View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Szeged

Acta Cybernetica **21** (2014) 367–382.

Monitoring Evolution of Code Complexity and Magnitude of Changes

Vard Antinyan[*], Miroslaw Staron[*], Jörgen Hansson[*],
Wilhelm Meding[†], Per Österström[†], and Anders Henriksson[‡]

Abstract

Complexity management has become a crucial activity in continuous software development. While the overall perceived complexity of a product grows rather insignificantly, the small units, such as functions and files, can have noticeable complexity growth with every increment of product features. This kind of evolution triggers risks of escalating fault-proneness and deteriorating maintainability. The goal of this research was to develop a measurement system which enables effective monitoring of complexity evolution. An action research has been conducted in two large software development organizations. We have measured three complexity and two change properties of code for two large industrial products. The complexity growth has been measured for five consecutive releases of the products. Different patterns of growth have been identified and evaluated with software engineers in industry. The results show that monitoring cyclomatic complexity evolution of functions and number of revisions of files focuses the attention of designers to potentially problematic files and functions for manual assessment and improvement. A measurement system was developed at Ericsson to support the monitoring process.

Keywords: complexity, metrics, risk, lean, agile, correlation, measurement systems, code, change, revision

1 Introduction

Actively managing software complexity has become an important aspect of continuous software development. It is generally accepted that software products developed in a continuous manner are getting more and more complex over time. Evidence shows that the rising complexity drives to deteriorating quality of software [2,3]. The continuous increase of code base and growing complexity can lead to large, virtually unmaintainable source code if left unmanaged.

[*]Computer Science and Engineering, University of Gothenburg | Chalmers, E-mail: {vard.antinyan,miroslaw.staron,jörgen.hansson}@chalmers.se

[†]Ericsson, E-mail: {wilhelm.meding,per.osterstrom}@ericsson.com

[‡]Volvo Group Truck Technology, E-mail: anders.J.henriksson@volvo.com

DOI: 10.14232/actacyb.21.3.2014.6

A number of metrics have been suggested to measure various aspects of software complexity and evolution over development time [7]. Those metrics has been accompanied with a number of studies indicating how adequately the proposed metrics relate to software quality [6, 17]. Complexity and change metrics have been used extensively in recent years for assessing the maintainability and fault-proneness of software code [4]. Despite the considerable amount of research conducted for investigating the influence of complexity on software quality, little results can be found on how to effectively monitor and prevent complexity growth. Therefore a question remains:

How to monitor code complexity and changes effectively when delivering feature increments to the main code branch?

The aim of this research was to develop method and tool support for actively monitoring complexity evolution and drawing the attention of industries' software engineers to the potentially problematic trends of growing complexity. In this paper we focus on the level of self-organized software development teams who often deliver code to the main branch for further testing, integration with hardware, and ultimate deployment to end customers. We address this question by conducting a case study at two companies, which develop software according to Agile and Lean principles. The studied companies are Ericsson which develops telecom products and Volvo Group Truck Technology (GTT) which develops electronic control units (ECU) for trucks.

Our results show that using two complementary measures, McCabes cyclomatic complexity of functions and number of revisions of files supports teams in decision making, when delivering code to the main branch. The evaluation shows that monitoring trends in these measures draws attention of the self-organized Agile teams to a handful of functions and files. These functions and files are manually assessed, and the team formulates decisions before the delivery on whether they can cause problems.

2 Related Work

Continuous software evolution: A set of measures useful in the context of continuous deployment can be found in the work of Fritz [8], in the context of market driven software development. The metrics presented by Fritz measure such aspects as continuous integration as pace of delivery of features to the customers. These metrics complement the two indicators presented in this paper with business perspective which is important for product management.

The delivery strategy, which is an extension of the concept of continuous deployment, has been found as one of the three key aspects important for Agile software development organizations in a survey of 109 companies by Chow and Cao [5]. The indicator presented in this paper is a means of supporting organizations in their transition towards achieving efficient delivery processes.

Ericssons realization of the Lean principles combined with Agile development was not the only one recognized in literature. Perera and Fernando [14] presented

another approach. In their work they show the difference between the traditional and Lean-Agile way of working. Based on our observations, the measures and their trends at Ericsson were similar to those observed by Perera and Fernando.

Measurement systems: The concept of an early warning measurement system is not new in engineering. Measurement instruments are one of the cornerstones of engineering. In this paper we only consider computerized measurement systems i.e. software products used as measurement systems. The reasons for this are: the flexibility of measurement systems, the fact that we work in the software field, and similarity of the problems e.g. concept of measurement errors, automation, etc. An example of a similar measurement system is presented by Wisell [21] where the concept of using multiple measurement instruments to define a measurement system is also used. Although differing in domains of applications these measurement systems show that concepts which we adopt from the international standards (like [11]) are successfully used in other engineering disciplines. We use the existing methods from the ISO standard to develop the measurement systems for monitoring complexity evolution.

Lowler and Kitchenham [12] present a generic way of modeling measures and building more advanced measures from less complex ones. Their work is linked to the TychoMetric [15] tool. The tool is a very powerful measurement system framework, which has many advanced features not present in our framework (e.g. advanced ways of combining metrics). A similar approach to the TychoMetrics way of using metrics was presented by Garcia et al. [9]. Despite their complexity, both the TychoMetric tool and Garcias approach can be seen as alternatives in the context of advanced data presentation or advanced statistical analysis over time. Our research is a complement to [13] and [15]. We contribute by showing how the minimal set of measures can be selected and how the measurement systems can be applied regularly in large software organizations.

Meyer [10, pp. 99-122] claims that the need for customized measurement systems for teams is one of the most important aspects in the adoption of metrics at the lowest levels in the organization. Meyers claims were also supported by the requirements that the customization of measurement systems and development of new ones should be simple and efficient in order to avoid unnecessary costs in development projects. In our research we simplify the ways of developing Key Performance Indicators exemplified by a 12-step model of Parmenter [13] in the domain of software development projects.

3 Design of the Study

This case study was conducted using action research approach [1,16]. The researchers were part of the companys operations and worked directly with product development units. The role of Ericsson in the study was the development of the method and its initial evaluation, whereas the role of Volvo GTT was to evaluate the method in a new context.

3.1 Studied Organizations

Ericsson: The organization and the project within Ericsson developed large products for mobile packet core network. The number of the developers in the projects was up to a few hundreds. Projects were executed according to the principles of Agile software development and Lean production system, referred to as Streamline development within Ericsson [20]. In this environment, different development teams were responsible for larger parts of the development process compared to traditional processes: design teams, network verification and integration, testing, etc.
Volvo GTT: The organization which we worked with at Volvo GTT developed ECU software for trucks. The collaborating unit developed software for two ECUs and consisted of over 40 designers, business analysts and testers at different levels. The development process was in the transaction from traditional to Agile.

3.2 Units of Analysis

During our study we analyzed two different products software for a telecom product at Ericsson and software for two ECUs at Volvo GTT.
Ericsson: The product was a large telecommunication product composed by over two million lines of code with several tens of thousands C functions. The product had a few releases per year with a number of service releases in-between them. The product has been in development for a number of years.
Volvo GTT: The product was an embedded software system serving as one of the main computer nodes for a product line of trucks. It consisted of a few hundred thousand lines of code and several thousand C functions. The analyses that were conducted at Ericsson were replicated at Volvo GTT under the same conditions and using the same tools. The results were communicated with designers of the software product after the data was analyzed. At Ericsson the developed measurement system ran regularly whereas at Volvo the analysis was done semi-automatically, that is, running the measurement system whenever feedback was needed for designers.

3.3 Reference Group

During this study we had the opportunity to work with a reference group at Ericsson and a designer at Volvo GTT. The aim of the reference group was to support the research team with expertise in the product domain and to validate the intermediate findings as prescribed by the principles of Action research. The group interacted with researchers on a bi-weekly meeting basis for over 8 months. At Ericsson the reference group consisted of a product manager, a measurement program leader, two designers, one operational architect and one research engineer. At Volvo GTT we worked with one designer.

3.4 Measures in the Study

Table 1 presents the complexity measures, change measures and deltas of complexity measures over time. The definitions of measures and their deltas are provided also.

Table 1: Metrics and their definitions

Complexity Measures	**Abbrev.**	**Definition**
McCab's cyclomatic complexity of a function	M	The number of linearly independent paths in the control flow graph of a function, measured by calculating the number of "if", "while", "for", "switch", "break", "&&", "||" tokens
Structural $Fan-out$	$Fan-out$	The number of invocations of functions found in a specified function
Maximum Block Depth	MBD	The maximum level of nesting found in a function
Cyclomatic complexity of a file	M_f	The sum of all functions M in a file
Change Measures	**Abbrev.**	**Definition**
Number of revisions of a file	NR	The number of check-ins of files in a specified code integration branch and its all sub-branches in a specified time interval
Number of designers of a file	ND	The number of developers that do check-in of a file on a specified code integration branch and all of its sub-branches during a specified time interval
Deltas of Complexity Measures	**Abbrev.**	**Definition**
Complexity deltas of a function	ΔM, $\Delta Fan-out$, ΔMBD	The increase or decrease of M, *Fan-out* and MBD measures of a function during a specified time interval.

3.5 Research Method

According to the principles of action research we adjusted the process of our research with the operations of the company. We conducted the study according to the following pre-defined process:

- Obtain access to the source code of the products and their different releases
- Calculate complexity measures of all functions and change measures of all files in the code
- Calculate the complexity deltas of all functions through five releases of both products
- Sort the functions by complexity delta through five releases
- Identify possible patterns of complexity change
- Identify drivers for complexity changes for functions with functions having highest overall delta
- Correlate measures to explore their dependencies and select measures for monitoring complexity and changes
- Develop a measurement system (according to ISO 15939) for monitoring complexity and changes
- Monitor and evaluate the measurement system for five weeks

The overall complexity change of function is calculated by:

$$\text{Overall delta} = |\Delta M_{1-2}| + |\Delta M_{2-3}| + |\Delta M_{3-4}| + |\Delta M_{4-5}| \,.$$

$|\Delta M_{i-j}|$ is the absolute value of change of M of a function between i and j releases. Overall complexity change of Fan-out and MBD is calculated the same way.

4 Analysis and Results

In this section we explore the main scenarios of complexity evolution. We carry out correlation analysis of collected measures in order to understand their dependencies and chose measures for monitoring.

4.1 Evolution of the Studied Measures Over Time

Exploring different types of changes of complexity, we categorized changes into 5 groups.

1. Group 1 - Functions that are newly created and become complex in current release and functions that existed but disappeared in current release.
2. Group 2 - Functions that are re-implemented in current release.
3. Group 3 - Functions that have significant change of complexity between two releases due to development or maintenance.

4. Group 4 - Test functions, which are regularly generated, destroyed and re-generated for unit testing.

5. Group 5 - functions that have minor complexity changes between two releases.

Group 1 and group 5 functions were observed to be the most common. They appeared regularly in every release. Engineers of the reference group characterized their existence as expected result of software evolution. Group 2 functions were re-implementation of already existing function. The existed functions were re-implemented with different name and the old one was destroyed. After re-implementation the new functions could be named as the old one. Re-implementation usually took place when major software changes were happening: In this case re-implementation of a function sometimes could be more efficient than modification. Figure 1 shows the cyclomatic complexity evolution of top 200 functions through five releases of products. Each line on the figure represents a C function.

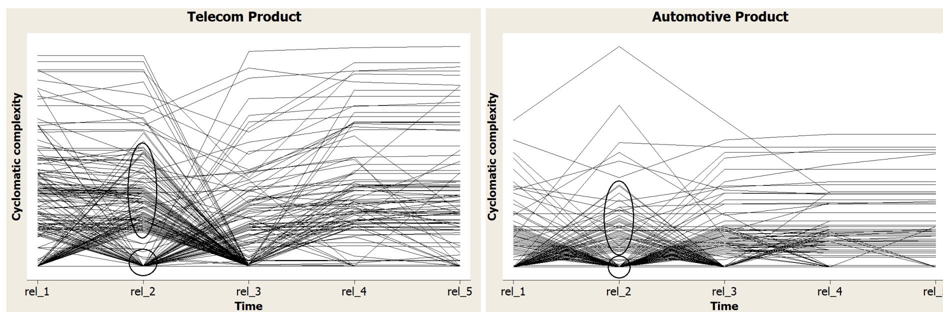

Figure 1: Evolution of M of functions

In Figure 1 re-implemented functions are outlined by elliptic and old ones by round lines. In reality the number of re-implemented functions are small (about 1%), however considering the big magnitude of complexity change of them, many of them ended-up in the top 200 functions in the picture, giving an impression that they are relatively many. Figure 2 similarly presents the evolution of Fan-out in the products. Group 3 functions are outlined by elliptic line in Figure 2.

Group 3 functions were usually designed for parsing a huge amount of data and translating them into another format. As the amount and type of data is changed the complexity of the function also changes. Finally the Group 5 functions were unit test implementations. These functions were destroyed and regenerated frequently in order to update running unit tests. Figure 3 presents the MBD evolution of products. As nesting depth of blocks can be relatively shallow, many lines in Figure 3 overlap each other thus creating an impression that there are few functions. We observed that functions in group 1, ones were created, stayed complex over time. These functions are outlined with a rectangular line in Figure 3.

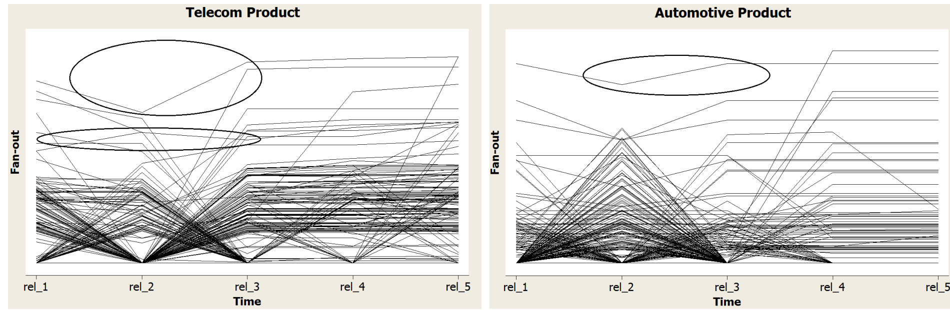

Figure 2: Evolution of Fan-out of functions

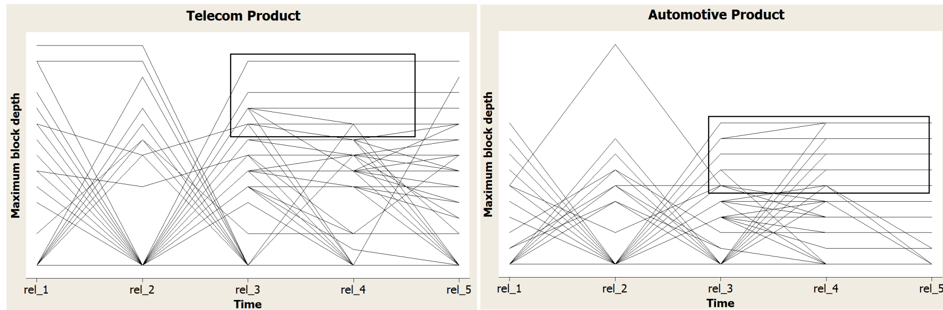

Figure 3: Evolution of MBD of functions

The statistics of functions of all groups are represented in Table 2. The table shows how all functions, that had complexity change, are distributed in groups. We would like to mention that the number of all functions in telecom product is about 65000 and in automotive product about 10000, however only top 200 functions out of those are presented in the figures. This might result in disproportional visual relationship between the relation of different groups of functions in the table and in the figures as the figures contains only top 200 functions.

Table 2: The distribution of functions with complexity delta in groups

Group	Group 1	Group 2	Group 3	Group 4	Group 5
Percentage	27%	1%	1%	1%	70%

We observed the change of complexity for both long time intervals (between releases) and for short time intervals (between weeks). Figure 4 shows how the complexity of functions changes over weeks. The initial complexity of functions is provided under column M in the figure. We can see the week numbers on the

top of the columns, and every column shows the complexity growth of functions in that particular week. Under column we can see the overall delta complexity per function that is the sum of weekly deltas per function.

File name	Function name	M	Total: ΔM	w1306	w1307	w1308	w1309	w1310	w1311	w1312
file 1	function 1	14	0	0	0	0	0	0	0	0
file 2	function 2	15	15	0	0	0	0	0	15	0
file 2	function 3	1	0	0	0	0	0	0	0	0
file 3	function 4	10	5	4	-9	11	-11	10	0	0
file 4	function 5	11	3	0	0	0	0	11	0	0
file 5	function 6	58	13	17	0	11	-11	0	0	-4
file 5	function 7	22	22	0	0	0	0	0	0	22
file 6	function 8	20	20	0	0	0	18	2	0	0
file 6	function 9	17	17	0	0	0	17	0	0	0
file 7	function 10	11	11	0	0	0	11	0	0	0
file 8	function 11	13	13	0	0	0	0	13	0	0
file 9	function 12	28	28	0	28	0	0	0	0	0
file 10	function 13	12	12	0	0	0	12	0	0	0

Figure 4: Visualizing complexity evolution of functions over weeks

The fact that the complexity of functions fluctuates irregularly was interesting for the designers, as the fluctuations indicate active modifications of functions, which might be due to new feature development or represent defect removals with multiple test-modify-test cycles. Functions 4 and 6 are such instances illustrated in Figure 4. Monitoring the complexity evolution through short time intervals we observed that very few functions are having significant complexity increase. For example in a week period the number of functions that have complexity increase $\Delta M > 10$ can vary between 5-10 while overall number of functions reaches a few tens of thousands in the product.

4.2 Correlation analyses

The correlation analyses of measures were conducted in order to eliminate dependent measures and select a minimal amount of measures for monitoring. The correlation analysis results of complexity measures for the two software products are presented in Table 3. The visual presentation of the relationship of complexity measures is presented in Figure 5. As the table illustrates there is a strong correlation between M and Fan-out for the telecom product and M and MBD for the automotive product. There is a moderate correlation between M and MBD for the telecom product. Generally designers of reference group concluded that monitoring the cyclomatic complexity among all complexity measures is good enough as there was a moderate or strong correlation between three complexity measures. M was chosen because of two reasons:

1. MBD is rather a characteristic of a block of code than a whole function. It is a good complementary measure but it cannot characterize the complexity of a whole function.

2. Fan-out seemed to be a weaker indicator of complexity than M because it rather showed the vulnerability of a function towards other functions that are in that function.

Considering aforementioned conclusions M was chosen among complexity measures to be monitored.

Table 3: Correlation of complexity measures

Telecom / **Automotive**	MBD	M
M	0.41 / **0.69**	
Fan-out	0.34 / **0.20**	0.76 / **0.26**

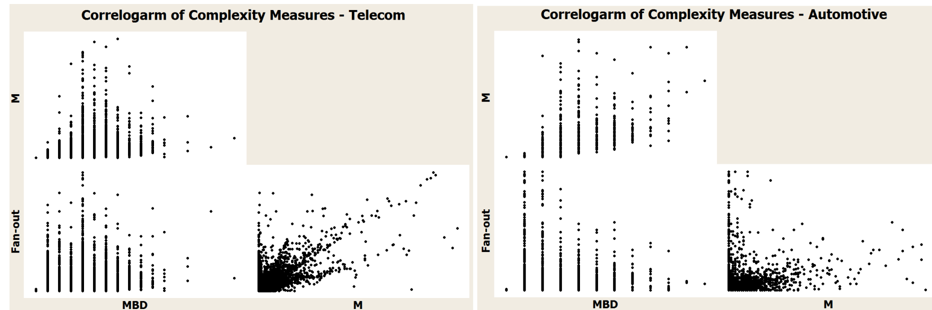

Figure 5: Correlogram of complexity measures

NR and ND are measures that indicate the magnitude of changes. Previously a few studies have shown that change metrics are good indicators of problematic areas of code, as observed Shihab [18]. The measurement entity of NR and ND is a file. Therefore in order to understand how change measures correlates to complexity we decided to define the M measure for files (Table 1). Table 4 presents the correlation analysis results for ND, NR and M_f measures.

An important observation was the strong correlation between the number of designers and the number of revisions for the telecom product (Table 4). At the beginning of this study the designers of the reference group at Ericsson believed that a developer of a file might check-in and check-out the file several times which probably is not a problem. The real problem, they thought, could be when many designers modify a file simpultaneously. Nonetheless, a strong correlation between the two measures showed that they are strongly dependent, and many revisions

is mainly caused by many designers modifying a file in a specified time interval (Figure 6).

Table 4: Correlation of change and complexity measures

Telecom / **Automotive**	M_f	ND
ND	0.40 / **0.37**	
NR	0.46 / **0.72**	0.92 / **0.41**

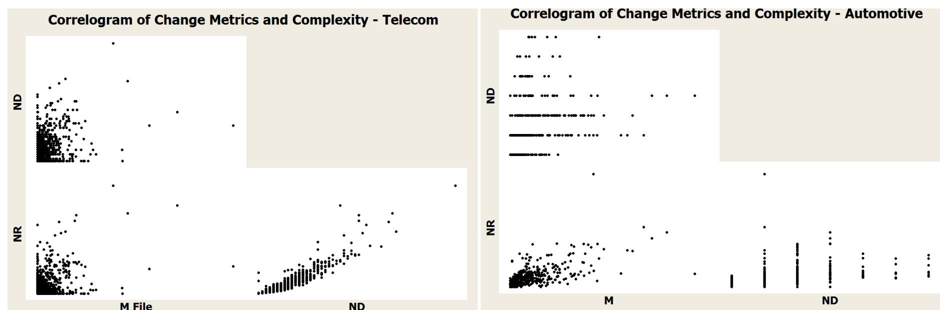

Figure 6: Correlogram of change and complexity measures

In case of automotive product correlation of ND and NR was moderate which can be due to small number of designers who have rather firmly assigned development areas and usually change the same code. Moderate correlation between M_f and NR for the telecom product indicates that complex files are prone to changes. There are always simple files that are changed often due to development.

Considering the correlation analysis results we designed a measurement system at Ericsson for monitoring code complexity and magnitude of changes over time. The description of design and application of measurement system is discussed in the next section.

4.3 Design of the Measurement System

Based on the results that we obtained from investigation of complexity evolution and correlation analyses, we designed two indicators based on M and NR measures. These indicators capture the increase of functions complexity and highlight the files with highest change magnitude over time. These indicators were designed according to ISO/IEC 15959. The design of complexity indicator is presented in Table 5. The other indicator based on NR is defined in the same way: the files that had NR > 20 during last week development time period should be identified and reviewed. The measurement system was provided as a gadget with the necessary information updated on a weekly basis (Figure 7). The measurement system relies on a previous

study carried out at Ericsson [19]. For instance the total number of files with more than 20 revisions since last week is 5 (Figure 7). The gadget provides the link to the source file where the designers can find the list of files or functions and the color-coded tables with details.

We visualized the NR and Δ M measures using tables as depicted in Figure 4. As in Streamline development the development team merged builds to the main code branch in every week it was important for the team to be notified about functions with drastically increased complexity (over 20).

Table 5: Measurement system design based on ISO/IEC 15939 standard

Information Need	Monitor cyclomatic complexity evolution over development time
Measurable Concept	Complexity change of delivered source code
Entity	Source code function
Attribute	Complexity of C functions
Base Measures	Cyclomatic complexity number of C functions M
Measurement Method	Count cyclomatic number per C function according to the algorithm in CCCC tool
Type of measurement method	Objective
Scale	Positive integers
Unit of measurement	Execution paths over the C/C++ function
Derived Measure	The growth of cyclomatic complexity number of a C function in one week development time period
Measurement Function	Subtract old cyclomatic number of a function from new one: $\Delta M = M(week_i)M(week_{i-1})$
Indicator	Complexity growth: The number of functions that exceeded McCabe complexity of 20 during the last week
Model	Calculate the number of functions that exceeded cyclomatic number 20 during last week development period
Decision Criteria	If there are functions that exceeded M number 20 then software designers should review these functions refactor if necessary

5 Threats to Validity

In this paper we evaluate the validity of our results based on the framework described by Wohlin et al. [22]. The framework is recommended for empirical studies in software engineering.

The main external validity threat is the fact that our results come for an action research. However, since two companies from different domains (telecom and

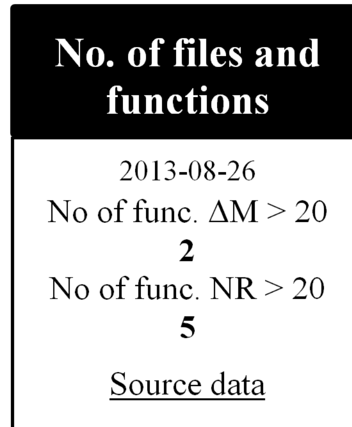

Figure 7: Information product for monitoring M and NR metrics over time

automotive) were involved, we believe that the results can be generalized to more contexts than just one specific type of software development.

The main internal validity threat is related to the construct of the study and the products. In order to minimize the risk of making mistakes in data collection we communicated the results with reference groups at both companies to validate them.

The limit 20 for cyclomatic number established as a threshold in this study does not have any firm empirical or theoretical support. It is rather an agreement of developers of large software systems. We suggest that this threshold can vary from product to product. The number 20 is a preliminary established number taking into account the number of functions that can be handled on weekly basis by developers.

The main construct validity threats are related to how we identify the names of functions for comparing their complexity numbers over time. There are several issues emerging in this operation. Namely, what happens if a function has changed its list of arguments or what happens if a function is moved to another file? Should this be regarded as the same function before and after changing the list of arguments or the position? We disregarded the change of argument list however this can be argued.

Finally the main threat to conclusion validity is the fact that we do not use inferential statistics to monitor relation between the code characteristics and project properties, e.g. number of defects. This was attempted during the study but the data in defect reports could not be mapped to individual files. This might be a thread for jeopardizing the reliability of such an analysis. Therefore we chose to rely on the most skilled designers perception of how fault-prone and unmaintainable the delivered code is.

6 Conclusions

In continuous software development quick feedbacks on developed code complexity is crucial. With small software increments there is a risk that the complexity of units of code can grow to an unmanageable level. In this paper we explored how complexity evolves, by studying two software products one telecom product at Ericsson and one automotive product at Volvo GTT. We identified that in short periods of time a few out of tens of thousands functions have significant complexity increase. We also concluded that the self-organized teams should be able to make the final assessment whether the potentially problematic is indeed problematic.

By analyzing correlations between three complexity and two change metrics we concluded that it is enough to use two measures, McCabe complexity and number of revisions, to draw attention of the teams to potentially problematic code for review and improvement.

The automated support for the teams was provided in form of a MS Sidebar gadget with the indicators and links to statistics and trends with detailed complexity development data. The measurement system was evaluated by using it on an ongoing project and communicating the results with software engineers in industry.

In our further work we intend to study how the teams formulate the decisions and monitor their implementation.

Acknowledgment

The authors thank the companies for their support in the study. This research has been carried out in the Software Centre, Chalmers, University of Gothenburg and Ericsson, Volvo Group Truck Technology.

References

[1] Baskerville, R.L. A Critical Perspective on Action Research as a Method for Information Systems Research. Journal of Information Technology, 1996(11), 235-246.

[2] Boehm, B. A view of 20th and 21st century software engineering. Paper presented at the Proceedings of the 28th international conference on Software engineering, 2006.

[3] Bosch, Jan. From integration to composition: On the impact of software product lines, global development and ecosystems. Journal of Systems and Software, 83(1), 67-76. doi: http://dx.doi.org/10.1016/j.jss.2009.06.051

[4] Catal, Cagatay. A systematic review of software fault prediction studies. Expert Systems with Applications, 36(4), 7346-7354. doi: http://dx.doi.org/10.1016/j.eswa.2008.10.027

[5] Chow, Tsun. A survey study of critical success factors in agile software projects. Journal of Systems and Software, 2008, 81(6), 961-971.

[6] Fenton, Norman E. A critique of software defect prediction models. Software Engineering, IEEE Transactions on, 1999, 25(5), 675-689.

[7] Fenton, Norman E. Software metrics (Vol. 1): Chapman and Hall London, 1991.

[8] Fitz, Timothy. Continuous Deployment at IMVU: Doing the impossible fifty times a day. from http://timothyfitz.wordpress.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/

[9] Garcia, F. Managing Software Process Measurement: A Meta-model Based Approach. Information Sciences, 2007, 177(2), 2570-2586.

[10] Harvard Business School. Harvard business review on measuring corporate performance. Boston, MA: Harvard Business School Press, 1998.

[11] International Bureau of Weights and Measures. International vocabulary of basic and general terms in metrology = Vocabulaire international des termes fondamentaux et gnraux de mtrologie (2nd ed.). Genve, Switzerland: International Organization for Standardization, 1993.

[12] Lawler, J. Measurement modeling technology. IEEE Software, 2003, 20(3), 68-75.

[13] Parmenter, David. Key performance indicators : developing, implementing, and using winning KPIs. Hoboken, N.J.: John Wiley and Sons, 2003

[14] Perera, G. I. U. S. Enhanced agile software development - hybrid paradigm with LEAN practice. Paper presented at the International Conference on Industrial and Information Systems (ICIIS), 2007.

[15] Predicate Logic. TychoMetrics. Retrieved 2008-06-30, 2008, from http://www.predicatelogic.com

[16] Sandberg, Anna. Agile Collaborative Research: Action Principles for Industry-Academia Collaboration. IEEE Software, 2011, 28(4), 74-83.

[17] Shepperd, Martin. A critique of cyclomatic complexity as a software metric. Software Engineering Journal, 3(2), 30-36.

[18] Shihab, Emad. An industrial study on the risk of software changes. Paper presented at the Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012.

[19] Developing measurement systems: an industrial case study. Journal of Software Maintenance and Evolution: Research and Practice, 23(2), 89-107. doi: 10.1002/smr.470

[20] Tomaszewski, Piotr. From Traditional to Streamline Development - Opportunities and Challenges. Software Process Improvement and Practice, 2007(1), 1-20. doi: 10.1002/spip.355

[21] Wisell, David. Considerations when Designing and Using Virtual Instruments as Building Blocks in Flexible Measurement System Solutions. Paper presented at the IEEE Instrumentation and Measurement Technology Conference, 2007.

[22] Wohlin, Claes. Experimentation in Software Engineering: An Introduction. Boston MA: Kluwer Academic Publisher, 2000.