Effective Complexity Management by Investigating Trade-offs between Several Aspects of Software Complexity

Vard Antinyan¹), Miroslaw Staron¹), Erik Wikström²), Jesper Derehag²), Wilhelm Meding²), Anders Henriksson³), Mattias Rusten³), Jörgen Hansson¹)

¹Department of Computer Science and Engineering, Chalmers | University of Gothenburg, Sweden
²Ericsson, Sweden, ³AB Volvo, Sweden
Hörselgången 11,
417 56 Gothenburg, Sweden
vard.antinyan@cse.gu.se miroslaw.staron@cse.gu.se

ABSTRACT
The complexity management of software code has become one of the major problems in software development industry. With growing complexity the maintenance effort of software code increases. Moreover, multiple aspects of complexity create difficulties for optimal complexity management. The objective of this study is to investigate the relationships of several aspects of code complexity and assess the possibilities of optimizing overall code complexity. We conducted an action research project in two software development companies and complemented it with a study of three open source products. Four complexity metrics were measured, and their nature and mutual influence were investigated using statistical methods. The results and possible explanations were discussed with software engineers in industry. The results showed that there are two distinguishable aspects of complexity of source code functions; internal and external complexities. Those have an inverse relationship. Moreover, the product of them cannot be greater than a certain limit, regardless of software size. We developed a method that permits identification of most complex functions considering the two aspects of complexities. The evaluation showed that the use of the method is effective in industry; it enables identification of few (0.5%) most complex functions out of thousands of functions for manual assessment and reengineering.

Categories and Subject Descriptors

General Terms
Design, Management, Measurement

Keywords
Complexity, analysis, trade-off, correlation, maintenance

1. INTRODUCTION
The effect of complexity on software maintenance and fault-proneness has been studied extensively in recent years. It is generally accepted that the growing complexity has a direct impact on maintainability and fault proneness of developed software. Many studies have observed that growing complexity triggers decreasing maintainability and escalating fault proneness of software [1-3]. Banker, et al. [4] was one of the first studies that observed a direct positive relationship of software code complexity and maintenance effort. Later, in order to neutralize the effect of size in investigation of complexity-maintainability relationship, relative complexity measures were applied; researchers showed that in case of similar size more complex code is significantly more hard-to-maintain and fault-prone [5, 6]. For quantification of different aspects of complexity many measures have been developed [7], among which the most used ones are; McCabe’s cyclomatic complexity, Halstead’s metrics, Chidamber’s and Kemerer’s coupling metrics, depth of nesting, structural fan-in and fan-out etc. In recent years, these metrics have been used extensively to develop models for assessing the maintainability and fault-proneness of software code [8, 9]. Despite the success of those models the industrial practitioners are still asking how to reduce code complexity. There is one particular question, posed often: When reducing one aspect of code complexity, how it affects other aspects of complexity? Reformulation this question from complexity management perspective we get:

How to manage code complexity considering trade-offs between several aspects of complexity?

We consider that there is a trade-off between two types of complexities if there is evidence that complexities of both types cannot increase simultaneously for a source code function. In other words, from certain complexity level, the increase of one type of complexity might necessitate the reduction of the other type of complexity. In this paper two distinct aspects of complexity of functions are delineated; internal and external complexities, which have a hyperbolic relation. Furthermore, the more complex the functions become, the more obvious this relation is. We created a method that assesses the complexity of functions and identifies the most complex ones, considering the two aspects of complexity. A preliminary evaluation at Ericsson and Volvo showed that the method supports software engineers in industry for effective complexity management.

The remainder of this paper is divided into seven sections: First we introduce the motivation of this study, next we describe the design of the study, then we proceed with reporting results. In section five and six we document the complexity management method and possible threats to its validity. Finally we present the related work and conclude the paper.
2. MOTIVATION OF THE STUDY
This research has been initiated based on a practical issue that software engineers of industry encountered, when using our previously developed method for software reengineering. In our previous research we had developed a method and supporting measurement system that identifies fault-prone and hard-to-maintain files in continuous development [10]. The tool supported software engineers to focus on very few files out of thousands for refactoring. The method was relying on a combination of two metrics – cyclomatic complexity and changes of source code. During the use of measurement system (about seven months) the engineers of software development teams at Ericsson and at Volvo GTT posed a question regarding refactoring of software. The question was: What happens if the engineers reduce the cyclomatic complexity of a source code function by splitting it? Some engineers argued that while decomposing a complex function into several functions the dependencies between functions will increase, and a new type of complexity might be introduced to the functions.

In order to better understand the complexity from practical point of view and answer the above question (refactoring and redesigning of source code) we initiated the research presented in this paper. Particularly we were interested if there are trade-offs between different aspects of complexity, and if there are, how complexity can be managed by considering these trade-offs. These two considerations motivated us to emphasize the importance of the unit for which the complexity is defined, because complexity management activities are applied on specific units of code, such as functions.

3. STUDY DESIGN
Action research has been applied to conduct this study. We adhered to action research principles described in [11, 12] which means that the research process has been carried out by close collaboration of industrial professionals and researchers.

3.1 Industrial Context
Ericsson and Volvo Group Truck Technology (GTT) have been involved in this research. The organization at Ericsson developed a software product for mobile packet core network. It comprised several hundreds of developers and several cross-functional development teams situated in different geographical positions. The organization at Volvo comprised several hundreds of developers who develop tens of Electronic Control Units (ECU) for trucks of different brands. Our collaboration unit developed two specific ECUs.

3.2 Target Software Products
While the research was conducted by the request of two organizations, we also included three open source product in our research, thus making target number of products to be analyzed being five. Different sizes, domains and development strategies of software products permitted to obtain more generalizable results. The brief description of studied products is as follows:

Telecom software was developed by Ericsson. It provides an embedded software system for a packet core network. The software consisted of over two million lines of code.

Automotive software was developed by Volvo GTT. It provides an embedded software system for two ECUs of a Volvo truck. The software consisted of over 200 000 lines of code.

Mozilla Firefox was free open source software developed by Mozilla Corporation. It provides free web browser for Windows, Linux and other operating systems. The software consisted of over three million lines of code.

LibreOffice was free open source software developed by The Document Foundation. It provides office suite for word processing, drawings, slideshows etc. The software consisted of 3.5 million lines of code.

Gimp was free software developed by a group of volunteers. It provides graphic editor for image manipulations. The software consisted of over 300 000 lines of code.

All software products were developed partly by C and partly by C++ languages. All products have been in development for several years.

3.3 Collected Measures
In order to assess different aspects of software code complexity we have collected four basic complexity measures and seven derived measures [13] (normalizations of measures are not considered as new measures). The evolution of measures is also counted between two different releases of products. The four basic measures have been selected by considering the credibility of how well they are recognized as software complexity measures in literature and in practice [14, 15]. These measures are McCabes cyclomatic complexity, structural fan-out, structural fan-in and maximum nesting level of a function. The other seven measures have been developed during the study based on basic measures. Source code function has been selected as a unit of measurement; this unit is chosen by software engineers of industry and researchers as everyone agreed that the essential building block and the subject of interest in source code is the function. The measures and their definitions are presented in Table 1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural fan-in (basic measure)</td>
<td>Fin</td>
<td>The number of invocations of a specified function found in entire source code</td>
</tr>
<tr>
<td>Structural fan-out (basic measure)</td>
<td>Fout</td>
<td>The number of invocations of functions found in a specified function</td>
</tr>
<tr>
<td>Maximum block depth (basic measure)</td>
<td>MBD</td>
<td>The maximum level of nesting found in a function</td>
</tr>
<tr>
<td>Internal complexity</td>
<td>C(I)</td>
<td>The sum of normalized M and Fout</td>
</tr>
<tr>
<td>External</td>
<td>C(E)</td>
<td>Identical as Fin; C(E) = Fin</td>
</tr>
</tbody>
</table>

Table 1. Measures and Definitions
The source codes of all five software systems have been used for our measurements. The measurements at companies were done by acquiring access to source code repositories of organizations. We have done the measurements on two releases of all software products. The time interval between releases has been chosen to be approximately two years. The length of the time interval between two releases has been chosen considering the precondition that a significant development time is required for observing complexity evolution of overall product.

Two focus groups have been formed [16] at two companies to support the researchers with their insights and expertise when evaluating newly obtained results. The software engineers of focus group at Ericsson were an operational architect and two design architects/designers. The focus group at Volvo was formed by a design architect, a designer, a tester and a line manager. Both formal meetings which took place on about monthly basis and informal discussions have been main means of communications. During these meetings we have shared the intermediate results and clarify the proceeded research direction.

The communications in two companies have been carried out in a similar manner. The following steps have been applied to address the research question in the five phases of action research [11]:

1. **Diagnosing**: Formulate the research question
2. **Action design**: Discuss the four basic complexity measures with software engineers in industry and determining what aspects of code complexity these measures reveal (use focus group)
3. **Action design**: Define main (two) complexity forms, internal and external complexity, for a source code function
4. **Action design**: Develop correlograms of four basic complexity measures to visually observe their relationship

<table>
<thead>
<tr>
<th>complexity</th>
<th>Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delta cyclomatic complexity</td>
<td>$\Delta M$</td>
</tr>
<tr>
<td>Delta structural fan-in</td>
<td>$\Delta \text{Fin}$</td>
</tr>
<tr>
<td>Delta structural fan-out</td>
<td>$\Delta \text{Fout}$</td>
</tr>
<tr>
<td>Delta maximum block depth</td>
<td>$\Delta \text{MBD}$</td>
</tr>
<tr>
<td>Normalized Measures</td>
<td>e.g. $M_n$, $C_n(I)$</td>
</tr>
<tr>
<td>Trade-off score</td>
<td>$\text{TS}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Measures</th>
<th>Normalization of measures in $[0, 1]$ interval</th>
</tr>
</thead>
</table>

3.4 **Research Method**

The source codes of all five software systems have been used for our measurements. The measurements at companies were done by acquiring access to source code repositories of organizations. The source code of open source products has been obtained by publicly available repositories. We have done the measurements on two releases of all software products. The time interval between releases has been chosen to be approximately two years. The length of the time interval between two releases has been chosen considering the precondition that a significant development time is required for observing complexity evolution of overall product.

Two focus groups have been formed [16] at two companies to support the researchers with their insights and expertise when evaluating newly obtained results. The software engineers of focus group at Ericsson were an operational architect and two design architects/designers. The focus group at Volvo was formed by a design architect, a designer, a tester and a line manager. Both formal meetings which took place on about monthly basis and informal discussions have been main means of communications. During these meetings we have shared the intermediate results and clarify the proceeded research direction.

The communications in two companies have been carried out in a similar manner. The following steps have been applied to address the research question in the five phases of action research [11]:

1. **Diagnosing**: Formulate the research question
2. **Action design**: Discuss the four basic complexity measures with software engineers in industry and determining what aspects of code complexity these measures reveal (use focus group)
3. **Action design**: Define main (two) complexity forms, internal and external complexity, for a source code function
4. **Action design**: Develop correlograms of four basic complexity measures to visually observe their relationship

5. **Action design**: Calculate external and internal complexities based on basic complexity measures. Basic measures are normalized in $[0, 1]$ interval as a precaution for gaining an equivalent effect of them in their combination
6. **Action design**: Suggest a hypothesis that for complex functions these two complexity forms might be in an inverse relationship, indicating an existing trade-off between them.
7. **Action design**: Propose that the trade-off line equation of internal and external complexity is hyperbola
8. **Action design**: Conduct nonlinear regression and a test of significance to determine if there is truly a hyperbolic relationship between the two types of complexity for complex functions.
9. **Action design**: Measure the evolution of four complexity measures in order to observe the mutual influence of complexity measures, when they change simultaneously.
10. **Action design**: Define two thresholds of complexity trade-off score (Table 1) for functions and indicating the possible highest limit of complexity that a function can have
11. **Action taking**: Provide functions with highest TS number to software engineers of industry to manually assess and refactor the functions if necessary
12. **Evaluating**: Assess the effectiveness of how good indicator of complexity TS number is (focus group)
13. **Specifying learning**: Calibrate results and diagnose the next problem (future research)

4. **RESULTS**

Action research principles lead us to obtain results, discuss it with focus group members, and clarify research direction. For this reason the obtained results and our conclusions with focus groups are reported in a chronological order. We believed that it will make this section comprehensive for reader.

4.1 **Delineating Two Types of Complexity**

To start with, we discussed the essence of four basic complexity measures with software engineers at Ericsson and Volvo GTT. The main idea was to understand what these measures show and what they mean for software engineers from development perspective. This knowledge was important from semantic point of view when analyzing the data with statistical operations.

Generally McCabe’s complexity and Fout were regarded as measures that introduce complexity to a function and make them difficult to understand and maintain. Big number of M basically means that there are many conditions to be recalled in a function, when maintaining or understanding it. McCabe defined a limit of ten for M number, which latter was extended to even bigger numbers as thresholds. However, there are two major factors that can influence on establishing a threshold for McCabe complexity number: One is that the source code function can be composed by many different independent code
blocks, which make it significantly easy to understand what that function does, irrespective the overall number of condition statements in a function. This was agreed by all engineers of both focus groups unanimously. The second factor is that if there is a need for recalling a number of conditions (items) at ones, the human memory is retrieving it with groups and associations Miller [17]. These means that the mind is sorting different patterns into meaningful groups and enhancing the effectiveness of recall process. For instance if there are five similar “if” statements and ten similar “case” statements, human memory most likely will sort them into two groups thus remembering two different things instead of 15. This effect is the same for Fout measure also; calling five different functions 45 times in a function will require for a software designer to recall five groups of things. Of course, if the arguments of functions are different, things will become more difficult but the number of groups will not change.

According to designers the maximum nesting level of a function is a strong indicator of complexity. The reason is that a deeply nested block obliges a designer to remember all pre-written conditions inside the block in order to understand the current statement. Surely, in a large nested block, sometimes there are similar patterns, which make the recall of them easier, but generally it was agreed that deeply nested blocks are one thing that should be avoided without further consideration, because they are one of main reasons for emerging bugs. One problem though, with MBD as a complexity measure, is that it does not characterize the complexity of a whole function because a function can have many blocks, and their different nesting levels cannot be effectively merged into one number. However the MBD is a strong indicator of complexity which we wanted to include in our analysis to understand its relation with other measures. The above discussed three measures are indicators of internal complexity aspects of a function. They all indicate how the elements (statements, conditions, called functions) of functions interoperate inside the functions for completing a particular purpose. Therefore we call these three measures internal complexity measures of a function.

Fin is a measure which shows the extent that a given function makes other parts of code complex. From a given function’s perspective Fin does not introduce any kind of complexity to that function. But it can make other parts of code volatile to faults if it is not designed properly. High Fin in a given function means that in order to make changes in that function a designer needs to understand how several callers are using that function, to be able to determine the effects of changes on the product. The bigger Fin is the greater the dependencies of other functions are to the given function. Conventionally we call Fin external complexity measure of a function.

4.2 Visualization of Measures’ Dependencies

Subsequently, we developed correlograms of complexity measures for all five software products in order to understand how the four complexity measures relate to each other.

Figure 1 presents five correlograms of measures for all products. In the figure we can see that for all products, the scatterplots of M and Fout have an emphasized tail which shows a number of crowded functions along with vertical (Fout) axis. We have outlined them with elliptic lines.

Figure 1 Correlograms of complexity measures of functions for five software products
These functions are mostly executing unit tests. In the same scatterplots we can see a few other functions that have significantly bigger M and Fout numbers then other functions. These are outlined with round circles. These functions are generated state machines. For example in case of Mozilla the outlined two functions are building hierarchical trees, in case of Ericsson a function is reading and translating vast amount of signals etc. These state machines have relatively high M and Fout numbers for all products but are much simpler than the complexity measurements indicate; they are composed by many “switch-case” and simple “if” statements. The rest of the data for M and Fout is scattered irregularly showing no specific pattern. The scatterplots of MBD with other measures do not have specific pattern either. The most interesting scatterplots are probably the scatterplots of Fin with other complexity measures. As we can see in the figure three bottommost scatterplots of all correlograms have a hyperbolic shape, clearly indicating an inverse relationship between Fin and other measures (in Figure 1 outlined by a rectangular line for Gimp’s correlogram). Functions having bigger external complexity (Fin) tend to have smaller internal complexity (M, Fout, MBD), and functions with bigger internal complexity tend to have smaller external complexity. The explanation of this is simple, yet very important: Functions, which have significant internal and external complexities at the same time, are hard to manage and maintain. The software engineers of focus groups claim that external complexities at the same time, are hard to manage and maintain. The software engineers of focus groups claim that there are two main aspects of difficulty with this kind of functions:

- It takes significant amount of time to understand how the modified complex function can affect on all its callers
- The functions that have significant internal and extended complexity sometimes can create a chain of changes in the system which dramatically slows down the maintenance activities.

For functions that are complex generally, there was a clear trade-off line between internal and external complexities that we quantify in the next section.

### 4.3 Complexity Distribution

In order to observe the trade-off between internal and external complexities we decided to combine M and Fout in one number to express better the internal complexity of functions. In our previous research [10] we showed that there is not a strong correlation between M and Fout which means their combination will provide a better internal complexity indicator. We decided not to include MBD measure because as explained before it is not a complexity characteristic of a whole function but rather a block which might cause inaccuracies in developing a combined measure. We normalized M and Fout of functions in $[0, 1]$ interval in order to balance their influence in their combination. Denoting the combined internal complexity measure by $C(I)$ we calculate it as:

$$C(I) = M_n + Fout_n \quad \text{Eq. 1}$$

We consider that the weights of M and Fout are approximately equal therefore simple sum of them is chosen. Conventionally we denote extended complexity by $C(E)$ which is the same as Fin:

$$C(E) = Fin \quad \text{Eq. 2}$$

By initial observation $C(I)$ and $C(E)$ have inverse relationship therefore we test the significance of this relationship by denoting:

$$TS = C_d(I) * C_d(E) \quad \text{Eq. 3}$$

where TS is called a trade-off score. The value of TS was calculated as product of normalized internal and external complexity numbers. Functions having high TS number are considered complex, because TS indicates the joint magnitude of three complexity measures (M, Fout, and Fin). According to our hypothesis the internal and external complexities have hyperbolic relationship for complex functions. We chose top 300 functions with highest TS number and developed a fitted line (Figure 2) for all five software. The coefficient of determination for data points (functions) fitting hyperbola was $R^2 \in [0.85, 0.98]$ for five software products. Similarly we chose the next groups of functions that had the highest TS score (excluding the first group that was already chosen) and repeated the test. We identified that for approximately 5% of all functions with highest TS numbers the hyperbola is the line that characterize the relationship of Internal and External Complexity. This observation was true for all five products irrespective their size. We observed that $R^2$ dropped sharply with decreasing TS number of functions. Thus the higher the TS was the better hyperbola fitted to scatterplots of subgroups.

![Figure 2 Fitted line plot of Internal and External complexities for top 300 functions with highest TS numbers](image)

We concluded that from certain TS level both complexity measures can no longer increase together in a function, otherwise the function becomes too hard-to-maintain. From semantic point of view, a function that is fulfilling multiple tasks cannot be reused many times because every reuse might require a slight change which is difficult to adjust in a complex function.

Next we observed the original complexity numbers of top functions with highest TS number. Two focal observations were made for this study:

1. 0.20% of all functions in each products with highest TS number had similar M, Fout, and Fin numbers across all products irrespective product sizes.
2. The number of functions having high TS numbers was bigger for bigger products.

We would like to remind the reader that the sizes of investigated products range from 200 000 lines of code to 3.5 million lines of code, and despite this tremendous difference the complexity...
numbers for top functions with highest TS numbers were approximately the same for every software product. This observation suggests that no matter how much the overall size and complexity of software increase, there is a limit to how complex its functions (units) can become. By approximation the actual complexity numbers in this limit, on the equilibrium of M, Fout and Fin was \( M = 80, Fout = 80 \) and \( Fin = 80 \).

### 4.4 Evolution of Complexity

To assure that there is a possible limit for complexity we measured the complexity evolution over long time. We measured deltas of four complexity measures between two years for all five products. The intention was that for older versions of products, with given established TS limit, we could observe if there have been functions that exceeded that limit after two years of development period. Figure 3 shows the correlogram of deltas for complexity measures of LibreOffice. Every dot represents a function. The vertical and horizontal axes are drawn on zero points to show the position of functions with deltas. For example if a function is in upper right quarter of the scatterplot of \( \Delta M \) and \( \Delta MBD \) it means that these measures had positive delta together.

![Figure 3 Correlogram of \( \Delta \) complexity measures for LibreOffice](image)

As it can be seen on the diagram of Figure 3, there are functions in all quarters of all scatterplots (and also along with axes). This means that there have been all types of changes between two versions of software products: M, Fout, Fin and MBD have been growing together, decreasing together, growing independently or decreasing independently.

We investigated if external and internal complexity measures have increased together for the functions that have highest TS in older versions of product. The Figure 4 presents a correlogram of deltas of complexity measures for top 500 functions with highest TS. This correlogram is developed for LibreOffice but the correlograms for other software are the same. The bottommost scatterplots clearly show that there has been no simultaneous growth of Fin and other measures. Upper right quarters of all scatterplots are almost empty, indicating that no positive delta exists for top complex functions (emphasized by circles on Figure 4). Most of the changes are decreases of complexity.

By this simple visualization it was shown that there is indeed a complexity limit for functions which previously were articulated by engineers. This limit is not exceeded as it makes functions unmanageable. At this point we concluded that when considering code complexity management, the unit of code that complexity is defined for, is very important, as software designers are working rather with specific units of code than the whole code of product itself. It is worth to mention that when product is constantly growing by size, naturally the overall number and dependencies of functions increase constantly, nonetheless, as our results show, the complexity of a specific function in practice cannot increase perpetually. Above statement might imply that no matter how big software is, if the complexity of its units (functions) is kept in a manageable level then in practice the software can grow even bigger. However this view can be challenged if code units of other abstraction level are also considered, such as software modules and components. Keeping the complexity of functions in a manageable level might not be enough for overall complexity management and additional analysis of complexities of different units might be needed.

### 4.5 Complexity Trade-off

For effective complexity management we determined two thresholds which were used for identifying hard-to-maintain functions. Figure 5 presents two different hyperbolas; first bold hyperbola separates 0.5% of all functions with highest TS score. The 0.5% threshold is established by evaluating the list of top complex functions with software engineers of focus groups as well as considering the amount of functions that can be manually assessed and refactored (if needed) in a release time frame.

![Figure 5 Filtering functions with highest TS number](image)
These functions are depicted as round dark circles in the figure. We call the first hyperbola optimal limit of complexity trade-off. The second hyperbola with dashes is called tolerable limit of complexity trade-off. We consider that there is very little likelihood for functions to be beyond the tolerable limit. The functions that are between optimal limit and tolerable limit are called difficult functions. Considering 0.5% of all functions with highest TS number, by approximation, on the equilibrium of M, Fout and Fin, the optimal line was passing approximately through M = 23, Fin = 23, Fout = 23 points for all products (notice that these are actual and not normalized numbers of complexity).

In order to evaluate if difficult functions are really difficult to maintain we communicated the results with software engineers in industry. The evaluation results are presented in the next section.

4.6 Evaluation and Calibration of Results

The difficult functions at Ericsson and Volvo GTT were evaluated with engineers of focus groups. The evaluation showed that 80% of difficult functions are hard to maintain due to the nature of their two sides of complexity. Difficult functions were not necessarily the most complex functions considering only internal complexity. However, due to large number of other functions which were dependent on them made them hard-to-maintain. 15% of difficult functions were easy to maintain because of two reasons:

1. Certain functions were framework functions doing standard small operations. Those have low M number, M<5 but the high Fin of these functions caused them end up in the group of difficult functions.

2. A few functions were state machines with high M and Fout numbers and Fin~2 numbers. These functions were not hard-to-maintain also.

Briefly stating, a functions having very small C(I) could not be regarded as difficult function irrespective how big C(E) was, and a function having small C(E) could not be regarded as difficult functions irrespective how big C(I) was. Therefore we excluded functions that have either very small internal complexity (M & Fout < 5) or very small external complexity (Fin < 5).

The remaining 5% of difficult functions were functions which contained a large number of ‘switch’ and ‘case’ statements and large number of calls of the same function. These functions, despite big number of M or Fout were easy-to-maintain due to their simplicity. This observation indicates that the use of more sophisticated internal complexity measure can enhance the results obtained from Eq. 3.

The investigation showed that the group of difficult functions in open source products has similar patterns; simple framework functions and state machines with high TS number were identified. We concluded that in order to automatically exclude these few simple functions from analysis we need to apply more sophisticated complexity measures which was not feasible in this study because of time constraints. However, considering that the number of exceptions was we concluded that the method is good enough to be used.

Generally the results showed that when developing software it is important to have tolerable complexity for developed units which in our case were functions. It is very important to notice that external and internal complexities did not have a causal relationship; for example the decrease of internal complexity does not necessarily mean the increase of the external complexity. In fact, all functions that are not in the list of difficult functions have low internal and external complexities. This indicates that there is always a way of achieving low complexity on a unit (function) level.

5. A METHOD FOR CODE COMPLEXITY MANAGEMENT

Generalizing the obtained results we delineate a method for managing complexity in software code development. The necessary steps for applying the method are as follows:

1. Calculate the values of M and Fout as internal complexity, and Fin as external complexity for all functions

2. Normalize M, Fout and Fin complexity values in any non-negative interval, say [0, 1]

3. Calculate the internal complexity C(I) value for all functions, by summing normalized M and normalized Fout.

4. Calculate the trade-off score (TS) by multiplying internal and external complexity values for all functions.

5. Sort all functions in respect with the calculated TS number with descending order

6. Select 0.5% of all functions that have highest TS

7. Review the selected functions for refactoring needs.

The aforementioned six steps are executed during our research for five large software products as well as the seventh step for automotive and telecom products. The results have been shown to be greatly similar. Apart from this, in order to simplify the method for its industrial application, we created a simpler approximation of this method which was applied at Ericsson. In order to avoid additional steps of normalization of data we propose instead of normalizing the data calculate internal complexity as:

\[ C(I) = 3M + Fout \]  
Eq. 4

This simplification emanates from the fact that the mean value of Fout is about three times greater than mean value of M for all five products. Subsequently the trade-off score can be calculated as:

\[ TS = C(I)\times C(E) \]  
Eq. 5

After having the TS for all functions, 0.5% of functions with highest TS should be chosen and manually evaluated for possible refactoring.

6. THREATS TO VALIDITY

The validity evaluation of this research relies on a study of Baskerville and Wood-Harper [12] who regard the data validity to be a major problem in action research. While data collection and analysis can be considered “safe” in the phase1-2 in this action research, there are considerable validity concerns in phase 3-4. In Phases 1-2 the data is collected by standard tools, and in order to obtain more generalizable results three open source
products (cases) are also investigated. The use of rigorous tools and statistical techniques, and uniformity of the results for all five products secure the validity of conclusions that we had in the end of phase 2. However, the interpretive nature of results creates a tangible validity threat, when making conclusions of phase 3-4 (action taking). The problem is that combination of complexity metrics is not a well-known concept for the engineers of focus group therefore they interpret complexity numbers and their meanings in their own way while evaluating. If we, researchers explain the concepts to them before the evaluation then we intervene the evaluation by our own interpretations. In this study we did not intervene in the “action taking” and “evaluation” phases relying on the fact that all focus group members are familiar to the complexity concepts.

Using a single measure for internal complexity would permit to establish an absolute TS threshold which should not be exceeded for any function during development. The difficulty of such threshold’s establishment creates a validity threat. We could not define such a threshold in a simple way due to the fact that C(I) is not a basic measure which can be calculated directly, it is a combination of two measures which are normalized. Thus while applying Eq. 3 for a function we cannot easily check if the TS number of function is within the established threshold. One solution might be using only M or Fout in Eq. 3 as an internal complexity measure. However both M and Fout are revealing narrow aspects of complexity which can compromise the results. We believe that a single sophisticated complexity measure of internal complexity will enable the establishment of a simple threshold practicable in industry; therefore we have planned to address this issue in our forthcoming research. For now, we rely on selecting 0.5% of most complex functions instead of defining an absolute threshold.

The nature of trade-off between internal and external complexities can be debatable, which is an internal validity threat: If we observe the distribution of C(I) complexity measure separately we can notice that vast majority of functions in all products have small internal complexity numbers. In our case the number of functions that have M>10 cyclomatic complexity is less than 8% of all functions in the five software products, the number of functions that have M=20 is less than 2.5%, etc. The similar effect is observed for Fin measure. This means that the outcome of higher complexity number has a smaller probability. Hence, assuming that C(I) and C(E) have independent relation, the probability of an outcome of a function that has both high C(I) and C(E) number is much smaller due to probability multiplication effect. So, the relation of C(I) and C(E) can be expected to be inverse. However, contemplating the nature of observed trade-off, we can observe two important facts:

1. The statement “the outcome of higher complexity number has a smaller probability” is the manifestation of the nature of complexity itself. This means that small number of complex functions is caused because of complexity; complex functions are more difficult to produce therefore they are more rare in a product.

2. Since there is very small, yet existing probability of a function with both high C(I) and C(E) number, there must be a few outlier functions in at least one of the five software products, however as the Figure 2 illustrates there is no such outlier.

Concluding the above discussions we must notice that the tolerable limit for all five products are the same, and this supports the claim that no matter what product is developed, what the overall size of the product is, or who the developers are; there is a limit in how far the complexity of functions can grow. Notice that this last statement does not contradict to Lehman’s law of growing program complexity [18], since Lehman’s law concerns with whole program’s complexity in terms of that program’s deteriorating structure. In this work we emphasize the functions (as units of program) that a developer directly works with.

7. RELATED WORK

Probably one of the first studies that reflects on two aspects of code complexity, internal and external complexity, is Henry’s and Kafura’s study where they define complexity metrics based on information flow between software components Henry and Kafura [19]. In this paper the authors define the complexity based on the product of Fan-out and Fan-in. In another paper of them, they investigate the correlation of three widely known complexity metrics to understand their relationship [20]. We believe that their research had a pivotal role in exploring complexity management issues and we continued the tradition in terms of scrutinizing the nature of different aspects of complexities.

A comprehensive list of software complexity types can be found in Mens [21] work but the discussion of nature and relations of complexity types was out of the scope of that study. Card and Agresti [22] distinguish local and inter-module complexity for software code and define the sum of them as a total complexity. Nonetheless, they did not investigate the relationship of the two complexity aspects, which was a topic for our study. Concas, et al. [23] are studying the use of new complexity metrics, which are based on social network analysis. They correlate them with traditional complexity metrics and investigate their dependencies.

In their work Councill and Mubarak investigated the relationship of Fan-in and Fan-out of classes and found that there are ‘server’ and ‘client’ types of classes which just have high fan-in and high fan-out respectively [24] and [25]. What is more, they found classes which have relatively high fan-in and fan-out but they never investigated how much these two measures can be scaled up. Our study is a relevant complement to their study as it investigates further the trade-off between complexity measures and ultimately defines two types of complexity that have an inverse relationship for complex functions.

Tran-Cao, et al. [26] define the code complexity as a three dimensional vector, which is composed by data movement, data manipulation and system complexity. They show that effort estimation models based on combination of these complexity metrics is more effective than previously suggested models.

Xiao [27] proposed a hybrid metric for internal complexity of functions, which is based on Halstead’s, McCabe’s complexity metrics and fan-out. We have planned in our future work to use such a metric for measuring the internal complexity, as it is more sophisticated and can reveal wider aspects of internal complexity of functions.
8. CONCLUSIONS

The purpose of our research was to investigate the relationship of different aspects of software complexity for optimal complexity management. We analyzed two industrial and three open source products for this purpose. The results showed that there are two main types of complexity that emerge when developing software functions, internal complexity and external complexity. The former is complexity related to internal operations of functions which makes the function difficult to understand. The latter shows the ability and extent that a given function can make other parts of code complex.

Analysis showed that:

1. External and internal complexities have inverse relationship

2. Irrespective the size and domain of software there is a clear maximum limit for the product of external and internal complexity measures

These two investigations outlined the existing trade-off between two complexities, that is, in practice the product of two complexity values for a function cannot exceed a certain limit.

The paper also presents a method for effective complexity management. The evaluation showed that the method enables to identify the few most complex functions out of thousands for manual assessment and reengineering.

In our future work we plan to integrate this method with the measurement system that we had developed for Volvo GTT and Ericsson. This will both strengthen the effectiveness of measurement system and permit the evaluation of currently presented method in a longer run.

9. ACKNOWLEDGMENTS

Our thanks to the companies for their support in the study. This research has been carried out in the Software Centre, Chalmers, University of Gothenburg, Ericsson, Volvo Group Truck Technology.

REFERENCES


