Sizing Maintenance Tasks for Web Applications

Harry M. Sneed
Anecon GmbH
Vienna, Austria
Harry.Sneed@t-online.de

Shihong Huang
Computer Science & Engineering
Florida Atlantic University
shihong@cse.fau.edu

Abstract

Web applications are fast becoming the new legacy systems of today. While there is considerable similarity between traditional software systems and Web-based systems, there are also significant differences between them. One area that illustrates this dual nature is cost and effort estimation. There exists a mature body of knowledge for performing such estimates on traditional software systems, but such methods may not appear to be directly applicable to Web applications. This paper presents an effort estimation technique for maintaining a large-scale Web application by measuring and tracking the size and complexity of the Web-based system. Specifically, a combination of function-points and static impact analysis is used, tracing the change requests to different components of the Web application, and then measuring their size and complexity to aid the cost estimation for that particular change request based on function point productivity measurements. To illustrate the use of this technique, a case study from a real-world industrial product is presented.

Keywords: software maintenance, metrics, Web size measurement, cost estimation, impact analysis, Web

1. Introduction

Financial institutions in Central Europe are under strong pressure to reduce their costs while at the same time increasing their services and fulfilling more legal requirements such as the Basel-II convention. On top of that, many banks have been merged together and have acquired subsidiary banks in Eastern Europe. Globalization now gives them the opportunity to have their development made either on site or to have it made by their subsidiary companies in Eastern Europe.

Since the cost of labor differs greatly from more than €800 per person day in Austria to less than €100 per person day in Romania, the IT managers are now faced with the question of where they should have their IT systems maintained. They can choose between many sites. To help them in making this decision, they need to know the productivity of the different development groups and the size of the tasks to be performed. This way, they can decide whether it is more cost effective to develop and maintain their applications at the central office in Vienna or to move development and maintenance off to a subsidiary company in Prague, Warsaw, Budapest or Bucharest. The goal is to have applications developed, and maintained there where it is most cost effective to do. However, the question of cost effectiveness is not an easy one to answer, since there are many related issues such as productivity, quality, and the nature of the applications involved.

1.1 Maintenance vs. Development

In sizing software tasks, one must distinguish between development and maintenance. It is one thing to develop a new system from the requirements specified by the bank analysts and another to maintain an existing system based on the user error reports and change requests. If the requirements are not specified with sufficient detail, then users, analysts, developers, and testers have to be in close and constant contact with one another, as advocated by the proponents of agile development, meaning they have to be in the same proximity. This can inhibit outsourcing [20].

In the case of maintenance projects, the application exists in the form of software artifacts (e.g., code, documentation, and test cases). The maintainers have to be able to understand the error reports and the change requests and to link them to the software artifacts. The communication between users, analysts, and maintainers is much more formalized. Therefore, maintenance projects can be more easily outsourced to remote locations. In this case, the maintenance personnel would only have to understand German language comments, documents, error reports, and change requests.

1.2 Maintenance Cost Factors

In outsourcing software maintenance, the question arises as to how much the maintenance work will cost. This raises two important issues: (1) what is the scope of the maintenance tasks, and (2) what is the productivity of the maintenance personnel?

To answer the first question, one must know the size of the application in terms of some unit of measure of which there many candidates, among them lines of code, statements, function-points, object points, use-case points, etc. The literature on software measurement is abundant with alternative measures of software size [12]. Since the complexity and quality of software have a definite effect on the maintenance effort, these two factors should also be considered in adjusting the size of the application system.

To answer the second question, one must know how productive the maintenance team is in terms of effort per size unit for each type of maintenance task. In software maintenance, there are at least two different maintenance task types: (1) corrective maintenance (error corrections), and (2) adaptive and perfective maintenance (changes) [10].

For both types, it is necessary to associate the hours worked with the size of the domain affected by the task. This requires data to be collected on the number of hours spent on correcting errors, the number of errors occurring and the impact domain of those errors as well as the numbers of hours spent on making changes, the number of changes made, and the impact domain of those changes. Collecting such data requires not only an error tracking and change management system, but also a precise time accounting system to be in place over a longer time period. It is not something that can be done within a short time frame [4].

1.3 Project Objectives

The goal of the project was restricted to measuring the size of existing web applications and to finding a method of predicting the impact domain of a particular maintenance task (i.e., error correction or change request). In accordance with the goal / question / metric approach advocated by Basili, this lead to two questions: (1) how many size units does the software system have, and (2) what proportion of those size units are affected by a given maintenance task [6].

To answer the first question, one needs to apply one or more sizing metrics. To answer the second question, one has to measure the size of the impacted domain. Thus, two metrics are required: (1) a size metric, and (2) an impact metric. These two metrics are discussed in the Sections 2 and 3, respectively.

2. Measuring Web Application Size

While there is considerable similarity between traditional software systems and Web-based systems, there are also significant differences between them. One area that illustrates this dual nature is cost and effort estimation. There exists a mature body of knowledge for performing such estimates on traditional software systems, but such methods may not appear to be directly application to Web applications. Section 2.1 below summarizes such of this related work.

As with traditional software systems, there are several ways to measure the size of Web applications. Sections 2.2-2.4 discuss three specific techniques: function points, statements, and object points. Section 2.5 describes how the measurements resulting from these techniques are normalized and used for measuring Web application size.

2.1 Related Work

There is some research on cost estimation models for Web applications, such as the COBRA model [24]. To have an appropriate cost estimation for web applications, one must have an accurate assessment of the size of the software system. Web-based applications' unique features require new size metrics to accurately represent the effort involved in projects, since the traditional approaches such as LOC and FPs may not apply. Therefore, some size metrics tailored for Web development have emerged, such as object points, application points, and multimedia points [11].

Some sizing methods are based on existing software size measurement. For example, Candido et al use a simplified function point based on IFPUG (International Function Point User Group) and NESMA (Netherlands Software Metrics Association) to estimate the size of application developed in PHP, HTML, Java, and MySQL. In [1], the authors evaluated a functional size measurement called OO-Method Functional Points for the Web (OOmPFWeb) and shown that OOmPFWeb can improve function point counting.

Reifer [23] developed a new sizing metric called Web Objects, and an adaptation of the COCOMO II model called WebMo to accurately estimate Web-based software development effort and duration. Based on over 40 projects, the author reports that WebMo is quite useful for quick-to-market development efforts.

2.2 Measuring Size in Function Points

User are often convinced by consulting companies that function points are the universal measure of software size. There are, however, many problems with the function point sizing method. One problem is that there are many variations which claim to be counting function points, but are all counting something quite different, including the original Albrecht method [3], the official IFPUG method [16], the Mark II method [27], the COSMIC full function- point method [2], the IBM Germany function-point method [22] and the fast function-point method [18]. It seems there are almost as many function-point counting methods as there are organizations counting function-points.

Another problem is that function-point advocates claim that the method is independent of the information technology used, which it never was. Therefore, the count will differ depending on the type of database and data communication systems employed. Web-based systems will exhibit another function-point count that is different than client/server systems or monolithic mainframe systems. Normalized relational databases will render more function points than hierarchical file systems. The definition of what is an input and what is an output is influenced by the technical environment that an application is running in.

2.3 Measuring the Size in Statements

In light of the problem with function points described above, counting statements (in addition to function points) is seen as a wise choice. Statements are arguably a better measure for sizing software than lines of code because in the era of modern text editors, a line does not mean anything. Lines can vary from 72 to 720 characters. It is a question of the editor and the programming style how many lines there will be. With modern languages such as Java, JSP, XML and XSL lines tend to be particularly long with many elements and statements on one line. With older programming languages there are often many lines per statement such as with complex if-statements in COBOL. Therefore, lines of code are a highly unreliable measure of software size, despite the coincidental correlations which some researchers have found between lines of code and program complexity [13].

Statements can, on the other hand, be defined as syntactical units in terms of the language used. C-type languages have statements that end with a semicolon or with a sweeping bracket as with a function or method definition. There are some exceptions to this rule (e.g., macros), but these exceptions can be managed. XML-type languages have statements that begin and end with paired tags. Thus each end tag like </name> can be counted as a statement.

2.4 Measuring Size in Object Points

Some years ago, one of the authors introduced the object-point method for sizing object-oriented systems based on weighted counts of the object model [26]. The units of measure are classes, interfaces, attributes, methods and associations. Classes are weighted 4, interfaces 3, associations 2 and attributes 1. Methods are weighted from 1 to 5, depending on the complexity of their flow graphs, i.e. McCabe metric [19].

In the case of relational databases, each table is counted as a class and each foreign key and index as an association. In the case of XML data structures, data types are counted as classes whereas the data elements are counted as attributes. This method has proven to be very stable in measuring the code of existing, object-oriented applications and corresponds well to the number of statements.

Since function-points are intended to measure the communication between an application system and its environment, whereas object-points are intended to measure the architecture of a system, these two size measures will not always correlate. In fact, they seldom do. By using both of them, the user is given two independent size measures: (1) one for the system communication, and (2) one for the system architecture.

Statement counts tend to correlate with object-point counts, except in cases where the methods are particularly large. This happens quite frequently when programmers are working in a procedural mode. To balance this off, the statement count should also be used as an indicator of code size.

2.5 Normalizing the Raw Size Measures

The counts described above give the raw size of a system independently of the complexity and quality of that system. However, as is known from the literature, both complexity and quality have an effect on the maintenance effort [5]. Therefore, the raw size of a system should be adjusted by both measures. It goes beyond the scope of this paper to deal with all of the possible complexity and quality measures. In this project, eight complexity attributes and eight quality attributes were used.

The complexity attributes used were class hierarchy, data usage, data flow, interface, control flow, coupling, cohesion, and language complexity. All of them were normalized to a rational scale of 0 to 1 with 0.5 as the median complexity as recommended by the ISO standard 9126 [17]. The complexity of a given component is the average complexity of the eight different normalized metrics. (See Sample 1.)

```
+----------------------------------------------------------------------+
|             C O M P L E X I T Y       M E T R I C S                  |
+----------------------------------------------------------------------+
|       DATA USAGE COMPLEXITY (Chapin Metric)      =======>  0.537      |
|       DATA FLOW COMPLEXITY (Elshof Metric)       =======>  0.119      |
|       COHESION COMPLEXITY (Constantine Metric)  =======>  0.080      |
|       INTERFACE COMPLEXITY (Henry Metric)        =======>  0.920      |
|       CONTROL FLOW COMPLEXITY (McCabe Metric)    =======>  0.334      |
|       COUPLING COMPLEXITY (Card Metric)          =======>  0.153      |
|       CLASS COMPLEXITY (Chidamer Metric)         =======>  0.889      |
|       LANGUAGE COMPLEXITY  (Halstead Metric)     =======>  0.571      |
|                                                                      |
|       AVERAGE  PROGRAM COMPLEXITY                =======>  0.450      |
+----------------------------------------------------------------------+
```

To adjust the raw size of a system by its complexity, the actual complexity, e.g. 0.6, is divided by the median complexity 0.5, giving for example a multiplication factor of 1.2.

$$Complexity\_Adjustment\_Factor = \frac{Actual\_Complexity}{Median\_Complexity}$$

If the complexity is low, say 0.4, the adjustment factor is 0.8. Multiplying the raw size by the adjustment factor will give the adjusted size of the software. If the raw count of function points is 1000 and the complexity metric is 0.6, the adjusted size will be 1200 function-points.

$$Quality\_Adjustment\_Factor = \frac{Median\_Quality}{Actual\_Quality}$$

The quality attributes used were modularity, portability, flexibility, testability, reusability, readability, conformance, and maintainability. These too were normalized to a rational scale of 0 to 1 with 0.5 as the median complexity. The quality of a given component is the average quality of the 8 different normalized metrics. (see Sample 2)

Sample 2: Code Quality Metrics:

```
+----------------------------------------------------------------------+
|                 Q U A L I T Y     M E T R I C S                      |
+----------------------------------------------------------------------+
|       DEGREE OF MODULARITY                       =======>  0.921      |
|       DEGREE OF PORTABILITY                      =======>  0.199      |
|       DEGREE OF FLEXIBILITY                      =======>  0.709      |
|       DEGREE OF TESTABILITY                      =======>  0.553      |
|       DEGREE OF READABILITY                      =======>  0.900      |
|       DEGREE OF REUSABILITY                      =======>  0.111      |
|       DEGREE OF CONFORMITY                       =======>  0.100      |
|       DEGREE OF MAINTAINABILITY                  =======>  0.520      |
|                                                                      |
|       AVERAGE  PROGRAM QUALITY                   =======>  0.550      |
+----------------------------------------------------------------------+
```

To adjust the raw size of a system by its quality, the median quality of 0.5 is divided by the measured quality, e.g. 0.6, giving for example a multiplication factor of 0.8.

$$Quality\_Adjustment\_Factor = \frac{Median\_Quality}{Actual\_Quality}$$

If the quality is high, say 0.6, the adjustment factor is 0.8. Multiplying the raw size by the adjustment factor will give the adjusted size of the software. If the raw count of function points is 1000 and the quality metric is 0.6, the adjusted size will be 800 function-points.

$$Adjusted\_Count = Raw\_Count \cdot \left(\frac{Actual\_Complexity}{Median\_Complexity} \right)$$

3. Measuring Web Maintenance Impact

In maintenance it is not enough to know the size of the system being maintained. One must also know which portion of the system is being affected by the change. To answer that question, it is necessary to know which entities of a system are affected down to a very low level of granularity and to measure the size and complexity of those entities. This is referred to as impact analysis [7].

3.1 Identifying Elementary Software Entities

In procedural systems, the lowest-level entities are the procedures, which in C are functions, in PL/I are procedures, in FORTRAN are subroutines, and in COBOL are paragraphs. To them must be added the variables they use and the files, masks, and database entities they access.

In object-oriented systems, the lowest-level entities are the methods and attributes of classes, as well as the interfaces through which the methods are accessed and the files and databases, which the methods access.

In web-based systems, elementary entities are the widgets of the web pages and the operations of the web services, together with the procedures that are embedded in the page definitions, as is the case with Java Script. These web-specific entities are combined with other object-oriented entities to form a complex network of interacting, interdependent units of code.

3.2 Tracing Source Entity Relationships

Since software entities are highly interrelated, changing any one entity will have an effect on all its dependent entities. Therefore, these relationships have to be identified and stored for tracing. In procedural programs, there are two basic interdependencies between separate procedures: (1) the sharing of common data attributes, and (2) the calling of other procedures.

To trace these dependencies, one must store all of the procedure calls, giving the caller, the callee and the parameters, as in:

<calling_procedure><called_procedure><parameters>

as well as all of the common data usages, giving the user, the used data, and the usage type, input, output, ID:

<using_procedure><used_data_group><usage_type>

In object-oriented systems, there are three basic interdependencies: (1) inheritance, (2) association, and (3) sharing of common attributes. Classes are dependent on higher level classes in so far as they inherit methods and attributes from them:

<inheriting_class><inherited_class><overriding_parameters>

Methods are dependent on other methods within the same class or class hierarchy, if they share the same data attributes. This gives the dependency

<using_method><used_data_attribute><usage_type>

A further type of entity to be found in the source that is often neglected by the research community is the database. Database tables are defined in the data base schemas, e.g. in the create table SQL declarations. A table declaration will give the table name, the names and types of attributes, the key names and the index name. Data base tables will refer to one another. This gives the relationship:

<base_table><target_table><connecting_key>

Besides that, methods access database tables. So when analyzing the source, it is necessary to recognize database access operations. These relationships should be stored together with the database relationships. They are of the type:

<target_table><accessing_method><access_type>

Finally, there are file type entities defined by a DTD or XML schema. These schema descriptions are kept in separate libraries and are connected to the physical files by the schema name. Every file name will be associated with one or more schema names, thus giving the relationship

<schema><file><file_type>

As is the case with database tables, there are methods or procedures which access the files described by the schemas. When parsing the source code, it is possible to recognize file operations. These methods to file relations should be extracted from the source and stored as tuples, together with the database access relationships:

<file><accessing_method><access_type>

In web-based systems, some additional dependencies are added to the conventional ones. There are four additional dependencies: (1) XML data types, (2) templates, (3) Web pages, and (4) Web services.

In XML, any data element of one type can refer to another data type, giving a type relationship. This relationship can be conditional or non-conditional, e.g.

<referring_type><referred_type><reference_condition>

Template dependencies occur when a style sheet contains a template and when a template contains another template as depicted by the tuple:

<base_template><target_template><conditionality>

Web pages are implemented as either HTML forms or XSL style sheets. Both contain templates to define the layout of their data presentations. Any one webpage will be related to one or more templates with the relationship:

<webpage><template><conditionality>

Web service dependencies are a chain of dependencies. Operations have inputs and outputs, inputs and outputs are messages, and messages have data types. Operations in the web service interface are also methods in some server class. Thus, an entity type is required:

<base_type><base_entity><target_type><target_entity>

Operation	Input & Output
Input	Message
Output	Message
Message	Type
Type	Type

Web services have to be traced from the operation name to the name of a method or interface within the server source. In addition, they have to be traced from the message to the data types that represent that message. It is imperative to first extract the data type relations from the schema descriptions and the methods from the server source before processing the web services.

3.3 Tracing Error Reports and Change Requests

All of the entities and relationships outlined above can be extracted from the source code. However, there are entities outside of the software itself, namely the error reports and the change requests that are actually requirements. As such, they need to be stored and processed in the requirements management system. The requirements, including their error reports and change requests, will have their own structure and their own ontology. The entities are the names of artifacts perceived by the user, namely interfaces and reports.

Other applications that use this system see the import and export files and the web service interface offered. Thus, any change request or error will refer to an entity perceived by the end user or the using system. These are in total the user interfaces (e.g. web pages with links), the reports, the import/export files, and the web services.

In making an error report or a change request, it is up to the initiating user to relate his or her requirement to an existing entity, which could be any of the four user entities listed above. The user will want to change or correct a user interface, a report, an import/export file, or a web service.

Since these entities are also among the source entities, they represent a union of the set of user entities with the set of source entities.

Change request ->	UserInterfaces ^ Webpages Reports <= Files Imports / Exports <= (Files & Databases) WebServices <= Web Service Descriptions

Once a link has been established from the user request to some user entity to the top-level source entity, it is then only a question of searching through all of the related source entities to identify which entities can be affected by the change. In so far as each source entity has been analyzed, it will have at least two size measurements in the metric table: (1) in function points or object points, and (2) in statements. The size of the impact domain is then the sum of these sizes of all of the source entities affected by that change.

Impact size = Sum {Source.Entty_Size}

Not the size of the system as a whole but the size of the impact domain is used to estimate the costs of a maintenance task. If several such tasks are being performed at one time, when preparing a new release, then the impact domain is the super set of all entities impacted by the various error corrections and changes.

Since maintenance estimates must be both quick and cheap, the impact analysis must be automated and combined with the component sizing activities.

4. A Case Study: Measuring the Size of a Web Application at an Austrian Bank

This two section of the paper presents a case study of a project launched in the summer of 2006 to measure a web application in a Viennese bank. One of the authors was assigned the task of coming up with at least two independent size measures (including function points.) The objects of measurement were:

- C++ sources in the backend library
- Java sources in the front end library
- SQL sources in the data base library

and in the web library:

- XML sources for the web pages
- XSL sources for the templates
- XSD sources for the data schemas
- WSDL sources for the web services

4.1 Measuring C++ and Java Sources

The first three source types (C++, Java, and SQL) could be analyzed independently from one another. For this, the author already had source analyzers from previous projects, which have also been described in previous papers [25]. The source analysis tools, C++Audit and JavAudit, parse the source to identify and count classes, methods, attributes interfaces and foreign method calls. This gives the object points. In addition, the statements are counted.

Counting function-points in C++ and Java Code is done by recognizing file operations and report writing. The file operations are weighted from 3 to 6 or from 4 to 7 depending upon whether the file is read from or written to. The complexity level is determined by the complexity rating of the source member as a whole, thus deviating somewhat from the IFPUG counting rules which determine the weight of a file depending on the number of individual data elements (DETS) and the number of data groups (RETS). Reports are considered to be outputs and are weighted from 4 to 7. (See Samples 3 & 4)

Sample 3: Procedural Quantity Metrics:

```
+------------------------------------------------------------------------+
|          P R O C E D U R A L   Q U A N T I T Y   M E T R I C S         |
|                                                                        |
|        Number of Statements                    =======>     3208       |
|        Number of Input Operations              =======>      274       |
|        Number of Output Operations             =======>      233       |
|        Number of File & Database Accesses      =======>       19       |
|        Number of Function References           =======>     1153       |
|        Number of Foreign Functions referenced  =======>      879       |
|        Number of Macro References              =======>        0       |
|        Number of Macros referenced             =======>        0       |
|        Number of If Statements                 =======>      165       |
|        Number of Switch Statements             =======>        1       |
|        Number of Case Statements               =======>        3       |
|        Number of Loop Statements               =======>       21       |
|        Number of Exception Conditions          =======>      126       |
|        Number of Return statements             =======>      438       |
|        Number of Control Flow Branches         =======>      939       |
|        Number of all Control Statements        =======>      215       |
|        Number of Literals in Statements        =======>      316       |
|        Number of Nesting Levels (Maximum)      =======>        7       |
|        Number of Test Cases (Minimum)          =======>      502       |
|        Number of different Statement Types     =======>     1654       |
|        Number of Assertions made               =======>       51       |
|        Number of Function-Points               =======>      456       |
+------------------------------------------------------------------------+
```

Sample 4: Structural Quantity Metrics:

```
+------------------------------------------------------------------------+
|              C O D E   Q U A N T I T Y   M E T R I C S                 |
|                                                                        |
|        Number of Source Members analyzed       =======>       65       |
|        Number of Source Lines in all           =======>    10174       |
|        Number of Genuine Code Lines            =======>     5371       |
|        Number of Comment Lines                 =======>     3318       |
|                                                                        |
|          S T R U C T U R A L   Q U A N T I T Y   M E T R I C S         |
|                                                                        |
|        Number of Modules                       =======>        9       |
|        Number of Includes                      =======>       65       |
|        Number of Classes  declared             =======>       61       |
|        Number of Classes  inherited            =======>        0       |
|        Number of Methods  declared             =======>      766       |
|        Number of Methods  inherited            =======>       15       |
|        Number of Interfaces implemented        =======>       32       |
|        Number of Interfaces declared           =======>        4       |
|        Number of Object-Points                 =======>     2943       |
+------------------------------------------------------------------------+
```

4.2 Measuring SQL Services

Counting function-points in the SQL schemas is done in a different manner. Database tables have to be classified as either internal or external. Internal files should be weighted from 7 to 15 depending on the number of attributes (DETS) and data groups (RETS). External files that serve to transport data from one system to another are weighted from 5 to 10. Since normalized tables have no data groups, the author decided to count the number of indexes for each table to give the RETS. This is not exactly in accordance with the IFPUG counting rules, but one must realize that the function-point counting methods prescribed by the IFPUG are rather dated. However, if the customer insists on counting them, then it must be done, so whoever counts them must come up with individual interpretations. For distinguishing between internal and external files, the user had to prepare a special parameter table, since this could not be recognized in the source.

Counting object points and statements in SQL is a rather straightforward matter. Tables are counted as classes and attributes as attributes. Foreign keys are interpreted as associations. Thus, a table with 10 attributes and one foreign key has an object point count of 4 + 10 + 2 = 16. Statements are counted as each attribute, key, index, table and name space declaration, ending with a closed parenthesis, a comma or a semicolon. (See Sample 5.)

Sample 5: Database Quantity Metrics

```
+---------------------------------------------------------------------+
|          D A T A B A S E     Q U A N I T Y      M E T R I C S       |
+---------------------------------------------------------------------+
|       Number of Database Descriptions          =======>       73    |
|       Number of SQL Tables                     =======>       82    |
|       Number of SQL-Statements                 =======>     1364    |
|       Number of Data Attributes                =======>     1004    |
|       Number of different Data Types           =======>      535    |
|       Number of Primary  Keys                  =======>       94    |
|       Number of Foreign  Keys                  =======>      155    |
|       Number of DB-Indexes                     =======>       50    |
|       Number of Relationships                  =======>      235    |
|       Number of Database Views                 =======>       03    |
|       Number of Database View Attributes       =======>       17    |
|       Number of Object-Points                  =======>     2542    |
|       Number of Function-Points                =======>      606    |
|       Number of Deficiencies                   =======>      582    |
+---------------------------------------------------------------------+
```

4.3 Measuring XML Sources

For measuring XML web pages, templates, schemas, and web services, a new set of analyzers had to be developed:

- HTMLAudit for the XML/HTML definitions
- XSLAudit for the XSL style sheets
- XSDAudit for the XSD schema descriptions
- WSDLAudit for the WSDL interfaces.

The XSLAudit Analyzer analyzes the web page templates to count the variables contained within them and to trace the links to other templates. Each variable is counted as a data item (DET) and each included template as a data access (FTR). Embedded Java procedures are counted as methods, variables as attributes and the template as a class to give the object-points. Statements are counted as each pair of tags, i.e. each tag </ indicates a statement. The DETs and FTRs per template are stored in a transient template table.

The HTMLAudit analyzer then parses the HTML/XML page descriptions to identify which templates are referred to therein. Their DETs and FTRs are then taken from the template table to compute the function-points for that page. Here again a question comes up: whether to count the webpage as a whole as an input or output, or to count each template as an input or output depending on whether it is filled by the system or the user or both. The author opted for the template, i.e. widget, as the input or output.

Counting object-points and statements was again a simple matter. The web page is considered a class and each referenced template as an association. Data elements are counted as attributes. Statements are as with all XML type documents paired tags. (See Sample 6.)

Sample 6: User Interface Function-Points

```
+--------------------------------------------------------------------+
|Type;GUI_Interface_Name;GUI_Widget_Name        ;Dets;Ftrs;FcPts |
+--------------------------------------------------------------------+
|EI  ;BUTTON.XML       ;ac:button               ;0021;0001;0006  |
|EO  ;CALENDER.XML     ;ac:calendarinitializer ;0033;0001;0007  |
|EI  ;CHECKBOXES.XML   ;ac:checkbox             ;0026;0001;0006  |
|EO  ;DROPDOWN.XML     ;ac:dropdown             ;0068;0002;0007  |
|EI  ;LABELS.XML       ;ac:label                ;0016;0001;0006  |
|EO  ;PARAGRAPH.XML    ;ac:paragraph            ;0016;0001;0005  |
|EI  ;PINEDIT.XML      ;ac:pinedit40            ;0003;0001;0003  |
|EI  ;SELECT.XML       ;ac:select               ;0042;0002;0006  |
|EI  ;TEXTS.XML        ;ac:text                 ;0015;0001;0004  |
|EI  ;TEXTAREAS.XML    ;ac:textarea             ;0038;0001;0006  |
|EO  ;TEXTINPUTS.XML   ;ac:textinput            ;0042;0002;0007  |
|EO  ;PARAGRAPHS.XML   ;ac:paragraph            ;0016;0001;0005  |
+--------------------------------------------------------------------+
|GUI ;EISXML           ;All Entities            ;0336;0015;0068  |
+--------------------------------------------------------------------+
```

As a prerequisite to the web service interface analysis, the XSD schemas had to be parsed first. The XSDAudit analyzer did this. It navigates through the XML schemas picking up the references to the enclosed complex data types and counting the number of data elements in each type. The number of data fields – DETs – assigned to a base data element is the sum of the data elements of all data types referred to in the type hierarchy of that element. Instead of counting the number of databases accessed – FTRs – as prescribed by the IFPUG counting method, the author counted the number of nested complex data types. These counts were attached to the base element name and stored in a table.

The last step was to analyze the web service descriptions themselves. For this, the WSDLAudit analyzer was used. It identifies the operations referred to by the Java sources that were analyzed previously. For this, the tool must search the method table for each WSDL operation. If it is found, the input and output messages of that operation are traced to the data element that describes them. The data element is then located in the table of data types from which the number of elementary data elements and the number of data groups is taken. This gives the function-point count for the input and the output of that particular operation.

This tracing of web service operations to messages and from there to the data type descriptions also gives the number of attributes, the number of interfaces and the number of objects for the object-point count. In addition, the statements are counted as pairs of XML tags. Both the function-points and the object-points taken from the web services are assigned to the client method that invokes them, in lieu of assigning them to the server component that implements them. The statement counts were also assigned to the invoking component. (See Sample 7.)

Sample 7: Web Service Function-Points:

```
+----------------------------------------------------------------------+
|Type;WSDL_Interface_Name;XSD_Data_Type                 ;Dets;Rets;FcPts
+----------------------------------------------------------------------+
|EI  ;RSFWBMService      ;saveTopicRequestType           ;0001;0001;0003
|EO  ;RSFWBMService      ;saveTopicResponseType          ;0001;0001;0004
|EI  ;RSFWBMService      ;getTopicRequestType            ;0003;0001;0003
|EO  ;RSFWBMService      ;getTopicResponseType           ;0001;0001;0004
|EI  ;RSFWBMService      ;getDocMetaForTopicRequestType  ;0001;0001;0003
|EO  ;RSFWBMService      ;getDocMetaForTopicResponseType ;0001;0001;0004
|EI  ;RSFWBMService      ;getDocMetaForAccessRequestType ;0002;0001;0003
```

In this way, the user interfaces were sized as separate components to be stored along side the client; server component sizes derived from the C++ and Java Sources and the database table sizes were taken from the SQL sources. The size counts of the web services were, however, assigned to the invoking Java classes to give the following size metric tables:

Source Elements	FuncPts.	ObjtPts.	Stmts.
C++ Server Classes	339	13874	32479
Java Client Classes *	202	2943	3208
SQL Tables	606	2542	1564
XML GUIs	98	2744	8995

This master table was broken down in four sub-tables, one for each type. In the application analyzed, there were 533 C++ classes, 61 Java classes, 82 SQL tables and 24 XML GUIs with a total of 1245 function points, 22103 object points and 46246 statements.

4.4 Measuring the Impact Domain

To validate the estimation of a maintenance task, three sample tasks were selected and estimated:

1. To correct an incorrect result, displayed in a user interface;
2. To change an algorithm to compute a result written out in an XML export file by a web service;
3. To add an attribute to a data base table.

Each of these tasks is briefly described below.

Measuring the Impact Size of an Error Correction

In the case of the first maintenance task, it was necessary to link the error report to the target GUI. The GUI was then linked to the classes that processed the GUI, the classes inheriting from these classes, and the classes associated with them. The size of those classes and the size of the GUI were added together to give the total size of the impact domain for the error correction.

Measuring the Impact Size of an Interface Change

In the case of the second maintenance task, the export file was connected to the class that wrote it, which in turn was connected to the inherited and associated classes. The web service sizes were as pointed out above, included in the classes using those web services. (See Sample 8.)

Sample 8: Change Request Cost Estimation:

```
+----------------------------------------------------------------------+
|                     Code based Impact Analysis (Unadjusted)          |
| Product    : RZB-WEB                                                 |
| System     : EIS                                                     |
| Task       : CR-2                                    Date : 23.07.06 |
| Repository : d:\tools\maintain\softrepo\EIS\tables                   |
+----------------------------------------------------------------------+
| Source          Source      Unadjusted                        Change|
| Member          Type   Fct-Pts Obt-Pts  Stmts    Comp     Qual   Rate  |
+----------------+------+-------+-------+-------+-------+-------+------+
| Exchange       | XML  |   7   |   33  |   41  | 0.317 | 0.624 | 0.20 |
| PrepareExport  | Java |   0   |   19  |   47  | 0.411 | 0.587 | 0.20 |
| WriteXMLFile   | WSDL |   4   |    7  |   11  | 0.431 | 0.653 | 0.20 |
| XMLFile        | CPP  |   0   |   28  |  315  | 0.595 | 0.496 | 0.20 |
+----------------+------+-------+-------+-------+-------+-------+------+
| CR-2           | Mixed|  11   |   87  |  414  | 0.438 | 0.590 | 0.20 |
+----------------+------+-------+-------+-------+-------+-------+------+
```

```
+----------------------------------------------------------------------+
|                     Code based Impact Analysis (Adjusted)            |
| Product    : RZB-WEB                                                 |
| System     : EIS                                                     |
| Task       : CR-2                                    Date : 23.07.06 |
| Repository : d:\tools\maintain\softrepo\EIS\tables                   |
+----------------------------------------------------------------------+
| Source          Source        Adjusted           Factor  Factor Change|
| Member          Type   Fct-Pts Obt-Pts  Stmts    Comp    Qual   Rate  |
+----------------+------+-------+-------+-------+-------+-------+------+
| Exchange       | XML  |  3.5  |   16  |   37  | 0.634 | 0.801 | 0.20 |
| PrepareExport  | Java |   0   |   16  |   68  | 0.822 | 0.851 | 0.20 |
| WriteXMLFile   | WSDL |   3   |    5  |    7  | 0.862 | 0.793 | 0.20 |
| XMLFile        | CPP  |   0   |   33  |  378  | 1.190 | 1.008 | 0.20 |
+----------------+------+-------+-------+-------+-------+-------+------+
| CR-2           | Mixed|  6.5  |   70  |  490  | 0.877 | 0.863 | 0.20 |
+----------------+------+-------+-------+-------+-------+-------+------+
| CR-2           | x CR |  1.3  |   14  |   98  |       |       | 0.20 |
+----------------+------+-------+-------+-------+-------+-------+------+
```

Measuring the Impact of a Database Change

In the case of the third maintenance task, the data base table was linked via the access relationships to the C++ classes that accessed it. These were then linked to the inherited and associated classes and their sizes added to the sizes of the data base table itself to give the size of the domain impacted by the database change.

5. Conclusions

Unfortunately, the project did not include estimating the maintenance tasks and comparing the estimates with the actual effort involved, since this was not a pure research project, but an industrial pilot project aimed solely at sizing impact domains of maintenance tasks; the estimation was not part of the contract. The author did the best job possible in the limited time allowed to measure the size of individual software artifacts and to link those sizes to the maintenance tasks planned.

The project demonstrated that it is necessary to create a metrics database with the sizes of all classes, components, interfaces and database tables. Furthermore, a repository is required to store all of the relationships between the software artifacts. Then it proved to be possible to link error messages and change requests to the exterior artifacts visible to the user and to link them to the interior artifacts invisible to the users. The artifacts of the repository are then linked to the entries in the metric database to aggregate the sizes of the artifacts affected. The linking of the sizes of related individual sources members leads in the end to the total size of the impact domain.

With that, the goal of the project was reached, albeit with more effort than was allocated. The project was budgeted for 20 days. In the end it required 32 days. At this time, it is not clear how the customer will proceed. The outsourcing of software maintenance to the subsidiary companies in Eastern Europe is a highly political issue since the newly elected socialist government wants to restrict the exporting of jobs. The rationale for this project was to offer upper management a proof of concept, that the size of maintenance tasks can be automatically extracted from the source code. To that end it was a success.

5.1 Lessons Learned

The main lesson learned in this project was that it is not easy to measure the size of software. A line of code is a useless measurement that is totally dependent on how the code is formatted. Even statements are hard to count if there is not a specific definition of what a statement is for every language analyzed.

Function-points are claimed to be a technology independent form of sizing a system by measuring its external behavior in term of inputs, outputs, interfaces and database tables [14]. In practice however it turns out that these entities are not at all independent of the technical environment. The languages and technologies being used determine inputs, outputs, interfaces, and databases. Inputs and outputs in a web-based system are quite different from inputs and outputs in a mainframe online system or a client/server system. Relational database tables are also far removed from the VSAM files and hierarchical databases that the authors of the function-point method had in mind when they proposed the method. The sheer number of data items in a user or system interface has little bearing on the complexity of the algorithm behind those interfaces. Besides that, they are, as this project demonstrates, not at all easy to count, especially with web applications.

For these and many other good reasons, the authors oppose the use of function-points to estimate maintenance tasks – particularly for Web applications. The fact that they were counted here was due solely to the insistence of an unwary customer under the influence of deceitful business consultants. If given a choice, the authors would have restricted themselves to statements and object-points. It was the attempt to count function-points rigidly according to the IFPUG rules that caused the project budget to be exceeded by over 50 %.

5.2 Future Work

As concerns the impact analysis, it was discovered that there are several flaws to detecting related methods via static analysis. Polymorphism and dynamic binding (i.e. determining the method to be called at runtime) are impossible to resolve with only the sources. Either you consider all of the potential callees to be within the impact domain or you leave all of them out. This flaw has been pointed out by L. Briand and others [8], but their alternative to performing dynamic analysis is not a viable solution in industry. Thus, the problem remains unsolved.

Another flaw was the external coupling of C++ modules by means of external files. In case the user did not use the same file names in both the producing and the consuming modules, the link went unnoticed. The same applies to data objects addressed via pointers set at run time. For these and other reasons, the impact domain measured was certainly only a subset of the real impact domain. Here much more research is required than what is currently available. In that respect, this project serves to demonstrate how far removed the research community sometimes is from solving real industrial problems.

References

[1] Abrahao, S.; Poels, G.; Pastor, O.: "Evaluating a Functional Size Measurement Method for Web Applications: An Empirical Analysis." *Proceedings of the 10th International Symposium on Software Metrics* (METRICS'04).

[2] Abran, A.; Silva, I.; Primera, L.: "Field Studies Using Functional Size Measurement in Building Estimation Models for Software Maintenance." *Journal of Software Maintenance and Evolution*, 14(1), March 2002, pp. 31 – 64.

[3] Albrecht, A.; Gaffney, J.: "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation." *IEEE Transactions on Software Engineering,* 9(6), November. 1983, p. 639 -648.

[4] April, A.; Hayes, J.; Abran, A.; Dumke, R.: "Software Maintenance Maturity Model: the software maintenance process model" *Journal of Software Maintenance and Evolution*, 17(3), May 2005, pp. 197-223.

[5] Basili, V.; Briand, L.; Melo, W.: "A Validation of Object-Oriented Design Metrics as Quality Indicators." *IEEE Transactions on Software Engineering,* 22(10) October 1996, pp. 751-761.

[6] Basili, V.; Caldiera, C.; Rombach, H.-D.: "Goal Question Metric Paradigm." *Encyclopedia of Software Engineering*, Vol. 1, John J. Marciniak (Editor), John Wiley & Sons, 1994, p. 528-532.

[7] Bohner: "Impact Analysis in the Software Change Process: a Year 2000 Perspective." *Proceedings of the 12th International Conference on Software Maintenance* (ICSM'96: Monterey, CA; November 4-8,1996), IEEE Computer Society Press, pp. 42-51.

[8] Briand, L.; Di Penta, M.; Labiche, Y.: "Assessing and Improving State-based Class Testing: a Series of Experiments." *IEEE Transactions of Software Engineering*, 30(11), November 2004, pp. 770-783.

[9] Candido, E.; Sanches, R.: "Estimating the Size of Web Applications by Using a Simplified Function Point Method." *Proceedings of the WebMedia & LA-Web 2004*, pp.98-105, 2004.

[10] Chapin, N.; Hale, J.; Khan, K.; Ramil, J.: "Types of Software Evolution and Software Maintenance." *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1), Feburay 2001, pp. 3-30.

[11] Cowderoy, A.J.C.: "Size and Quality Measures for Multimedia and Web-site Production." *Proceedings of the 14th International Cocomo Forum*, 1999.

[12] Ebert, C.; Dumke, R.; Bundschuh, M.; Schmietendorf, A.: *Best Practices in Software Measurement: How to Use Metrics to Improve Project and Process*, Springer Berlin, October 2004.

[13] El Emam, K.; Benlarbi, S.; Goel, N.; Rai, S.: "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics." *IEEE Transactions on Software Engineering*, 27(7), July, 2001, pp. 630 – 650.

[14] Garmus, D.; Herron, D.: *Function-Point Analysis: Measurement Process for successful Software Projects*, Addison-Wesley, Reading MA., December 15, 2000.

[15] IFPUG (International Function Point User Group) online at http://www.ifpug.org/

[16] IFPUG (International Function Point Users Group), *Function Point Counting Practices Manual*, release 4.2, Westerville, Ohio, 1999.

[17] ISO/IEC: Software Product Evaluation: Quality Characteristics and Guidelines for their use, ISO/IEC Standard 9126, International Standards Organization, Genf, 1994

[18] Jones, C.: *Estimating Software,* McGraw-Hill, New York, July 1998.

[19] McCabe, T.: "A Complexity Measure." *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, December 1976, p. 308.

[20] Nerur, S.; Mahapatra, R.; Mangalaraj, G.: "Challenges of Migrating to Agile Methodologies." *Communication of the ACM*. 48(5), May 2005, pp. 72-78.

[21] NESMA (Netherlands Software Metrics Association) online at http://www.nesma.nl/sectie/home/

[22] Poensgen, B. and Bock, B. *Function-Point Analyse*, dpunkt.verlag, Heidelberg, 2005

[23] Reifer, D.: "Web Development: Estimating Quick-to-Market Software." *IEEE Softeware*, November/December 2000.

[24] Ruhe, M.; Jeffery, R.; Wieczorek, I.: "Cost Estimation for Web Applications." *Proceedings of the 25th International Conference on Software Engineering* (ICSE 2003: May 3-10, Portland, Oregon).

[25] Sneed, H.: "Impact Analysis of Maintenance Tasks for A Distributed Object-Oriented System" *Proceedings of 17th International Conference on Software Maintenance* (ICSM 2001: Florence, Italy, November 7-9, 2001) IEEE CS Press, pp. 180-189.

[26] Sneed, H.M.: "Estimating the Development Costs of Object-Oriented Software." *Proceedings of 7th European Software Control and Metrics Conference*, Wilmslow, UK, 1996, p. 135.

[27] Symons, C.: "Function-Point Analysis: Difficulties and Improvements." *IEEE Transactions on Software Engineering*, Vol. 14, Nr. 1, January 1988, pp. 2-11.