An Empirical Study of Software Reuse and Quality in an Industrial Setting

Berkhan Deniz[1] and Semih Bilgen[2]

[1] Aselsan Electronics Inc.,
Defense System Technologies (SST) Group, Ankara, Turkey
[2] Middle East Technical University,
Electrical and Electronics Engineering Dept., Ankara, Turkey
berkhand@aselsan.com.tr,
semih-bilgen@metu.edu.tr

Abstract. Software reuse is known to be generally effective in reducing development and maintenance time and cost as well as increasing quality. In this paper, the effects of reuse on software quality in an industrial setting are empirically investigated within the framework of three different case studies. Throughout this study, we worked with Turkey's leading defense industry company Aselsan's software engineering department. We collected and calculated reuse and quality metrics as well as performance measures of individual embedded software modules and staff productivity rates. By analyzing these measurements, we developed suggestions to further benefit from reuse through systematic improvements to the reuse infrastructure and process.

Keywords: Software Reuse, Quality Metrics, Embedded Software, Fault-proneness, Industrial Study.

1 Introduction

Software reuse is generally accepted to reduce development and maintenance time and cost. Any software life cycle product can be reused, not only fragments of source code [1].

As software assets are reused, the accumulated defect fixes result in higher quality [2]. Therefore, a high degree of reuse correlates with a low defect density.

In this study, we investigated the reuse and quality relations in real-life software projects carried out by Turkey's leading defense industry company Aselsan. We designed three separate case studies in which we collected and calculated object-oriented quality metrics, reuse rates and performance of individual modules, fault-proneness of components, and productivity rates of the products. Then, by analyzing these metrics, we reached some useful conclusions: Based on these case studies, we developed suggestions to further benefit from reuse through systematic improvements to the reuse infrastructure and process.

The remaining sections of this paper are organized as follows: In Section 2, background information about reuse types, methods of measuring reuse and quality are presented. Also, previous studies on the effects of reuse on software quality are briefly reviewed. In Section 3, our research hypotheses and the designed case studies are presented. The collected reuse and quality metrics are introduced and the discussions of these measurements are provided for all case studies, respectively. Section 4 is about validity of the study. In Section 5, the collected data is analyzed and the research hypotheses are verified. Finally, in Section 6, suggestions to improve the reuse infrastructure are formulated based on the reported case studies. Section 7 concludes the paper; the work done and obtained results are summarized, achievements and difficulties of the study are reviewed; suggestions for future studies are offered.

2 Background

2.1 Reuse Types

Developers can resort to reuse via most software related entities such as requirements, system specifications, design reports, and processes such as domain engineering for product lines, as well as any other process artifact [1, 3]. There are two main reuse categories: Components-based and transformation-based [12]. In the first category, developers choose appropriate components, and reuse them, with modifications, if and where. Open source software components, commercial-off-the-shelf components, software architecture modules, and product-line components are some examples of reusable components [6]. In the second category, an automated engine produces outputs by transforming appropriate inputs. Most reuse-engines are examples of transformation-based reuse.

2.2 Measuring Reuse

Source line of code (SLOC) is the most often suggested metric for reuse size measurements in literature [1, 11]. Furthermore, researchers suggest object-oriented function points for measuring reuse, since this metric is more straightforward to implement [3]. Additionally, for component-based software, "size" is not an available metric as it is for standard systems. Therefore, "the number of use cases" is suggested as an alternate means of size measurements [4].

2.3 Measuring Quality

ISO/IEC 25000 Quality Model. The latest quality model released by ISO/IEC is Software product Quality Requirements and Evaluation (SQuaRE). This model covers software quality requirements with a systems perspective. The new standard includes a mechanical parts section including mechanics, hydraulics, electronics, and human processes. Hence, the new system-description investigates a wide range of applications [14]. ISO/IEC 25023 Measurement of the system and software product quality model of SQuaRE collects and replaces ISO/IEC 9126-2 and ISO/IEC 9126-3 revised [15].

Code-Based Software Quality Metrics. The above standard suggests both internal and external metrics for measuring quality, and various other metrics derived using this model. However, since the standard measures a software product via a large number of aspects; it does not provide specific code-based metrics; hence using this standard does not make sense for code-based measurements. Therefore, we decided to use another model for code-based metrics. The advantages of code-based metrics are summarized as:

- System level forecasting [8, 17]
- Prior identification of unsafe components [13]
- Development of safety design and programming instructions [8, 17]
- Identify the quality and structure of the software design and code [13]
- Prediction of fault-proneness [13]
- Prediction of development and testing efforts
- Validation of the software design quality [5]
- Improve software quality and productivity [5]
- Rapid response when a new technology is adopted [3]
- Reduction of implementation and maintenance cost [3]

The literature on CK metrics signifies that, in most of the studies, the measured metrics correlate with software quality, especially with fault-proneness [5]. Additionally, it is widely observed that OO concepts of coupling and complexity are strongly correlated with fault-proneness [16].

Effects of Reuse on Software Quality. Software products' quality usually increases with reuse; because as software artifacts are reused, the collection of the defect corrections in sequential versions brings about a higher quality [2]. Furthermore, the defects detected in the reused components are given higher priority. Additionally, components to be reused are designed and tested more carefully.

3 Research Hypotheses and Case Studies

In this study, we examined the relations between software reuse and quality in Aselsan. We formulated four hypotheses as shown below and in order to confirm these, we designed three different case studies.

- Hypothesis 1 – Code-based Quality: The quality of software products is improved as reuse rates of the products increase.
- Hypothesis 2 – Performance of Embedded Software: Performance of the embedded software products decays as reuse rate of the products increase.
- Hypothesis 3 – Fault-proneness: The number of defects detected in components decreases as these components are reused in various products.
- Hypothesis 4 – Productivity: The productivity rates of products increase as the reuse rates of these products increase.

Throughout this study, we worked with three different software teams. These teams were selected due to their accessibility and the possibility of communication with them arising from the author's employment.

The summary of metrics employed in each case study is displayed in Table 1.

Table 1. Case studies and the corresponding metrics employed

Case Study / Metrics	Case Study 1	Case Study 2	Case Study 3
Code-based Quality	+		+
Performance	+		+
Fault-proneness		+	
Productivity		+	+

3.1 Case Study 1

In this study, we worked with team 1. This team develops real-time embedded software for various air defense weapon systems produced in Aselsan and creates software in C++ language.

Three different software modules are investigated in this case study.

The first module is the User Command and Control Interface of an embedded system. This module opens up a TCP/IP socket interface and the external users of the system connect and control the system through this interface. This module is responsible for getting commands through socket, parsing them, and entering the commands into the rest of the system. This module also sends updates and requests to external users; it formats messages in bytes level and sends through the socket.

Since message taking and sending include parsing and formatting in bytes level, it is time-consuming to add a new message to the command interface. In order to simplify message management, a new reuse engine was developed and used by the developers in this team. This engine creates a middleware for all parsing and formatting parts of the socket interface. The user of this tool just decides on the interface functions, and the auto-created middleware is inserted into the main code.

This reuse engine is an example of transformation-based reuse since this engine reuses all the process needed for message sending, receiving, and parsing. The developed middleware can be used without any changes in many different systems. Additionally, this engine can also be used by the test engineers and reduces test efforts. Furthermore, the documentary outputs of this engine can be used for documentation purposes as an example for documentation reuse.

The second module does the same work as the first module, but is implemented using the above-mentioned reuse engine.

The third module is also a product of the reuse engine; however, its developers are different from the second one. It does work that is similar to the work in the second one, but it is a small module: It has fewer messages than the second one.

In addition to CK metrics at the class level, we employed additional complexity metrics in the experiment. In the OO environment, design concepts such as inheritance, coupling, and cohesion have been argued to embrace complexity [5].

Also, complexity metrics have been shown to correlate with defect density in a number of case studies [13]. Hence, we selected additional complexity metrics from [7]: McCabe Cyclomatic Complexity (McCabeCC - the number of flows in a part of the code), Nested Block Depth (NBD - the extent of nested blocks of code), and Percent Branch Statements (% Branches – percentage of the statements that create a break in the sequential execution of statements).

In this study, the only available physical metric is the number of cycles for the same work to be done for each module. Therefore, we used this metric to investigate the change of performance metrics for embedded systems with changing reuse rates.

The reuse rates of the modules were calculated by using reused (auto-generated) non-comment line of codes, and total non-comment line of codes (Module 1: 0%, Module 2: 81%, Module 3: 52%).

Table 2. Extracted software quality metrics

Metrics Type	Module 1	Module 2	Module 3
CBO	2,311111	1,944444	2,166667
DIT	0,333333	0,651166	0,066667
NOC	0,422222	0,686047	0,133333
WMC	4,777778	2,166667	2,9
LCOM	0,0820	0,0495	0,0823
SLOC	2819	4117	765
McCabeCC	2,49	1,30	1,59
NBD	1,71	0,84	1,10
% Branches	18,2	7,4	8,9
NoCycles	7914	9756	9648

The only performance metric calculated is the number of cycles (NoCycles) for the modules to receive the command from the system and send the corresponding data. (Table 2).

Discussion of the Measurements. We group the measured metrics according to their primary OO concepts i.e. Size, inheritance, coupling, and complexity, and compare these metrics with respect to increasing reuse rates. We have found a strong correlation between complexity metrics and reuse rate (Figure 1). Between coupling metrics and reuse rate, a positive relationship is observed (Figure 2). We did not observe a strong relation between the other quality metrics and reuse rate.

The use of a reuse engine causes a reduction in complexity. In Figure 1, a reduction is noticed in all complexity metrics (WMC, McCabeCC, NBD, and % branches) with increasing reuse rate.

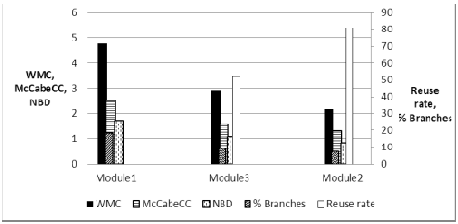

Fig. 1. Comparing complexity metrics and reuse rate

In Figure 2, an improvement is observed in terms of coupling as reuse rate increases. The change of the architecture for reuse and introducing interface classes in the system make the system less coupled.

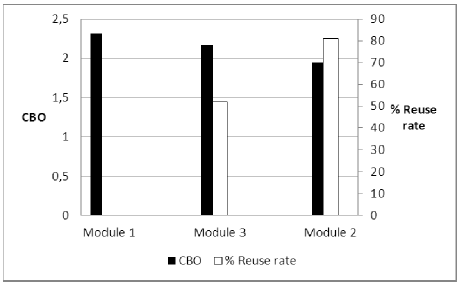

Fig. 2. Comparing coupling metric and reuse rate

Figure 3 shows the variation of performance metric (NoCycles) with different reuse rates. As expected, the number of cycles increases with increasing reuse. In the first module, after the command is received from the system, it is sent through related socket directly; however in second and third modules the system architecture changed in order to reuse the middleware and now between sending and receiving, a middleware is introduced which increases the number of cycles.

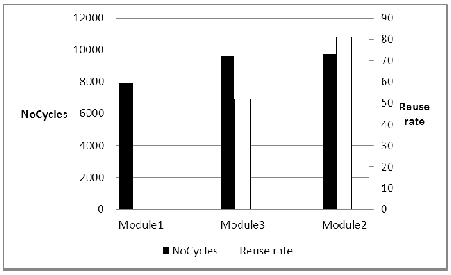

Fig. 3. Comparing performance metric and reuse rate

3.2 Case Study 2

In this study, we worked with team 2. This team develops command and control software for tactical fire support systems using a software product line. The team creates software in .NET environment. SPL of team 2 is composition-oriented. In it, there are two types of components: common platform components reused in various projects and product specific components developed for every single product. A product developed on this SPL consists of various numbers of common platform components and product specific components in it.

The measurements in case study 2 include defect counts of the components developed by this team, which are reused in different products and the productivity rates of these products.

Defect counts are obtained from the problem reporting system used in Aselsan. As a result of company politics, all defects detected during system integration and acceptance testing and also those reported by customers are kept in the problem reporting system i.e. Defects during software development process are not included in these measurements.

Measurements about requirement counts are acquired from the requirements management tool used in Aselsan.

Total efforts of the products are measured by the business management software used in Aselsan. However, due to the commercial confidentiality, we do not provide exact measures of the efforts in this study.

About the specifications of the SPL's various products, and about the properties of the common and product-specific components in these products, we worked with one of the configuration managers of team 2.

In this work, three different products are explored. All products have at least one product specific component, and other components are common platform components. The three products were developed sequentially with six months between the completions of each one.

Table 3. New and reused component counts in three different products

Product No / Component count	New	Reused
Product 1	19	0
Product 2	3	16
Product 3	6	15

We classified the components which we analyzed as "new" and "reused" components. New components are not used in earlier products, and reused components are used previously in other products. Table 3 shows the new and reused component counts in the products analyzed. Table 4 displays new and total requirement counts in all components for each product. Table 5 displays total effort in man-hour for each product.

Defect counts of the components used in three different products are shown in Table 6. Components 1-8 are common in all three products; component 11 is partly common; and components 9, 10, and 12 are new components (i.e. They are used in corresponding products for the first time). All 12 components are common-platform components.

Table 4. New and total requirement counts in all components for each product

Components / Product No	Product 1	Product 2	Product 3
C1	291 / 291	4 / 295	0 / 295
C2	383 / 383	10 / 293	0 / 293
C3	261 / 216	0 / 216	1 / 217
C4	167 / 167	0 / 167	0 / 167
C5	301 / 301	4 / 305	0 / 305
C6	304 / 304	11 / 315	0 / 315
C7	126 / 126	21 / 147	1 / 148
C8	220 / 220	32 / 252	27 / 279
C9	-	-	211 / 211
C10	-	-	177 / 177
C11	275 / 275	-	14 / 289
C12	-	141 / 141	-

Table 5. Total effort for each product

Product No	**Total Effort (man-hour)**
Product 1	2,75 * N
Product 2	1,5 * N
Product 3	N

Discussion of the Measurements. According to Figure 4, the average defect count in the first product is more than 50, less than 3 in the following product and less than 1 in the third product. More than 95 % of the total defects are detected in the first product. There are various reasons for this improvement: the reused components are less modified than non-reused ones; therefore, they are more stable. Additionally, the reused components are designed more intensely; since defects in them affect different products. Furthermore, the employment of the common components in various products causes them to become faultless and finished components.

Table 6. Defect counts of the components in three different products

Components / Product No	Product 1	Product 2	Product 3
C1	87	2	0
C2	35	5	0
C3	54	1	1
C4	20	0	0
C5	63	0	0
C6	100	6	0
C7	24	0	1
C8	48	3	0
C9	-	-	24
C10	-	-	34
C11	55	-	7
C12	-	27	-

In Figure 5, defect counts of the product-specific components (i.e. Components 9, 10, and 12), and the average defect counts of common components are shown. For all three components, defect counts are more than 20. Since, these components are not reused, we observe a similar distribution as the defect counts of the common components in the first product they are used.

Defect count of component 11 is shown in Figure 6. This component is partially-common in products 1 and 3 (see Table 4 and Table 6). In product 1, more than 50 defects are detected, and in product 3 almost 10 defects are detected. The defect distribution of this component is similar to common components' distribution of products 1 and 2.

Table 7 shows productivity rates calculated by dividing number of new requirements (Table 4) by total efforts (Table 5).

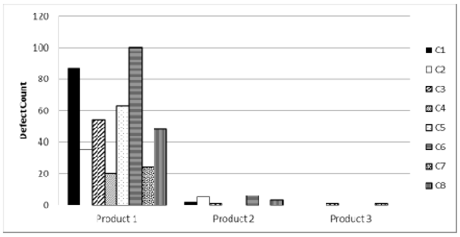

Fig. 4. Defect counts of the common components (C1-C8)

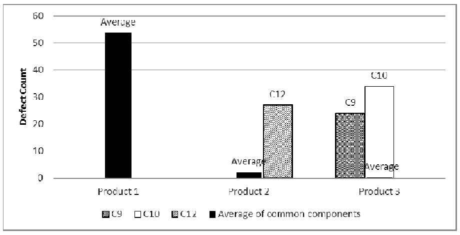

Fig. 5. Defect counts of the product-specific components (C9, C10, and C12)

Table 8 shows productivity rates calculated with the total number of requirements.

In Figure 7, productivities are compared. As components are reused in different products, productivity rates increase remarkably, which is not surprising.

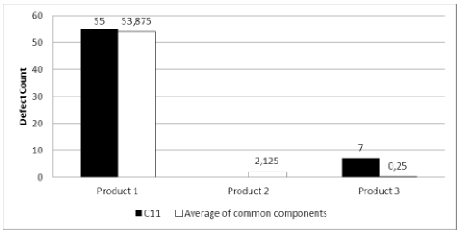

Fig. 6. Defect counts of the partially-common component (C11)

Comparison of the productivity rates with new requirements exhibits a sharp reduction between the first and second products; and a noteworthy expansion between the second and third products. We interpret the first case as a conclusion of the development by employing a product-line. Since the reused components are already developed in previous products, the productivity rates increase significantly.

Table 7. Productivity rates using new requirements

Product No	New Requirements	Productivity (requirements / man-hour*N)
Product 1	2283	830,2
Product 2	223	148,7
Product 3	431	431

Furthermore, the first reduction in the second case is explained as an adaptation period of the development with the product line. Although there are developed-products, ready to be used, it is still time-consuming to gather up these components and integrate them with the recently developed components. Finally, the expansion between the second and third products in the second case is interpreted as an evidence of being trained in development with the product line. It is expected that, in future products, the productivity rates using the new requirements will exceed the productivity rate of the first product.

Table 8. Productivity rates using total requirements

Product No	Total Requirements	Productivity (requirements / man-hour*N)
Product 1	2283	830,2
Product 2	2231	1487,3
Product 3	2796	2796

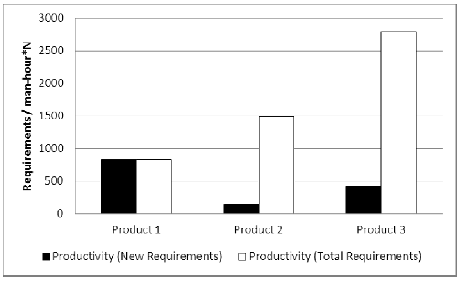

Fig. 7. Comparison of productivity rates of each product using new requirements and total requirements

3.3 Case Study 3

In this study, we worked with team 3. This team develops real- time embedded software for fire control systems using a composition-oriented SPL in C++ language. The capabilities, which can be included in the product line or excluded from the product line, are modeled as separate components. In this SPL, there are various common-platform components reused in different products and also there are product-specific components.

Measurements of case study 3 include of changing productivity rates as reuse rates vary for different products developed sequentially using SPL of team 3. In addition, the performance measurements of a critical scenario in this domain will be measured and compared before and after the SPL is employed.

Reuse rates are calculated using reused non-comment line of codes and total non-comment line of codes. Productivity rates are calculated by dividing total non-comment line of codes by total effort to develop the so called product. Total efforts are measured by the business management software used in Aselsan. However, due to commercial confidentiality, we do not provide total efforts and total source line of code metrics here. For making these measurements, we worked with one of the configuration managers of the team.

In Table 9, reuse and productivity rates for the products developed using SPL of team 3 are given.

In order to compare the performance before and after employing the product line, one of the most critical scenarios in the system, the automatic video tracking scenario is investigated. The performance metrics used are time delay and CPU usage in this scenario. These metrics are measured using the provided embedded operating system functions.

Table 9. Reuse and productivity rates for products in SPL of team 3

Product No	% Reuse Rate	Productivity (SLOC / man-hour)
Product 1	35,73	54,22
Product 2	39,25	54,92
Product 3	48,86	43,38
Product 4	48,9	68,39

During the development process of the product line, the team introduced two more layers into the scenario. During the SPL design the team worked on two different approaches i.e. Pull and push strategies, while the first approach was more reusable with higher abstraction level.

Separately for three scenarios, the delay from reception of the track data from VT System to the transmission of the platform data to the Servo Controller System and the CPU usage during the scenario are measured. Measurements are shown below in Table 10.

Table 10. Measurements of the AVT scenario

Scenarios / Measurements	Minimum Delay (ms)	Maximum Delay (ms)	Average Delay (ms)	% CPU Usage
Before SPL	0,85	1,05	0,9	72,3
Pull strategy	2,12	35,0	20,0	81,5
Push strategy	2,12	5,5	3,2	79,9

Discussion of the Measurements. In Figure 8, the comparison of reuse and productivity rates of the products is displayed. Reuse rates increase from product 1 to product 4; however productivity change does not have the equivalent attitude. Productivity rates increase slightly between the first two products, and then productivity rate decreases from product 2 to product 3. However, between the last two products, productivity rate differs noticeably. When we discussed this situation with the team, we found out the following factors:

- There was a serious waste of time during the development of the non-reused (new) parts in product 3,
- Most of the developers of the product 3 were unfamiliar with software development by employing the product-line.

We can conclude that utilization of some normalizing factors i.e. Code complexity for the non-reused parts and experience of the developers, in measuring productivity rates can be useful. Furthermore, during the initial products, it is not surprising to observe productivity decays; since it is time consuming to get used to the product line in a software development team.

Previously, three different implementation methods of the AVT scenario are explained. The first method was before the team developed the SSRM SPL. In the second and third methods, there are two additional layers which are due to the product line employment and in order to increase the reuse of the scenario software.

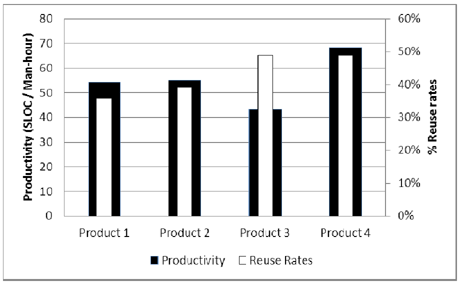

Fig. 8. Comparison of reuse and productivity for products in SPL of team 3

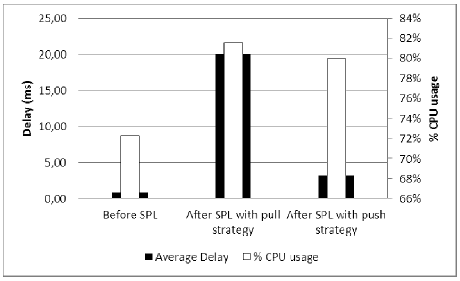

Fig. 9. Comparison of average delays and CPU usages of three different implementations of AVT scenario

The AVT scenario delay is measured for three cases (see Table 10), and comparison of the average delay and CPU usage are shown in Figure 9. According to these comparisons, we can conclude that while transforming the software into more reusable, and more abstract from the interfaces; we lose from the performance. Therefore, the developers should decide on the limit of this trade-off. Sometimes, the performance requirements allow these improvements; however sometimes performance requirements are too heavy. The second method was the most appropriate one in terms of reusability, however the developers had to perform and apply the third method. It was also more reusable compared to the first method, and it was acceptable when the performance requirements were considered.

4 Validity of the Case Studies

Four aspects of validity are summarized for empirical studies as “construct validity”, “internal validity”, “external validity”, and “reliability” [10].

- Construct validity is about the compliance of the measurement and interpretation of the theoretical constructs. In our case studies, the metrics are either measured by the researcher or collected from the company databases.
- Internal validity is about whether the causal relations are studied. Since our case studies are not scientific experiments, we did not suffer for this validity.
- External validity focuses on the generalizability of the results. In our case studies, we aimed collecting all related metrics from all the teams; but for current states of the teams and projects, it was not possible. Therefore, our results and suggestions mainly are advantageous for our company.
- Reliability is concerned with the replicability of the study by different researchers. We claim that any researcher accessible to the internal metrics of a software company may come up with the similar results.

5 Verification of the Hypotheses

In this section, the results of the measurements in the case studies will be analyzed, and it will be stated whether or not the hypotheses are verified.

Case study 1 findings show that some CK metrics and size metrics do not correlate with changing reuse rate: SLOC, DIT, NOC, and LCOM. However, Coupling and Complexity CK metrics and the additional complexity metrics show a strong correlation with the changing reuse rate. In accordance with the relevant literature, the improvements in coupling and complexity metrics are sufficient to claim an increase in software quality. Therefore, we can conclude that Hypothesis 1 is verified.

In case study 1, we measured and compared the performance of a message receiving and transmitting scenario in three different embedded software modules. We find a strong negative correlation between performance and reuse rate; which is consistent with the related arguments in the literature.

In case study 3, we measured and compared the performance metrics of a critical scenario of an embedded software system before and after employing a product-line approach. We observed that, the case before the product line approach was the best regarding the performance, and as the software turned into more reusable and more abstract from other parts of the software, the performance of the software decayed.

Consequently, the measurements from two different case studies have verified Hypothesis 2.

In case study 2, measurements are taken from a product line which is used in subsequent products. When the components are not reused, we observe a large number of defects. Furthermore, we find that the decrease in the defect counts is independent of the product types. When the component is firstly used in product 3, again we detect a similar distribution as if the component is firstly used in product 2.

To conclude, as components are reused in several products, we observed that their defect counts decrease significantly and so their fault-proneness. Therefore, we can conclude that hypothesis 3 has been verified in this study.

In case study 2, productivity rates of the three products developed using the SPL approach are presented. Productivity rates are measured using the number of requirements using the requirements count and total effort. The results showed that, if the productivity is measured using the total number of requirements in the deployed product, the productivity rates improve significantly. Additionally, productivity is measured also by using the new requirements. In that case, we observed a reduction in productivity between the first and second products; and an increase in productivity between the second and third products. This situation is interpreted as an adaptation period of the product line approach.

In case study 3, we compared productivity rates of products implemented by another product line approach with increasing reuse rates. In this measurement, we also observed a positive correlation with reuse and productivity rates. However, the change of the members of the team during the development of product 3 caused a reduction in the productivity rate of this product. This situation is interpreted, similarly in case study 2, as an adaptation period of the product line.

Hence, we concluded that, if the effects of the adaptation period of the product line approach are ignored, the productivity rates improve significantly as the rate of reuse increases in a product line. Therefore, hypothesis 4 has been verified.

6 Suggestions for Further Benefit from This Study

In this section, suggestions regarding the reuse infrastructure and process to improve benefits of reuse are formulated.

- Use of Reference Metrics

Software developers should incorporate software quality metrics into their software development processes, and before and after serious decisions on design, technology or infrastructure; the change of these metrics should be investigated.

Therefore, in order to succeed in the employment of these metrics, the software developers should select reference metrics specific to their software domain and periodically monitor the changes of these metrics.

- Automated Detection of Architectural Effects

Real time embedded software developers should monitor the performance requirements after employing extensive architectural modifications; furthermore, they must update the modifications if the performance of the software eventually becomes unacceptable for the system.

Thereupon, the embedded software developers should develop methods in order to automate the process of detecting the architectural modifications which include the chance of worsening the software performance below system requirements.

- Recording Software Development Process Defects

In order to improve the management of defects, and investigate the defects intensely; the severity of the defects should also be provided after being corrected.

Additionally, the defects detected during the software development process should also be recorded; since the defects of the components during the development process is a key metric in order to improve the reuse infrastructure of a product line.

- Association of Defects with Design Concepts

  In order to improve the management of defects, and investigate the defects intensely; the software developers should identify each defect with corresponding component, and the design concept.

- Recording Rework Efforts and Efforts Associated with Reuse

  In order to be able to measure and analyze rework and reuse efforts, with changing reuse rates; these metrics should be recorded, and for this purpose the relevant infrastructure should be developed.

- Explicit Accounting for Code Reuse

  During analysis of the productivity rates of the products, it was found that lack of the experience of the developer team was a significant factor of the declines in productivity; since, the employment of product lines requires an extra effort such as an adaptation period. Henceforth, during effort estimations of the products developed by a product line approach; the experiences of the developers, about the product line, should also be considered. Finally, the developers should also estimate and record efforts separately for reused and non-reused components, in order to analyze the impacts of reuse on the productivity rates deeply.

7 Conclusion

In this study, we worked with software engineering department of a defense industry company Aselsan. We examined their software projects and follow reuse and quality relations for these projects. For this purpose, we collected and compared some software measurements such as OO quality metrics, fault-proneness, performance, and productivity with changing reuse rates. Finally, we have formulated suggestions in order to improve the reuse infrastructure and process, after verifying reuse and quality relations in this setting. Throughout this study, we accomplished three different case studies and measured and compared different concepts in all three cases; however we were not able to obtain all these measures in all cases. Therefore, for future studies, it would be a significant improvement if all types of the measurements could be collected and compared with changing reuse rates, for all case studies separately.

References

1. Frakes, W., Terry, C.: Software reuse: metrics and models. ACM Computing Surveys 28(2), 415–435 (1996)
2. Lim, W.C.: Effects of reuse on quality, productivity, and economics. IEEE Software 11(5), 23–30 (1994)
3. Jamali, S.M.: Object Oriented Metrics (A Survey Approach). Department of Computer Engineering Sharif University of Technology, Tehran, Iran (2006)

4. Sedigh-Ali, S., Ghafoor, A., Paul, R.A.: Metrics and models for cost and quality of component-based software. In: Proceedings of IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 149–155 (2003)
5. Subramanyam, R., Krishnan, M.S.: Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. IEEE Transactions on Software Engineering 29(4), 297–310 (2003)
6. Mohagheghi, P., Conradi, R., Killi, O.M., Schwarz, H.: An Empirical Study of Software Reuse vs. Defect-Density and Stability. In: Proceedings of International Conference on Software Engineering, pp. 282–291 (2004)
7. Oliveira, M.F.S., Redin, R.M., Carro, L., da Cunha Lamb, L., Wagner, F.R.: Software Quality Metrics and their Impact on Embedded Software. In: 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software, MOMPES 2008, pp. 68–77 (2008)
8. El-Emam, K.: Object-oriented metrics: A review of theory and practice. In: Advances in Software Engineering, pp. 23–50. Springer-Verlag New York, Inc., New York (2002)
9. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6), 476–493 (1994)
10. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14(2), 131–164 (2009)
11. Mohagheghi, P., Conradi, R.: Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empirical Software Engineering 12(5), 471–516 (2007)
12. Dusink, L., van Katwijk, J.: Reuse Dimensions. In: SSR 1995 Proceedings of the 1995 Symposium on Software Reusability, pp. 137–149 (1995)
13. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: ICSE 2006 Proceedings of the 28th International Conference on Software Engineering, Shanghai, China, pp. 452–461 (2006)
14. Boegh, J.: A New Standard for Quality Requirements. IEEE Software 25(2), 57–63 (2008)
15. ISO/IEC, Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality, ISO, ISO/IEC WD 25023 (2011)
16. Deniz, B.: Investigation of The Effects of Reuse on Software Quality in an Industrial Setting. M.S. thesis, Electrical and Electronics Engineering Dept., Middle East Technical University, Ankara, Turkey (2013)
17. Lincke, R., Lundberg, J., Löwe, W.: Comparing software metrics tools. In: ISSTA 2008 Proceedings of the 2008 International Symposium on Software Testing and Analysis, Seattle, Washington, USA, pp. 131–142 (2008)