inputs*: "data or user control input ... that enters the external boundary of the application being measured, and adds or changes data in a logical internal file type."
- *external outputs*: "data or control output ... that leaves the external boundary of the application being measured."
- *external inquiries*: "unique input/output combination, where an input causes and generates an immediate output."
- *internal logical files*: "logical group of user data or control information in the application ... from the viewpoint of the user, that is generated, used, and maintained by the application."
- *external interface files*: "files passed or shared between applications ... logical group of user or control information that enters or leaves the application."

Figure 1 depicts the five components.

The number of function points awarded to a component depends on its "functional complexity". This is determined from from the numbers of file types, record types, and data element types involved. Tables 1, 2 and 3 respectively show how to determine the functional complexity of an input, output or inquiry,

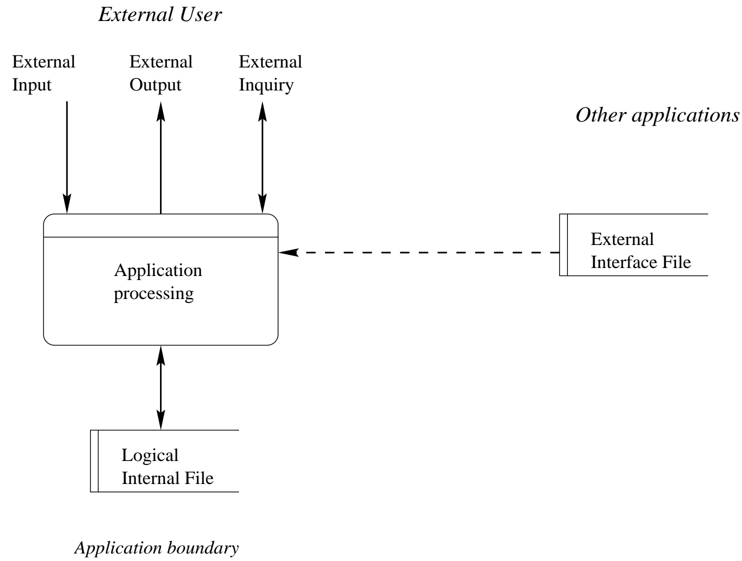

Figure 1: Components in Albrecht Function Points

File types	Data element types		
referenced	1–4	5–15	16+
0–1	Low	Low	Average
2	Low	Average	High
3+	Average	High	High

Table 1: Functional complexity of External Inputs

File types	Data element types		
referenced	1–5	6–19	20+
<2	Low	Low	Average
2–3	Low	Average	High
4+	Average	High	High

Table 2: Functional complexity of External Outputs, Inquiries

Record	Data element types		
types	1–19	20–50	51+
0–1	Low	Low	Average
2–5	Low	Average	High
6+	Average	High	High

Table 3: Functional complexity of files

Component	Functional complexity		
	Low	Average	High
Input	3	4	6
Output	4	5	7
Inquiry	3	4	6
Internal file	7	10	15
External file	5	7	10

Table 4: Function points per component

and internal or external file. Table 4 shows the number of function points awarded for each type of component and each functional complexity level.

To recap, the first phase of FPA involves defining the system boundary; identifying the inputs, outputs, inquiries, internal logical files, and external interface files; awarding a number of function points to each individual input, output, inquiry, internal file and external file according to its functional complexity; and summing all of those function point values. This results in a total unadjusted function point ("UFP") value, which is seen as capturing the user-oriented functionality inherent in the system.

2.2.2 Adjusted Function Points

The second phase of FPA involves calculating a Value Adjustment Factor ("VAF"), based on fourteen general system characteristics ("GSC's"), and using it to adjust the UFP value.

The general system characteristics are supposed to capture other aspects of the functionality of the system: "pervasive general factors that are not sufficiently represented by the previously discussed transactional and data functions" [30]. They are evaluated for the system as a whole, rather than applying to individual elements.

The fourteen characteristics are:

- *Data communications*: the degree to which the application communicates directly with the processor. (Batch applications are at one extreme; real-time, telecommunication, or process control systems at the other.)

- *Distributed data processing*: the degree to which distributed data or processing functions are a characteristic of the application, within the application boundary.

- *Performance*: the degree to which response time and throughput performance must be considered in developing the application.

- *Heavily used configuration*: a heavily used operational configuration requires special design considerations.

- *Transaction rate*: a high transaction rate influences the design, development, installation and support of the system.

- *Online data entry*: online data entry and control functions are provided in the application.

- *End user efficiency*: the degree to which online functions emphasize a design for end user efficiency.
- *Online update*: the application involves online update of internal files.
- *Complex processing*: the degree to which processing logic influenced the development of the application.
- *Reusability*: the degree to which the application and the code in the application have been specifically designed, developed, and supported to be reusable
- *Ease of installation*: the degree to which conversion from previous environments influenced the development of the application; conversion and installation ease are characteristics of the application.
- *Ease of operation*: the degree to which the application attends to operational aspects, such as startup, backup and recovery processes, without requiring manual intervention.
- *Multiple sites*: the degree to which the application is designed, developed and supported to be installed at multiple sites for multiple organizations.
- *Facilitate change*: the degree to which the application is designed, developed and supported for easy modification of processing logic or data structure.

Seven of these characteristics were present in Albrecht's initial formulation of function points. Three of the original ten were dropped (element complexity — now assessed separately for each element; reliable backup; and whether online data entry involves complex transactions). Seven characteristics have been added: transaction rate, end user efficiency, reuse, ease of installation, ease of operation, multiple sites, facilitate change.

The GSC's are used to adjust the UFP value as follows. The degree of influence of each characteristic is scored on a scale from 0 to 5; the sum of fourteen scores is a result from 0 to 70; this is converted linearly to a value from 0.65 to 1.35. That value (the VAF) is multiplied by the UFP to give the adjusted function point ("AFP") value.

2.3 IFPUG

In 1986, several users of function points formed a non-profit organization called the International Function Point Users Group ("IFPUG").

IFPUG's aims are to spread the use of function points, and to standardize the way people count and use function points. It does this by holding regular conferences, publishing guides and manuals, and running exams (people who pass IFPUG's exam are styled "Certified Function Point Specialists").

IFPUG has made minor changes to terminology, but the basic structure of function points has remained essentially unchanged since 1984.

IFPUG has several committees, the most important being the Counting Practices Committee. This committee defines how function points should be counted in various situations — creating standard guidelines and resolving inconsistencies. The results are published in the Counting Practices Manual. This

document has gone through several Releases, including 3.0 [34] in 1990, 4.0 [36] in 1994, 4.1 [40] in 1999, and 4.2 [41] in 2004.

Another part of IFPUG's activity is to make recommendations on how types of software and aspects of software that were not considered in 1984 can be mapped into the original function point structure [35, 37, 38, 39].

Function points as Albrecht defined them and IFPUG maintains them are referred to hereafter as "IFPUG function points". Where "function points" are referred to without qualification, it is IFPUG function points we mean.

3 Experience with IFPUG Function Points

Much has been learned over the years about the strengths and weaknesses of function points.

The strength of IFPUG function points is pragmatic: they have been used successfully in practice for many years.

But they have been criticized on several grounds. These include theoretical problems concerning their construction; doubts about the detail of various steps; disagreements about exactly what they measure; subjectivity of measurement; some important aspects of software or types of software are not measured.

This section describes some of the experience and knowledge gained about function points. We begin with their successful use in practice, and then look at their problems and limitations.

3.1 Success of Function Points

We have seen that function points were developed in the context of application development in IBM in the 1970's and early 1980's. They are geared towards data-rich business applications, a dominant application domain in IBM then.

Functional size measurement has some key characteristics that are easy to promote. The orientation towards users, and early availability (measurable at the specification stage) are genuinely important and attractive features.

Albrecht/IFPUG function points "got in early", at a time when the need for functional size measurement was beginning to be appreciated and there was nothing much else around in the area. Since they were indeed found to be a useful measure for MIS software, they became a pragmatic success.

Function points were marketed heavily by IBM, IFPUG, and several consultants. Estimation tools based on function points were developed successfully. Books were written describing FPA (eg [23, 29, 30, 58]) and experience with it in different industry sectors (eg [58]), and many articles written promoting its use (eg [28]).

By the mid-1990's, it was forecast that "function points would become the primary means for measuring application size, reaching a penetration of approximately 50 percent of development organizations by the year 2000" [32] (quoted in [28]). FPA was probably used to some extent by the information systems departments of most major companies and government departments in North America, much of Western Europe, and other parts of the world [98].

Though their use may have declined [98], IFPUG function points are still very much the dominant functional size measure in the world today. They have almost complete hold of the functional sizing market everywhere except

the UK, where IFPUG function points and Mark II function points (described in Section 4) have about half the market each. COSMIC-FFP (described in Section 6) is gaining industry acceptance, but is still in its early days.

The key word is "pragmatic". Practitioners have found function points to be a useful measure, even while recognizing some of the criticisms. "Are function points a perfect metric? No. Are they a useful metric? In my experience, yes" [28]. "If [a measure] proves to be empirically correlated to some software phenomena we wish to forecast and control ... why should we refuse it only because the measure is not formally validated?" [79].

A considerable body of knowledge and experience has accumulated. Practitioners have learned how to use function points effectively, what variations to expect between different industry sectors, and even how to exploit some of the structural characteristics of function points that worry researchers. This is most true in the MIS domain; function points are used much less in real-time software and communications software.

3.2 Theoretical problems of construction

Abran and Robillard investigated the whole underlying structure of function point analysis, considering all the measurement scales and mathematical operations used throughout the process [5, 6].

They observe that the process begins with measurement on an absolute scale (identifying components, recognizing their types, counting the data and record types they involve). Abran called the collection of operations to this point the Function Point Measurement Model ("FPMM") [3]; the rest of the process he called the Function Point Productivity Model ("FPPM").

The process then moves to ordinal scales (classifying components into three levels of functional complexity), losing information in the process but still a valid operation according to measurement theory. Thereafter, things go awry. The assignment of points to components of different levels takes measurement from the ordinal scale back to the ratio scale — adding information, but without a theoretical underpinning.

Calculation of the VAF is similarly flawed. Invalid operations are performed, as ordinal scale measurements are converted to absolute scale values with no underpinning theory to justify the conversion. Effectively, arithmetic is performed on ordinal scale values as though they were measured on a ratio scale.

Hence, from a theoretical point of view function points cannot be considered a proper measure according to measurement theory. "The mapping, or measurement space, of FPA undoubtedly needs to be clarified if it is to be trusted as a valid measurement system" [6].

Empirical research showed that for the purposes of understanding productivity — the original purpose of function points, after all — none of the steps of the FPPM added significant information. Effort could be explained as well from the elementary measurements from the FPMM as from the final function point value resulting from the full process [6].

Kitchenham *et al.* proposed a framework for software measurement validation in 1995 [67]. They too were critical of the construction of IFPUG function points. They noted the scale transformations, and invalid operations performed on measurements from different scales. Further criticisms were that function points are discontinuous (no value of 1 or 2 FPs is possible), and there is no

"unit" value. They are not fully based on a ratio scale. The result is a measure that is not valid in various ways. The risk is that it may behave in unexpected ways, for example asserting incorrectly that one program is larger than another [64].

3.3 Weighting the components

The origin, validity and objectivity of the weights (function points per component type) have long been questioned.

Albrecht saw the weights as measures of the relative value of the function to the user/customer. They were chosen by debate and trial within IBM.

Symons published the first criticisms of function points [99]. He had several concerns about the weights. He questioned whether they would be valid in all circumstances. He felt that they contained some anomalies: components containing very many data elements receive at most twice the function points of a component containing only one data element. He desired some more objective assessment of the weights.

The concept of function value to the user has troubled others as well. Skeptics of this subjective concept view the weights as being really related to effort. For example:

> in the absence of any convincingly objective measure of user function or value of a transaction ... independent of the cost of implementation, we are forced to relate function or value to some cost-related reference measure [106].

In devising Mark II function points (Section 4), Symons equated value with production effort.

Symons has reported Albrecht as saying the weights were chosen "on the basis of the relative numbers of data elements appearing on each of these four component types, based on an examination of a number of systems which were being studied at that time in IBM" [100]. If the weights reflect the amount of data present in a component, there is some link with effort: programming effort is required to validate input data elements, and generate and format output elements.

There has been one attempt to derive weights that really do reflect perceived value to the user. Wittig *et al.* asked users to give their perceptions of amount of functionality provided by various FPA components. Numerous pairwise comparisons were made in which users assessed which of two components was larger, and by how much. With sufficient samples of all combinations of component types and functional complexity levels, it becomes possible to develop a set of weights empirically (using the Analytic Hierarchy Process [92] process) that reflect the users' perceptions of relative functionality. Initial results from a pilot study of 23 projects [111] were confirmed by further study of another 22 projects [90]. They are presented in Table 5.

These differ from Albrecht's weights in several respects. Albrecht weights files more heavily than transactions, but here it is generally the other way around. With Albrecht's weights, the maximum weight for a component type is always about twice the minimum; here the maximum is about three times the minimum for transaction types, and four times the minimum for file types. Inputs have the least weight under Albrecht, and the most weight here.

	Functional complexity		
Component	Low	Average	High
Input	4	7	13
Output	3	5	10
Inquiry	3	5	9
Internal file	3	5	12
External file	2	5	9

Table 5: Weights derived by Wittig

Wittig's results provide an empirical basis for changing the weights in FPA so that function points really do reflect functionality to the user. The research has gone no further, though, and the new weights have not been adopted by IFPUG.

3.4 What is actually measured?

Kitchenham *et al.* noted that it is not clear what function points actually measure: functionality is one concept, product size as perceived by a user is a different concept [67].

Whether IFPUG function points measure size alone is a point of strong debate. The aim is to measure the problem, independent of the technology used to implement the solution. Critics argue that function points don't actually work that way. Technology dependence is implicit in the choice of weights for function point components [99]. Technology dependencies can occur in all aspects of the FPA approach [106].

Further, several of the characteristics measured in the adjustment phase of FPA are really effort drivers, not size drivers [72]. Including them in the sizing process corrupts function points as a "pure" measure of size.

The problem is that organizations use size measures for more than just sizing the problem. They are also used for estimating effort, a task requiring technology dependence. A size measure that is truly unrelated to technology is no help for estimation [80]. Right from Albrecht's initial papers, validation of function points has commonly been done by relating size in FPs to size in LOC or to development effort, thus relating functional measurement to technical measurement.

Proponents of FPA (eg [28, 30, 58]) assert that function points *are* purely a measure of functionality, independent of technology, unrelated to effort. The argument amounts to a philosophical divide.

3.5 Subjectivity

IFPUG's work in developing successive releases of the Counting Practices Manual is aimed at standardizing function point counting and reducing subjectivity. Even so, counting function points is a subjective task, involving judgements by the person doing the count. This means that two analysts counting the same specification are unlikely to arrive at the same answer.

Early studies showed that within organizations, the variation between different function point counters was up to 30%; between organizations, it exceeded

30% [77]. Later studies showed the variation between counters to be about 12% [63], or "as little as plus or minus 11 percent" [28].

Training and experience is the key. Rule reports that the variation between recently trained project staff is typically around 23% [91]. For experienced specialists, following rigorous standards, the variation can be less than 5% [15, 91].

This does not necessarily mean that counters will agree with each other to within a few percent on all aspects of a function point count. In practice there can be many differences in identifying and counting individual components, but they often cancel each other out when they are aggregated [83].

3.6 What is missed out?

Function points are oriented towards data-strong systems, typified by business software. Processing in these systems is simple. Most effort goes into defining data structures.

Not all systems fit this pattern. Scientific and engineering software is often function-strong: dominated by the internal processing required to transform inputs to outputs. Real-time software is often control-strong: dominated by behavioural or control issues. Hybrid systems have elements of each of these types of software.

Traditional function points are less effective for software from these other domains.

Further, software development is very different now to when function points were first proposed. Then, bespoke MIS software was much more important. Now, software development is much more likely to involve package customization. The nature of software is often different: real-time, multi-layer, client-server software, rather than mainframe- and database-oriented software.

Although IFPUG has developed guidelines to help in several situations (eg [24, 73]), it can take some effort to relate the five component types that suit data-strong software to other types of software.

This is the prime motivation behind most of the alternatives to IFPUG function points.

3.7 The General Systems Characteristics and VAF

The adjustment phase of FPA is widely used by practitioners within the IFPUG world. This is partly because of a view that the general systems characteristics are genuinely important aspects of software projects, and partly for compatibility with the large number of past FP measurements that used it.

However, the GSC's and (especially) the VAF are probably the most criticized aspects of function points.

3.7.1 General Systems Characteristics

Symons observed that it is confusing to consider internal processing complexity as one of the GSC's, and also to give it a role in allocating function points to a component according to the the numbers of file and element types referred to in the component [99].

Symons also felt that the 14 specific characteristics were unlikely to be satisfactory for all time. He felt that more than 14 were needed (in Mark II function points he extended it to 19), and that the particular set of 14 might need to change over time. This last point is supported by empirical evidence that two characteristics (online entry, and data communications) are inevitably scored as 5 in today's online world. They should be redefined if they are to keep any discriminative value [71].

Empirical study has also shown that patterns can be observed in the GSC's for different types of software [29, 71]. For example, management information systems and decision support systems tend to be low in complexity, and place high importance on facilitating change by the user. Transaction/production systems and office information systems have more constraints on performance, and are less concerned with facilitating change by the user.

The definitions of some of the GSC's overlap. Relationships between them have been studied several times, invariably finding that several GSC's tend to vary together: they tend to be either low together or high together. This suggests that instead of 14 separate things being captured by the GSC's, a smaller number of underlying dimensions is involved, with some being captured several times over by different GSC's. The number of underlying dimensions might be 5 [71], 6 [66] or 7 [21], depending on the data set, but it is clear that there are not 14 independent factors.

As noted above, several of the GSC's (for example, performance requirements, reusability, heavily used configuration) really do not belong in a size measure. They are effort drivers, not size drivers [72]. Including them in the sizing process corrupts function points as a size measure.

3.7.2 Value Adjustment Factor

The VAF has come in for heavy criticism.

It involves inadmissible transformations according to measurement theory: arithmetic should not be performed on ordinal scale values [67].

Other criticisms are that it is not right to give all adjustment factors the same weight [21, 99]; a multiplicative model would make more sense than an additive model, and the range of possible technical complexity adjustments is too narrow [21].

Tellingly, the VAF turns out not to provide any significant value. It makes no difference to the accuracy of effort estimates. The relationship between adjusted function points and development effort is no stronger than that between unadjusted function points and development effort [3, 54, 66, 71]. This is probably because in most systems the VAF turns out to be very close to 1.0 [71]. The VAF is not even reliable as a simple indicator of whether effort is likely to be more or less than "average" for a given UFP.

Outside the IFPUG community, use of the VAF is declining. An adjustment phase is no longer a recommended part of Mark II function points, it has never been part of COSMIC-FFP, and it is not part of the recently-adopted standards for functional size measurement. Symons describes the VAF as "totally irrelevant to modern software development" [98].

For a while IFPUG described the VAF as an optional part of the process, in "Release 4.1 Unadjusted" of the Counting Practices Manual [40]. This meant that Release 4.1 Unadjusted was able to comply with the new ISO standard for

Component	Bock & Klepper	ISBSG
Input	3.18	4.3
Output	5.33	5.4
Inquiry	3.92	3.8
Internal file	8.41	7.4
External file	5.54	5.5

Table 6: Average function points per component

functional size measurement [44], and could itself be approved as a standard [48]. But the latest Release (4.2 [41]) of the IFPUG Counting Practices Manual no longer describes the VAF as optional, and Release 4.1 is no longer marketed on IFPUG's web site. It is clear that in IFPUG's eyes the VAF is still an integral part of Function Point Analysis.

3.8 Relationships between FP component types

The relative proportions of each of the five component types, and the three levels within each component type, turn out to be remarkably stable.

This is worrying to researchers, concerned about the validity of the internal structure of function points: strong correlations between the different component types suggests that some things are effectively counted more than once. On the other hand, there are advantages for practitioners, who can make use of this knowledge in several ways.

3.8.1 Function point breakdown

Bock and Klepper [15] observed that the proportion of low, average and high components of each type was stable in their environment. They used this knowledge to simplify the function point counting process. Instead of separately determining the functional complexity of each individual component, they simply gave each component the average number of function points (determined by multiple regression) for that component type in their environment. Those averages are shown in Table 6.

Similar results (also shown in Table 6) have been found by the International Software Benchmarking Standards Group (ISBSG) [42].

The most time-consuming aspect of function point analysis is counting all the element types and file types referenced in a function point component, in order to determine whether the component has low, average or high functional complexity. Assigning average numbers of function points has the potential to speed up the counting process, "with no significant reduction in accuracy" [15].

We can compare the averages in Table 6 with Albrecht's weights. For example, Albrecht's gives an output 4, 5, or 7 function points respectively for low, average and high functional complexity. As the average is about 5.3 to 5.4, outputs tend to be "average". Other transactional components (inputs and inquiries) also tend to have low to average functional complexity. Data components (internal logical files, and external interface files) tend to have low functional complexity. High functional complexity is rare. These observations

Component	New	Enhance
Input	37 %	36 %
Output	24 %	32 %
Inquiry	13 %	12 %
Internal file	22 %	15 %
External file	4 %	5 %

Table 7: Contribution of component types to total UFP

suggest there could be some benefit in recalibrating the tables used to determine functional complexity, particularly for files.

The relative contribution to total FP from each of the component types has also been analyzed by ISBSG [42]. For new developments there tends to be about 3 inputs, 1.5 outputs, and 1 inquiry per internal logical file. For enhancement projects there tends to be about 4 inputs, 3 outputs, and 1.4 inquiries per internal logical file.

The contribution of each component type to overall FP is shown in Table 7. New developments and enhancement projects are tabulated separately, because there are statistically significant differences in their breakdowns. Not surprisingly, enhancement projects involve less change to data and more creation of extra outputs.

Knowing the typical breakdown of FP components can be helpful early in a project, when a quick estimate of the total size in function points can be very useful. If one of the five component types can be counted early, and the relationship between that component and total function points is stable enough, an early estimate of total function points might be possible.

In many projects, a data model is available right from the start, or is the first part of the system to be documented. This means that the number of internal logical files can often be counted very early. Strong linear relationships have repeatedly been found between the number of internal files and total UFP, with approximately 30 UFP per file [70]. In the ISBSG data set, better predictions of total function points are given if the number of inputs is used as the predictor. This is less useful to an estimator, though, because the number of inputs is not usually known as early in a project.

Knowing the typical breakdown of FP components can also be useful for validation of a function point count. One form of validation can be to check that the FP breakdown is close to that expected [85]. While any given project may well vary from the average, large departures from expectation can indicate that a counting error has been made. Things to check include the relationship between internal logical files and UFP; the ratios of numbers of component types to each other; the percentage contribution of each component type to total UFP; and the average functional complexity of each function type. Checklists can indicate errors that might have been made if particular deviations from average are seen.

3.8.2 Correlations between component types

Three studies have been made into the correlations between the five component types. The first two agreed on some findings and disagreed on others [54, 65].

The third confirmed the agreements of the previous two, and suggested explanations for the disagreements [69].

It is clear that correlations between the five component types are inherent in FPA. Although the strengths of the correlations might vary from one data set to another, some common patterns are observed:

- Inputs, inquiries, and internal files are always correlated.
- External files are rarely correlated with the other components.
- Most correlations are weak, but some are strong enough (0.7 or greater) to indicate that some things are effectively counted more than once in FPA.

The strength and statistical significance of correlations between the components are greater for projects developed using 4GLs and application generators. They are weaker and vary more in projects developed using 3GLs. The correlations are also stronger in new developments than in maintenance projects [69].

Since 4GLs are most used in the domain where FPA is most widely accepted, this might imply that some simplification of the FPA process is possible in the areas where it works.

3.9 Backfiring

One aspect of function points that has been studied empirically from the very beginning — and by proponents of function points — is the relationship between function points and lines of code. This relationship is an interesting one. Function points and LOC are supposed to measure different things: respectively, the size of the problem independent of any solution technology, and the size of the solution.

But from the beginning, strong correlations have been observed between function points and lines of code for a given language [10]. Albrecht and Gaffney suggested this as the basis of a two-step process for estimating effort, using observed linear relationships to estimate lines of code from function points, and then effort from lines of code. A different formula was needed for each different language, but the correlation between lines of code and function points was very strong for each language.

Many thousands of projects have been sized now with both lines of code and function points. Formulas relating function points and program length are built into several commercial software estimation tools.

The concept of "backfiring" is simply the reverse of estimating length from function points: function points are estimated for an application by dividing its length by the average number of statements per function point for the language concerned [57]. The value of backfiring lies in providing a quick way to estimate the total size in function points of an organization's software portfolio, as a step in planning the maintenance of that portfolio [30]. Backfiring too is supported by commercial estimation tools.

In 1995 Capers Jones described backfiring as a "useful technique", that provides "a powerful way of sizing, or predicting, source-code volume for any known programming language or combination of languages" [57]. He noted that "the margin of error in converting LOC data into function points or back is high, but it's improving in both directions as more data becomes available."

In 1996, the second edition of Jones's book *Applied Software Measurement* [58] included a table identifying the number of logical source statements needed on average to encode one function point in each of 464 programming languages. He noted that code complexity could affect the relationship for any given program, but wrote about backfiring in matter-of-fact terms as a practical thing to do.

Backfiring presents an ironic problem. A basic tenet is that functionality and length are *different*; and that function points, being user-oriented, are a much better measure than lines of code, which are subject to variations between programmers and languages. From this viewpoint, Jones has written famously that using lines of code as a software size measure should be considered professional malpractice [59]. The argument is undermined if it turns out that function points and length are strongly related.

Jones is now much less positive about backfiring. His advice is to "strongly discourage its use in virtually every conceivable circumstance", and that "if it seems too good to be true, it probably is. Sooner or later, real counting will need to be done" [61]. Nevertheless, the "Programming Languages Table" is still maintained and is still available for purchase from Jones's company [61], albeit with warnings about its use.

Fundamentally, backfiring probably can be used effectively to estimate function points from program length or vice versa. The correlations between function points and length are undeniable. But two important qualifications must be noted.

First, as Jones points out, "local development practices and the way languages deliver functionality differently will always make backfiring a troublesome, dangerous exercise — especially if the backfiring is not meticulously calibrated to local conditions." In other words, knowledge is needed of relationships between function points and length in one's own environment; generic tables such as Jones's Programming Languages Table should not be relied on.

Second, backfiring should only be used to estimate portfolios of multiple projects. For a single project, the risk of an inaccurate estimate is high. Across a set of projects, there is some chance that individual errors will even out and averages will be reasonable.

3.10 Summary

The section has presented criticisms and research that suggest the need for other approaches to functional size measurement. Indeed, the rest of this chapter describes the evolution of several other approaches.

The criticisms and research findings described above have met with various responses.

Some have been disputed or ignored. Challenges to function points as purely a *size* measure are disputed on philosophical grounds. Problems of construction are overlooked, because function points have been found to be useful despite them.

Others have led to some evolution of how IFPUG function points are counted. Successive releases of IFPUG's Counting Practices Manual aim to improve standardization, reduce subjectivity, and indicate how things that were not initially considered should be counted within the core FP structure. IFPUG works hard to ensure the relevance of function points in a changing software world.

IFPUG's changes are all concerned with how to make counting decisions. The underlying structure of function points, and the process of calculating function points, has remained unchanged.

Several people have attempted to overcome perceived problems with IFPUG function points, by proposing extensions, or alternatives, to them. It is time to look at some of the most significant of those other proposals.

4 Mark II Function Points

The first significant alternative to Albrecht's function points was put forward in the late 1980's by Charles Symons, from the United Kingdom.

While praising Albrecht for breaking important ground, Symons saw several weaknesses in Albrecht's approach. Symons' criticisms were published in 1988 [99], along with an outline of an alternative formulation for function points: dubbed "Mark II Function Points". Full details of the new approach were set out in a book, published in 1991 [100].

Although Symons described Mark II function points as an evolutionary step, the aims and approach are quite different to Albrecht function points.

Mark II function points gained sufficient industry acceptance (though scarcely used outside the UK) to join IFPUG function points as a mainstream function point approach.

4.1 Symons' criticisms

Symons criticized several aspects of Albrecht function points (most of these have been discussed in Section 3):

- Classifying each component as having low, average or high functional complexity was too much of a simplification.
- The origin, validity and objectivity of the weights (function points per component type) were doubtful.
- The treatment of internal processing complexity was confusing.
- The general systems characteristics had several flaws.
- Function points are not summable in the way one would expect. If several separate collaborating systems are replaced by one integrated system, the integrated system has fewer function points than the component systems. In effect, the whole is less than the sum of the parts.

Symons' main point was that Albrecht's approach was developed in a particular environment; Symons questioned the suitability of the method, and particularly the weights, for general application.

In 1991 Symons added two more criticisms [100]. First, some types of software cannot be sized reliably using Albrecht function points. Examples are expert systems and system software: software that features complex internal processing, or complex algorithmic processing. Second, the count of files should be weighted by usage. A file that is used in many transactions should be counted multiple times, not just once.

The latter point reflects a crucial change in philosophy, that informed Symons' new formulation of Mark II function points. Albrecht sought to measure things of value to the user, and measured them once each. Symons retained the aims of technology independence and user focus, but moved away from the subjective concept of value to the user. Instead he sought explicitly to relate the system size measure to the effort involved in developing the system. This was more objective than trying to measure value to the user, and directly suits the aims of having a size measure suitable for measuring productivity and estimating effort. From this point of view, "The size of the databases, measured in function points ... is not directly relevant to the size of the applications ... What matters in measuring the size of systems for performance measurement and estimating purposes, is the usage of the files by the transactions within the applications" [100].

Symons took most of these observations into account when formulating Mark II function points. He did away with the low/average/high classification, and simply counted the numbers of data elements. He preferred the relational database concept of "entity" to the ambiguous concept of "logical file", and changed the basic components to be counted in a way that meant files were counted as many times as they were used. He determined the weights for different element types through a calibration process relating specifically to project effort, and proposed that the weights might change over time to retain the relationship with effort. He modified the adjustment process based on the general systems characteristics.

The one thing he did not try to do was measure the size of an "algorithm", regarding that as a problem for the future. Thus, Mark II function points have the same problems as IFPUG function points in terms of range of applicability. They are still geared towards file-oriented business systems.

4.2 Mark II Function Points

In Mark II function points, the system is regarded as consisting of a collection of "logical transactions". Each has input, processing, and output components. The size of a logical transaction is the sum of the sizes of the input, processing, and output components. The size of the system is the sum of the sizes of the logical transactions.

The sizes of the input and output components of a logical transaction are proportional to the numbers of data element types they involve. As the number of element types goes up, the size goes up; there is no simplification to just low, average or high. The idea is that the effort to format and validate an input, and to format an output, is proportional to the number of elements involved.

For the size of the processing component of a transaction, Symons looked to the ideas of McCabe's cyclomatic complexity [78] and Jackson's mapping of data structure to code logic [53]. Symons proposed that a measure of processing complexity is to count the number of data entity types (data entity types are the same as entities in relational data modelling) referenced (created, read, updated, deleted) by the transaction.

The size of a logical transaction, expressed in Unadjusted Function Points, is thus:

$$\text{UFP} = N_i W_i + N_e W_e + N_o W_o$$

where

N_i = number of input data element types,
W_i = weight of an input data element type,
N_e = number of entity types referenced,
W_e = weight of an entity type,
N_o = number of output data element types,
W_o = weight of an output data element type.

In 1988 Symons' initial calibration exercise yielded weights of $W_i = 0.44$, $W_e = 1.67$, $W_o = 0.38$. By 1991 they had become 0.58, 1.66, and 0.26 respectively. The weights have not changed since 1991.

Mark II function points include an adjustment phase based on some general systems characteristics. The final calculation of the adjustment factor is similar to IFPUG: scoring the characteristics on a scale of 0 to 5, adding the scores, scaling the result, and adding it to 0.65 to produce a Technical Complecity Adjustment factor. The unadjusted size is multiplied by the TCA to produce the system size in adjusted function points.

There are two differences between the IFPUG and Mark II calculations of the TCA. First, the impact of each factor is halved when calculating the TCA. Second, Mark II adds five extra system characteristics (interfaces to other applications, special security features, direct access for third parties, documentation requirements, special user training facilities) to the 14 of IFPUG, making 19 in total. The possibility of adding further client-defined characteristics was also envisaged.

Symons saw little incentive to put much work into redefining the adjustment process. He observed that the range of TCA values was small in practice, and seemed already to be smaller than in 1984. He expected the TCA to continue to decline in significance. Although it still appears in the Mark II FPA Counting Practices Manual [104] as an optional part of the Mark II FPA process, its use is no longer recommended.

4.3 Experience with Mark II Function Points

Kitchenham *et al.* consider that Mark II function points satisfy the properties of a valid measure, but only if they are regarded as an an effort measure, not a size measure [67].

A natural question is whether or not there is a relationship between IFPUG function points and Mark II function points. Symons reported initially that they did not correlate well, and the scatter was random, so it would not be possible to predict Mark II size from IFPUG size; this implies that they measure different aspects of size [100]. He found that the Mark II method gives a higher UFP score than the IFPUG method as system size increased.

Dolado [22] found the opposite. He found a strong correlation between IFPUG and Mark II function points. In his data set, the IFPUG method gave a higher UFP score than the Mark II method on all projects but one.

Symons later reported conversion formulas between IFPUG function points and Mark II function points, for new developments, for two ranges of project size [101]. He noted again that the Mark II method gives a higher UFP score than the IFPUG method as system size increased, as expected since files may be counted multiple times. Up to 1500 IFPUG UFP's, a quadratic formula relates the two sizes.

$$\text{MkII} = 0.9\times \text{IFPUG} + 0.0005 \times \text{IFPUG}^2$$

This formula was derived empirically, from a collection of systems counted with both methods. Above 1500 IFPUG UPF's, no empirical data was available. Some approximate MkII/IFPUG ratios were derived by extrapolation.

The UK Software Metrics Association maintains the Mark II FPA Counting Practices Manual. Mark II function points have achieved a strong share of the function point market in the UK, but are little used elsewhere.

5 Some other early variations

Mark II function points are by no means the only alternative proposed to IFPUG function points. Many variants exist in the literature (see [76] for a broader survey than is presented here). Apart from the main approaches identified in this chapter, none has achieved much currency.

Most have been proposed because someone thought something was missing from IFPUG FPA, and thought they knew how to fill the gap.

Three early alternatives or variations to function points are described now. The first is interesting because it appeared at about the same time as function points. The others extend function points in different ways, attempting to improve the measurement of different types of software.

5.1 DeMarco's Function Weight

A proposal with similar aims to function points, called "System Bang", was published in 1982 by DeMarco [19]. It too was proposed as an implementation-independent indication of the function to be delivered, as perceived by the user.

DeMarco based his measure on things that can be counted from a specification model: the function, data and state models of the Structured Analysis and Design Technique. He identified twelve different primitive things that should be counted as soon as the specification model was complete. The primitives included processes on data flow diagrams, objects and relationships on entity-relationship diagrams, and states and transitions on state-transition diagrams.

DeMarco did not suggest calculating Bang as a weighted sum of all twelve primitives. He suggested instead that projects should be divided into a small number of domains, and a different formulation of Bang should be developed for each domain.

His main classification of systems was function strong, data strong, and hybrid.

Function strong systems "can be thought of almost entirely in terms of the operations they perform upon data." It is the internal processing required to transform inputs into outputs that matters, and the data is fairly simple. For a function strong system, DeMarco suggested that the principal component of Bang is Functional Primitives (lowest level pieces on data flow diagrams). Each functional primitive was given a weight depending on the number of input and output "tokens" (data items that need not be subdivided) it worked with. The weighted sum of Functional Primitives was called "Function Bang" (since renamed "Function Weight").

A data strong system "is one with a significant database, and most of the effort for this system is allocable to tasks having to do with implementing the

database itself." For such a system, the principal component of Bang is the number of objects in the data model. Each object receives a weight depending on the number of relationships it participates in. "Data Bang" is the weighted sum of objects.

Hybrid systems fall between function strong and data strong systems. DeMarco suggested calculating both Function Bang and Data Bang for such systems, and using the two values separately as predictors when forecasting cost.

DeMarco also classified systems on a second dimension, based on the relative importance of data movement compared to computation. Commercial systems have more data movement, and scientific systems more computation. This classification did not affect how Bang was computed, but rather how the value was used. Bang was intended to be a predictor of project cost, and DeMarco warned that different projections should be used for projects from different domains.

It is interesting to draw some comparisons between function points and Bang. Data is essentially measured the same way: DeMarco's objects correspond to files in function points. Processing is counted differently: Bang counts processes on data flow diagrams, which is a low-level unit, while function points count flows across the system boundary. Function points are geared towards data strong systems, while Bang recognizes function-strong systems as needing to be counted differently. Neither approach considers systems where states and transitions are significant. Function points do not consider tham at all; DeMarco identifies states and transitions as things to count, but then gives them no further consideration.

DeMarco has cited anecdotal evidence of the usefulness of his method for estimating effort [20]. No formal validations have been published. Its only other appearance in the literature is in a simulation study from 1993 [87]. Bang never achieved the critical mass of users to become a mainstream functional sizing method.

5.2 3D Function Points

3D function points were developed by Whitmire at Boeing in the early 1990's [108, 110]. The motivation was his perception that traditional function points did not properly measure scientific and real-time software.

3D function points follow DeMarco in regarding all applications as having three dimensions: data, function, and control. "Each dimension contains characteristics that contribute to overall problem complexity or size, and these characteristics can be measured directly." [108].

Traditional function points were accepted as capturing the data dimension. They are incorporated directly into 3D function points to measure that dimension.

Like DeMarco, Whitmire viewed function strong systems as being dominated by the processing required to transform inputs into outputs. But where DeMarco counts processes, weighted according to the number of input and output tokens they work with, 3D function points measure processing differently.

A "transformation" is defined to be "the set of process steps and governing semantic statements [predicates that must remain invariant throughout the sequence of operations, or the pre- and post-conditions defined for each operation] required to transform one set of input data to output data" [108]. Depending on the number of processing steps and the number of semantic statements involved,

Processing steps	Semantic statements 1–4	5–15	16+
0–1	Low	Low	Average
2	Low	Average	High
3+	Average	High	High

Table 8: Functional complexity of Transformations in 3D FP

a transformation is classed as having low, average or high functional complexity (see Table 8), and then receives 7, 10 or 15 3D function points respectively.

The third dimension is control. This dimension is measured by counting the states and transitions on a Finite State Machine. Initial and terminal states are not counted, and the transition count is reduced by the number of states so that only multiple paths leading out of a state are counted. The resulting number of transitions is added to the 3D function points (*ie* each transition is counted separately, with a weight of 1 3D function point).

Whitmire recommended representing the 3D function points of an application as a triple (data, function, control) rather than just adding them together. The single combined value is also useful, but considering the three values separately helps spot characteristics of an application that are hidden by a single value [107].

DeMarco suggested counting the control dimension of an application, but didn't actually do it. 3D function points were the first proposal to include all three dimensions. They have never become a mainstream sizing method though. Symons [98] reported in 2001 that 3D function points were still used successfully within Boeing, but no further information has been published outside Boeing.

5.3 Feature Points

Feature Points are an extension of Albrecht's 1984 definition of function points. They were proposed by Capers Jones in 1986, in an attempt to measure software that is high in algorithmic complexity (for example systems software, real-time software) [56, 58].

A sixth component was added to the five standard function point component types. An "algorithm" was defined as "the set of rules which must be completely expressed in order to solve a significant computational problem" [58]. Examples are a square root extraction routine or a Julian date conversion routine.

Algorithms receive 3 points each in the feature points method. There is no classification of low, average and high functional complexity for any type of component (continuing with a change to the function point approach that Jones had already made, in an attempt to simplify the process, in his SPQR/20 estimation model). The weight given to internal logical files is reduced from 10 points to 7, to reflect the reduced significance of data files in systems software.

Jones reported that for classical MIS projects — the natural domain of IFPUG function points — feature points and function points often gave almost identical results. But for harder forms of systems software, feature point counts were significantly higher.

Feature points were the first attempt to capture the processing aspects of

an application in a tool and method that was marketed for widespread use. Although they were supported in the SPQR/20 estimation tool, they were still described as "experimental" in 1996 [58]. They never got beyond that stage. Although they are still documented on SPR's web site [56], they are no longer supported by SPR.

5.3.1 How to count algorithms?

The significant contribution of feature points was the idea of counting algorithms. But that was also its weakness. Whitmire felt that the definition of algorithms was not sufficient for counting purposes [108], and that they were deficient in considering processing steps but not semantic statements. Symons noted the "inherent difficulty of agreeing standard ways of defining and assigning a weight to algorithms of increasing size and complexity" [98], and considers that no functional sizing method has made any progress towards handling algorithmic complexity [97].

One recent proposal [88, 89] involves sizing an algorithm by regarding it as consisting of data to be operated on (an input), a local storage area to hold intermediate results (one or more internal logical files), and a result (an output). Their functional complexity is determined using normal IFPUG methods; a design aim of the approach is that it should fit directly within IFPUG methods. The function point value for the algorithm is the sum of the values for the input, storage, and output. The approach has been demonstrated on two examples. Whether this approach scales up has yet to be seen, and there remains the crucial question of deciding which algorithms to count.

5.4 Summary

Each of the three variants discussed in this section has interesting aspects. But none has achieved widespread success.

Bang and feature points were proposed by consultants, who used them in their own practices. Neither supports them any more. Presumably the methods were not successful enough or promoted enough to provide sufficient business value to their creators. 3D function points have only ever been used within Boeing, and have not been promoted in the literature for over 10 years. With no promotion outside Boeing, they have always been noted as an important but localized idea.

The problem that these three approaches tried to address was measuring real-time software and measuring algorithms. Measuring algorithms is still an unsolved problem. Measuring real-time software may have been solved: it is time to turn to COSMIC-FFP.

6 COSMIC

All of the function point approaches discussed above were the ideas of individuals. COSMIC is different.

COSMIC stands for Common Software Measurement International Consortium. Established in 1998, it is a consortium of academics and practitioners with an interest in functional sizing of software. Alain Abran and Charles Symons

were (and remain) joint project leaders. The original participants came from Canada, Europe and Australia. Most were participants in the ISO Working Group that was then engaged in developing a standard for functional size measurement.

COSMIC's aim was to develop and promote a new approach to functional size, useable for performance measurement and estimation, applicable to as wide a range of software domains as possible. Priority was given to business software, real-time software, and hybrids of the two, but not software of high algorithmic complexity.

The resulting measure built on work that was already under way on "Full Function Points".

6.1 Full Function Points

In 1997, Abran and his colleagues proposed an extension to IFPUG function points, with the aim of better measuring real-time software [96].

They felt that an extension to IFPUG function points was needed, because IFPUG function points had two problems when dealing with the control aspect of real-time software. First, real-time software frequently involved single-occurrence groups of data; these are difficult to map to internal logical files and external interface files. Second, real-time processes varied widely in their number of sub-processes; this could not be captured properly by measurement at the process level which can only give a narrow range of function points (eg 3 to 6 points for an input).

They proposed that the functional user requirements of real-time software included data-rich files and transactions, plus control data and control transactions. The data-rich aspects could be measured using IFPUG function points. The control aspects needed to be measured separately, for which they defined new control data and transaction types.

The sum of the two measurements represented the total system size, measured in Full Function Points ("FFP").

For control data they defined control groups. A control group is a group of control data used by the application, identified from a functional perspective; it exists for more than one transaction. They drew a distinction between read-only control groups, that are used but not changed by the application, and updated control groups that do get changed.

For control transactions, they defined four function types. Each represents a type of data movement. An external control entry represents a group of control data coming in from outside the application boundary. An external control exit represents a group of control data going outside the application boundary. An internal control read represents a group of control data being read from internal storage. An internal control write represents a group of control data being written to internal storage.

The principle behind measuring a process is that it has a separate sub-process for each data movement, and the functional size is directly proportional to the number of sub-processes. Control groups are not sized directly; only the data movements contribute to the functional size.

There is a very different measurement perspective involved here, compared to the IFPUG process. IFPUG sizes a transaction by counting the data elements involved, converting that to one of three functional complexity levels,

and giving a consequent number of between 3 and 7 function points. Files are also sized, once each, using a similar procedure that gives each file between 5 and 15 function points. FFP also measures size by counting data, but it is data movement rather than static data structure that is measured. Moreover, FFP sizes a control process by simply counting the data movements, which can be anything from 2 upwards with no upper limit.

The two measurement methods are different enough that it was necessary to justify the validity of adding the two results to give a single size figure [86].

Validation of FFP took the form of industrial field trials, to check the relevance and useability of the approach and to compare the results from IFPUG and FFP measurement. Early trials showed that for MIS software, IFPUG and FFP measurements were similar, while for real-time software FFP measurements were much higher.

By the time version 2.0 of the FFP measurement manual was issued in July 1999 [105], FFP had dropped the IFPUG part of the process. *All* data and transactions, not just control data and transactions, were measured in terms of entry, exit, read and write sub-processes. (Simpler terminology was possible as a result: for example, "entry" was now used, instead of "external control entry".)

Another fundamental difference from the IFPUG point of view had emerged by then: "layers" were being considered [84]. IFPUG views functionality as entirely based on business functions seen by external users. The developers of FFP considered that other aspects of software should be able to be included as well, perhaps involving different layers of functionality.

FFP and IFPUG function points were now entirely different measures.

6.2 COSMIC Full Function Points

By the end of 1998, COSMIC had been formed. The aim was to develop, and gain acceptance as an industry standard, a new approach to functional size measurement.

COSMIC sought to develop a measure that should: be useful to software project managers; be widely applicable, with priority given to business and real-time domains; conform to the emerging ISO standard for functional size measurement. They began by reviewing the main existing function point methods, and came up with a set of principles. They moved on to defining their own functional size measure.

The resulting measure drew heavily on ideas from Mark II and FFP. FFP version 2.0 was adapted to suit COSMIC's measurement principles, and rebadged as COSMIC Full Function Points ("COSMIC-FFP"). The first Measurement Manual was issued in October 1999. The most recent version is dated January 2003 [4].

The principles behind COSMIC-FFP are that:

- The functional user requirements of software are defined as a set of *functional processes.* A functional process is "an elementary component of a set of Functional User Requirements comprising a unique cohesive and independently executable set of data movements. It is triggered by one or more triggering events ... It is complete when it has executed all that

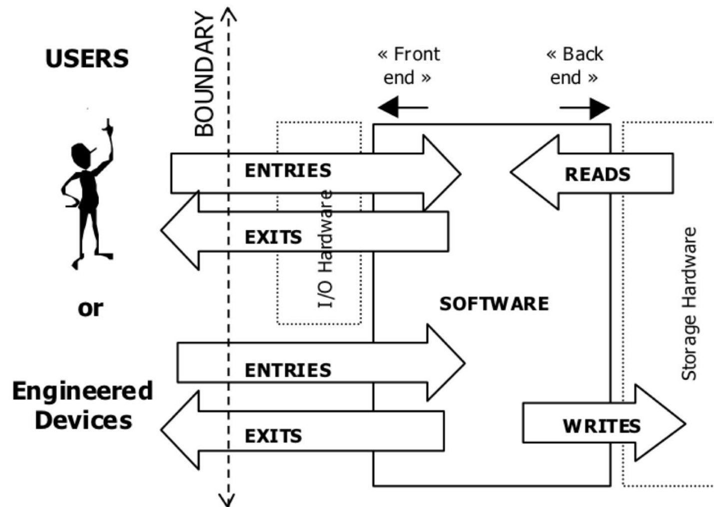

Figure 2: Data movements in COSMIC-FFP

is required to be done in response to the triggering event" [4]. Triggering events occur outside the software boundary.

- Software manipulates pieces of information, designated as data groups, which consist of data attributes. Figure 2 depicts the flow of data groups[1].
- Functional processes involve sub-processes, concerned with movement (entries, exits, reads and writes) and transformation of data groups.
- The functional size of a functional process is directly proportional to its number of data movements.
- The functional size of an application is the sum of the sizes of its functional processes.

At heart, sizing an application with COSMIC-FFP involves identifying the functional processes; identifying data groups; and identifying and counting the movements of data groups.

Note that data transformations are not considered. Some processing is assumed to be associated with each data movement (validating inputs, formatting outputs, *etc.*). The processing involved in data transformations is assumed to be covered by, or at least proportional to, this baseline processing. That won't be true for software involving complex algorithmic transformations; COSMIC does not claim that COSMIC-FFP applies to such software.

Each functional process involves at least two data movements. There must be at least one entry, and one exit or write. There is no upper limit on data movements; instances of functional processes with over 100 data movements have been observed.

[1] Figure 2 is a reproduction of Figure 2.4.1.1 from [4], page 19. Reproduced with permission.

The unit of measurement in COSMIC-FFP is not called a "function point". It is the "COSMIC functional size unit", or "Cfsu". It is defined to be equivalent to one single data movement type at the sub-process level.

COSMIC-FFP excludes a technical complexity adjustment. It does not attempt to take into account the effect on size of technical or quality requirements. Those are recognized as important in a project, but should not be included when measuring functional size.

COSMIC-FFP has two important innovations: layers, and measurement viewpoints.

Layers The concept of layers in COSMIC-FFP is similar to the concept of a layered software architecture. A layer is the result of functional partitioning, such that all included functional processes perform at the same level of abstraction. A layer provides functional services to its users; subordinate layers provide services to software in higher layers; software in a subordinate layer can perform without assistance from software in layers that use its services.

The point is that functional user requirements may be assigned to multiple layers in an application. This matters from a sizing point of view. If just the external behaviour is measured, the allocation of functionality to layers is not relevant. If layers are measured separately, overall functional size goes up, as inter-process communication is measured. For example, in a non-layered view, reads and writes are only measured once as internal operations. If they are delegated to a subordinate layer in a layered model, an extra level of entry/write and exit/read is measured. Writes from layer 1 become entries in layer 2, exits from layer 2 become reads in layer 1, and all are measured. Communication between peers in the same layer can also be measured.

Measurement viewpoints Regarding software as a black box, with only externally visible behaviour, is one viewpoint: suited to an end user, who cares only about what they may do with the software. This is the "end user measurement viewpoint". Regarding software as a collection of functions, spread across different layers, each providing separate functionality to the user, is a different viewpoint: suited to a developer, who cares about the totality of the software that they must write to meet the user's needs. This is the "developer measurement viewpoint". The contrast between these viewpoints looks a lot like the functionality/length difference — except that layers in COSMIC-FFP still represent functionality that must be provided, rather than lines of code that must be written.

The difference between viewpoints is important. Different viewpoints lead to different sizes being measured. The end user viewpoint corresponds to the IFPUG view of functionality. End users are not interested in a layered view, for example. Developers, and managers of development projects, are concerned more with the major components to be developed: these might reflect different layers, or peers within the same layer.

It is important to identify the viewpoint for measurement, if comparisons are to be attempted between developments.

6.3 Experience with COSMIC

COSMIC-FFP were tested in industrial field trials [7], to test that the documentation about COSMIC-FFP is understandable across different domains; that the method can be applied with reasonable effort; and that the sizes reflected the functionality as perceived by experts.

Results showed that experience with both COSMIC-FFP and the application domain was required to ensure repeatability. The COSMIC-FFP approach was seen as easy to apply. The effort involved was comparable to that for other functional sizing methods (though if only poor or incomplete documentation is available, IFPUG is probably slightly easier to apply than COSMIC).

The field trials also produced some data about the relationship between Cfsu and effort. The initial data set was small, so the results were not definitive, but strong relationships were observed.

The COSMIC Measurement Manual comments on conversions between COSMIC-FFP and other function point measures. There are few projects yet that have been sized with both COSMIC-FFP and another method (though one case study exists in which the same system was measured with five different approaches [26]), so most of the comments are based on theory and impressions rather than statistics. A small number of projects have been sized with both IFPUG and COSMIC FFPs. It seems that, on average, IFPUG and COSMIC sizes (if both are measured from the same end-user viewpoint) are roughly similar for new development projects of over about 100 Cfsu. For enhancement projects, and projects measured from different viewpoints, poorer correlations are expected. Direct data is not available relating COSMIC sizes to Mark II or early FFP sizes, but it is expected that on average they should all give roughly similar sizes.

In Section 3.8 we saw that in the IFPUG world, knowing the balance to expect between the five component types is useful for early estimation and for validating a count. Similar value can be expected in the COSMIC world. Study of 52 projects from the ISBSG Data Repository found the proportions of the four data movement types (entries, exits, reads, writes) to total functional size to be 33%, 35%, 19% and 13% respectively. Entries and exits contribute about equally, with a big drop to reads and a further drop to writes [43].

The use of COSMIC-FFP is increasing, particularly for real-time software, though its take-up is slower in the USA where IFPUG holds sway.

7 Function points for Object-Oriented software

From a user's point of view, object oriented software development is an implementation approach. In principle, it should be possible to apply function points as readily to OO software as to software developed using any other technology.

Specification documents include some form of model of the system, from which function points are counted. Traditional function points are counted from a data model and descriptions of the transactions. Those items may not be represented directly in an object oriented development, where the system model is probably expressed in UML. If function points are to be counted for an OO system, the issue is how to relate things in a UML model to the files and transactions required for a function point count.

Garmus and Herron [30] include a chapter on counting an object-oriented application. But there is almost nothing OO-specific in the chapter, beyond noting that a different approach is often needed because the documents are different. Their advice is to first become familiar with the entire system model, including functional description, object models and system diagrams. Once you understand what the system provides for the user, you can then count function points normally.

Several authors have proposed methods for adapting function points to object oriented software. Some retain a focus on traditional function points as the output from a count. They concentrate on relating OO concepts to FP components. Others develop new measures, tailored to OO software but analogous to FPs in some aspects of their construction.

In contrast to traditional methods, where IFPUG, Mark II and COSMIC dominate, the range of proposals for OO software is wide. None has yet achieved broad recognition. This indicates that measurement in the OO domain is not as well understood.

7.1 Mapping OO concepts to Function Points

If traditional function points are to be counted from a UML model, the issue is how to relate OO concepts to FP components.

Most proposals are based on class diagrams. A draft proposal [38] by IFPUG treats classes as files, and methods as transactions. Longstreet [75] treats classes that represent non-transient data as files, and describes sending and receiving messages as outputs and inputs respectively.

Longstreet [74] also sees a natural link between use cases and FPA. He gives some small examples but not much other detail.

There are four main proposals for identifying IFPUG function point components from UML models. Three give particular sets of rules and guidelines. The fourth draws on the others, seeking to determine empirically how best to make various counting decisions. All of these proposals aim to replicate (in most cases automatically) the result that would be produced by an expert human counter of IFPUG function points.

Fetcke [27] defined a set of rules for mapping a use case model and class model to concepts from the IFPUG Counting Practices Manual [36]. Some actors are identified as users or external systems. Use cases that interact directly with users or external systems, and other use cases that extend those with direct external interaction, are candidates for transactional components. Domain objects (specifically, entity objects, if that level of analysis is provided) correspond to files. Rules for handling aggregation and inheritance are given. Fetcke concentrated on demonstrating that his method was unambiguous enough to be applied in practice; how close the results were to a normal FP count was not studied.

Uemura *et al.* [103] considered class diagrams and sequence diagrams, obtaining most information from sequence diagrams. Classes that might represent data files are identified from the full set of sequence diagrams. Those whose data are changed by at least one transaction are internal files, the others are external files. Class attributes map to DETs, and RETs are essentially ignored (assumed always to be 1). Transactional components are identified from sequences of messages, initiated by external actors. Rules are given to decide the

type of transactional component (input, output, inquiry). Message arguments map to DETs, and the number of FTRs is 1 or 2 depending on which rule is applied. Provided the class and sequence diagrams meet certain conditions, these rules can all be automated; their paper reports early success with a tool they have developed.

Abrahão *et al.* [2] use yet another combination of UML diagrams. They consider that an OO conceptual model has four aspects: data (object model: class diagram), behaviour (dynamic model: state transition diagrams and interaction diagrams), process (functional model: describing the semantics of changes to an object's state), and presentation (presentation model: describing user interaction with the system). Classes and legacy views correspond to internal and external files. Services that change the state of a file correspond to inputs. Outputs and inquiries are recognized from the presentation model. Rules are given for identifying and counting each type of function point component, and the final result is intended to match an IFPUG function point count. Results from a case study are promising [1], and they too have built a tool to automate their process.

Cantone *et al.* [16] are building on the work of Fetcke, Uemura, and also Antoniol *et al.* [14]. They consider class diagrams, use case diagrams, and sequence diagrams. They do not specify a particular set of rules for what to count and how. They provide some guidelines and heuristics, and give their comments on the appropriateness of rules proposed by earlier researchers. They have also identified many situations where different decisions can be made on how to treat certain aspects of an OO model. They are conducting empirical work to see how different decisions affect the outcome. The aim is to develop a tool that can automatically analyze a UML model and generate a function point count that is as close as possible to that given by an expert in FPA.

Some work has also been done on mapping UML concepts to the data movement subprocesses in COSMIC-FFP [55]. The main resource is the sequence diagram. Discrete interactions between actors and the system correspond to functional processes; messages that cross the system boundary correspond to entries and exits; messages sent to objects correspond to reads or writes. The method can be automated if the sequence diagrams are fully specified and optional return arrows from read processes are not included in the diagram.

7.2 Function Point-like measures for OO

Another set of proposals is somewhat different. Measures are defined that are tailored to OO software, and are analogous to function points in some aspects of their construction. In most cases the same sorts of things are counted: objects for files, and methods or messages for transactions. But there is no expectation that the result would closely match an IFPUG FPA count.

Whitmire [109] considered each class as an internal file; messages sent across the system boundary were treated as transactions.

Schooneveldt [94] treated classes as files, and considered services delivered by objects to clients as transactions.

Sneed [95] proposed *object points* as a measure of size for OO software. Object points are derived from the class structures, the messages, and the processes or use cases, weighted by complexity adjustment factors.

Predictive Object Points (POPs) [82] are based on counts of classes and weighted methods per class, with adjustments for the average depth of the inheritance tree and the average number of children per class. Methods are weighted by considering their type (constructor, destructor, modifier, selector, iterator) and complexity (low, average, high), giving a number of POPs in a way analogous to traditional FPs.

Graham [31] proposed *task points* as a size measure that can be computed at the requirements analysis stage. A task point is an atomic task that the system will carry out in support of user requirements. Task points are equivalent to the leaf nodes in the Task Object Model — a hierarchical model of the business tasks to be supported by the system.

Two proposals are worth a slightly more detailed look, because they have more presence in the literature.

7.2.1 Object Oriented Function Points

Antoniol *et al.* [14] introduced *Object Oriented Function Points* ("OOFPs"). All measurement was based on the class diagram.

Though drawing heavily on IFPUG function points for inspiration, the philosophy behind OOFPs was not to find a way to count IFPUG function points. It was to develop a measure based on OO concepts that is useful for estimating project attributes such as LOC and effort.

OOFPs map classes to files, and methods to transactions. No attempt was made to distinguish between inputs, outputs and inquiries: all were simply regarded as "service requests". For classes, attributes and associations were mapped to DETs, and multi-valued attributes/associations to RETs. For methods, simple and compound arguments were mapped to DETs and RETs respectively.

Each class and method was given a number of points, based on the numbers of DETs and RETs, in the same way as traditional function points. As an initial formulation, OOFPs adopted the classification and weighting tables from the traditional FP method.

It can be seen that OOFPs are similar in structure to most of the other approaches, in mapping classes to files and services or messages to transactions. But OOFPs had the advantage that they could be counted automatically. Some design decisions were made to ensure that was the case. Tools were constructed to automate the process.

Pilot studies indicated that OOFPs were good predictors of size in LOC [13], though other primitive OO measures did as well.

An interesting outcome from the empirical research conducted with OOFPs was concerned with different ways to handle generalization and aggregation. With aggregation, is it best to count an entire aggregation structure as a single logical file, recursively joining lower level aggregations, or should the classes stay separate? With generalization, is it best to count each individual class with only its own attributes and associations, or should one consider as a different logical file the collection of classes comprised in the entire path from the root superclass to each leaf subclass? These decisions affect the number of classes counted, and the number of attributes within each class. The best approach, judging by how accurately OOFPs predicted LOC, was to perform full inheritance before counting files (so all inherited attributes are counted for a file, and only leaf

classes are counted), but not to aggregate classes (so just treat aggregation as equivalent to association),

Development of OOFPs has lapsed. Some of the ideas in OOFPs have fed into current work by Cantone *et al.* [16].

7.2.2 Use Case Points

Use case points have been proposed as an early predictor of software development effort. They are calculated from a use case model.

The components that are measured are different to the components of FPA (actors and use cases, rather than data and transactions). Another key difference from FPA is that the resulting number is seen as an indicator of effort, not size. But the stages involved in calculating use case points are modelled on the calculation of IFPUG function points.

The method was proposed by Karner in 1993 [62] (accessible descriptions are in [93] and [68]).

The method is based on identifying and classifying actors and use cases, and then adjusting for various technical and environmental factors that would influence development effort.

There are three stages to the process.

1. Each actor in the use case model is classified as simple, average, or complex. The classification is based on the nature of the interface between the actor and the application. The idea is that more complex interfaces involve more programming effort, so more use case points are given to them. Simple, average, and complex actors receive 1, 2 and 3 points respectively.

2. Each use case in the model is classified as simple, average, or complex. The classification is based on the number of "transactions" involved in the use case. A transaction is an atomic set of activities occurring between an actor and the system, occurring entirely or not at all. Again, the idea is that the more tasks that are involved in a use case the more programming effort there will be, so more use case points are awarded to use cases with more transactions. Simple, average, and complex use cases receive 5, 10 and 15 points respectively.

   The total number of points for all use cases and actors is summed, giving Unadjusted Use Case Points ("UUCP"). UUCP are meant to account for effort that is related to the inherent size of the task.

3. The use case points are adjusted to take into account 13 technical factors and 8 environmental factors that are expected to influence effort. Technical factors relate to aspects of the application itself. Environmental factors relate to the development team and environment.

   Each of these 21 factors is scored from 0 to 5: 0 means the factor is irrelevant for the project; 5 means the factor is essential. The scores for the technical factors are used to compute the Technical Complexity Factor ("TCF"), a number ranging from 0.6 to 1.35. The scores for the environmental factors are used to compute the Environment Factor ("EF"), a number ranging from 0.425 to 1.7.

   The UUCP is multiplied by both of the adjustment factors to give Adjusted Use Case Points ("UCP").

The analogies with IFPUG function points are obvious. Relevant components are classified to one of three levels and given use case points according to level; adjustment factors are calculated and applied in a manner similar to FPA; several of the technical adjustment factors are even the same as in FPA.

A difference is that the adjustment factors do not all have the same weights. The Environmental Factor is also new. It would be clearly inappropriate in a purported size measure, but it makes sense for an indicator of effort.

All the criticisms about inappropriate scale transformations and operations that apply to IFPUG FPA apply to use case points too. But the criticism that adjustment factors are effort drivers, not size drivers, no longer matters. Uncertainty about what the component weights represent is also removed: use case points are unequivocally intended to predict effort.

Some case studies (eg [12]) have found use case points to be good predictors of effort; about 15 to 30 staff-hours per use case point seems typical. How use case points compare with other estimation approaches has not been evaluated though, apart from a study in which they outperformed expert judgement [11].

Use case points are sensitive to an issue which is common in the world of use case modelling: writing the use cases and their constituent transactions at a standard level of granularity. Standards are needed on how to write use cases if comparisons across organizations are to be attempted. This is improving, with the publication of books like [17].

As with any other approach of this sort, manual counting of use case points is undesirable. It is time-consuming, and there is some subjectivity. Tools exist to identify actors and use cases, and to handle the calculation steps, but there remains the problem of classifying actors and use cases. Kusumoto *et al.* [68] have proposed heuristics for doing this automatically, and have developed a prototype tool.

7.3 Summary

Several proposals have been described above. The various methods may be compared on several criteria, including structure, how much information is considered, and when the information is available; objectivity and the potential for automation; and value to a user.

There is a trend towards considering ever more information from a UML model in order to come up with a size measure. This may give more useful results. It is also likely to improve the identification of FP component types, if that is the aim, as more information is available. The down-side is that some methods can only be applied later in the life cycle, perhaps well into the design stage.

In the end, the value of each method depends on how useful the measures turn out to be for project management. This is difficult to judge from the literature, because the proposals have been validated in different ways. For example, Fetcke concentrated on demonstrating that his method was unambiguous enough to be applied in practice. Schooneveldt showed in a case study that his method gave a result similar to a traditional FP count. Graham developed a tool to estimate effort from task points, but its accuracy is not reported in the literature. OOFPs were compared with LOC, not function points or effort. Use case points are validated by relating them to effort.

This is an area of active research. Among current activity, the work of Abrahão's group and Cantone's group look most promising, along with development of use case points. We can expect new proposals for mapping UML concepts to IFPUG and COSMIC components, and new FPA-like proposals for OO and other software. We can also expect more work on validation of different proposals, including comparative evaluations.

8 Function Point Standards

Different functional sizing methods take different interpretations on the concepts of software functionality. This leads to inconsistencies between methods, and also between different counters using the same method.

To resolve these inconsistencies, and define more rigorously what functional size measurement means, a standardization project was established within ISO/IEC. A working group (WG12) was put together in 1993, under ISO/IEC JTC1 Subcommittee 7 (whose area of work is software and systems engineering). ISO/IEC JTC1/SC7/WG12 had representatives from 12 countries, and also from the main function point users groups. Its task was to develop standards for functional size measurement.

The first task was to define standards for functional size measurement as a general concept. It was 10 years before a series of five standards were all approved, that collectively make up ISO/IEC 14143 *Functional Size Measurement*:

1. *Definition of concepts* (1998) [44]. This part identifies the common fundamental characteristics of functional size measurement methods, and defines a set of generic mandatory requirements for a method to be called a Functional Size Measurement Method ("FSMM").

2. *Conformity evaluation of software size measurement methods to ISO/IEC 14143-1:1998* (2002) [45]. This part is used by people needing to verify that a given software FSMM complies with the requirements of 14143-1.

3. *Verification of functional size measurement methods* (2003) [50]. This part is used by people needing to check the effectiveness of a particular FSMM as a measurement technique.

4. *Reference model* (2003) [51]. This provides a collection of reference user requirements, that can be used to test the effectiveness of a particular FSMM for different software types in different environments. It also provides the means to compare measurement results between FSMMs. methods.

5. *Determination of functional domains for use with functional size measurement* (2004) [52]. An important issue with FSMMs is their applicability to the functional domain of the software they are measuring. This part describes how to define functional domains, and provides guidance for classifying functional requirements over functional domains.

A sixth part *(Guide for the use of ISO/IEC 14143 series and related international standards)* provides guidance on how to select a suitable functional size measurement method. This part is presently at the balloting stage of the approval process.

Following the publication of the ISO/IEC 14143 series, four major function point approaches have been evaluated against that series and have gained recognition as ISO/IEC standards themselves:

- IFPUG's Counting Practices version 4.1[40] unadjusted (i.e. without the VAF) is published as ISO/IEC 20926 [48].
- Mark II function points[104] is published as ISO/IEC 20968 [46].
- The Netherlands Software Metrics Association's variant of IFPUG function points is published as ISO/IEC 24570 [49].
- The COSMIC-FFP method is published as ISO/IEC 19761 [47].

It is important to note that a Value Adjustment Factor is not part of functional size measurement as defined by ISO/IEC 14143-1. COSMIC-FFP never included an adjustment phase anyway. The others have had to drop that component in order to conform to ISO/IEC 14143-1. Many people may still use a VAF, but that part of their counting does not comply with ISO/IEC's standard definitions. For example, Release 4.2 of IFPUG's Counting Practices Manual reinstates the VAF that had been optional in Release 4.1 Unadjusted, and so does not comply with ISO/IEC 14143-1.

ISO/IEC has not attempted to decide between the relative merits of different sizing methods. It is up to the market to decide.

9 Conclusions

Many approaches to functional size measurement have been proposed over the 25 years since function points were first described. We have described the main proposals and a few other variants.

There are now three major methods in use: IFPUG function points, 25 years old; Mark II function points, 15 years old; and COSMIC-FFP, 5 years old.

IFPUG has history and the marketing behind it. A large base of experience permits benchmarking in many industry sectors. IFPUG holds the dominant share of the functional sizing market today, and will for some time.

IFPUG function points are not likely to develop further. Though IFPUG considers possible research areas (see for example [102]), it is a basic tenet that new ideas must require no change to the basic IFPUG structure.

Mark II is in a similar, though less dominant, position. Furthed development is unlikely, because Mark II is essentially considered to be replaced by COSMIC-FFP.

COSMIC-FFP is likely to increase its share of the functional sizing market, particularly in the real-time domain where other approaches have never been very successful.

With all of these approaches, we can expect to see continued refinement of guidelines, to reduce subjectivity and improve consistency between counters. There will also be some interest in mappings between other software models (such as UML) and the components counted in these approaches.

With IFPUG and Mark II, most work is likely to be based around exploiting the large experience base that has accumulated: for example, in benchmarking [42, 60], and development of estimation techniques [81]. The sizing methods

are mature (COSMIC would say dated), so emphasis will be on using the measurements rather than developing the measure and developing the experience base.

Work on COSMIC-FFP will also involve refinement of measurement guidelines (for example, the definition of "layers" was updated in August 2004). The main effort in the short term will be based on promoting and expanding its use, developing tools to support its use, and building an experience base.

Business software and real-time software are now well addressed by functional sizing methods. The problem of how to size scientific software, featuring complex algorithms, remains.

ACKNOWLEDGEMENTS

Thanks to Alain Abran and Pam Morris for their reviews of an earlier version of this chapter.

References

[1] S. Abrahão, G. Poels, and O. Pastor. Comparative evaluation of functional size measurement methods: An experimental analysis. Technical Report Working Paper 2004/234, Ghent University, March 2004.

[2] S. Abrahão, G. Poels, and O. Pastor. Functional size measurement method for object-oriented conceptual schemas: Design and evaluation issues. Technical Report Working Paper 2004/233, Ghent University, March 2004.

[3] A. Abran. *Analysis of the Measurement Process of Function Point Analysis.* PhD thesis, École Polytechnique de Montréal, March 1994.

[4] A. Abran, J.-M. Desharnais, S. Oligny, D. St-Pierre, and C. Symons. *COSMIC-FFP Measurement Manual, Version 2.2, The COSMIC Implementation Guide for ISO/IEC 19761:2003.* École de technologie supérieure, Université du Québec, Montréal, 2003. www.lrgl.uqam.ca/cosmic-ffp/manual.jsp.

[5] A. Abran and P. N. Robillard. Function points: a study of their measurement processes and scale transformations. *Journal of Systems and Software*, 25(2):171–184, May 1994.

[6] A. Abran and P. N. Robillard. Function points analysis: an empirical study of its measurement processes. *IEEE Transactions on Software Engineering*, 22(12):895–910, December 1996.

[7] A. Abran, C. Symons, and S. Oligny. An overview of COSMIC-FFP field trial results. In *Proc. ESCOM 2001*, London, April 2001.

[8] A. J. Albrecht. Measuring application development productivity. In *Proc. IBM Applications Development Symposium*, pages 83–92. IBM, October 1979. (Reprinted in *Programming Productivity: Issues for the Eighties*, 2nd edition, IEEE Computer Society, 1986).

[9] A. J. Albrecht. AD/M productivity measurement and estimate validation. Technical Report CIS & A Guideline 313, IBM, November 1984.

[10] A. J. Albrecht and J. Gaffney. Software function, source lines of code and development effort prediction: a Software Science validation. *IEEE Transactions on Software Engineering*, 9(6):639–648, June 1983.

[11] B. Anda. Comparing effort estimates based on use case points with expert estimates. In *Proc. EASE2002 (Empirical Assessment in Software Engineering)*, Keele, UK, April 2002.

[12] B. Anda, H. Dreiem, D. I. K. Sjøberg, and M. Jørgensen. Estimating software development effort based on use cases — experiences from industry. In M. Gogolla and C. Kobryn, editors, *UML 2001 — Proc. 4th International Conference on the UML*, pages 487–504. Springer-Verlag, October 2001. LNCS 2185.

[13] G. Antoniol, R. Fiutem, and C. Lokan. Object-oriented function points: An empirical validation. *Empirical Software Engineering*, 8(3):225–547, September 2003.

[14] G. Antoniol, C. Lokan, G. Caldiera, and R. Fiutem. A function-point like measure for object oriented software. *Empirical Software Engineering*, 4(3):263–287, September 1999.

[15] D. B. Bock and R. Klepper. FP-S: a simplified function point counting method. *Journal of Systems and Software*, 18(3):245–254, July 1992.

[16] G. Cantone, D. Pace, and G. Calavaro. Applying function point to Unified Modeling Language: Conversion model and pilot study. In *Proc. 10th International Symposium on Software Metrics*. IEEE, September 2004.

[17] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.

[18] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin-Cummings, 1986.

[19] T. DeMarco. *Controlling Software Projects*. Yourdon Press, 1982.

[20] T. DeMarco. An algorithm for sizing software products. *Performance Evaluation Review*, 12(2):13–22, 1984.

[21] J.-M. Desharnais and G. Hudon. Adjustment model for function point scope factors — a statistical study. In *Proc. Spring Conference, Florida*. IFPUG, 1990.

[22] J. J. Dolado. A study of the relationships among Albrecht and Mark II function points, lines of code 4GL and effort. *Journal of Systems and Software*, 37(2):161–173, May 1997.

[23] J. B. Dreger. *Function Point Analysis*. Prentice-Hall, 1989.

[24] European Function Point Users Group. *Function Point Counting Practices for Highly Constrained Systems*, 1993.

[25] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* Thomson, 2nd edition, 1997.

[26] T. Fetcke. The warehouse software portfolio — a case study in functional size measurement. Technical Report Research Report 1999–20, TU Berlin, 1999.

[27] T. Fetcke, A. Abran, and T.-H. Nguyen. Mapping the OO-Jacobson approach into function point analysis. In *Proc. TOOLS-23'97.* IEEE, August 1997.

[28] S. Furey. Why we should use function points. *IEEE Software*, 14(2):28–30, March/April 1997.

[29] D. Garmus and D. Herron. *Measuring the Software Process — a practical guide to functional measurements.* Prentice-Hall, 1996.

[30] D. Garmus and D. Herron. *Function Point Analysis: Measurement Practices for Successful Software Projects.* Addison-Wesley, 2001.

[31] I. Graham. Making progress in metrics. *Object Magazine*, 6(8):68–73, 1996.

[32] M. Hotle. Understanding and improving the AD estimating process. Technical report, Gartner Group, November 1996.

[33] W. S. Humphrey. *A Discipline for Software Engineering.* Addison-Wesley, 1995.

[34] IFPUG. *Function Point Counting Practices Manual, Release 3.0.* International Function Point Users Group, Westerville, Ohio, 1990.

[35] IFPUG. *Function Point Counting Practices: Case Study 2 (Analysis — ERD, DFD; Construction — DB2 data base, Graphical User Interface GUI Windows).* International Function Point Users Group, Westerville, Ohio, 1994.

[36] IFPUG. *Function Point Counting Practices Manual, Release 4.0.* International Function Point Users Group, Westerville, Ohio, 1994.

[37] IFPUG. *Function Point Counting Practices: Case Study 1 (Analysis — ERD, Process hierarchical model; Construction — IMS data base, Text Base Screen Implementation).* International Function Point Users Group, Westerville, Ohio, 1996.

[38] IFPUG. *Function Point Counting Practices: Case Study 3 (Object-Oriented Analysis, Object-Oriented Design).* International Function Point Users Group, Westerville, Ohio, 1996.

[39] IFPUG. *Function Point Counting Practices: Case Study 4 (TRACS — A Traffic Control System with Real-Time Components).* International Function Point Users Group, Westerville, Ohio, 1998.

[40] IFPUG. *Function Point Counting Practices Manual, Release 4.1.* International Function Point Users Group, Westerville, Ohio, 1999.

[41] IFPUG. *Function Point Counting Practices Manual, Release 4.2*. International Function Point Users Group, Princeton Junction, New Jersey, 2004.

[42] ISBSG. *Worldwide Software Development — the Benchmark. Release 5*. International Software Benchmarking Standards Group, 1998.

[43] ISBSG. Projects sized using COSMIC full function points. In *The Benchmark. Release 8*, chapter 8. International Software Benchmarking Standards Group, 2004.

[44] ISO/IEC. *14143-1:1998 Functional size measurement — Part 1: Definition of concepts*, 1998.

[45] ISO/IEC. *14143-2:2002 Functional size measurement — Part 2: Conformity evaluation of software size measurement methods to ISO/IEC 14143-1:1998*, 2002.

[46] ISO/IEC. *20968:2002 Mk II Function Point Analysis - Counting Practices Manual*, 2002.

[47] ISO/IEC. *19761:2003 COSMIC-FFP — A functional size measurement method*, 2003.

[48] ISO/IEC. *20926:2003 IFPUG 4.1 Unadjusted functional size measurement method — Counting practices manual*, 2003.

[49] ISO/IEC. *24570:2003 NESMA functional size measurement method version 2.1*, 2003.

[50] ISO/IEC. *TR 14143-3:2003 Functional size measurement — Part 3: Verification of functional size measurement methods*, 2003.

[51] ISO/IEC. *TR 14143-4:2003 Functional size measurement — Part 4: Reference model*, 2003.

[52] ISO/IEC. *TR 14143-5:2004 Functional size measurement — Part 5: Determination of functional domains for use with functional size measurement*, 2004.

[53] M. Jackson. *Principles of Program Design*. Academic Press, 1975.

[54] D. R. Jeffery and J. Stathis. Function point sizing: Structure, validity and applicability. *Empirical Software Engineering*, 1(1):11–30, March 1996.

[55] M. S. Jenner. Automation of counting of functional size using COSMIC-FFP in UML. In *Proc. 2002 International Workshop on Software Measurement*, Magdeburg, Germany, October 2002.

[56] C. Jones. What are feature points? Software Productivity Research, 1992.

[57] C. Jones. Backfiring: Converting lines of code to function points. *Computer*, 28(11):87–88, November 1995.

[58] C. Jones. *Applied Software Measurement*. McGraw-Hill, 2nd edition, 1996.

[59] C. Jones. Should the 'lines of code' metric be viewed as professional malpractice? *Voice*, 1(2):10–14, 1997.

[60] C. Jones. *Software Assessments, Benchmarks, and Best Practices.* Addison-Wesley, 2000.

[61] C. Jones. Programming languages table.
. www.spr.com/products/programming.htm, August 2003.

[62] G. Karner. Metrics for Objectory. Diploma thesis, University of Linköping, Sweden. No. LiTH-IDA-Ex-9344:21, December 1993.

[63] C. F. Kemerer. Reliability of Function Points measurement: a field experiment. *Communications of the CACM*, 36(2):85–97, February 1993.

[64] B. Kitchenham. The problem with function points. *IEEE Software*, 14(2):29–31, March/April 1997.

[65] B. Kitchenham and K. Känsälä. Inter-item correlations among function points. In *Proc. 15th International Conference on Software Engineering*, pages 477–480. IEEE, May 1993.

[66] B. A. Kitchenham. Empirical studies of assumptions that underlie software cost-estimation models. *Information and Software Technology*, 34(4):211–218, April 1992.

[67] B. A. Kitchenham, S. L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 12(12):929–944, December 1992.

[68] S. Kusumoto, F. Matukawa, K. Inoue, S. Hanabasa, and Y. Maegawa. Effort estimation tool based on use case points method. In *Proc. 10th International Symposium on Software Metrics.* IEEE, September 2004.

[69] C. J. Lokan. An empirical study of the correlations between function point elements. In *Proc. 6th International Symposium on Software Metrics*, pages 200–206. IEEE, November 1999.

[70] C. J. Lokan. Statistical analysis of ISBSG data and function point analysis. In *Proc. Australian Software Metrics Conference.* ASMA, November 1999.

[71] C. J. Lokan. An empirical analysis of function point adjustment factors. *Information and Software Technology*, 42(9):649–659, June 2000.

[72] C. J. Lokan and A. Abran. Multiple viewpoints in functional size measurement. In *Proc. 1999 International Workshop on Software Measurement*, pages 121–131, Lac Superieur, Quebec, Canada, September 1999.

[73] D. Longstreet. Function points applied to new and emerging technologies. www.softwaremetrics.com, 2000.

[74] D. Longstreet. Use cases and function points. www.softwaremetrics.com, 2000.

[75] D. Longstreet. OO and function points. www.softwaremetrics.com, 2001.

[76] M. Lother and R. Dumke. Points metrics — comparison and analysis. In *Current Trends in Software Measurement*, pages 228–267, Aachen, Germany, 2001. Shaker Publishing.

[77] G. C. Low and D. R. Jeffery. Function points in the estimation and evaluation of the software process. *IEEE Transactions on Software Engineering*, 16(1):64–71, January 1990.

[78] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[79] R. Meli. Functional metrics: Problems and possible solutions. In *Proc. FESMA'98*, pages 503–514, Antwerp, 1998.

[80] R. Meli. Functional and technical software measurement: Conflict or integration. In *Proc. FESMA 2000*, 2000.

[81] R. Meli and L. Santillo. Function point estimation methods: a comparative overview. In *Proc. FESMA'99*, pages 271–286, Amsterdam, 1999.

[82] A. Minkiewicz. Measuring object-oriented software with predictive object points. In *Proc. ASM'97 — Applications in Software Measurement*, Atlanta, October 1997.

[83] P. Morris. Function point audits. In *Proc. Australian Software Metrics Conference*. ASMA, September 2004.

[84] P. Morris and J.-M. Desharnais. Measuring ALL the software not just what the business uses. In *Proc. Fall Conference*, Orlando, September 1998. IFPUG.

[85] P. Morris and J.-M. Desharnais. Function point analysis: Validating the result. Technical Report Version 1.3, Total Metrics Pty Ltd, February 1999.

[86] S. Oligny and A. Abran. On the compatibility between full function points and IFPUG function points. In R. Kusters, A. Cowderoy, F. Heemstra, and E. van Veenendaal, editors, *Project Control for Software Quality (Proc. ESCOM'99)*. Shaker Publishing, 1999.

[87] R. Rask, P. Laamanen, and K. Lyytinen. Simulation and comparison of Albrecht's function points and DeMarco's function bang metrics in a CASE environment. *IEEE Transactions on Software Engineering*, 19(7):661–671, July 1993.

[88] N. Redgate and C.B. Tichenor. Measure size, complexity of algorithms using function points. *Crosstalk*, pages 12–15, February 2001. www.stsc.hill.af.mil/crosstalk/.

[89] N. Redgate and C.B. Tichenor. Measuring calculus integration formulas using function point analysis. *Crosstalk*, pages 24–27, June 2002. www.stsc.hill.af.mil/crosstalk/.

[90] E. E. Rudolph, G. E. Wittig, G. R. Finnie, and P. M. Morris. Verifying function point values. In *Proc. FESMA'98*, Antwerp, 1998.

[91] P. G. Rule. The importance of the size of software requirements. In *Proc. NASSCOM Conference*, Mumbai, India, February 2001.

[92] T. L. Saaty. *The Analytic Hierarchy Process*. McGraw-Hill, New York, 1980.

[93] G. Schneider and J. P. Winters. *Applying Use Cases*. Addison Wesley, 2nd edition, 2001.

[94] M. Schooneveldt, T. Hastings, J. Mocek, and R. Fountain. Measuring the size of object-oriented systems. In *Proc. 2nd Australian Conference on Software Metrics*, pages 83–93. Australian Software Metrics Association, November 1995.

[95] H. Sneed. Estimating the Development Costs of Object-Oriented Software. In *Proceedings of 7th European Software Control and Metrics Conference*, Wilmslow, UK, May 1996.

[96] D. St-Pierre, M. Maya, A. Abran, and J.-M. Desharnais. Adapting function points to real-time software. In *Proc. Fall Conference*, Scottsdale, 1997. IFPUG.

[97] C. Symons. Personal communication. December 2003.

[98] C. Symons. Come back function point analysis (modernised) — all is forgiven! In *Proc. 4th European Conference on Software Measurement and ICT Control*, pages 413–426, Heidelberg, Germany, May 2001. FESMA-DASMA.

[99] C. R. Symons. Function Point Analysis: Difficulties and improvements. *IEEE Transactions on Software Engineering*, 14(1):2–11, January 1988.

[100] C. R. Symons. *Software Sizing and Estimating: Mk II FPA*. Wiley, 1991.

[101] C. R. Symons. Conversion between IFPUG 4.0 and MkII function points, version 3.0. Software Measurement Services. www.gifpa.co.uk, 1999.

[102] C. B. Tichenor. Recommendations for further function point research. www.softwaremetrics.com, 2000.

[103] T. Uemura, S. Kusumoto, and K. Inoue. Function-point analysis using design specifications based on the Unified Modelling Language. *Software Maintenance and Evolution: Research and Practice*, 13(4):223–243, July/August 2001.

[104] UK Software Metrics Association. *MkII FPA Counting Practices Manual Version 1.3.1*, October 1998. www.gifpa.co.uk.

[105] UQAM Software Engineering Management Research Laboratory. *Full Function Points Measurement Manual Version 2.0*, 1999.

[106] J. M. Verner, G. Tate, B. Jackson, and R. G. Hayward. Technology dependence in Function Point Analysis: a case study and critical review. In *Proc. 11th International Conference on Software Engineering*, pages 375–382. IEEE, 1989.

[107] S. A. Whitmire. Posting to function point mailing list. 21 September 1995.

[108] S. A. Whitmire. 3D function points: Scientific and real-time extensions to function points. In *Proc. 10th Pacific Northwest Software Quality Conference*, Portland, Oregon, 1992. Pacific Agenda.

[109] S. A. Whitmire. Applying function points to object oriented software. In J. Keyes, editor, *Software Engineering Productivity Handbook*, chapter 13, pages 229–244. McGraw-Hill, 1993.

[110] S. A. Whitmire. An introduction to 3D function points. *Software Development*, pages 43–53, April 1995.

[111] G. E. Wittig, G. R. Finnie, E. E. Rudolph, and P. M. Morris. Project research design to validate FPA coefficients using AHP. In *Proc. 3rd Australian Software Metrics Conference*. ASMA, November 1996.