Tracing Complexity from Requirements to Implementation

Gabriela Arévalo[1,2], Gabriela Robiolo[1], Miguel Martínez Soler[1]

[1] Universidad Austral
Avda. Juan de Garay 125
(1054), Buenos Aires, Argentina
[2] CONICET
Avda. Rivadavia 1917
(1033), Buenos Aires, Argentina
{garevalo,grobiolo}@austral.edu.ar, miguelmsoler@gmail.com

Abstract. Complexity metrics are useful measurements to ascertain the quality of software. However, the metrics are only applied and isolated in one specific development phase and they can not trace the quality (and the evolution) of the same software artifacts in all development phases within the context of the software lifecycle. In this paper, we propose a methodology to use a traceable metric to measure software artifacts from requirements to implementation in a system. We validate our approach on transactions defined in use cases and implemented in source code. Some initial results show that our approach can solve the problem of tracing software artifacts during the software development process.

1   Introduction

UML models provide a useful means to have a controlled (clean) process from requirements to implementation levels for developers during software development. The most known models are Use-Case Model, Analysis Model, Design Model and Implementation Model [8]. These models are not independent one from each other, but they keep different relationships between them. For example, the Use Case Model describes the proposed functionality of a target system. The Analysis Model describes the structure of the system or application, detailing the logical implementation of the functional requirements identified in the Use Case Model. The Design model builds on the Analysis Model by describing in detail the structure of the system and how the system will be implemented. Finally, the Implementation Model represents the physical composition of the implementation in terms of subsytems and implementation elements (directories and files, including source code, data and executable files).

Even when we have described briefly all the models, we observe that there are (implicit) dependencies that tightly link one model to another one. These dependencies are named *traces* [6]. They can be defined through the historical or process relationships between elements that represent the same concept in

the different models. We consider them as implicit because there are no rules to derive the dependencies from one model to another one.

However, the possibility of tracing different elements and their relationships between the models is important mainly in software maintenance. During this phase, the software engineers must cope with changes in a system (for example, due to new user needs or new platforms). These changes can affect entities in the implementation or in any intermediate products of a system. But in any development level, the modelled entities are not just a lonely set of software artifacts, but they result from a mapping of high/low level entities designed in another model. For example, if we consider the design and implementation levels, a Java class in a package in a system is the low level representation of a modelled class of the class diagram of the system. Any change can generate improvements or errors, and the developers should be able to (forth and back) track them. But the process of tracing is not trivial because these dependencies (between entities) are implicit.

During software development, we apply different metrics on software elements and relationships to measure different attributes of the software. Specifically in case of size or complexity metrics, which provides a measurement to a specific internal product feature in the target software, we have specific metrics for each particular model or phase of a system lifecycle. Briefly, Function Points (FP) [4] are applied to models during modeling/requirements phase, Use case Points [9], Transactions and Paths [15] are based on Use case Models, Chidamber-Kemerer's [7] metrics are design-oriented, cyclomatic complexity [11] is graph-oriented and Lines of code is code-oriented. Even when the choice of metrics models is wide, there are no metrics that could be applied and traced through the complete process of software development (from requirements to implementation).

The application of specific metrics for each development phase is a drawback to track the evolution of the software elements and relationships in a system. In other words, the developers can define and measure the requirements during the first stages of an application development. They can also calculate requirements size and complexity [15], but these metrics will be only kept as information of this development phase, because it implies a particular characteristic (in our case, requirements) in a specific model. However, software engineers must have a global view of the system, and must be able to identify each software element (and its characteristics) from requirements to implementation phases. It is important for them to know if they were implemented (or not), decomposed in other software components, integrated in an existing one and its size and complexity, just to mention some.

To cope with this requirement, we develop an approach based on two metrics proposed in Robiolo et al. work [15, 10]: Number of Transactions (nT) and Number of Paths (nP). They are based on use cases and computed from textual descriptions of use cases and UML Models (e.g., secuencial diagram, which is a design model). nT is the number of transactions in the use case identified from the actor stimulus in the text by the verbs, i.e. the actor actions interacting with the system. nP is the number of paths of a transaction of the use case textual

description. Each transaction has a main path, which is the textual description of the regular interaction between the actor and the system, and alternative paths, which are identified by the IF ... THEM expression. Thus, the computation of nP in use case textual description is based on the same idea that MacCabe complexity [11] when applied to code, so each IF ... THEM expression adds 1 to the complexity computation. Summarizing, nT measures the use case size and nP, the transaction complexity

In this paper, we show how both metrics can be calculated in source code, and we show how the traceability of the metrics helps to understand how the different requirements were implemented in a target application, and which are the explicit differences between requirements in terms of models and source code.

This paper is structured as follows: Section 2 details our approach to analyze the source code to obtain the metrics in the system. Section 3 presents the case study, the obtained measurements and analysis of the results. Section 4 cites some related work, and finally Section 5 presents some conclusions and future work.

2 Our Approach

As mentioned previously, we need to calculate two metrics: Number of Transactions (nT) and Number of Paths (nP) starting from source code using the concept of complexity metrics. We must remark that our approach is mainly based on object-oriented languages.

The starting point is the identification of a transaction in source code. To map the definition of a transaction based on use cases into code, the key aspect was to identify the method(s) that implement the actor' stimulus in the source code. As the identified method(s) is (are) entry points to the system, we name them as *access-points*. As a transaction is made up of a set of sequential method calls, the access-point is the method that is not called but it calls other method. Thus, the nT is the number of access-points.

Once we identified the access-points of the systems, the complexity of a transaction is computed in terms of number of the paths of the transaction where the access-point(s) is(are) the starting point(s). Each transaction has a principal path, built with the chain of method calls from the starting point to the end of the transaction, and alternatives paths (AP), which are identified by the *if-then* expressions. This definition of transaction is similar to the one suggested in [18], which makes cyclomatic complexity easy to compute, where each alternative path adds one to the complexity of the principal path whose complexity is also 1. In terms of formulas, if a transaction T has a chain of methods calls $m_1 ... m_k$, then

$$nP(T) = 1 + AP(starting\ point\ method) + \sum_{1}^{k} CC(m_i) - k \qquad (1)$$

if *AP (starting point method) = 0* (i.e. the access point method has no alternative paths), then the formula will remain as in [18], where k is the number of methods calls, $CC(m_i)$ is the cyclomatic complexity metric of the method m_i.

As said previously, the identification of transactions in source code is not trivial. Therefore, we developed an approach consisting of 5 steps:

1. *Mapping from Source Code to a Metamodel.* Our goal is not to link our approach to a specific object-oriented programming language. Thus, instead of analyzing the source code itself, we generate a model of the source code to keep our analysis independent of the target source code. Our case study is implemented in Java, so we chose *Recoder* to generate the model of the source code. Recoder is a Java framework for source code metaprogramming aimed to deliver a sophisticated infrastructure for many kinds of Java analysis and transformation tools [3]. This framework parses all the source code and generates a Abstract Syntax Tree (AST) for each Java file in the system. The set of trees then are a highly detailed syntactic model, where no information is lost. They represent the model of the analyzed source code.

2. *Computation of Mc Cabe Complexity of Methods.* As we base our computation of the nP based on the Mc Cabe Complexity, we calculate it for each method in the model, represented by its Abstract Syntax Tree (AST) generated by Recoder.

3. *Reducing the Recoder model.* Even when Recoder tool provides complete information regarding the target source code, we need a reduced model of the source code to make further analysis step easier. Specifically this reduced model only keeps the information regarding method calls (to detect the access points to build the transactions) of the generated Abstract Syntax Trees (AST). Thus, we only keep the information of classes, methods, attributes, method calls (including constructors) and inheritance relationships, and the corresponding Mc Cabe complexity of all the methods.

4. *Computation of McCabe complexity of Methods. .* As we base our computation of the *nP* based on the McCabe Complexity, we calculate it for each method in the reduced model, represented by its Abstract Syntax Tree (AST) generated by Recoder.

5. *Mapping reduced model to logic facts.* In order to detect access points, the reduced model of the source code is mapped as logical facts using CLIPS [2], which is a productive development and delivery expert system tool which provides a complete environment for the construction of rules and/or object-based expert systems. Thus, any interpretation regarding the target model is simplified to writing logic rules. This logic model of the source code is used to infer more useful information of the system.

6. *Adding lost inheritance characteristics.* With Recorder, we just analyzed the object- oriented model from a syntactic viewpoint. But we have to consider semantic relationships, such as inheritance relationships that have influence in the computation of nP. Specifically, we have to add logical facts to the model that show that a subclass B of a class A, can answer a call of a method m, which is implemented in the superclass and not overwritten in B. Also, other level of inheritance have to be considered: *i.e.* if a class A is superclass of class B, and class B is superclass of class C. As the transitive inheritance information of C is a subclass of A is lost, it is added as new logical facts in the model generated by CLIPS.

7. *Detecting access points.* The detection of the access points to build the transactions is designed with logic rules implemented in CLIPS. The rules are the following ones:
   (a) Class `A` has a method `m` as an access point, if `m` is defined in class `A`, class `A` implements the interface `ActionListener`, and the signature of `m` is `void actionPerformed (java.awt.event.ActionEvent)`.
   (b) Class `A` which implements `main(String[])` as static method.
   (c) Other rules for the specific implementation environment, added manually.

8. *Building Transactions using access points.* The previous step identified the access points of the target program. We have to use them to build the transactions on the source code. As a first step, we identify which are the transactions in the different analyzed use cases. Once we have identified them, we can have different mapping strategies:
   (a) A transaction with a unique access point.
   (b) A transaction with multiple access points: For example, in a form with different fields, a piece of code is executed whenever the user fills in one. However, the transaction ends when all the fields were filled in.
   (c) Several transactions with the same access point. A transaction can be an alternative path inside the code, that is executed starting from an access point, which is common to several transactions.

9. *Computation of the nP of each identified transaction.* Once we have identified the transactions, we compute the complexity of them based on the number of paths of the method calls, as said previously in this section.

3   Case Study

To validate our approach, we use a specification of an ATM System [1] to validate the presented approach. We chose this case study because it is a middle-sized one, it was developed by third parties and we have the textual descriptions of the use cases and the Java source code. Table 1 shows the traceability between

two models: use case model and code model. Use case model is represented by the name of use cases and the actor's actions identified in use case textual descriptions, which are identified in the use case transaction. Code model is represented by Java classes, and the access points, that are identified in code using logic rules in the reduced Recorder model. The nT identified from code is 27; however, the nT in use case model is 21. Table 2 shows the use case names, the measurements values of nP obtained manually on use cases and automatically on source code. The latter ones are calculated as follows: We apply the formula 1 to each transaction that was identified by the access point and mapped to the use case. Finally, the nP of each use case transaction were summed up.

Use Case Model	Use Case Model	Code Model	Code Model
UC	Actor's actions of UC T	UC	Access point
System Shutdown	turns off	atm.ATM	(performShutdown)
System Startup	enter (Startup)	atm.ATM	(performStartup)
Session	insert, enter (Session)	atm.Session	(performSession)
Deposit Transaction	Accept	atm.transaction.Deposit	(complete Transaction)
Deposit Transaction	Choose_D	atm.transaction.Deposit	(getSpecificsFromCustomer)
Inquiry Transaction	Choose_I	atm.transaction.Inquiry	(complete Transaction)
Inquiry Transaction	Choose_I	atm.transaction.Inquiry	(getSpecificsFromCustomer)
Transaction	Choose_T	atm.transaction.Transaction	(performTransaction)
Transfer Transaction	Choose_Tr	atm.transaction.Transfer	(complete Transaction)
Transfer Transaction	Choose_Tr	atm.transaction.Transfer	(getSpecificsFromCustomer)
Withdrawal Transaction	Choose_W	atm.transaction.Withdrawal	(complete Transaction)
Withdrawal Transaction	Choose_W	atm.transaction.Withdrawal	(getSpecificsFromCustomer)
Invalid Pin Extension	re-enter	atm.transaction.Transaction	(performinvalid PIN)
Transaction not specified		ATMMainE	(actionPerformed)
Transaction not specified		simulation.ATMPanel.72. . . 09	(actionPerformed)
System Startup	enter (Startup)	simulation.BillsPanel.26. . . 84	(actionPerformed)
Session	Insert	simulation.CardPanel.61. . . 43	(actionPerformed)
Transaction not specified		simulation.LogPanel.45. . . 72	(actionPerformed)
Transaction not specified		simulation.LogPanel.45. . . 63	(actionPerformed)
Session	Insert	simulation.SimCardReader. . .	(actionPerformed)
Deposit Transaction	Accept	simulation.SimEnvelopeAc. . .	(actionPerformed)
Session, Deposit Trasa, Transfer Trasa	enter(Session),choose (all)	simulation.SimKeyboard. . . 08	(actionPerformed)
Session, Deposit Trasa, Transfer Trasa	enter(Session),choose (all)	simulation.SimKeyboard. . . 09	(actionPerformed)
Session, Deposit Trasa, Transfer Trasa	enter(Session),choose (all)	simulation.SimKeyboard. . . 38	(actionPerformed)
Session, Deposit Trasa, Transfer Trasa	enter(Session),choose (all)	simulation.SimKeyboard. . . 74	(actionPerformed)
System Startup, System Shutdown	turns on, turns off	simulation.SimOperatorPanel.	(actionPerformed)
Transaction not specified		simulation.SimReceiptPrinter.	(actionPerformed)

Table 1. Class, Access Point, Use Case and Transaction identified in a Traceability Relationship

Analysis of the Results. From Table 1, we can observe that our hypothesis that there is a dependency between transactions at use case level and source code level is confirmed. Thus, we consider that the transaction is traceable. However, the traceability relationship between transactions at use case level and code level is not one to one, but it is zero to many. This means that a code transaction may be not defined at the use case level, and that a use case transaction may be decomposed in more than one transaction at the code level. But when analyzing the results, we do not keep traces between paths at use case level and at code level (as we do in transactions), because an alternative path is not identified as a requirement unit, as the requeriment unit is defined by a principal path. Even

when we could find traces between paths, so far the information is not relevant in our analysis.

Table 2 shows an important difference between *nP (UC)* and *nP (Code)*, the value of *nP(Code)* are higher than the values of *nP (UC)* . There are also differences between use case transaction and code transactions. This result can be considered normal because the level of detail of developers is different when they work in requirements phase and programming phase.

Discussion. Regarding the methodology, the most difficult aspect was the identification of the access points in the source code to build the transactions, because it is based on rules that we have built. In this paper, we limit the number of rules, but analyzing other case studies, other ones can be added to refine the results. Moreover, the developed methodology to identify a transaction in source code is complex. The reason of this problem is that the implementation environment does not provide facilities to trace elements from requirements to code. Also the fact that the use case model is informal results a disadvantage of the tracing detection and metrics calculation.

Use case name	nP (UC)	nP (Code)+ k
System Startup Use Case	2	18
System Shutdown Use Case	1	8
Session Use Case	4	81
Transaction Use Case	4	94
Withdrawal Trasaction Use Case	2	33
Deposit Trasaction Use Case	3	115
Transfer Trasaction Use Case	1	52
Inquiry Trasaction Use Case	1	17
Invalid Pin Extension	3	85

Table 2. Path measured on UC and Code

4 Related Work

Several authors have dealt with the traceability issue in different contexts, such as databases information, maintaining, requirements or analysis, although none cope with tracing metric from requirement to code. Pfleger et al. [14] cope with processing measurements because they are more difficult to track, as they often require traceability from one product or activity to another one. They argue that in this case, databases of traceability information are needed, coupled with software to track and analyze progress. Moreover, Murray and Shahabuddin [12] point out that in the flight software in an Earth-orbiting science instrument named Aquarius they have institutional requirements on software development that include requirements traceability. Requirements must be traced from source requirements, and down to both verification scenarios and implementing design

elements. They need to be able to produce *traceability matrices* for planning and review purposes, which they do easily from the use case descriptions, with their embedded requirement trace information, using Perl.

Paul et al. [13] present approaches needed for current software metrics database environments to achieve efficient execution and management of large projects. They proposed a combination of critical metrics and analytic tools that can enable highly efficient and cost effective management of large and complex software projects. Concerning changes in requirements metrics, they require considerable effort to determine the extent of necessary revisions. These measures include requirements traceability and requirements stability metrics, which point out that the database requirement traceability will be another extension of our work.

Shepperd [16] investigates the various existing metrics for system component size. An alternative metric is proposed, based upon the traceability of functional requirements from a specification to design. It is suggested that the multidimensional model is more effective at identifying problem modules than any single metric.

Antoniol et al. [5] presents a method to establish and maintain traceability links between subsequent releases of an object-oriented (OO) software system. Maintaining traceability links between subsequent releases of a software system is important to evaluate version deltas, to highlight effort/code delta inconsistencies, and to assess the change history. This can help in planning the future steps of evolution and evaluating the reliability and cost of changes before the actual intervention takes place. The method recovers an *as is* design from C++ software releases, compares recovered designs at the class interface level, and helps the user to deal with inconsistencies by pointing out regions of code where differences are concentrated. Results as well as examples of applications to the visualization of the traceability information and to the estimation of the size of changes during maintenance are reported in the paper. Although, the maintainability aspect is out of the scope of this paper is a topic to be included in our future works.

Visaggio [17] proposes a metric for expressing the entropy of a software system and for assessing the quality of its organization from the perspective of impact analysis. The metric is called *structural information* and is based on a model dependency descriptor. The metric is characterized by its independence from the method of building the system and the architectural styles which represent it at the various levels of abstraction. It takes into account both the structure of the components at all levels of abstraction and the structure derived from the links between the different levels of abstraction. Even though it uses the concept of internal and external traceability, it may be replaced by only one direction dependency or mutual. Also the definition of internal traceability does not match with the UML definition of traceability used in this paper.

5 Conclusions and Future Work

In this paper, we have presented an approach and our initial validation to detect and measure the complexity of traceable transactions from requirements (use cases) to implementation (source code) of a target software system. With our initial experiments, we have shown that our metrics (nP and nT) are useful to evaluate if the requirements were implemented or not, and how they were implemented. Thus, we can offer the developers a trace view from requirements to code. Even when the results are promising, there are still some future work. We want to analyze if there is a correlation between nP measured in use cases and in source code. We want also to test our approach in other object-oriented languages to see if we need to adapt the methodology in other case studies.

References

1. ATM Example, `http://www.cs.gordon.edu/courses/cps122/ATMExample/index.html`
2. CLIPS, `http://clipsrules.sourceforge.net/WhatIsCLIPS.html`
3. Recoder, `http://recoder.sourceforge.net/index.html`
4. ISO/IEC. 20926:2003 IFPUG 4.1 Unadjusted functional size measurement method - Counting practices manual (2003)
5. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A.: Maintaining traceability links during object-oriented software evolution. Software Practice & Experience Journal 31, 331–355 (April 2001)
6. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1997)
7. Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering Journal 20, 476–493 (June 1994)
8. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley (2003)
9. Karner, G.: Metrics for Objectory. Diploma Thesis, University of Linköping (1993)
10. Lavazza, L., Robiolo, G.: Introducing the evaluation of complexity in functional size measurement: a uml-based approach. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 25:1–25:9. ESEM '10, ACM, New York, NY, USA (2010)
11. McCabe, T.: A Complexity Measurement. IEEE Transactions on Software Engineering Journal 2, 308–320 (1976)
12. Murray, A.T., Shahabuddin, M.: Object-Oriented Techniques applied to a Real-time Embedded, Spaceborne Application. In: Companion to the 21st ACM SIG-PLAN symposium on Object-oriented programming systems, languages, and applications. pp. 830–838. OOPSLA '06, ACM, New York, NY, USA (2006)
13. Paul, R.A., Kunii, T.L., Shinagawa, Y., Khan, M.F.: Software Metrics Knowledge and Databases for Project Management. IEEE Transactions on Knowledge Data Engineering 11(1), 255–264 (1999)

14. Pfleeger, S.L., Jeffery, R., Curtis, B., Kitchenham, B.: Status Report on Software Measurement. IEEE Software 14, 33–43 (1997)
15. Robiolo, G., Badano, C., Orosco, R.: Transactions and Paths: Two use cases based metrics which improve the early effort estimation. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. pp. 422–425. ESEM '09, IEEE Computer Society, Washington, DC, USA (2009)
16. Shepperd, M., Ince, D.: Multi-dimensional Modelling and Measurement of Software Designs. In: Proceedings of the 1990 ACM Annual Conference on Cooperation. pp. 76–81. CSC '90, ACM, New York, NY, USA (1990)
17. Visaggio, G.: Structural information as a quality metric in software systems organization. In: Proceedings of the International Conference on Software Maintenance. pp. 92–99. IEEE Computer Society, Washington, DC, USA (1997)
18. Watson, A.H., Mccabe, T.J., Wallace, D.R.: Special Publication 500-235, Structured Testing: A Software Testing Methodology using the Cyclomatic Complexity Metric (1996)