XV Jornadas de Ingeniería del Software y Bases de Datos
JISBD 2006
José Riquelme - Pere Botella (Eds)
© CIMNE, Barcelona, 2006

EXPERIENCE MEASURING MAINTAINABILITY IN SOFTWARE PRODUCT LINES

Gentzane Aldekoa[1], Salvador Trujillo[2], Goiuria Sagardui[1], Oscar Díaz[2]

1: Departamento de Informática
Mondragon Unibertsitatea
Loramendi s/n. 20500 Mondragon, Spain
e-mail: {galdekoa,gsagardui}@eps.mondragon.edu, web: http://www.mondragon.edu

2: Onekin Research Group
Departamento de Lenguajes y Sistemas Informáticos
Universidad del País Vasco
Manuel Lardizabal, 1. 20009 San Sebastián, Spain
e-mail: {struji, oscar.diaz}@ehu.es, web: http://www.onekin.org

Keywords: Maintainability, Software Product Lines, Feature Oriented Programming.

Abstract. *Families of applications are steadily emerging for distinct settings such as embedded systems, navigational systems, financial applications or even web applications. This moves the attention from single application development to Software Product Line (SPL) development where the focus is on constructing reusable artefacts of the assembly line from which final products are obtained. This paper presents a first reported experience on measuring maintainability index for SPLs where the maintainability index of each feature is measured. This yields a number of benefits towards the global improvement of maintainability before the customer product of the SPL is built.*

1. INTRODUCTION

Software Product Lines (SPLs) are defined as "*a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" [10].

The SPL paradigm distinguishes between domain engineering and application engineering, where construction of the reusable assets (a.k.a. platform) and their variability is separated from production of the product-line applications. The platform includes the architecture, software components, design models and, in general, any artefact that is liable to be reused [22].

The platform is the base on top of which products are created adding variability, which is expressed as features (i.e. increments in application functionality). Distinct techniques exist to realize feature implementations. Among them, AHEAD [1] provides an algebraic model (and tools), where the realization of each feature is separated. An SPL product is then obtained by composing the base with customer desired features.

Quality measurement is a main requirement in software development. SPLs are not an exception. However, most measures and techniques use a holistic approach where the artefact to be measured is assessed as a whole. This could be contra-intuitive for SPL products which are obtained as the synthesis of a set of features. From this perspective, the question is whether the quality (whatever this means) of an SPL product can be obtained from the quality of the features the product supports. And this in turn, poses the question of how to assess feature quality. Notice that features are “inconclusive” pieces of code which do not make sense isolately but assembled to a base or platform. In this sense, they are different from components which can be run isolately.

This feature-base measurement would permit users to select features based on the expected contribution the feature makes to the final quality of the SPL product. Or even to restrict the available products to those below a certain value for a specific quality attribute along the lines of the work described in [3].

To assess the feasibility of this approach, this work focuses on the Maintainability Index (MI) as the measure to be assessed. The MI is largely used to help reduce system's tendency toward "code entropy" or degraded integrity. Existing tools enable to measure MI for software products.

The findings of this work include the difficulty of ascertaining MI by directly measuring the feature. Rather the feature impact is assessed as the difference between SPL products, some exhibiting the feature and others lacking the feature. This indirect way permits to assess the contribution of a feature by measuring proper products rather than adapting existing MI techniques to the “inconclusive products” that features represent.

The rest of this paper is organized as follows. Section 2 reviews maintainability and AHEAD. Section 3 introduces the main ideas behind measuring SPL maintainability. Section 4 presents our experience in doing so. Section 5 describes some future work and section 6 concludes.

2. BACKGROUND

2.1. Maintainability

Maintainability is the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changing environment [14].

The literature shows several efforts to characterize and quantify software maintainability. The most widely accepted propose the use of a polynomial regression model where regression analysis is used as a tool to explore the relationship between software maintainability and software metrics [25, 26].

Zhou determines maintainability measures as (i) a managerial assessment to quantify the cost of maintaining existing software systems, (ii) as a quality assessment and control mechanism to drive software development efforts, and as (iii) a mechanism to enforce maintainability standards prior to acceptance and/or delivery [26].

Van Emden defines complexity as "*the way in which a whole is different from the composition of its parts*" and analyses software complexity via Entropy [24]. Factor analysis is another statistical technique wherein metrics are orthogonalized into unobservable underlying factors, which are then used to model system maintainability [20].

A program's maintainability is calculated through a *Maintainability Index* (MI) [23], which uses a combination of widely-used and commonly-available measures. Note that all are based on average-per-code-module measurement. The MI of a program is a polynomial:

$$MI = 171 - 5.2 \times \ln(aveV) - 0.23 \times aveV(g') - 16.2 \times \ln(aveLOC) + 50 \times \sin\sqrt{2.4 \times perCM} \qquad (1)$$

where:

`aveV` is the average Halstead Volume V per module [13], which is the computational complexity of a program's module measured directly from source code.

`aveV(g')` is the average extended cyclomatic complexity per module [18], which measures the number of linearly-independent paths through a program module.

`aveLOC` is the average count of lines of code (LOC) per module. Non Commenting Source Statements (NCSS[i]) are measured.

`perCM` is the average percent of lines of comments per module.

The larger the MI, the more maintainable is the program. MI measurement is not a trivial task, which requires sophisticated tooling. *SemanticDesigns*[ii] offers tools to measure MI for Java and C code [5]. MI has been widely used and tested in industrial cases [26,21]. These cases were compared by [11,20].

[i] JavaNCSS. http://www.kclee.de/clemens/java/javancss/
[ii] Semantic Designs. http://www.semdesigns.com/

Recent works evaluate SPL quality focusing on SPL architecture. They study different quality attributes such as modifiability [7,9], flexibility [16], and reusability [19]. Bengtsson studies maintainability by developing different change scenarios where the change impact is analyzed [6].

In these works, architecture's maintainability was evaluated. Neither the whole SPL, nor the products were studied.

2.2. AHEAD and Feature Oriented Programming

Feature Oriented Programming (FOP) is a paradigm of SPL synthesis where features are the building blocks of products [4]. Features are units (i.e. incrementing application functionality) by which different products can be distinguished and defined within an SPL.

In general, an SPL is characterised by the set of features it supports, e.g.

$$MySPL = \{ feature2, feature1, base \} \tag{1}$$

whereas a product is obtained as the synthesis of some of those features in the base or platform of the SPL. For instance

$$\begin{aligned} \Pr oduct1 &= feature2 \bullet feature1 \bullet base \\ \Pr oduct2 &= feature1 \bullet base \\ \Pr oduct3 &= feature2 \bullet base \end{aligned} \tag{2}$$

where $\bullet$ stands for synthesis or composition.

Algebraic Hierarchical Equations for Application Design (AHEAD) is a model of FOP where each feature implementation (a.k.a. layer) encapsulates the set of files (a.k.a. artefacts) realizing its functionality [4].

Feature realization is described as increments to functionality (a.k.a. refinements) that provides the required feature. The refinement depends on the artefact at hand. It is not the same the notion of refinement in Java that in XML. For Java artefacts, a refinement can introduce new data members, methods and constructors to a target class, as well as extend or override existing methods and constructors of that class[iii]. However, as refinement capability is not present in those vocabularies, AHEAD provides its own language. Hence, *Jak* is a Java-like language supporting refinements[iv].

```
(a)                         (b)                        (c)
feature BasePlatform ;      feature Feature1 ;         feature Feature2 ;
class Foo {                 refines class Foo {        refines class Foo {
  int counter ;               void reset ()              void reset ()
  int getCounter ()           { counter=0; }             { counter=10; }
   { return counter; }      }                          }
}
```

Figure 1. Jak Refinement

[iii] Refinement resembles regular inheritance. However, note that it is a mixin, i.e. a class whose superclass is parameterized [8]. The link to the superclass is not fixed until composition (in regular inheritance is fixed).
[iv] Jak2java would transform Jak to Java [1].

Figure 1a shows a base artefact *Foo* defining a variable member (*counter*), and a method (*getCounter*) realizing feature *BasePlatform*. Now consider that the realization of feature *Feature1* implies leveraging previous class with a new method (*reset*). Figure 1b shows the definition of this refinement function in Jak [4]. Figure 1c shows another refinement *Feature2* where the same *reset* method is defined. *Feature2*'s *reset* overrides previous method if the composition is *Feature2•Feature1•BasePlatform*.

3. MEASURING MAINTAINABILITY INDEX FOR AHEAD'S SPL

Focusing on Java, feature realization is achieved through *Jak* files. A first approach would be to measure directly *Jak* files. However, a number of problems prevented us from doing so. First, existing tools are targeted to Java or C. So, *Jak*-specific measurement tooling would be required. Second, *Jak* can be measured adapting existing MI. However, the measurement of refinements would require further study because a refinement is not a complete Java class, but an extension. So, it is likely that a new MI for refinements would be necessary. Third, refinements may override existing code which would make the MI result to change. Thus, there are some interactions among features we may want to detect. These feature interactions would affect to product's MI [17].

In other words, feature addition does not always have a monotonic effect on the MI measure. The addition of a new feature could result in a reduction of the MI for the resulting product since a feature can make the code of the resulting product harder to maintain. Therefore, calculating the MI for each feature separately, and then, aggregating those values to obtain the MI of the product could be wrong. Unless with the AHEAD model for feature realization.

As a result, an indirect way to measure feature MI was devised. Let P be the set of total products an SPL can generate. Let P+ and P- be a total cover of P, which refer to the products exhibiting feature F and the products that lack feature F, respectively. The MI impact for feature F can be ascertained from MI(P+) and MI(P-).

This intuition is formalized in terms of a matrix. Each row stands for a product, and each column is an "incognita" that represents the MI contribution of a given feature F. Next section describes our experience on this for a sample case.

4. EXPERIENCE FOR A SIMPLE CASE

A simple calculator was developed to evaluate our ideas. *MUKalk* is a basic windows-GUI calculator offering basic operations (e.g. addition, multiplication, sinus, etc), different languages (e.g. English, Spanish, Basque, Polish), different presentation styles (e.g. casual, modern, classic), and different modes of operations (e.g. scholar, scientific). *MUKalk* consisted of 15 features, yielding hundreds of distinct calculator products [3].

We use all features (15) to build only a feature-representative subset of products (100). Then, the MI of each product was measured using *SemanticDesigns'* tooling.

Table 1 shows partially these results on the left-side. At first sight, note that the MI variation is low. Each built product size ranges from 133 KB to 170 KB. It takes around 10 seconds to build each product.

Product	**Features**	**MI**	
P000	base	128.37	$128{,}37 = x_{base}$
P001	base•plus	128.07	$128{,}07 = y_{plus} + x_{base}$
P002	base•plus•minus	127.91	$127{,}91 = z_{\min us} + y_{plus} + x_{base}$
P003	base•plus•mul	127.96	$127{,}80 = m_{mul} + z_{\min us} + y_{plus} + x_{base}$
P004	base•plus•div	127.74	$// more\_equations\_omitted$
P005	base•plus•minus•mul	127.80	
P006	base•plus•mul•minus	127.80	
P007	base•plus•minus•div	127.60	
...	...		

Table 1. Product (partial) measures

Next step was to create a system of equations with these results to get the values of MI for each feature (the equations system for *MUKalk* is partially shown on right-side at Table 1). To this end, we create a matrix where the rows contain equations. We have one row per product. The number of columns is the number of features. While creating this system, we assumed that • (composition operator) was addition. This assumption was backed by recent studies showing AHEAD's duality where the rate of refinements within total is around 10%. Thus, the most of the code (90 %) are introductions [2]. Therefore, we can argue that • would be addition if the majority of the code is introduced. This premise was later confirmed in our experiments.

MatLab[v] was used to resolve the matrix equation system. To this end, a vector (*MI_vector*) with all the MI values for products was created. Next, we show a partial vector with 16 values:

```
MI_vector=[128.37,128.07,127.91,127.96,127.74,127.80,127.60,
           127.65,127.51,128.01,127.91,127.69,127.60,128.07,
           127.74,127.83]'
```

Then, a matrix specifying product features (*PF*) was created. Note that 1 designates the feature was presented, whereas 0 means was not. Next, a partial matrix with features for 16 products is shown.

```
PF=[1,0,0,0,0;1,1,0,0,0;1,1,1,0,0;1,1,0,1,0;1,1,0,0,1;
    1,1,1,1,0;1,1,1,0,1;1,1,0,1,1;1,1,1,1,1;1,0,1,0,0;
    1,0,1,1,0;1,0,1,0,1;1,0,1,1,1;1,0,0,1,0;1,0,0,1,1;
    1,0,0,0,1]
```

MatLab used those entries (*MI_vector* and *PF*) to resolve equation as follows:

```
x=solveAxb (PF, MI_vector) // MatLab command to resolve this
```

[v] MatLab. http://www.mathworks.com/

This command returned the values of each MI impact (x_{base} is feature base's MI, y_{plus} is feature addition or plus's MI) for each feature:

$x_{base} = 128.2225$

$y_{plus} = -0.1225$

$z_{\min us} = -0.1750$

$m_{mul} = -0{,}1225$

$n_{div} = -0{,}3425$

...

The variation of the values (i.e. the partial impact on MI) were low (always below 1), it ranged from 0.1225 to 0.3425. MI decremented in all cases a feature was added. Remind that larger MI meant more maintainable. So, this result seemed reasonable.

MatLab resolution provides a range of errors. The error for base feature was the highest. So, as the MI for the base must be fixed, we decided to reformulate `MI_vector` by decreasing all values with the value of Base's MI, and resolving equation again. Then, we obtained slightly different values where the error decreased.

4.1. Improving Feature MI

The impact of MI is known from previous experience. As we noted, there are differences among feature's MI. These differences are related to the complexity of a given feature implementation where the impact of MI is large (i.e. the decrement of MI is large).

In this scenario, it enables to locate features where implementation should be improved in order to improve product's MI. The focus should be placed on features most used. This depends on feature commonality (i.e. the number of products in an SPL where the feature appears) [3].

We may focus on features where either MI, either feature commonality are large. On these features, we can take design decisions such as (i) simplify feature code, (ii) refactor feature code, or (iii) split feature code among new created features.

4.2. Optimizing Product MI

Feature models (FM) are models where features are arranged in [12, 15]. It is the customer which uses FM to configure a product selecting desired features. In this scenario, we provide MI to those customers interested in future product maintainability. This information would help them during the decision taking process. Hence, MI together with other attributes (e.g. quality, cost or time) may be used to extract knowledge about the SPL.

To do so, it is necessary to transform the FM into a reasoning framework such as the one presented in [3]. Doing so, it is possible to obtain information about the FM such as how many potential products are available from the FM, which is the lowest price for a product, and which is the best product in terms of quality, usability, performance, security, and so forth.

As we know the MI impact for each feature, we can use [3] to automate reasoning on maintainability. At this point, customer may ask something like "*offer me only the set of products with a specific level of quality attributes*". The measures obtained before together with automated reasoning frameworks would provide an answer to these problems.

5. FUTURE WORK

Additional Maintainability Metrics. This work focused on the measurement of SPL's MI based on current software product metrics. Nonetheless, there exists other software metrics such as the aptitude to be analyzed, the aptitude to be changed, stability, etc. Further studies are needed to measure them.

Quality Attributes. Following the approach presented in this work, other quality attributes indexes can be measured for SPLs. Attributes such as productivity, usability, modifiability need further attention.

Tool support. Some infrastructure was developed to automate the building of products and the gathering of measures. This infrastructure relies on scripting techniques. Specific tool support is needed in this field to ease SPL measurement.

Different SPL approaches. There are distinct approaches to SPL from which we chose AHEAD. It would be interesting to construct the same SPL using different variability implementation techniques, and evaluate whether the same MI are obtained for the same product constructed with different approaches.

Artefact heterogeneity. Currently quality attributes are intended to measure source code. However, other artefacts such as XML documents are increasingly used to develop applications. Mechanisms to measure quality attributes for them are envisioned.

Feature dependency. A feature usually *requires* or *excludes* other features. These dependencies are represented through FM. We did not consider these dependencies. They should be considered in further studies.

6. CONCLUSIONS

The need for SPL measurement, and particularly maintainability, is a major issue currently in SPL development. We concentrated on AHEAD because it provides a systematic approach to SPL development where a model to separate artefacts within features is provided.

This paper presented a first reported experience on measuring SPL maintainability. It introduced an initial model to measure it, and presented some experimental results with an in-house developed case study. Our results confirmed partially our assumptions that need to be evaluated with further cases.

This work proposed the use of maintainability index to take design decisions that would improve globally maintainability. As well, we provided some insights on automated reasoning. Finally, some open issues for future work were proposed.

Acknowledgments

This work was partially done while Gentzane Aldekoa was visiting Don Batory at the University of Texas at Austin. It was co-supported by the Spanish Ministry of Science & Education and the European Social Fund under contract TIC2005-05610. Salvador Trujillo enjoys a doctoral grant from the Ministry of Science & Education. Special thanks to Ira Baxter and Mike Hayes (Semantic Designs), which kindly provides us a free version of DMS.

REFERENCES

[1] AHEAD Tool Suite. http://www.cs.utexas.edu/users/schwartz/ATS.html

[2] S. Apel, and D.Batory. *A Quantitative Analysis of Aspectual Mixin Layers*. GPCE. 2006. Portland, Oregon (USA)

[3] D. Benavides, A. Ruiz Cortés, and P. Trinidad. *Automated reasoning on feature* models. Advanced Information Systems Engineering: 17th International Conference, CAiSE. 2005.

[4] D. Batory, J.N. Sarvela, and A. Rauschmayer. *Scaling Step-Wise Refinement*. IEEE TSE. June 2004.

[5] D. Baxter, C. Pidgeon, and M. Mehlich. *DMS®: Program Transformations for Practical Scalable Software Evolution*. ICSE 2004.

[6] P. Bengtsson and J. Bosch. *Architecture Level Prediction of Software Maintenance*. 3rd European Conference on Software Maintenance and Reengineering. 1999.

[7] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. *Architecture-level modifiability analysis*. Journal of Systems and Software, 69. 2004.

[8] G. Bracha and W. Cook. *Mixin-Based Inheritance*. ECOOP/OOPSLA 1990.

[9] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison Wesley. 2001.

[10] P. Clements and L.M. Northrop. *Software Product Lines - Practices and Patterns*. Addison-Wesley. 2001.

[11] D. Coleman, D. Ash, B. Lowther, and P.W. Oman. *Using Metrics to Evaluate Software System Maintainability*. IEEE Computer, 27(8). 1994.

[12] K. Czarnecki, S. Helsen, and U.W. Eisenecker. *Staged Configuration Using Feature Models*. In 3rd International Conference on Software Product Lines, SPLC. 2004.

[13] M. H. Halstead. *Elements of Software Science*. Operating, and Programming Systems Series, Volume 7. Elsevier. 1977.

[14] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. 1990.

[15] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. *Feature–Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute. Carnegie Mellon University. 1990.

[16] N. Lassing, D. Rijsenbrij, and H. van Vliet. *Towards a Broader View on Software Architecture Analysis of Flexibility*. Proceedings of the Asian-Pacific Software Engineering Conference (APSEC). 1999.

[17] J. Liu, D. Batory, and C. Lengauer. *Feature Oriented Refactoring of Legacy Applications*. ICSE, 2006.

[18] T. J. McCabe, and A. H. Watson. Software Complexity, *Crosstalk*, *Journal of Defense Software Engineering 7*, 12 . December 1994.

[19] G. Molter. *Integrating SAAM in Domain-Centric and Reuse-Based Development Processes*. Proc. 2nd Nordic Workshop Software Architecture (NOSA '99). 1999.

[20] J.C. Munson, and T.M. Khoshgoftaar. *The detection of fault-prone program*. IEEE Transactions on Software Engineering. Volume 18, Issue 5. May 1992.

[21] T. Pearse, and P. Oman. *Maintainability measurements on industrial source code maintenance activities*. 11th International Conference on Software Maintenance (ICSM'95). 1995.

[22] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer. 2005.

[23] SEI. Maintainability Index Technique for Measuring Program Maintainability. http://www.sei.cmu.edu/str/descriptions/mitmpm.html

[24] M. Van Emden. *An Analysis of Complexity*. Mathematical Centre Tracts. 1971.

[25] K. D. Welker, and P. W.Oman. *Software Maintainability Metrics Models in Practice*. Crosstalk, Journal of Defense Software Engineering 8, 11 November/December 1995.

[26] F. Zhuo, B. Lowther, P. Oman, and J. Hagemeister. *Constructing and Testing Software Maintainability Assessment Models*. Proceedings of the First International Software Metrics Symposium. 1993.