Embedded System Construction – Evaluation of Model-Driven and Component-Based Development Approaches

Christian Bunse[1], Hans-Gerhard Gross[2], and Christian Peper[3]

[1] International University in Germany, Bruchsal, Germany
Christian.Bunse@i-u.de
[2] Delft University of Technology, Delft, The Netherlands
h.g.gross@tudelft.nl
[3] Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany
Christian.Peper@iese.fraunhofer.de

Abstract. Model-driven development has become an important engineering paradigm. It is said to have many advantages, such as reuse or quality improvement, over traditional approaches, even for embedded systems. Along a similar line of argumentation, component-based software engineering is advocated. In order to investigate these claims, the MARMOT method was applied to develop several variants of a small micro-controller-based automotive subsystem. Several key figures, like model size and development effort were measured and compared with two main-stream methods: the Unified Process and Agile Development. The analysis reveals that model-driven, component-oriented development performs well and leads to maintainable systems and a higher-than-normal reuse rate.

Keywords: Exploratory Study, Embedded, Model-Driven, Components

1 Introduction

Embedded software design is a difficult task due to the complexity of the problem domain and the constraints from the target environment. One specific technique that may, at first sight, seem difficult to apply for the embedded domain, is modeling and Model-Driven Development (MDD) with components. It is frequently used in other engineering domains as a way to solve problems at a higher level of abstraction, and to verify design decisions early. Component-oriented development envisions that new software can be created with less effort than in traditional approaches, simply by assembling existing parts. Although, the use of models and components for embedded software systems is still far from being industrial best practice. One reason might be, that the disciplines involved, mechanical-, electronic-, and software engineering, are not in sync, a fact which cannot be attributed to one of these fields alone. Engineers are struggling hard to master the pitfalls of modern, complex embedded systems. What is really lacking is a vehicle to transport the advances in software engineering and component technologies to the embedded world.

Software Reuse is currently a challenging area of research. One reason is that software quality and productivity are assumed to be greatly increased by maximizing the (re)use of

(part of) prior products instead of repeatedly developing from scratch. This also stimulated the transfer of MDD and CBD [12] techniques to the domain of embedded systems, but the predicted level of reuse has not yet been reached. A reason might be that empirical studies measuring the obtained reuse rates are sparse. Studies, such as [7] or [8] examined only specific aspects of reuse such as specialization or off-the-shelf component reuse, but did not provide comparative metrics on the method's level. Other empirical studies that directly focus on software reuse either address non-CBD technology [14], or they focus on representations on the programming language-level [15]. Unfortunately, there are no studies in the area of MDD/CBD for embedded systems.

This paper shortly introduces the MARMOT system development method. MARMOT stands for *Method for Component-Based Real-Time Object-Oriented Development and Testing*, and it aims to provide the ingredients to master the multi-disciplinary effort of developing embedded systems. It provides templates, models and guidelines for the products describing a system, and how these artifacts are built. The main focus of the paper is on a series of studies in which we compare MARMOT, as being specific for MDD and CBD with the RUP and Agile Development to devise a small control system for an exterior car mirror. In order to verify the characteristics of the three development methods, several aspects such as model size [13] and development effort are quantified and analyzed. The analysis reveals that model-based, component-oriented development performs well and leads to maintainable systems, plus a higher-than-normal reuse rate, at least for the considered application domain.

The paper is structured as follows: Section 2 briefly describes MARMOT, and Sections 3, 4, and 5 present the study, discuss results and address threats to validity. Finally, Section 6 presents a brief summary, conclusions drawn, and future research.

2 MARMOT Overview

Reuse is a key challenge and a major driving force in hardware and software development. Reuse is pushed forward by the growing complexity of systems. This section shortly introduces the MARMOT development method [3] for model-driven and component-based development (CBD) of embedded systems. MARMOT builds on the principles of KobrA [1], assuming its component model displayed in Fig. 1, and extends it towards the development of embedded systems. MARMOT components follow the principles of encapsulation, modularity and unique identity that most component definitions put forward, and their communication relies on interface contracts (i.e., in the embedded world these are made available through software abstractions). An additional hardware wrapper realizes that the hardware communication protocol is translated into a component communication contract. Further, encapsulation requires separating the description of what a software unit does from the description of how it does it. These descriptions are called specification and realization (see Fig. 1).

The specification is a suite of descriptive (UML [11]) artifacts that collectively define the external interface of a component so that the component can be assembled into or used by a system. The realization artifacts collectively define a component's internal realization. Following this principle, each component is described through a suite of models as if it was an independent system in its own right.

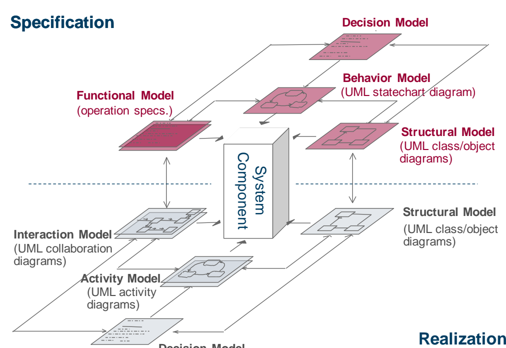

Fig. 1. MARMOT component model.

The fact that components can be realized using other components, turns a MARMOT project into a tree-shaped structure with consecutively nested abstract component representations. A system can be viewed as a containment hierarchy of components in which the parent/child relationship represents composition. Any component can be a containment tree in its own right, and, as a consequence, another MARMOT project. Acquisition of component services across the tree turns a MARMOT project into a graph. The four basic activities of a MARMOT development process are composition, decomposition, embodiment, and validation as shown in Fig. 2.

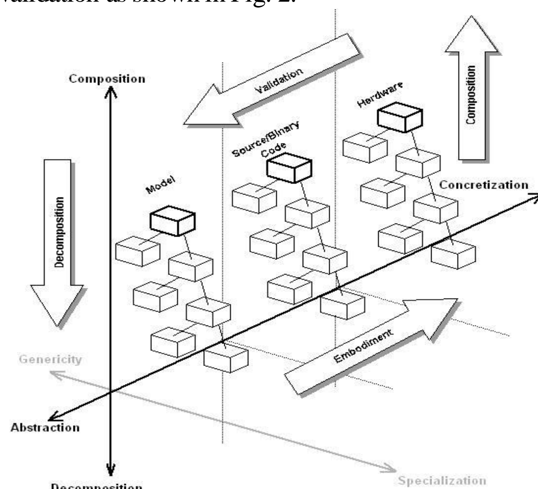

Fig. 2. Development Activities in MARMOT.

Decomposition follows the divide-and-conquer paradigm, and it is performed to subdivide a system into smaller parts that are easier to understand and control. A project always starts above the top left-hand side box in Fig. 2. It represents the entire system to be built. Prior to specifying the box, the domain concepts must be determined, comprising descriptions of all relevant domain entities such as standard hardware components that will appear along the concretization dimension. The implementation-specific entities determine the way in which a system is divided into smaller parts. During decomposition,

newly identified logical parts are mapped to existing components, or the system is decomposed according to existing components. Whether these are hard- or software is not important since all components are treated in a uniform way, as software abstractions.
Composition represents the opposite activity, which is performed when individual components have been implemented or reused, and the system is put together. After having implemented some of the boxes and having some others reused, the system can be assembled according to the abstract model. The subordinate boxes with their respective super-ordinate boxes have to be coordinated in a way that exactly follows the component description standard introduced above.
Embodiment is concerned with the implementation of a system and a move towards executable representations. It turns the abstract system (i.e., models) into concrete representations that can be executed. MARMOT uses refinement and translation patterns for doing these transformations. MARMOT supports the generation of code skeletons and can thus be regarded as a semi-automatic approach.
Validation checks whether the concrete representations are in line with the abstract ones. It is carried out in order to check whether the concrete composition of the embedded system corresponds to its abstract description.

3 Description of the Study

In general, empirical studies in software engineering are used to evaluate whether a "new" technique is superior to other techniques concerning a specific problem or property. The objective of this study is to compare the effects of MARMOT concerning reuse in embedded system development to other approaches such as the Unified process and agile development.

The study was organized in three runs (i.e., one run per methodology). All runs followed the same schema. Based on an existing system, documentation subjects performed a number of small projects. These covered typical project situations such as maintenance, ports to another platform, variant development, and reuse in a larger context. The first run applied MARMOT. The second run repeated all projects but used a variation of the Unified process, specifically adapted for embedded system development. The third run, applying an agile approach, was used to validate that modeling has a major impact and to rule out that reuse effects can solely be obtained at the code level. Metrics were collected in all runs and were analyzed in order to evaluate the respective research questions.

3.1. RESEARCH APPROACH

Introducing MDD and CBD in an organization is generally a slow process. An organization will start with some reusable components, and eventually build a component repository. But they are unsure about the return on investment gained by initial component development plus reuse for a real system, and the impact of the acquired technologies on quality and time-to-market. This is the motivation for performing the study and asking questions on the performance of these techniques.
Research Questions. Several factors concerning the development process and its resulting product are recorded throughout the study in order to gain knowledge about using MDD and CBD for the development of small embedded systems. The research

questions of the case-study focus on two key sets of properties of MDD in the context of component-oriented development. The first set of questions (Q1-Q4) lead to an understanding of basic and/or general properties of the embedded system development approach:

Q1: Which process was used to develop the system? Each run of the study used a different development approach (i.e., MARMOT, Unified Process, and Agile). These are compared in terms of different quality attributes of the resulting systems.

Q2: Which types of diagrams have been used? Are all UML diagram types required, or is there possibly a specific subset sufficient for this domain?

Q3: How were models transferred to source code? Developers typically work in a procedural setting that impedes the manual transformation of UML concepts into C [10].

Q4: How was reuse applied and organized? Reuse is central to MDD with respect to quality, time-to-market, and effort, but reuse must be built into the process, it does not come as a by-product (i.e., components have to be developed for reuse).

The second set of questions (Q5-Q9) deals with the resulting product of the applied approach (i.e., with respect to code size, defect density, and time-to-market).

Q5: What is the model-size of the systems? MDD is often believed to create a large overhead of models, even for small projects. Within the study, model size follows the metrics as defined in [13].

Q6: What is the defect density of the code?

Q7: How long did it take to develop the systems and how is this effort distributed over the requirements, design, implementation, and test phases? Effort saving is one promise of MDD and CBD [12], though, it does not occur immediately (i.e., in the first project), but in follow-up projects. Effort is measured for all development phases.

Q8: What is the size of the resulting systems? Memory is a sparse resource and program size extremely important. MDD for embedded systems will only be successful if the resulting code size, obtained from the models, is small.

Q9: How much reuse did take place? Reuse is central for MDD and CBD and it must be seen as an upfront investment paying off in many projects. Reuse must be examined between projects and not within a project.

Research Procedure. MDD and CBD promise efficient reuse and short time-to-market, even for embedded systems. Since it is expected that the benefits of MDD and CBD are only visible during follow-up projects [5], an initial system was specified and used as basis for all runs. The follow-ups then were:

R1/R2 Ports to different hardware platforms while keeping functionality. Ports were performed within the family (i.e., ATMega32) and to a different processor family (i.e., PICF). Implementing a port within the same family might be automated at the code-level, whereas, a port to a different family might affect the models.

R3/R4 Evolving system requirements by (1) removing the recall position functionality, and (2) adding a defreeze/defog function with a humidity sensor and a heater.

R5 The mirror system was reused in a door control unit that incorporates the control of the mirror, power windows, and door illumination.

3.2. PREPARATION

Methodologies. The study examines the effects of three different development methods on software reuse and related quality factors. In the first run, we used the MARMOT method that is intended to provide all the ingredients to master the multidisciplinary effort of developing component-based embedded systems. In the second run we followed an adapted version of the Unified Process for embedded system development [4] (i.e., RUP SE). RUP SE includes an architecture model framework that supports different perspectives. A distinguishing characteristic of RUP SE is that the components regarding the perspectives are jointly derived in increasing specificity from the overall system requirements. In the third run, an agile process (based on Extreme Programming) [9], adapted towards embedded software development, was used.
Subjects of the study were graduate students from the Department of Computer Science at the University of Kaiserslautern (1st run) and the School of IT at the International University (2nd and 3rd run). All students, 45 in total (3 per team/project), were enrolled in a Software Engineering class, in which they were taught principles, OO methods, modeling, and embedded system development. Lectures were supplemented by practical sessions in which students had the opportunity to make use of what they had learned. At the beginning of the course, subjects were informed that a series of practical exercises were planned. Subjects knew that data would be collected and that an analysis would be performed, but were unaware of the hypotheses that were being tested. To further control for learning and fatigue effects and differences between subjects, random assignment to the development teams was performed. As the number of subjects was known before running the studies it was a simple procedure to create teams of equivalent size.
Metrics. All projects were organized according to typical reuse situations in component-based development, and a number of measurements were performed to answer the research questions of the previous sub-section:

Model-size is measured using the absolute and relative size measures proposed in [13]. Relative size measures (i.e., ratios of absolute measures) are used to address UMLs multi-diagram structure and to deal with completeness [13]. Absolute measures used are: the number of classes in a model (NCM), number of components in a model (NCOM), number of diagrams (ND), and LOC, which are sufficient as complexity metrics for the simple components used in this case. NCOM describes the number of hardware/software components, while NCM is represents the number of software components. These metrics are comparable to metrics such as McCabe's cyclomatic complexity for estimating the size/nesting of a system. Code-size is measured in normalized LOC. *System size* is measured in KBytes of the binary code. All systems were compiled using size optimization.

The *amount of reused elements* is described as the proportion of the system which can be reused without any changes or with small adaptations (i.e., configuration but no model change). Measures are taken at the model and code level.

Defect density is measured in defects per 100 LOC, whereby defects where collected via inspection and testing activities.

Development effort and its distribution over development phases are measured as development time (hours) collected by daily effort sheets.
Materials. The study uses a car-mirror control system that moves a mirror horizontally and vertically into the desired position. Positions can be stored/recalled to support driver profiles. The simplified version of this study controls two servos via potentiometers, and

indicates movement on a LCD. A replication package is available from the authors.

For each run, the base system documentation was developed by the authors of this paper. The reason was that we were interested in the reuse effects of one methodology in the context of follow-up projects. Using a single documentation for all runs would have created translation and understanding efforts. Therefore, reasonable effort was spent to make all three documents comparable concerning size, complexity, etc. This is supported by the measures of each system.

4 Evaluation and Comparison

In the context of the three experimental runs, a number of measurements were performed with respect to maintainability, portability, and adaptability of software systems. Tables 1, 2, and 3 provide data concerning model and code size, quality, effort, and reuse rates. Table columns denote the project type[1].

Table 1. Results of the First Run (MARMOT)

		Original	R1	R2	R3	R4	R5
LOC		310	310	320	280	350	490
Model Size (Abs.)	NCM	8	8	8	6	10	10
	NCOM	15	15	15	11	19	29
	ND	46	46	46	33	52	64
Model Size (Rel.)	$\frac{NumberofStateCharts}{NumberofClasses}$	1	1	1	1	0.8	1
	$\frac{NumberofOperations}{NumberofClasses}$	3.25	3.25	3.25	2.5	3	3.4
	$\frac{NumberofAssociations}{NumberofClasses}$	1.375	1.375	1.375	1.33	1.3	1.6
Reuse	Reuse Fraction(%)	0	100	97	100	89	60
	New (%)	100	0	3	0	11	40
	Unchanged (%)	0	95	86	75	90	95
	Changed (%)	0	5	14	5	10	5
	Removed (%)	0	0	0	20	0	40
Effort (h)	Global	26	6	10.5	3	10	24
	Hardware	10	2	4	0.5	2	8
	Requirements	1	0	0	0.5	1	2
	Design	9.5	0.5	1	0.5	5	6
	Implementation	3	1	3	0.5	2	4
	Test	2.5	2.5	2.5	1	2	4
Quality	Defect Density	9	0	2	0	3	4

First Run Porting the system (R1) required only minimal changes to the models. One reason is that MARMOT supports the idea of platform-independent modeling (platform specific models are created in the embodiment step). Ports to different processor families (R2) are supported by MARMOT's reuse mechanisms.

---

[1] Project types are labeled following the scheme introduced in section 3 (e.g., "Original" stands for the initial system developed by the authors as a basis for all follow-up projects, "R1" – Port to the ATMEGA32 microcontroller (same processor family), "R2" – Port to the PIC F microcontroller (different processor family), "R3" – Adaptation by removing functionality from the original system, "R4" – Adaptation by adding functionality to the original system, and "R5" – Reuse of the original system in the context of a larger system.

Concerning the adaptation of existing systems (R3 and R4), data show that large portions of the system could be reused. In comparison to the initial development project the effort for adaptations is quite low (26hrs vs. 3/10hrs). The quality of the system profits from the quality assurance activities of the initial project. Thus, the promises of CBD concerning time-to-market and quality could be confirmed.

Interestingly, the effort for the original system corresponds to standardized effort distributions over development phases, whereby the effort of follow-ups is significantly lower. This supports the assumption that component-oriented development has an effort-saving effect in subsequent projects.

Porting and adapting an existing system (R1-R4) implies that the resulting variants are highly similar, which explains why reuse works well. It is, therefore, interesting to look at larger systems that reuse (components of) the original system (i.e., R5). 60% of the R5 system was reused without requiring major adaptations of the reused system. Effort- and defect density are higher than those of R1-R4, due to additional functionality and hardware extensions. However, when directly compared to the initial effort and quality, a positive trend can be seen that supports the assumption that MARMOT allows embedded systems development at a low cost but with high quality.

Table 2. Results of the Second Run (Unified Process)

		Original	R1	R2	R3	R4	R5
LOC		350	340	340	320	400	500
Model Size (Abs.)	NCM	10	10	10	8	12	13
	NCOM	15	15	15	11	19	29
	ND	59	59	59	45	60	68
Model Size (Rel.)	$\frac{NumberofStateCharts}{NumberofClasses}$	1.5	1.5	1.5	0.72	1.33	1.07
	$\frac{NumberofOperations}{NumberofClasses}$	4	3.5	3.5	3.25	3	3.46
	$\frac{NumberofAssociations}{NumberofClasses}$	2.5	2.3	2.3	2.5	2.16	1.76
Reuse	Reuse Fraction(%)	0	100	94	88	86	58
	New (%)	100	0	6	11	14	42
	Unchanged (%)	0	92	80	70	85	86
	Changed (%)	0	4	15	6	15	14
	Removed (%)	0	4	5	24	0	41
Effort (h)	Global	34	8	12	5.5	13	29
	Hardware	10	2	4	0.5	2	8
	Requirements	4	1	1	1.5	3	4
	Design	12	1	2	1	4	7
	Implementation	5	2	3	1.5	2	6
	Test	3	2	2	1	2	4
Quality	Defect Density	8	1	2	0	3	4

The **Second and Third Run** replicated the projects of the first run but used different development methods. Interestingly, the results of the second run are quite close to those of the first. However, the Unified Process requires more overhead and increased documentation, resulting in higher development effort. Ironically, model-size seems to have a negative impact on quality and effort. Interestingly, the mapping of models to code seems not to have added additional defects or significant overheads.

Although the amount of modeling is limited in the agile approach, it can be observed that the original system was quickly developed with a high quality. However, this does not hold for follow-up projects. These required substantially higher effort than the effort

required for runs 1 and 2. A reason might be that follow-ups were not performed by the developers of the original system. Due to missing documentation and abstractions, reuse rates are low. In contrast, the source-code is of a good quality.

Table 3. Results of the Third Run (Agile)

		Original	R1	R2	R3	*R4*	R5
LOC		280	290	340	300	*330*	550
Model Size (Abs.)	NCM	14	15	15	13	*17*	26
	NCOM	5	5	5	4	*7*	12
	ND	3	3	3	3	*3*	3
Model Size (Rel.)	$\frac{NumberofStateCharts}{NumberofClasses}$	0	0	0	0	*0*	0
	$\frac{NumberofOperations}{NumberofClasses}$	3.21	3.3	3.3	3.15	*3.23*	4.19
	$\frac{NumberofAssociations}{NumberofClasses}$	3.5	3.3	3.3	3.46	*3.17*	2.57
Reuse	Reuse Fraction(%)	0	95	93	93	*45*	25
	New (%)	100	5	7	7	*55*	75
	Unchanged (%)	0	85	75	40	*54*	85
	Changed (%)	0	14	15	40	*36*	10
	Removed (%)	0	1	10	20	*10*	5
Effort (h)	Global	18	5	11.5	6	*13.5*	37
	Hardware	6	2	4	1	*2*	8
	Requirements	0.5	0	0	0.5	*1*	1
	Design	2	0	0	1	*1.5*	3
	Implementation	7	2	5	2	*6*	18
	Test	2.5	1	2.5	1.5	*3*	7
Quality	Defect Density	7	0	2	1	*5*	7

5 Threats to Validity

The authors view this study as exploratory. Thus, threats limit generalization of this research, but do not prevent the results from being used in further studies.

Construct Validity. Reuse is a difficult concept to measure. In the context of this paper it is argued that the defined metrics are intuitively reasonable measures. Of course, there are several other dimensions of each concept. However, in a single controlled study it is unlikely that all the different dimensions of a concept can be captured.

Internal Validity. A maturation effect is caused by subjects learning as the study proceeds. The threat to this study is subjects learned enough from single runs to bias their performance in the following ones. An instrumentation effect may result from differences in the materials which may have caused differences in the results. This threat was addressed by keeping the differences to those caused by the applied method. This is supported by the data points as presented in table 1, 2, and 3. Another threat might be the fact that the studies were conducted at different institutes.

External Validity. The subjects were students and are, therefore, unlikely to be representative of software professionals. However, the results can be useful in an industrial context for the following reasons: Industrial employees often do not have more experience than students when it comes to applying MDD. Furthermore, laboratory settings allow the investigation of a larger number of hypotheses at a lower cost than field studies. Hypotheses supported in the laboratory setting can be tested further in industrial settings.

6 Summary and Conclusions

The growing interest in the Unified Modeling Language provides a unique opportunity to increase the amount of modeling work in software development, and to elevate quality standards. UML 2.0 promises new ways to apply object/component-oriented and model-based development techniques in embedded systems engineering. However, this chance will be lost, if developers are not given effective and practical means for handling the complexity of such systems, and guidelines for applying them systematically.

This paper shortly introduced the MARMOT approach that supports the component-oriented and model-based development of embedded software systems. A series of studies was described that were defined to empirically validate the effects of MARMOT on aspects such as reuse or quality in comparison to the Unified Process and an agile approach. The results indicate that by using MDD and CBD for embedded system development will have a positive impact on reuse, effort, and quality. However, similar to product-line engineering projects, CBD requires an upfront investment. Therefore, all results have to be viewed as initial. This has led to the planning of a larger controlled experiment to obtain more objective data.

References

[1] Atkinson, C., Bayer, J., Bunse, C., and others. Component-Based Product-Line Engineering with UML, Addison-Wesley, UK, 2001.
[2] Bunse, C., Gross, H.-G., Peper, C., Applying a Model-based Approach for Embedded System Development, 33rd SEAA, Lübeck, Germany, 2007.
[3] Bunse, C., Gross, H.-G., Unifying Hardware and Software Components for Embedded System Development, In: Architecting Systems with Trustworthy Components, Reussner, Staffort, Szyperski (Eds), Lecture Notes in Computer Science, Vol. 3938, Springer, 2006.
[4] Cantor, M., Rational Unified Process for Systems Engineering, the Rational Edge e-Zine, 2003, http://www.therationaledge.com/content/aug_03/f_rupse_mc.jsp.
[5] Crnkovic, I., Larsson, M. (Eds.), Building Reliable Component-Based Software Systems, Artech House, 2002.
[6] Douglass, B.P., Real-Time Design Patterns, Addison-Wesley, 2003.
[7] Briand, L.C., Bunse, C., Daly, J.W., A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs, IEEE TSE, 27(6), 2001
[8] Li, J., Conradi, R., Slyngstad, O.P.N., Torchiano, M., Morisio, M., Bunse, C., A State-of-the-Practice Survey of Risk Management in Development with Off-the-Shelf Software, IEEE Transaction on Software Engineering, 34(2), 2008
[9] Hruschka, P., Rupp, C., Agile SW-Entwicklung für Embedded Real-Time Systems mit UML, Hanser, 2002.
[10] Marwedel, P., Embedded System Design, (Updated Version), Springer, 2006.
[11] Object Management Group, UML Infrastructure and Superstructure, V2.1.2, 2007
[12] Szyperski, J., Component Software. Beyond OOP, Addison-Wesley, 2002
[13] Lange, C.F., Model Size Matters, Workshop on Model Size Metrics, 2006 (co-located with the ACM/IEEE MoDELS/UML Conference); October, 2006.
[14] Burkhard, J-M., Detienne, F., An Empirical Study of Software Reuse By Experts in Object-Oriented Design, INTERACT'95, Lillehammer Norway, June 27-29 1995
[15] Lee, N-Y., Litecky, C.R., An Empirical Study of Software Reuse with Special Attention to ADA, IEEE Transaction on Software Engineering, 23(9), 1997