# Source Code Based Function Point Analysis for Enhancement Projects

Steven Klusener
Software Improvement Group, Muiderstraatweg 58a, NL-1111 PT Diemen
& Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
Email: `steven@software-improvers.com`

## Abstract

*Function point analysis is a well known established method to estimate the size of software systems and software projects. However, because it is based on functional documentation it is hardly used for sizing legacy systems, in particular enhancement projects. In this short note we sketch briefly how a Function Point Analysis can be based on the source code.*

**Keywords**: Software maintenance, function point analysis.

## 1 Introduction

Function point analysis is a well known established method to estimate the size of software systems and software projects. Originally, the method was used in the early phases of the waterfall model such that the implementation effort could be estimated on the basis of the I/O-behavior as defined in the functional documentation. Later, one was able to obtain a first order estimation of the size of existing software systems, based on benchmarks and the number of lines of code of a system. This method is known as backfiring, see [8].

For existing software systems, in particular *legacy* systems, the functional documentation is often missing or obsolete. Hence, the standard Function Point Analysis (FPA) is not applicable. For the sizing of *enhancement projects* (the implementation of change requests) backfiring is not applicable either: it only refers to the complete software system and its precision is not sufficient to size individual enhancement projects.

In this note we present a method to perform FPA based on the source code. This method is instantiated for Cobol and JCL (Job Control Language). It can be integrated into the maintenance process, such that each change request is defined in term of the objects from the Function Point (FP) conceptual model. In this way the sizing of a change request is obtained for free. Moreover, the actual implementation process is improved because with our approach FPA bridges the gap between the functional world and the world of the source code. If a change request is indeed formulated in terms of the FP-objects, the maintenance programmer knows right away were to look in the source code. Hence, the traditionally expensive impact analysis can be shortened drastically. This integration of FPA into the maintenance process is not further discussed in this short note, neither is the software infrastructure that is needed to support this integration.

Because of the size limit of this short track we refer to the literature for background information about Function Point Analysis. FPA dates back to [1], for text books we refer to [3] and [5] and for the rules and guideline of the Internal Function Point Users Group (IFPUG) we refer to [7],

**Acknowledgments** In this note several results from an earlier study by Mirela Mustacevic are used, see [9]. Mustacevic developed an automated measurement for one small application, which came very close to the original, hand-counted, measurement of that application. However, it is not clear how this automated measurement acts in other cases.

Tobias Kuipers and Joost Visser (Software Improvement Group), Chris Verhoef (Vrije Universiteit), and Alain le Loux, Theo Thijssen and Henny Snijder (PinkRoccade Public) are thanked for support and discussing the contents of this note.

## 2 Constructing a Function Point model from the source code

In table 1 we give a brief overview of the the notions and concepts from the FP-model and their counterparts in the actual source code.

The rules and guidelines from [7], and also the textbook [5], divide FPA into 5 steps. Below we discuss these steps for source code based FPA:

1. **Determine the Type of FP Count**. In our approach we focus on enhancement projects.

| FP-objects | Technical objects |
|---|---|
| Application | (Sub-)system, collection of Cobol programs, copy books, JCL-jobs, etc. |
| Logical File | JCL data set, Cobol data file, SQL-table |
| Transactional function | Cobol I/O-statements, SQL-statements |
| Record Element Type (RET) | (Sub-)record declarations in Cobol, SQL-tables |
| RET Reference | Usage of fields and SQL-attributes in Cobol I/O statements and resp. SQL-statements. |
| Data Element Type (DET) | Cobol field declaration, SQL-attribute declaration. |
| DET Reference | Usage of fields and SQL-attributes in Cobol I/O statements and resp. SQL-statements. |

**Table 1. The mapping between FP-concepts and technical items**

2. **Identify counting boundary** The counting boundary, or application boundary, determines which `CALL`-statements are counted as external inquiries; calls to programs within the same application are not counted (see eq.call.1 in table 6 ).

   The average size of a system is too large to consider it as one application, otherwise hardly any `CALL`-statement will be counted as external inquiry. It doesn't make sense either to consider every module as one application, because then every `CALL`-statement will be counted. We propose to take all programs that are associated to a JCL-batch job into one application. For an analysis of how we can obtain the Cobol programs from the JCL-jobs we refer to table 2.

3. **Determine logical files** In this step we consider all data sets, Cobol files and DB2 tables that we encounter in the JCL jobs and the Cobol programs. For each data set, Cobol file and/or DB2 table which is permanent we create an associated logical file. All other files, like temporary files, sort files, etc. are skipped. This analysis is sketched in table 3.

   The record element types (sub-records) and data element types (elementary fields) can be obtained from the Cobol record declarations (as is also shown in table 3) or from the SQL-table definitions as is shown in table 4.

4. **Determine transactional functions and their element types** By analyzing the I/O-statements, SQL-statements and CALL-statements (as described in table 6), we obtain all external inputs, external outputs and external inquiries.

   Consider the following `SELECT`-statement in table 5 which is recognized as external input. It reads the table `RGDAT` (see table 4), only the subfields `DATE_REGISTER` and `REGISTERDAY` are data element types of the external input. The Cobol field `H-SQL-DAT-1-R` is also a data element type of this external input.

5. **Determine the adjusted Function Point value** As most of the 14 system characteristics are not directly deducible from the source code (like *Performance requirements* and *transaction rate*), we propose to omit this adjustment for source code based FPA.

```
EXEC SQL DECLARE RGDAT TABLE
( DATE_REGISTER  DATE NOT NULL,
  REGISTERDAY    CHAR(1) NOT NULL,
  CURRENCY       CHAR(1) NOT NULL,
) END-EXEC.
...
01  RGDAT.
  10 DATE_REGISTER  PIC X(10).
  10 REGISTERDAY    PIC X(1).
  10 CURRENCY       PIC X(1).
```

**Table 4. A SQL-table declaration and an associated Cobol record declaration**

```
EXEC SQL
 SELECT  MIN(DATE_REGISTER)
    INTO :RGDAT.DATE_REGISTER
    FROM  VNW200GTBOEKDAT
    WHERE  DATE_REGISTER >= :H-SQL-DAT-1-R
            AND  REGISTERDAY = ’1’
    END-EXEC.
```

**Table 5. The SQL-table of table 4 used in an external input**

## 3 Improving the counting of transactional functions using data flow analysis

In order to follow the original FPA, as stated in [7], as close as possible, the analysis of the previous section is not sufficient. Note, that we are not only interested in the final FPA-count values, but in a proper recognition of all FP-objects. So, false positives do not compensate false negatives. We can minimize both types of errors by applying an additional data flow analysis (for a possible implementation, see [2] and [4]). This holds particularly for the recognition of the transactional functions:
- Finding fields that are related to I/O statements, such that

**A simple case of calling a Cobol program from a JCL batch job**

```
//STEP1 EXEC PGM=<prog-id>
```

This JCL statement executes `<prog-id>`, which may be a Cobol program. In case you want to report missing programs, you have to be sure to skip utilities like `IEBGENER` (a copy-utility), `IEFBR14` (a dummy utility), and others.

**A more involved case which requires a data flow analysis of the JCL batch job**

```
//JOBC   JOB ,JOHN,MSGCLASS=H
//STEP2  EXEC PGM=UPDT
//DDA    DD   DSNAME=SYS1.LINKLIB(P40).DISP=OLD
//STEP3  EXEC PGM=*.STEP2.DDA
```

The `EXEC` statement named `STEP3` contains a backward reference to `DD` statement `DDA`, which defines system library `SYS1.LINKLIB`. Program `P40` is a member of `SYS1.LINKLIB`; `STEP3` executes program `P40`. (This example is taken from [6], page 16-23.)

**Table 2. Extracting Cobol calls from JCL jobs**

```
//TRXIK453 EXEC PGM=TRXI650
...
//INPUT1   DD DSN=A.SFBI.SELSORT2(+1),DISP=OLD
...
//OUTPUT1  DD DSN=A.SFBI.SELREST(+2),DISP=(,DELETE)
```

`INPUT1` is the input file of Cobol program `TRXI650` given below, it corresponds with the data set `A.SFBI.SELSORT2(+1)` (i.e. the actual file at the file system). It is a persistent file (`DISP=OLD`), so we create a logical file for it. In the Cobol source code we see, via the `FILE-CONTROL` and the `FILE SECTION`, that it corresponds with the Cobol record `RECORD-PERS-IN`. This record has one *record entity type* `HISTORY` (see the `OCCURS`-clause) and two *data element types* `PERS-ID` and `ACCNT`. The output file, in the JCL-job denoted by `OUTPUT1`, is temporary, because of `DISP=(,DELETE)`, so we do not create a logical file for it.

```
FILE-CONTROL.
  SELECT INFL            ASSIGN TO INPUT1.
  SELECT OUTFL           ASSIGN TO OUTPUT1.
...
FILE SECTION.
  FD  INFL BLOCK CONTAINS 0 RECORDS RECORDING MODE F.
  01  RECORD-PERS-IN.
    03 PERS-ID           PIC X(10).
    03 ACCNT             PIC X(10).
    03 HISTORY OCCURS 64.
       05 DATE           PIC X(8).
       05 AMOUNT         PIC S9(8)V9(2).

  FD  OUTFL BLOCK CONTAINS 0 RECORDS RECORDING MODE F.
  01  RECORD-CHECK-OUT.
    03 PERS-ID           PIC X(10).
    ...
```

**Table 3. Extracting data sets from JCL jobs and file structures from the Cobol declarations**

more data element types are counted for transactional functions.
- Search statements on local variables may be counted as external inputs, if these local variables are related to external files via data flow.
- By analyzing the actual usage of data element types, the amount of subfields that are counted as record element types and data element for transactional functions can be limited.

## 4 Future work

This note sketches an approach, which still has to be validated in future work:
- The method must be validated by applying it to a number of systems with a known FPA model (first without, and then with additional data flow analysis). The results have to be discussed with the system engineers.
- After the validation a software infrastructure can be developed. If needed, more detailed data flow analyzes have to be developed.
- Then a project has to be started in cooperation with the group of software maintainers; for a certain period it has to be checked if the change requests can indeed be formulated in terms of the FP-objects. If so, then the time reduction of the impact analysis has be estimated. If not, then it has to be determined whether certain FP-objects are missing, or whether (and why) certain change requests are not related to any FP-objects at all.
- The source code analysis has to be extended to other mainframe/mid-frame technologies such as PL-1, IMS (hierarchical database and transaction management), CICS (transaction management and screens) and IDMS (non-relational database).

## References

[1] A.J. Albrecht. Measuring application development productivity. In *Proceedings of the Joint*

| | **External Inputs** |
|---|---|
| eq.file.1 | `SORT .. USING <file-name>+`, where at least one `<file-name>` corresponds with an ELF |
| eq.file.2 | `MERGE .. USING <file-name>+`, idem |
| eq.file.3 | `DELETE <file-name>`, idem |
| eq.file.4 | `SEARCH <record-name>`, where `<record-name>` is declared with the `OCCURS` clause, and is declared in the `file section`. The associated file (see `FD clause`) must correspond with an ELF. |
| eq.file.5 | `WRITE <record-name> [<from> identifier]`, where `<record-name>` corresponds with an ILF and `identifier` corresponds with an ELF. |
| ei.sql.1 | `SELECT ... FROM <table-name>+`. |
| ei.sql.2 | `INSERT INTO <table-name>`. |
| ei.sql.3. | `UPDATE <table-name> SET <set-clause> FROM <from-clause>`. |
| ei.sysin.1 | `ACCEPT`, do not count cases `ACCEPT .. FROM <time>`, where `<time>` is `DATE, DAY, DAY-OF-WEEK` or `TIME`. |
| | **External Outputs** |
| eo.file.1 | `WRITE <record-name> [<from> identifier]`, where `<record-name>` corresponds with an ELF (directly or indirectly, see above). |
| eo.file.2 | `REWRITE`, idem |
| eo.file.3 | `SORT ... GIVING <file-name>`, where `<file-name>` corresponds with an ELF. Note that a `SORT` statement may have a `USING`-clause and a `GIVING`-clause. |
| eo.file.4 | `MERGE ... GIVING`, idem |
| eo.sql.1 | `UPDATE <table-name> SET <set-clause> FROM <from-clause>`, where `<table-name>` is an ELF. |
| | External outputs based on `SYSOUT` operations |
| eo.sysout.1 | `DISPLAY`, note that a sequence of `DISPLAY`-statements is counted as one external output. |
| | **External Inquiries** |
| eq.call.1 | `CALL <program-name>`, where `<program-name>` is part of another application. |
| eq.sysinout.1 | A sequence of `DISPLAY` statements, followed by an `ACCEPT` statement. (These statements must not be counted as well as external inputs and external output!) |
| | **Remarks** |
| remark.1 | Statements like `OPEN INPUT` and `CREATE TABLE` are *not* counted. |
| remark.2 | The case eq.call.1 is counted as external inquiry, because in general there is not sufficient information to make a more precise classification (i.e. in reality it can correspond with an external input or external output.) |
| remark.3 | If at least one ei.sysin.1 or eq.sysinout.1 is counted then we have to count `SYSIN` as ELF. Idem for SYSOUT with resp. eo.sysout.1 en eq.sysinout.1. |
| remark.4 | Currently we do count error messages, they can be skipped by further source code analysis |

**Table 6. The mapping of I/O-statements to transactional functions**

*SHARE/GUIDE/IBM Application Development Symposium*, pages 83–92, 1979.

[2] A. van Deursen and L. Moonen. An emperical study into Cobol type inferencing. *Science of Computer Programming*, 40:189–211, July 2001.

[3] J.B. Dreger. *Function Point Analysis*. Prentice Hall Advanced Reference Series, Computer Science, 1989.

[4] P.H. Eidorff, F. Henglein, C.Mossin, H. Niss, M.H. Sorenson, and M.Tofte. Anno domini: from type theory to Year 2000 conversion tool. POPL. ACM Press, 1999.

[5] D. Garmus and D. Herron. *Function Point Analysis, Measurement Practices for Successful Software Projects*. Adison-Wesley Information Technology Series, 2001.

[6] IBM. IBM MVS JCL reference manual, 2000.

[7] The International Function Point Group (IFPUG). Function point practices manual, release 4.1.1, 2000.

[8] T. Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, second edition, 1996.

[9] M. Mustacevic. *Automatisering van het tellen van functiepunten (Automated Function Point Counting)*, in dutch, 2000. Master's Thesis, University of Amsterdam, see `www.cwi.nl/~paulk`, *Supervised Master's Theses*.