Fraunhofer Institut Experimentelles Software Engineering

Proceedings of the

# First International Workshop on Quality Assurance in Reuse Contexts (QUARC 2004)

August 30, 2004, Boston, Massachusetts
In conjunction with the Third Software Product Line Conference

**Editors:**
Ronny Kolb
John D. McGregor
Dirk Muthig

IESE-Report No. 096.04/E
Version 1.0
August 30, 2004

A publication by Fraunhofer IESE

# Proceedings of the First International Workshop on Quality Assurance in Reuse Contexts (QUARC 2004)

## Organizers

Ronny Kolb
Fraunhofer Institute for Experimental
Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, GERMANY
kolb@iese.fraunhofer.de

John D. McGregor
Dept. of Computer Science
Clemson University
Clemson, SC 29634
johnmc@cs.clemson.edu

Dirk Muthig
Fraunhofer Institute for Experimental
Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, GERMANY
muthig@iese.fraunhofer.de

## Workshop Participants

Pekka Mäki-Asiala
VTT Technical Research Centre of Finland, VTT Electronics,
P.O.Box 1100
FIN-90571 Oulu, Finland
Pekka.Maki-Asiala@vtt.fi

Michael Eonsuk Shin
Department of Computer Science, Texas Tech University
Lubbock, TX 79409-3104
Michael.Shin@coe.ttu.edu

Grover G. Phillips
Engenio Information Technologies, Inc.
grover.phillips@engenio-it.com

Frank van der Linden
Philips Medical Systems QV-1
Veenpluis 4-6, PO Box 10000, 5680 DA Best
The Netherlands
frank.van.der.linden@philips.com

Tim Trew
Philips Research Laboratories
Cross Oak Lane, Redhill, RH1 5HA, UK
Tim.Trew@philips.com

Hideharu Teranishi
Ricoh Company Ltd.
1-15-5 Minami-Aoyama,
107-8544 Minatoku, Japan
Hiderahu.Teranishi@nts.ricoh.co.jp

Holger Diekmann
Robert Bosch GmbH

Rohit Sharma
Virtusa
2000 West Park Drive
Westborough, MA
rsharma@virtusa.com

Kiran Samartunga
Virtusa
2000 West Park Drive
Westborough, MA
kiran@virtusa.com

# Table of Contents

# 1 Introduction to Quality Assurance in Reuse Contexts

Ronny Kolb[1], John D. McGregor[2], and Dirk Muthig[1]

[1] Fraunhofer Institute Experimental Software Engineering, Sauerwiesen 6,
D- 67661 Kaiserslautern, Germany
christian.denger@iese.fhg.de

[2] Dept. of Computer Science, Clemson University, Clemson, SC 29634,
johnmc@cs.clemson.edu

Software development today faces several challenges. There is a critical need to reduce cost, effort, and time-to-market of software products, but, at the same time, complexity and size of products are rapidly increasing and customers are requesting more and more quality products tailored to their individual needs. A promising approach to address today's software development problems and to make the software development process more efficient is the systematic, large-scale reuse of software artifacts over multiple products. Recently, reuse-based software development paradigms such as component-based software development and software product lines have increasingly received attention not just in the software research community but even more in the software industry as they promise – and have shown – to shorten the development time of software systems and to reduce development and maintenance costs.

The potential benefits of all these approaches are based on the assumption that a significant portion of a new software product is built through the reuse of existing software components, but also other software artifacts such as architecture, design, or requirements. In order to achieve the promised improvements, however, a high level of quality of the artifacts intended for reuse is required. Therefore, more than for traditional software development, quality assurance becomes a crucial part of every reuse-based development effort. In fact, quality assurance is more critical for software product lines and other reuse-based software development paradigms since quality problems in an artifact not only lead to an end product with low quality but also propagate this low quality into all the products making use of it. Consequently, it is vital that before being reused the quality of all artifacts intended for reuse is assured using quality assurance techniques such as inspections and testing.

There are a number of specifics caused by software reuse such as variable usage of software components or genericity of artifacts, however, that have to be faced during quality assurance. In order to enable an organization to fully experience the expected efficiency gain through reuse, therefore, a quality assurance approach is required that enables to validate the products built from reusable artifacts as effective and as efficient as they are validated in a non-reuse context.

Despite the criticality of quality assurance and the special problems caused by reuse-based software development, however, research in the field of software product lines and component-based development has primarily focused on analysis, design, and implementation to date and only very few results address the quality assurance problems and challenges that arise in a reuse context. With the growing acceptance of reuse-based development paradigms such as software product lines, therefore, effective and efficient methods and techniques for ensuring the quality of reusable artifacts and products built by reusing existing artifacts are required.

The aim of this workshop is to establish a forum for the successful exchange of experience and ideas among practitioners and researchers, working together to improve the state-of-the-art and state-of-the practice in quality assurance for software product lines and other reuse-based software development approaches. The workshop will provide an opportunity for exchanging views, experiences, and lessons learned, advancing ideas, as well as discussing recent work and work in progress on topics dealing with quality assurance for software artifacts intended for reuse and products built using reusable artifacts. It intends to bring together researchers and practitioners from both academia and industry to share ideas on the foundations, techniques, methods, strategies, and tools of quality assurance for reuse-based software development paradigms.

The discussion of this workshop is particularly focused on the following issues:

- What are the implications of reuse-based software development paradigms such as software product lines or component-based development from the perspective of quality assurance?
- Are existing quality assurance approaches suitable with respect to reuse-based software development paradigms?
- How to improve existing quality assurance techniques and processes to be effectively and efficiently applicable in a reuse context?
- What is the best way to ensure the quality of software artifacts intended for the purpose of reuse?
- How can the quality of reusable components (in particular generic components used in product lines) be ensured efficiently and effectively?
- How to minimize the effort of ensuring the quality of products built from reusable artifacts?

- Which impact does the domain has on quality assurance techniques and strategies?
- When to use which quality assurance technique?
- Which quality assurance techniques should be applied to satisfy which quality attributes to the expected level?
- How do different variability implementation mechanisms influence quality assurance strategies?
- How to plan and prepare for quality assurance in a product line context?
- How should an organization invest its resources for quality assurance?
- How can costs and benefits be “traded-off” against resulting product quality?
- How can synergies between various quality assurance techniques be combined?

# 2 Inspections in Reuse Intensive Software Development Processes

Christian Denger[1] and Hideharu Teranishi[2]

[1] Fraunhofer Institute Experimental Software Engineering, Sauerwiesen 6,
D- 67661 Kaiserslautern, Germany
christian.denger@iese.fhg.de

[2] Ricoh Company LTD. 1-15-5 Minami-Aoyama, 107-8544 Minatoku
Tokyo, Japan, Hiderahu.Teranishi@nts.ricoh.co.jp

**Abstract.** High quality is an important goal for almost all software development projects. In the context of software product lines this goal is even more important. In such development approaches, the quality of the final system is a result of the quality of its comprising components. Moreover, reuse is an inherent element of product line engineering. Thus, the quality of each single component is of highest importance. Components that are built for reuse are even more important as a defect in such a component does not only affect the quality of the component but also the overall quality of all the systems that reuse this component. Inspections are one of the most effective and efficient quality assurance techniques but this quality assurance technique has so far not been tailored to the specific characteristics of reuse intensive systems. This paper discusses challenges and potential solutions on how make efficient use of limited inspection resources in a reuse intensive development context.

## 2.1 Introduction

Software developers strive to develop high quality software. In reuse intensive development approaches, such as software product lines, high quality of the single components that comprise a final system are of highest important [3]. The reason is that when reusing components in a product not only the components functionality and characteristics are reused but also the quality flaws contained in that component. In other words, defects in one component affect all those products in which that component is reused. Therefore, reusable components should achieve a highest level of quality in order to reduce the risk of defect propagation. Quality assurance techniques such as testing and inspections [4, 5, 2] have been developed to improve the quality of software products and components. However, so far these techniques are not tailored to the specials characteristics of reuse intensive systems [6].

One reason for this might be that it is assumed that traditional quality assurance approaches can be used within the context of product line engineering. This would imply that the challenging question of how to perform quality assurance in product line engineering companies is already solved. However, practice shows that this is not true. Product lines engineering imposes specific aspect that need to be addressed such as the modeling of variabilities and the use of generic components for reuse in specific applications. This aspect is not or insufficiently considered in existing quality assurance techniques. The challenges and the opportunities imposed by reusing components should be considered in tailored quality assurance techniques.

A crucial aspect of such tailored approaches is how a cost-effective enactment of quality assurance techniques might look like in the context of reuse. Every software development project has limited resources for quality assurance and therefore, these resources need to be spent in a most efficient and systematic way that is most likely to yield the best return on investment, and maximize the chances of successful defect

As software inspection are one of the most efficient software quality assurance techniques, especially for the early life-cycle phases [9], this technique should be in particular considered as a means to improve the quality in product line development approaches. In the following a balancing model for software inspection is discussed that tackles exactly the question on how to efficiently enact and perform software inspection to address the challenges and specialties imposed by the reuse intensive nature of product line engineering.

## 2.2 Inspections in Software Product Lines, Challenges and Chances

In the following the ideas of a balancing model for inspections are discussed. The following figure indicates the ideas of this model. The figure illustrates the overall issue of quality assurance in general and inspections in particular in a reuse centered development approach such as product line engineering. The inspections can be performed either on the generic or reusable components or within the projects that reuse a certain instantiation of the reusable components (white arrows).

Thus, a strategy is required on how to perform inspections as efficient as possible when reusing components. This imposes the following questions (green arrows):

- How to perform inspection on reusable components in a most effective and efficient way i.e. how to address issues such as variabilities and genericity?

- How to balance the effort for inspections of the reusable components and the inspections of the specific projects that reuse the generic components?

Beside these crucial questions and the related challenges the aspect of reusing components has also a positive effect. A reusable component is not only reused in one project but usually in several different ones. Thus, it is possible to continuously learn about the component's quality and to continuously improve the inspection approaches for generic components. The overall question with respect to this chance is:

- How to optimize inspections of generic components based on the knowledge of defects found on the instantiations of the components?

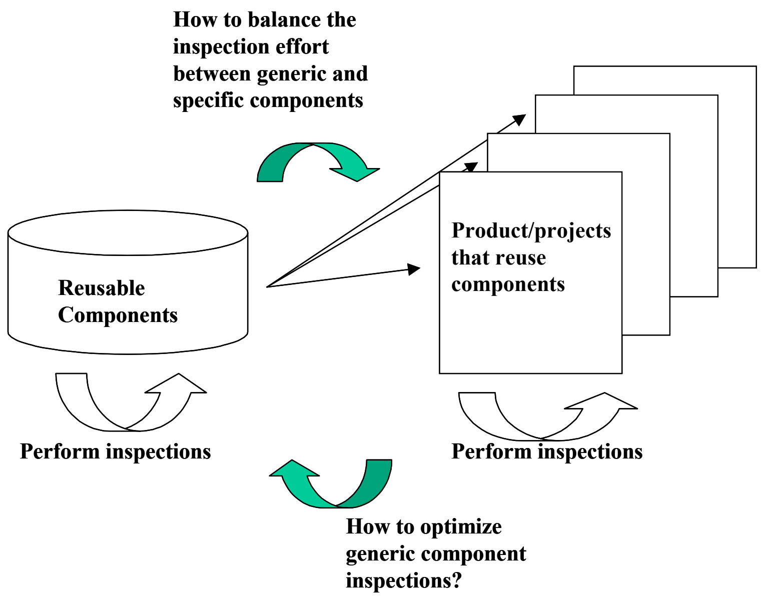

How to balance the inspection effort between generic and specific components
Reusable Components
Product/projects that reuse components
Perform inspections
Perform inspections
How to optimize generic component inspections?

Figure 2-1: Challenges of Inspecting in the Context of Reuse

## 2.3 Performing and Balancing Product Line Inspections

In the following some approaches are presented that should help to answer the questions mentioned above.

### *2.3.1 Addressing Genericity and Variabilities*

In order to address the question of how to cope with product line specific aspects such as variabilities and generic components, existing inspection approaches need to be tailored to these characteristics. Scenario-based approaches such as usage based reading [8] or perspective based reading [1, 7] proved to be the most efficient reading techniques in several application contexts.

The flexible nature of these approaches allows a tailoring to the specific requirements of product line inspections. The basic idea of perspective-based reading is that a software artifact is inspected from different viewpoint (perspectives). The selected perspectives represent relevant stakeholders of the software artifact that are interested in its quality. Mainly, these perspectives represent users of the object under inspection. For a component in general this could mean testers, implementer, code analysts, and maintainers. An inspector of each perspective gets a so-called reading scenario that provides active guidance on how to perform the inspection. Experience in applying this approach shows that the tester perspective seems to be the most efficient perspective. In that case the inspector has to develop test cases from the object under inspection and has to mentally simulate these test cases, which often reveals a lot of subtle defects. With these "traditional" perspectives it is possible to address typical quality criteria of a component such as its functional correctness, its consistency to other artifacts, its completeness and testability.

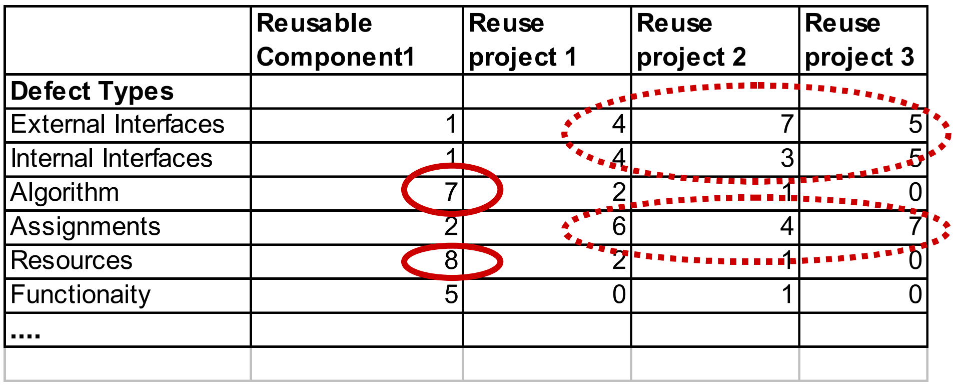
In the context of software product lines additional quality criteria are of importance such as maintainability, adaptability and reusability of a component. In a constructive way these aspects are achieved by variability modeling and the definition of generic components. In order to ensure that a component fulfills these aspect additional perspectives should be designed. The definition of a product line manager perspective and a product line architect perspective allow to focus the inspectors exactly on these aspects. The reading scenarios can be used to guide the inspectors in traversing variability resolutions through the component under inspection and to focus on the components reusability and adaptability by checking its interface definitions and by modeling some potential instantiations of the generic component to see whether all generic parts are reasonable.

### 2.3.2 *Balancing the Inspection Effort*

The second challenge that needs to be addressed is the efficient use of restricted inspection resources on the generic components and the specific projects that reuse these components. It is obvious that in an unsystematic inspection process one might redundantly inspect aspects of the reusable components, once on the generic component and later on the concrete instance. Therefore, a systematic planning process is essential. At first, two extreme approaches are thinkable. Either, the quality is checked only on the reusable components (i.e. the generic component) and no additional inspection of these components is performed during the specific product inspections. On the other end of the spectrum inspections are performed only on the product specific side, when the components are reused in a concrete context.

Both approaches have crucial drawbacks. Limiting inspection to the reusable components means that product specific aspects and the concrete instantiation of the component cannot be considered. This might lead to ineffective inspections as crucial defects due to the product specific instantiation are missed. Limiting the inspection to the product specific side leads to an inefficient inspection approach, as defects in a reusable component are propagated to all products and need to be corrected in all of these products. These drawbacks show that the inspection effort has to be balanced between inspections of reusable and reused components.

The basic assumption of this balancing model is that different type of defects and quality aspects can and should be addressed by inspecting the generic and the instantiated components. Based on this assumption it is then possible to measure which types of defects are easier to detect in each inspection and to balance the inspection effort in that way that those defects and qualities are addressed that are most easily found in the different inspections. The balancing model can be developed by a measurement program that evaluates the defect typed that are most efficiently found on the reusable components and on the specific project instantiations of the reused components. An initial hypothesis is, that all defect types that are related to the general quality of the component (e.g. functionality, internal consistency, algorithm, variable assignments) should be addressed on the reusable (generic) component. In addition reuse specific aspects such as modular design of the component, low coupling and high cohesion, reusability and integratability of the component should also be inspected on the reusable component. In the project specific inspection it is then recommended to focus on aspects such as the interfaces between reused and newly developed components and project specific changes that were implemented on the component. The following table shows how such a balancing model could look like.



| | **Reusable Component1** | **Reuse project 1** | **Reuse project 2** | **Reuse project 3** |
|---|---|---|---|---|
| **Defect Types** | | | | |
| External Interfaces | 1 | 4 | 7 | 5 |
| Internal Interfaces | 1 | 4 | 3 | 5 |
| Algorithm | 7 | 2 | 1 | 0 |
| Assignments | 2 | 6 | 4 | 7 |
| Resources | 8 | 2 | 1 | 0 |
| Functionaity | 5 | 0 | 1 | 0 |
| **....** | | | | |
| | | | | |

Figure 2-2: Excerpt of a balancing model

The entries in the table are derived over measurements on inspections performed on reusable components and reusing projects. The measurements indicate that the defect types Resources and Algorithm should be addressed on reusable components as these defect types are frequently identified in inspections of reusable components and almost not detected in the specific project. For the defect types Interfaces and Assignments the opposite is true (dotted lines). In almost all project these defect types are most frequently identified in product specific inspections.

For the measurement program the product line aspects and the inherent concept of reuse offers an important chance. As the reusable components are usually used in several products (and thus development project) it is possible to get enough data about which defect types are at best addressed when. In addition, from each project that reuses components and finds defects on them, it is possible to learn more how to optimize the inspections of the reusable inspections. An analysis method has to be implemented that analyzes a defect and decides whether this defect can be addressed in an inspection of the reusable component. The major benefit would be that such a defect can be detected and resolved already on the reusable component before it is propagated to all products that reuse the component.

All in all, the balancing model has several benefits:

- The inspection effort can be balanced between inspections of reusable components and project specific components by a systematic planning based on the model.
- Inspectors are focused on special defect types and quality aspects during inspection of reusable components and project specific inspections, which reduces the overlap and redundant checks. This also contributes to an improved efficiency of the inspection.
- The model comprises a continuous learning cycle which makes use of the inherent reuse concept of software product lines and thus helps to continuously optimize the balancing model.

## 2.4 Conclusion

The paper presented an approach for tailoring inspections to the context of reuse intensive development paradigms such as product line engineering. It showed how perspective based reading can be used to address special quality aspects imposed by a reuse intensive development paradigm. Perspective-based inspections proofed to be a highly efficient and effective defect detection technique that outperforms other inspection techniques such as checklists or experience (ad-hoc) inspections.

Moreover the paper showed that current inspection approaches do not consider the reuse aspect in a sufficient way and thus highly valuable development effort is wasted. By focusing the inspection on defects that are most efficiently found in inspections of reusable components helps to reduce this expensive drawback of recent inspection approaches. The focusing of the inspection process is at best guided by a balancing model for inspections in product line environments. This model shows that different defect types should be addressed in different phases (i.e. inspections of the reusable components and inspections on the specific products).

## References

[1] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, and M. Zelkowitz. The Empirical Investigation of Perspective-based Reading, Empirical Software Engineering 1 (1996) 133–164.

[2] B. Beizer. Software Testing Techniques. Second Edition, Van Nostrad Reinhold, New York, 1990.

[3] P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, August 2001.

[4] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. IBM System Journal, 15 (3); 1976.

[5] Thomas Gilb, Dorothy Graham. Software Inspections. Addison-Wesley Publishing Company, 1993.

[6] R. Kolb and D. Muthig, "Challenges in Testing Software Product Lines". CONQUEST 2003, Nuremberg, Germany.

[7] Oliver Laitenberger. Cost-effective Detection of Software Defects through Perspective-based Inspections. PhD Thesis in Experimental Software Engineering; Fraunhofer IRB Verlag, 2000.

[8] T. Thelin, P. Runeson, C. Wohlin. An Experimental Comparison of Usage-Based Reading and Checklist-Based Reading. IEEE Transactions on Software Engineering, 29-8 (August 2003), 687-704.

[9] Karl Wiegers. Peer Reviews in Software – A practical Guide. Addison-Wesley, 2002.

# 3 Improving Efficiency of Testing with Test Reuse: Development of Reusable Test Assets

Annukka Mäntyniemi and Pekka Mäki-Asiala

VTT Technical Research Centre of Finland, VTT Electronics, P.O.Box 1100,
FIN-90571 Oulu, Finland
{Annukka.Mantyniemi, Pekka.Maki-Asiala}@vtt.fi

**Abstract.** While systematic reuse promises to shorten software development time, testing may become a bottleneck of otherwise efficient software development process. This is because software testing practices have not advanced to the extent that of software reuse techniques and processes. Test asset reuse could provide similar efficiency gains as expected from software reuse. This paper proposes a tentative approach for development of reusable test assets. In this approach, software reuse techniques and principles are applied into software testing context.

## 3.1 Introduction

Software reuse has been practiced for decades, evolving from ad-hoc code reuse to today's component-based software engineering and product line engineering approaches. "Reuse is a simple concept", Basili et al. [1] state, "use the same thing more than once". However, as they say and experiences from industry [e.g. 11] prove, it is nothing but simple in practice. Although being a complicated process, promises of software reuse are tempting. Major benefits expected from software reuse are, for example, shortened time-to-market and higher quality of the software products [7], [9].

Unlike software reuse, test reuse is a fairly new and unexplored field of study. Demands for software testing and quality assurance are ever-increasing as the size and complexity of the systems expand and the markets demand higher quality and shorter development times. Also the software reuse itself sets demands for testing practices which are forced to keep up with the pace.

The amount of effort consumed by software testing varies from 30 to 50 per cent in a typical software development project [6]. Therefore, improving the efficiency of testing through test asset reuse could provide remarkable savings. Adhering to the definition of IEEE Std 1517-1999 [7] for an asset, test asset is defined to mean any test item (e.g. test case, test step, test specification) that is designed for reuse in multiple contexts. The expectations towards test reuse can be even higher than promised from software reuse, as in addition to reusing test assets in testing different products (e.g. product family), tests could be reused in different testing phases (e.g. unit and integration), types (e.g. functional and performance) and in regression testing.

In a typical software development project, the specifications and designs for the tests are developed concurrently with the design phases of the actual software, binding the tests tightly to the system under test. How to develop test assets that could be reused in variety of contexts?

The question has been approached earlier in the study of Korhonen et al. [10] that introduces a concept of feature-based testing. However, this study concentrates on reuse of tests in regression testing, which is only one of the possible targets for test reuse. The most extensive test reuse studies seem to be done in the telecommunication domain in the middle of 90's [4], [5]. Also in these studies the reuse target is quite limited, focusing only on protocol testing.

This paper proposes a tentative approach for development of reusable test assets. When compared to the earlier studies [4], [5], [10], this approach utilizes more extensively methods and practices known from software reuse and product line engineering, e.g. [2], [7], [8], [9], [13], [14], and provides a wider tour throughout the process of test development for reuse.

The rest of this paper is composed as follows: Section 3.2 introduces the approach for test asset development, Section 3.3 gives an example of a reusable test case and Section 3.4 draws conclusions on this paper.

## 3.2 Test Asset Development

Like software reuse [7], [9], test reuse can be divided to test development *for* and test development *with* reuse sides as illustrated in Figure 3-1. The specifications for the software systems form the basis of the test asset specifications. Based on these specifications reusable test assets are designed and implemented *for* reuse and reused later in the *with* reuse phase. However, unlike the *with* reuse approach of software development, which is mainly interested in building something new, the purpose of test development *with* reuse is to find errors or to give some level of assurance that the software meets its specification. Reusing test assets provides a way to speed up this process.

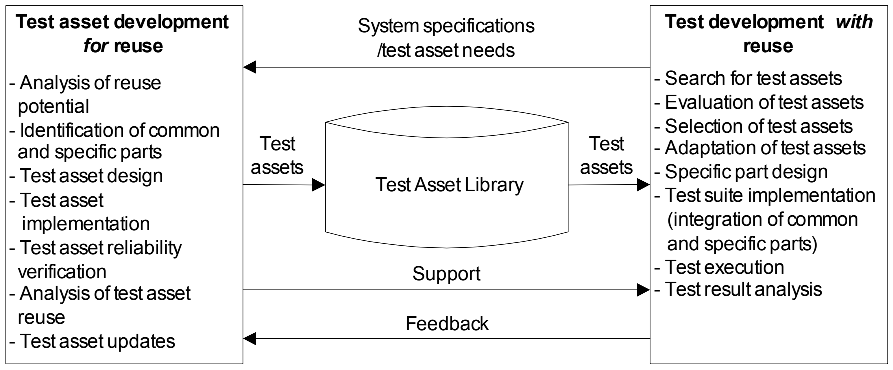

Test asset development *for* reuse
- Analysis of reuse potential
- Identification of common and specific parts
- Test asset design
- Test asset implementation
- Test asset reliability verification
- Analysis of test asset reuse
- Test asset updates
System specifications /test asset needs
Test assets
Test Asset Library
Test assets
Support
Feedback
Test development *with* reuse
- Search for test assets
- Evaluation of test assets
- Selection of test assets
- Adaptation of test assets
- Specific part design
- Test suite implementation (integration of common and specific parts)
- Test execution
- Test result analysis

Figure 3-1: Test asset development *for* and *with* reuse

According to Karlsson [9], in development *for* reuse, the main aim is to identify potential reusers with similar requirements, and to analyze the variations between their requirements. Domain analysis is the technique that can be applied for this purpose and it is a focal practice in software reuse. According to IEEE Std 1517-1999 [7] domain analysis is: "(A) The analysis of systems within a domain to discover commonalities and differences among them. (B) The process by which information used in developing software systems is identified, captured, and organized so that it can be reused to create new systems within a domain. (C) The result of the process in (A) and (B)" [originally in 12]. The generality may be expressed by using different component compositions or by designing an architecture that captures commonalities between different products and variability mechanisms that allow component customization when needed [13].

As software components, also test assets need to be sufficiently general to be reused in variety of contexts. This means that test developers must understand the common and the differing software items or features of the products to be tested, and design test assets for testing the identified common items or features. At the beginning of test asset development, the reuse potential of tests needs to be evaluated. It should be analyzed if the testing targets, i.e. software items and features, are specific to only one product to be tested or if they are common to multiple products. In the latter case, building of reusable tests is justifiable.

"First and the most important criterion for reusability is functionality" states Karlsson [9] meaning that it does not matter how well non-functional criteria are fulfilled if the reusable asset does not fulfill the functional needs. In test development for reuse, test assets should capture the right functionality. Defining this functionality may be difficult as similar features to be tested may be implemented differently in different products or common software items may have dependencies on other elements of a system. However, reuse of test assets is probably most productive when done concurrently with software reuse. In systematic software reuse, domain analysis models and architectural designs should address the common software items between different products. This makes generality analysis for test assets easier as their development can be targeted to cover the already identified common software items.

Figure 3-2 illustrates the possibility of splitting a test suite into common and specific parts. The common part concentrates on tests of items or features that are commonly included in the products to be tested, whereas the specific part concentrates on the tests of items that change for each product. This splitting follows the one described in EWOS/ETG 022 [5].

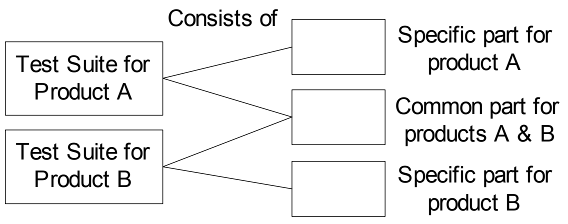

Consists of
Test Suite for Product A
Test Suite for Product B
Specific part for product A
Common part for products A & B
Specific part for product B

Figure 3-2: Common and specific parts of a test suite

This splitting idea is close to the domain analysis and architectural design phases of software development *for* reuse. One central criterion in software development *for* reuse is a modular architecture [9], [15]. Good modular architectures make dependencies explicit and lead to natural distribution of responsibilities [15]. In test asset design, principles of modularization should be followed to facilitate test assets' adaptability, understandability and maintainability. ETSI/ETR 190 [4] gives several guidelines for test modularization. Splitting a test suite into common and specific parts corresponds to organizational level modularization. This level can be applied to define the best partitioning of modules in order to minimize the development effort. Other levels of modularization are functional and language level. In the functional level, test suites are partitioned into several functional parts. This modularization is performed through the tree organization of the test cases. The language level is a naïve class of criteria that is based on a classification of language elements by the categories they belong to, such as constant declarations, test cases, test steps, defaults, etc. To facilitate reusability in modularization, structural complexity should be low [9], meaning that relationships between modules are avoided or kept simple and easily understandable.

In addition to capturing commonalities in the reusable test asset, it should be considered, how the diversity between the software items or features to be tested is addressed. Taking the diversity into account in test asset design makes test assets more reusable as reusers are able to customize them to their specific needs. Variability mechanisms provide the means to address the diversity between the products [13]. Variability is the ability to change or customize a system [14]. Those parts of the component that vary across contexts can be separated from the component itself at selected plug-points where that variation can be encapsulated and localized [3]. These variation or plug-points are places where the behavior of the component can be changed. A variation point represents a delayed design decision by providing possibilities for clients to create their own unique variants of a component [2].

In test asset development, the use of variability mechanisms depends on the test implementation language. According to D'Souza and Wills [3], reusable components can be built without object-oriented techniques, but it is a lot easier if these techniques are used. Jacobson et al. [8] define variability mechanisms that could be used to implement variability in test assets as well as in software components. These mechanisms are: parameterization, which is used if there are several small variation points for each variable feature; inheritance, which is applied if a method needs to be implemented for every application or if an application needs to extend a type with additional functionality; uses that applied in reusing an abstract use case to create a specialized use case; and extensions and extension points that are used, if parts of a component need to be extended with additional behavior selected from a set of variations for a particular variation point. In addition, several languages and tools can be used to implement variability [8][14].

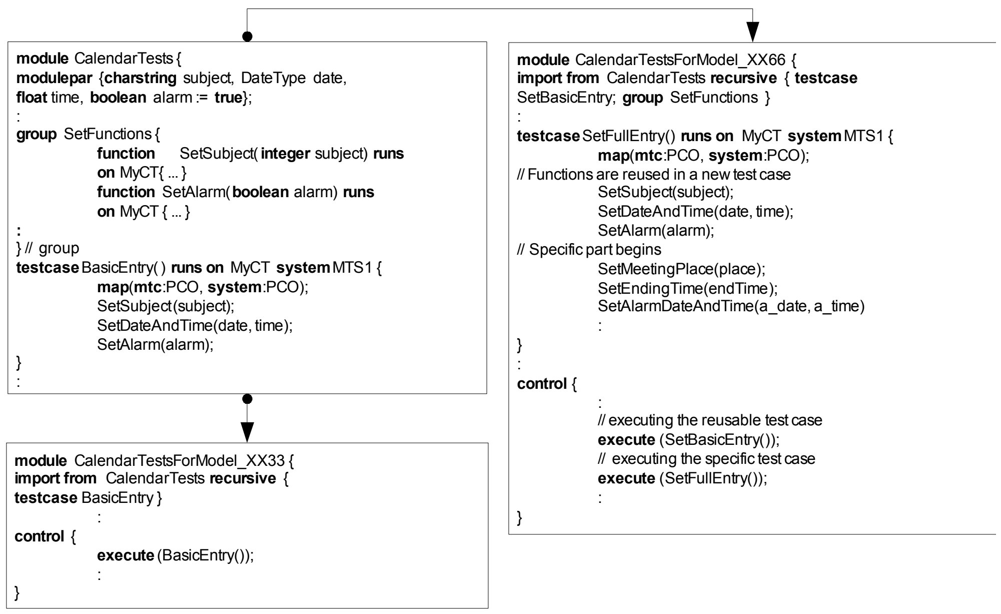
Test assets are implemented according to their designs. To facilitate reusability, the code should be self-descriptive [9], meaning that it should be commented to explain how the functionality is implemented. If possible, tests are executed in order to verify test assets reliability, i.e. that they function as designed. In practice, however, test asset's reliability is evaluated when it is executed in a real testing context in the *with* reuse side of test development.

Test assets are stored into a test asset library together with the documentation supporting reuse. Documentation includes information, for example, about the purpose of the test asset, suitable reuse contexts, and guidelines how to take test asset into use (e.g. interface descriptions). In addition, test asset developers provide support for the with reuse test developers. After the test asset has been used in testing, the *with* reuse side gives feedback to the test asset developers. Feedback is analyzed and it may lead to updates or development of new assets. The *for* reuse side maintains the test assets.

## 3.3 An Example: a Reusable TTCN-3 Test Case

The example presented in this paper is a simplified one. Today consumers are able to choose their mobile phones from a wide range of products. Even though more expensive phones will typically offer more features and more elaborate functions, some of these features are similar, and some of the functions are the same. Clearly, there is a possibility to use the same tests to validate these commonalities.

A short cursory example of a reusable test case implemented with TTCN-3 language is illustrated in Figure 3-3. This example presents a fictitious basic calendar test for testing the calendar feature found commonly in modern phones. In module *CalendarTests* test case *BasicEntry* is realized with the help of few functions and module parameters. This test case uses three functions to stress some of the basic functionalities found in phone calendars. The two other modules *CalendarTestsForModel_XX33* and *_XX66* respectively, represent a low-end and a high-end phone models of a phone vendor. The low-end model has only the basic functionality and therefore can reuse and execute the test case 'as is'. The high-end model can reuse the test case 'as is' but also take some parts of it and utilize them in a new test case.



```
module CalendarTests {
modulepar {charstring subject, DateType date,
float time, boolean alarm := true};
:
group SetFunctions {
            function    SetSubject( integer subject) runs
            on MyCT{ ... }
            function SetAlarm( boolean alarm) runs
            on MyCT { ... }
:
} // group
testcase BasicEntry( ) runs on MyCT system MTS1 {
            map(mtc:PCO, system:PCO);
            SetSubject(subject);
            SetDateAndTime(date, time);
            SetAlarm(alarm);
}
:
```

```
module CalendarTestsForModel_XX33 {
import from CalendarTests recursive {
testcase BasicEntry }
            :
control {
            execute (BasicEntry());
            :
}
```

```
module CalendarTestsForModel_XX66 {
import from CalendarTests recursive { testcase
SetBasicEntry; group SetFunctions }
:
testcase SetFullEntry() runs on MyCT system MTS1 {
            map(mtc:PCO, system:PCO);
// Functions are reused in a new test case
            SetSubject(subject);
            SetDateAndTime(date, time);
            SetAlarm(alarm);
// Specific part begins
            SetMeetingPlace(place);
            SetEndingTime(endTime);
            SetAlarmDateAndTime(a_date, a_time)
            :
}
:
control {
            :
            // executing the reusable test case
            execute (SetBasicEntry());
            // executing the specific test case
            execute (SetFullEntry());
            :
}
```

Figure 3-3: A reusable TTCN-3 test case

## 3.4 Conclusions

This paper introduced a tentative approach for development of reusable test assets and an example of a reusable TTCN-3 test case. The approach is based on applying reuse techniques and principles into software testing context. The approach is an intermediate result of a TT-Medal project that is going on at Technical Research Centre of Finland (VTT Electronics). It is currently being enhanced with a process perspective including both development *for* and *with* reuse sides. The project also studies possible test reuse targets (e.g. test reuse between different testing phases) and how TTCN-3 language suits for development of reusable test assets. During the latter year of the two year project, the approach for development and utilization of reusable test assets will be evaluated in a case study in cooperation between VTT Electronics and an industrial partner.

## References

[1] Basili, V., Caldiera, G. & Cantone, G. A Reference Architecture for the Component Factory. ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, (1992), 53–80.

[2] Bosch, J. Design and Use of Software Architectures. Adopting and Evolving a Product-line Approach. Harlow: Addison-Wesley (2000)

[3] D'Souza, D. F. & Wills, A. C. Objects, Components, and Frameworks with UML, The Catalysis Approach. USA: Addison-Wesley (1998)

[4] ETSI/ETR 190. Methods for Testing and Specification (MTS); Partial and multi-part Abstract Test Suites (ATS); Rules for the context-dependent reuse of ATSs (1995)

[5] EWOS/ETG 022. Test specifications for embedded protocols in application profiles (1992).

[6] Graham, D., Herzlich, P. & Morelli, C. CAST report, Computer aided software testing, 3rd ed. London: Cambridge Market Intelligence (1995)

[7] IEEE Std 1517-1999, IEEE Standard for Information Technology - Software Life Cycle Processes - Reuse Processes. USA, New York: IEEE (1999)

[8] Jacobson, I., Griss, M. & Johnsson, P. Software Reuse - Architecture, Process and Organization for Business Success. New York, USA: Addison-Wesley (1997)

[9] Karlsson, E-A. Software Reuse, A Holistic Approach. England: John Wiley & Sons Ltd (1995)

[10] Korhonen, J., Salmela, M. & Kalaoja, J. The reuse of tests for configured software products. VTT Publications 406. Espoo, Finland: Technical Research Centre of Finland. (1999)

[11] Morisio, M., Michel, E. & Tully, C. Success and Failure Factors in Software Reuse. IEEE Transactions on Software Engineering, Vol. 28, Issue 4, (2002), 340–357

[12] NIST Special Publication 500-222, Glossary of Software Reuse Terms (1994)

[13] Ommering, R. & Bosch, J. Widening the Scope of Software Product Lines – From Variation to Composition. Proceedings of the Second International Conference on Software Product Lines. Heidelberg: Springer LNCS, (2002), 328-347

[14] Svahnberg, M., Bosch, J. Issues Concerning Variability in Software Product Lines. Proceedings of the Third International Workshop on Software Architectures for Product Families, Heidelberg, Germany: Springer LNCS, (2000), 146-157

[15] Szyperski, C. Component Software, Beyond Object-Oriented Programming. England, Harlow: Addison Wesley Longman Limited (1997)

# 4 Quality Assurance in a Software Product Line

John D. McGregor

Dept. of Computer Science, Clemson University, Clemson, SC 29634, johnmc@cs.clemson.edu

**Abstract.** Quality is a multi-faceted concept whose very definition varies from one product to another. Software product lines have the potential to achieve very high levels of quality, but quality is not guaranteed merely by adopting the product line practices. Techniques and processes must be designed with a specific view of quality in mind. We describe an operational view of quality and how this may be incorporated into the product production process of a software product line organization.

## 4.1 Introduction

Quality is a nebulous concept that changes with the situation and the observer. While correctness is often the naïve definition of quality, it usually is just the starting point for what is needed for most products to be viewed as quality products. Recently, several groups have put forward the notion that "quality" is actually a composite of a number of attributes. These are referred to in one context as quality attributes [bass].

In a reuse environment, such as a software product line, quality assurance is even more critical than in traditional development because whatever the level of quality, it will broader implications than in one-off development. A software product line organization integrates business and technical aspects of planning, training, and process definition that influence the levels of quality for the organization's outputs.

The economics of software product lines should make it possible to expend more resources to produce higher quality, but simply expending more does not necessarily result in higher quality. Traditionally, quality assurance requires three essential elements.

1.) Quality assurance has to be a small voice in the head of every person on the project.
2.) Personnel must have adequate time and resources to do quality work.

3.) Tools, techniques, and processes should be defined in such a manner that quality work is a natural result rather than requiring extraordinary effort.

These three items show that quality is the result of commitment. Commitment on the part of the business and technical managers and the engineers, rather than just expenditures. The comprehensive nature of a software product line organization provides an environment conducive to quality but it is not guaranteed.

Some organizations view testing as their quality assurance while others separate testing from quality assurance departments. They view quality assurance as only those things that are done without touching code such as design reviews and inspections. We will first describe a viewpoint about quality processes and then we will focus on an inspection process for software product lines that bridges the gap between testing and quality assurance.

## 4.2 Quality Assurance Processes

Traditionally quality assurance is set up as a separate process that parallels the product production process, as illustrated in Figure 4-1. Many quality standards, such as the ISO-9000 family, would seem to support this approach by the ways in which they define the activities that constitute their process. There is good reason to do this. It emphasizes the independence of the organization which may be empowered to stop production if quality goals are not met. However, this arrangement often emphasizes the isolation of quality assurance from the realities of production.

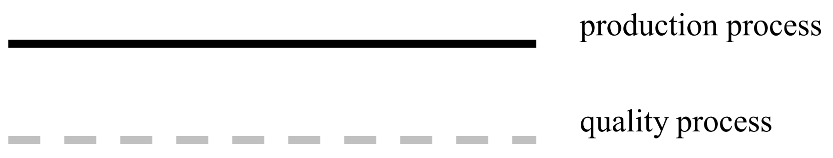


Figure 4-1: Isolated quality and production processes

Quality assurance is a cross-cutting concern that is a part of each production step. It should be woven into the product production process as illustrated in Figure 4-2. Every step in the production process definition should include activities that assure the quality of the outputs of the step.

product production
quality process

Figure 4-2: Integrated Quality and Production Processes

## 4.3 Guided Inspections

Inspection is a technique for examining designs, code, or documents produced in a development effort. Fagan described a procedure for code inspections that walked through the code in a sequential manner [Fagan 86]. Parnas showed that inspections could be improved by engaging the inspectors more fully with the artifacts being inspected [Parnas 85]. Techniques such as directed reading have been shown to be more effective than standard inspection techniques [Thelin 03], [Travassos 99].

Guided inspection is a technique developed by Luminary Software to increase the effectiveness of the typical inspection process. We developed this technique when we realized that existing inspection techniques only examined what is there and do not find what has been omitted. Guided Inspection is different in that the inspection is “guided” by test cases developed from the artifacts in the project rather than the inspector or a standard checklist.

For each guided inspection, a set of test cases is defined from the requirements and expectations for the artifact to be inspected. Constructing these test cases is an excellent way to focus the attention of the inspectors on the artifact they are about to inspect. The inspectors might begin with the use case model for the system and extract some of the scenarios to serve as test cases. Template test cases might be available for standard documents such as the system test plan.

Guiding the inspection process with test cases allows the inspectors to emphasize client priorities or to select the riskiest, or most critical, features to inspect most closely. Many use case templates provide for capturing risk information for each use case. The inspectors can use the usual case, alternative cases, and exceptional case scenarios for a highly critical use case while only using the usual case for a use that has low criticality.

As with any testing activity, “coverage”, the degree to which an artifact has been exhaustively explored, is an important measure. Using one scenario from each use case as the basis for test cases would achieve an “all use cases” level of coverage. By achieving high levels of coverage, inspectors are led naturally to identify necessary parts of the artifact that are missing. Tracking the degree to which coverage is achieved also maintains traceability between the requirements, the source of the test cases, and the portions of the inspected artifact that are touched by the test case.

### 4.3.1 Inspection Criteria for Software Product Lines

The models produced by a software product line organization present unique challenges. We first describe our criteria for individual product inspections and how they are applied in a product line context. Then we add an additional criteria.

**Correctness** is a measure of the accuracy of the model. When does a test case result in unexpected behavior? For a product line, behaviors that are correct in one product are not correct in another. Each test case requires a product scenario which is the context for the test.

**Completeness** is a measure of the inclusiveness of the model. Is it possible to write test cases that address concepts or operations not found in the model? For a product line, completeness covers all possible products. The product scenario is the context in which completeness is judged.

**Consistency** is a measure of whether there are contradictions within the model or between the current model and the model upon which it is based. Is it possible to write two test cases that are expected to produce the same result but that produce differing results? In a product line that is quite possible.

A software product line requires the additional consideration of **comprehensiveness**. We add this as a similar but slightly different notion of completeness.

### 4.3.2 Effectiveness of Inspections

All too often inspections are seen as hurdles to be cleared rather than opportunities for careful examination. Sometimes this is because it is not clear to the participants that the inspections are effective. The metrics used to evaluate inspections help shape the priorities of the inspection process. I once heard a manager bragging about the large number of lines of code that were inspected per hour in his inspections. Since then I have often wondered if that manager ever decided to skip reading the code altogether since that must have been the only thing slowing him down! Ultimately the effectiveness of the inspection process is measured by:

$$effectiveness = \frac{\text{the number of defects found by inspections}}{\text{number of defects found in the product over its lifetime}}$$

Although this value cannot be known immediately, a metric tracking program can support approximations based on actual data collected from previous projects. The denominator can also be taken to be the total number of engineer hours with the result x defects/hour.

## 4.4 Summary

The Guided Inspection technique has proven to be very effective part of a quality assurance effort. It provides early detection of defects in models from requirements models to detailed design models. Figures in the literature indicate that finding and fixing a defect at these early points in the life cycle costs 1% of what it would cost at system test time. The increased emphasis on MDA makes techniques such as Guided Inspection even more critical to the success of a product line. The technique does require effort but, as initiatives such as MDA push the boundary of automation, the effort to execute a model test case is being rapidly reduced.

# 5 Position – Quality Assurance in Reuse Contexts

Frank van der Linden

Philips Medical Systems QV-1, Veenpluis 4-6, PO Box 10000, 5680 DA Best, The Netherlands, frank.van.der.linden@philips.com

Quality is in the eye of the beholder. This means that every system developer, and every system user, may have different ideas on what is quality, and moreover they may have priorities on what they consider to be important quality.

In a reuse context the assets are reused over product borders, and that means that the way they deal with quality has to adapt over system border to new environments. Since different systems have different quality priorities, it may be the case that the asset solves different quality requirements, or solves them with the wrong priorities. For instance an asset may deal very well with security, but not with performance. This may be the wrong decision for a stand-alone system!

Certain architectures may be better than others to support certain quality. It may be useful to think about reusable quality architectures, in which other reusable assets fit. This may not be the final solution since the support of an architecture to a quality is only indirect. In fact, an architecture may support a certain quality, because it implements a certain solution to deal with some quality to a certain extent. This is often only qualitative. Moreover, a solution often supports a combination of more than one quality, whereas it may hamper the solution of another quality. A modular architecture may be good for maintenance, adaptability and flexibility, but it may hamper performance.

Many quality issues are emergent properties, which originate from the complete system configuration and implementation. It is often not easy to localize quality and assign them to specific parts of the system. This means that it is hard to decide whether a given asset supports a certain required quality well enough. There is presently no clear agreed upon way to describe them, and because the different qualities can be diverse, it may be the case that a complete description may be impossible, or require too much documentation effort, with too less pay back. The only way to solve this is to agree upon a small set of fixed solutions for quality requirements, and documenting the relationship of the asset on these solutions. Therefore in many cases test-runs and other forms of prototyping may be needed to get enough confidence in the support for the required qualities.

As a consequence, the following hard questions have to be solved for getting reusable quality assets.

1.) What does the developer need to know about a reusable asset, to be able to be able to deduce that the resulting system supports a certain quality to a certain extent?
2.) Can it be expected that a reusable asset developer knows what is required by all the users?
3.) Is it enough to restrict the reusable architecture to certain architectures only?
4.) Can this be supported by reusable architectures, or reusable architecture fragments?
5.) What are reusable architecture fragments or patterns?
6.) How to reuse other assets in reusable architecture fragments?
7.) How to combine reusable architecture fragments?
8.) Which kinds of assets are best supporting certain quality?
9.) How to relate asset attributes to quality?

As can be seen from the questions I have a lot of doubt about the possibility of general quality support at the reusable asset level. I like to discuss this. I have the opinion that architecture information is crucial to support qualities. In fact, it provides a certain way to satisfy the quality requirements. This may however enabling the quality support within a product-line setting. In that case it still has to be defined how reusable asset attributes can contribute to the quality.

It may be the case that architectures, or parts of them, can be made reusable as well. How to do that? Are patterns the way to proceed. Certain patterns seem to support certain solutions for certain quality requirements, for instance many patterns support flexibility.

# Consistency Checking in Multiple-View Meta-Models of Software Product Lines

Hassan Gomaa
*Dept. of Information and Software Engineering*
*George Mason University*
*Fairfax, VA 22030-4444*
*hgomaa@gmu.edu*

Michael Eonsuk Shin
*Dept. of Computer Science*
*Texas Tech University*
*Lubbock, TX 79409-3104*
*Michael.Shin@coe.ttu.edu*

## Abstract

*This paper describes an approach for quality assurance of software product lines by consistency checking in multiple-view UML meta-models. A multiple-view meta-model for software product lines describes how each view relates semantically to other views. The meta-model depicts life cycle phases, views within each phase, and meta-classes within each view. The relationships between the meta-classes in the different views are described. Consistency checking rules are defined based on the relationships among the meta-classes in the meta-model. These rules, which are specified formally using the Object Constraint Language (OCL), are used to resolve inconsistencies between modeling elements in the same view (intra-view consistency) or in different views (inter-view consistency), and to define allowable mappings between multiple views in different phases. Finally, tool support for the approach is described.*

## 1. Introduction

A multiple-view model [NKF94] of a software product line captures the commonality and variability among the software family members that constitute the product line. A better understanding of the software product line [CN02] can be obtained by considering the different perspectives, such as requirements modeling, static modeling, and dynamic modeling, of the product line. Using the UML notation, the functional requirements view is represented through a use case model, the static model view through a class model, and the dynamic model view through a collaboration model and a statechart model. While these views address both single systems and product lines, there is, in addition, a feature model view, which is specific to software product lines. This view describes the common and variant features of the product line.

Consistency checking between multiple views of a model is complex [NKF94], one of the reasons being the different notations that are needed. An alternative approach is to consider consistency checking between multiple views at the meta-model level, which uses one uniform notation instead of several. Furthermore, rules and constraints can be specified for the relationships between the meta-classes in the meta-model. This paper describes an approach for quality assurance of software product lines by consistency checking in multiple-view UML meta-models.

## 2. Multiple-View Models of Product Lines

A multiple-view UML model for a software product line defines the different characteristics of a software family [Parnas79], including the commonality and variability among the members of the family. The product line life cycle has three phases:

Product Line Requirements Modeling:
- Use Case Model View. The use case model view addresses the functional requirements of a software product line in terms of use cases and actors.

Product Line Analysis Modeling:
- Static Model View. The static model view addresses the static structural aspects of a software product line through classes and relationships between them.
- Collaboration Model View. The collaboration model view addresses the dynamic aspects of a software product line.
- Statechart Model View. The statechart model view, along with the collaboration model view, addresses the dynamic aspects of a software product line.
- Feature Model View. A feature model view captures feature/feature dependencies, feature/class dependencies, feature/use case dependencies, and feature set dependencies.

Product Line Design Modeling: During this phase, the software architecture of the product line is developed.

More information on multiple-view modeling for product lines is given in [GomaaShin02, Gomaa04].

## 3. Meta-Model for Software Product Lines

The meta-model describes the modeling elements in a UML model and the relationships between them. The meta-model is described using the static modeling notation of UML and hence just uses one uniform notation instead of several. Furthermore, rules and constraints are allocated to the relationships between modeling elements.

The multiple views are formalized in the semantic multiple-view meta-model, which depicts the meta-classes, attributes of each meta-class, and relationships among meta-classes.. A high level representation of the phases containing the views in this meta-model is shown in Fig. 1. A phase is modeled as a composite meta-class, which is composed of the views in that phase.

In the meta-class model, all concepts are modeled as UML classes. However, as the meta-classes have different semantic meaning, they are assigned stereotypes corresponding to the different roles they play in the meta-model. In Fig. 1, meta-classes representing the different views of a UML model are assigned the stereotype «view». Meta-classes representing development phases are assigned the stereotypes «phase» as they represent the different phases of the OO lifecycle, Requirements Modeling, Analysis Modeling, and Design Modeling.

Each view in Fig. 1 can be modeled in more detail to depict the meta-classes in that view. A view meta-class is a composite class that is composed of the meta-classes in that view. An example is given in Fig. 2, which depicts the meta-classes in the Class Model view and their relationships. Thus the Class Model view contains meta-classes such as class, attribute, relationship and class diagram, as well as the relationships between them.

Fig. 1 depicts underlying relationships among multiple views in development phases of a software product line:
Requirements phase:

- Use case model: This model describes the functional requirements of a software product line in terms of actors and use cases.

Analysis phase:
- Class model: This model addresses the static structural aspects of a software product line through classes and their relationships.
- Statechart model: This model captures the dynamic aspects of a product line through states and transitions.
- Collaboration model: This model addresses the dynamic aspects of a software product line by describing objects and their message communication.
- Feature model: This model captures the commonality and variability of a software product line by means of features and their dependencies.

The views in the Design phase are given in [Gomaa00].

## 3.1. Meta-Model Views

This section describes the meta-classes and their attributes, as well as the relationships between the meta-classes for the Class and Feature Model views in Fig. 2. Other views shown in Figure 1 are described in [Shin02].

Fig. 2 depicts meta-classes and relationships between the meta-classes for the class model view. A class diagram consists of classes and their relationships. A class may interact with an external class, such as an external input/output device or user interface. Each class may have attributes. Relationships between classes are specialized to aggregation, generalization/specialization, and association relationships. To capture variations of a software product line, the meta-model specializes a class to a kernel, optional or variant class. Kernel classes are

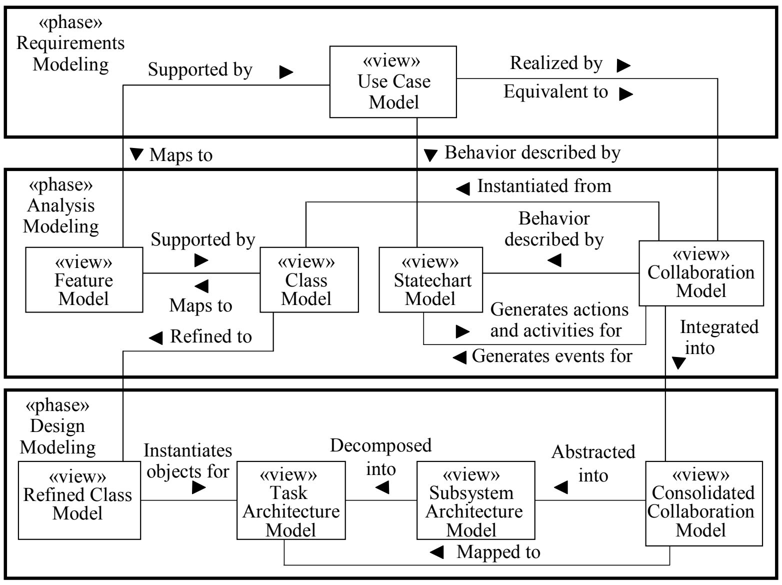


**Fig. 1. High-level relationships between multiple views for a software product line**

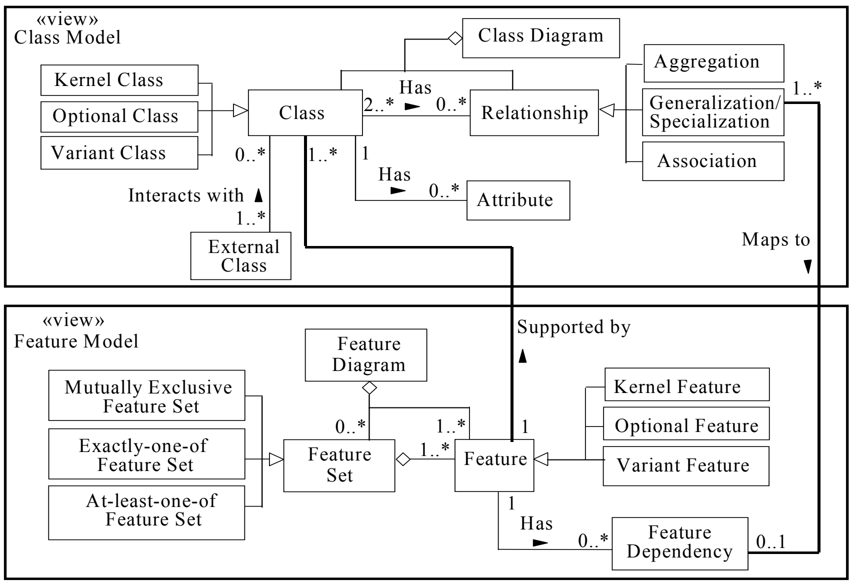


**Fig. 2. Meta-model for Class Model and Feature Model views in analysis phase**

required by all members of a software product line, whereas optional classes are required by only some members. Variant classes are required by the specific members to meet variations of kernel or optional classes. Fig. 3 depicts the meta-class attributes for the class model view. A class meta-class is classified by an application attribute (stereotype), whose possible values are control, algorithm, business logic, user interface or entity. A domain property of the class meta-class is captured through kernel, optional and variant class meta-classes.

Fig. 2 also depicts meta-classes and relationships between the meta-classes for the feature model view. A feature is an end-user functional requirement of an application system. Features are specialized to kernel, optional, and variant features depending on the characteristic of the requirements, that is, commonality and variability. Kernel features are requirements common to all members of systems, that is, required by all members of a product line. Optional features are required by only some members of a product line. A variant feature is an alternative of a kernel or optional feature to meet a specific requirement of some systems. Feature dependencies represent relationships between features, and feature sets refer to constraints on the choice of target features supported by a target system. A feature set is specialized to "mutually exclusive feature set," "exactly-one-of feature set," and "at-least-one-of feature set." In a mutually exclusive feature set, zero or one feature can be selected. An exactly-one-of feature set allows one and only one feature to be selected, whereas an one-or-more feature set permits one or more features to be selected. Fig. 4 depicts the meta-class attributes for the feature model view. Each feature dependency has a starting feature (fromFeature) and a destination feature (toFeature).

## 3.2. Relationships among Meta-Model Views

A meta-model for multiple views in each phase describes the relationships between the different views in each development phase. A meta-model for multiple views in a given phase of a software product line describes the relationships between different views in the same phase. The analysis phase of a software product line is viewed by means of the class model, collaboration model, statechart model, and feature model. Fig. 2 depicts a meta-model describing the relationships between the class and feature model views in the analysis phase. The relationships between the views are:

- A feature in the feature model is supported by classes in the class model.
- If there is a generalization/specialization relationship between two classes that support two different features respectively, the generalization/specialization relationship between two classes maps to a feature dependency between the two features.

A meta-model for multiple views in different phases of a software product line describes the relationships between the multiple views in the different phases. It shows how a meta-class in a view of a phase is mapped to a meta-class in the subsequent phase.

## 4. Consistency Checking between Multiple Views

Consistency checking rules are defined based on the relationships among meta-classes in the meta-model[MC01]. Model objects and their relationships in the multiple-view model are instantiated from meta-classes and their relationships in the multiple-view meta-model. The rules resolve inconsistencies between multiple views in the same phase or other phases, and define allowable mapping between multiple views in different phases. To maintain consistency in the multiple-view model, rules defined at the meta-level must be observed at the multiple-view model level. Consistency checking is used to determine whether the multiple-view model follows the rules defined in the multiple-view meta-model.

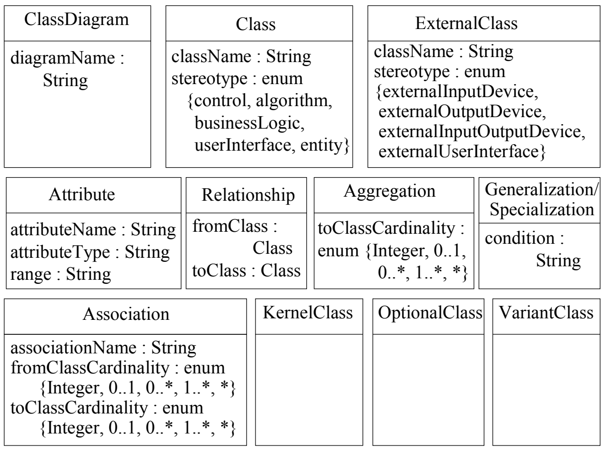

ClassDiagram
diagramName : String
Class
className : String
stereotype : enum {control, algorithm, businessLogic, userInterface, entity}
ExternalClass
className : String
stereotype : enum {externalInputDevice, externalOutputDevice, externalInputOutputDevice, externalUserInterface}
Attribute
attributeName : String
attributeType : String
range : String
Relationship
fromClass : Class
toClass : Class
Aggregation
toClassCardinality : enum {Integer, 0..1, 0..*, 1..*, *}
Generalization/ Specialization
condition : String
Association
associationName : String
fromClassCardinality : enum {Integer, 0..1, 0..*, 1..*, *}
toClassCardinality : enum {Integer, 0..1, 0..*, 1..*, *}
KernelClass
OptionalClass
VariantClass

**Fig. 3. Attributes of meta-classes in class model view**

Fig. 5 depicts consistency checking between a feature in the feature model and a class in the class model. Suppose an optional class “Class2” supports an optional feature “Feature2.” Class2 and Feature2 in the multiple-view model are respectively instances of Class and Feature meta-classes in the multiple-view meta-model. There is a relationship between Class and Feature meta-classes, which is “each optional class in the class model supports only one optional feature in the feature model.” For the multiple-view model to remain consistent, this meta-level relationship must be maintained between instances of those meta-classes, that is, Class2 and Feature2. Consistency checking confirms that each optional class in the class model supports only one optional feature in the feature model. Consistency checking rules are specified formally using the Object Constraint Language (OCL) [Warmer99], as well as informally in a natural language. The Object Constraint Language is a formal language that describes constraints on object-oriented models. A constraint is a restriction on one or more values of an object-oriented model.

As the multiple-view meta-model is developed sequentially, the consistency checking rules are derived from the meta-model. The rules from the meta-model for each view (in Section 3.1) are described first, then rules from the meta-model for the multiple views in each phase (in Section 3.2) are described and finally the rules from the meta-model for the multiple views in different phase (in Section 3.2) are described.

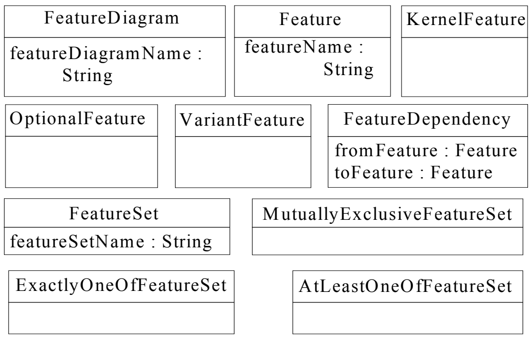


**Fig. 4. Attributes of meta-classes in feature model view**

## 4.1 Rules within One View

These rules describe consistency between meta-classes within each view, and include constraints on meta-classes. The rules derived from the class and feature meta-models in Fig. 2 are as follows:

1) A class must have a stereotype of “kernel,” “optional,” or “variant.” (Class Meta Model)
2) A class whose stereotype is “variant” must be a specialized class on a generalization/specialization hierarchy. (Class Meta-Model)
3) The stereotype of each feature must be “kernel,” “optional,” or “variant.” (Feature Meta-Model)
4) The stereotype of a feature set must be one of “Mutually Exclusive Feature Set,” “Exactly One-of Feature Set,” and “At-least-one-of Feature Set.” (Feature Meta-Model)
5) Zero or one feature must be selected from a mutually exclusive feature set. (Feature Meta-Model)
6) One or more features can be selected from an at-least-one-of feature set. (Feature Meta-Model)
7) Exactly-one-of Feature Set Constraint: One and only one feature can be selected from an exactly-one-of feature set. (Feature Meta-Model).

An example of these rules is “the stereotype of each class in the class model must be kernel, optional, or variant” (Fig. 2). This consistency checking rule is specified using OCL as follows:

```
context Class inv:
   self.allInstances->forAll(oclType = KernelClass or
     oclType = OptionalClass or oclType = variantClass)
```

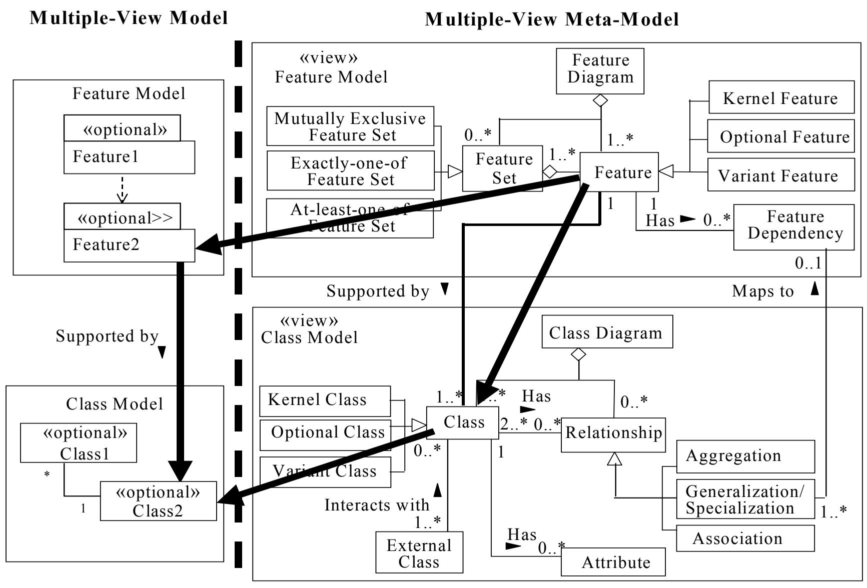


**Fig. 5. Meta-model for feature and class model views**

In this OCL specification, oclType is the type of Class instance, that is, KernelClass, OptionalClass or VariantClass.

Another example of these rules is "a class whose stereotype is variant must be a specialized class on a generalization/specialization hierarchy" (Fig. 2). This rule is specified using OCL as follows:

**context** VariantClass **inv**:
self.relationship->select( oclType = Generalization/Specialization **and** toClass = self.className)->size() = 1

In this OCL specification, the toClass is an attribute of Generalization/Specialization meta-class, which is inherited from the Relationship meta-class in the class model (Fig. 3). The className in "self.className" is an attribute of the Class meta-class.

## 4.2 Rules between Views

These rules address consistency between the different views in a given phase or in different phase. The rules derived from relationships between the class meta-model and the feature meta-model in the analysis phase (in Fig. 2) are:

1) Each "kernel" class in the class model must support only one kernel feature.
2) Each "optional" class in the class model must support only one "optional" feature.
3) Each "variant" class in the class model must support only one "variant" feature.
4) A kernel feature must be supported by at least one kernel class.
5) An optional feature must be supported by at least one optional class.
6) A variant feature must be supported by at least one variant class.
7) If there is a generalization/specialization relationship between two classes that support two different features respectively, the generalization/specialization relationship between two classes must map to a feature dependency between the two features.

An example of these rules is "each kernel class in the class model must support only one kernel feature" (Fig. 2), which is specified using OCL as follows:

**context** KernelClass **inv**:
self.feature->select(oclType = KernelFeature)->size() =1

Another example is a consistency checking rule, "if there is a generalization/specialization relationship between two classes that support two different features respectively, the generalization/specialization relationship between two classes must map to a feature dependency between the two features" (Fig. 2). This rule is specified using OCL as follows:

**context** Generalization/Specialization **inv**:
(self.class->size() = 2 **and** self.class->forAll(c1, c2 | c1.feature.featureName <> c2.feature.featureName))
**implies**
self.class->forAll(c1, c2 | ((c1.className = self.fromClass **and** c1.feature.featureName = c1.feature.featureDependency.fromFeature) **and**

```
(c2.className = self.toClass and
c2.feature.featureName =
c1.feature.featureDependency.toFeature)) or
((c2.className = self.fromClass and
c2.feature.featureName =
c2.feature.featureDependency.fromFeature) and
(c1.className = self.toClass and
c1.feature.featureName =
c2.feature.featureDependency.toFeature)))
```

In this OCL specification, the fromClass in "self.fromClass" and the toClass in "self.toClass" are attributes of the Generalization/Specialization meta-class in the class model (Fig. 3), which are inherited from the Relationship meta-class. The fromFeature in c1.feature.featureDependency.fromFeature and c2.feature.featureDependency.fromFeature is an attribute of the FeatureDependency meta-class in the feature model (Fig. 4). And the toFeature in c2.feature.featureDependency.toFeature and c1.feature.featureDependency.toFeature is an attribute of the FeatureDependency meta-class.

The consistency checking rules are specified in more detail in [Shin02].

## 5. Tool Support for Consistency Checking

In order to support consistency checking between multiple views, a proof-of-concept prototype, the Product Line UML Based Software Engineering Environment (PLUSEE) was developed. The scope of this proof-of-concept prototype includes the domain engineering phase. A domain model addressing the multiple views of a software product line is developed and checked for consistency between the multiple views.

Fig. 6 depicts the proof-of-concept prototype. A domain engineer captures a multiple-view domain model consisting of use case, collaboration, class, statechart, and feature models through the Rose tools, which save the model information in a Rose MDL file. From this MDL file, the domain model relations extractor extracts domain relations, which correspond to the meta-classes in the meta-model. Through the domain relations extractor, a multiple-view model is mapped to domain model relational tables. Using these tables, the consistency checker checks for consistency of the multiple-view model by executing the consistency checking rules described in Section 4. Tool support is described in more detail in [GomaaShin04]

## 6. Conclusions

This paper has described an approach for quality assurance of software product lines by consistency checking in multiple-view UML meta-models. The meta-model depicts life cycle phases, views within each phase, and meta-classes within each view. Consistency checking rules have been described based on the relationships among the meta-classes in the meta-model. These rules, which are specified formally using the Object Constraint Language (OCL), are used to resolve inconsistencies within each view and between multiple views, and to define allowable mapping between multiple views in different phases.

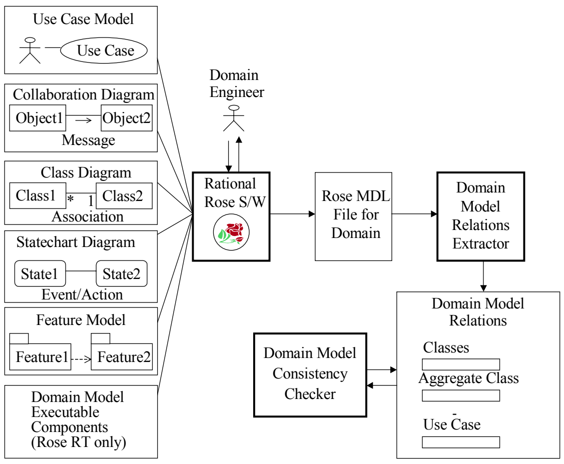
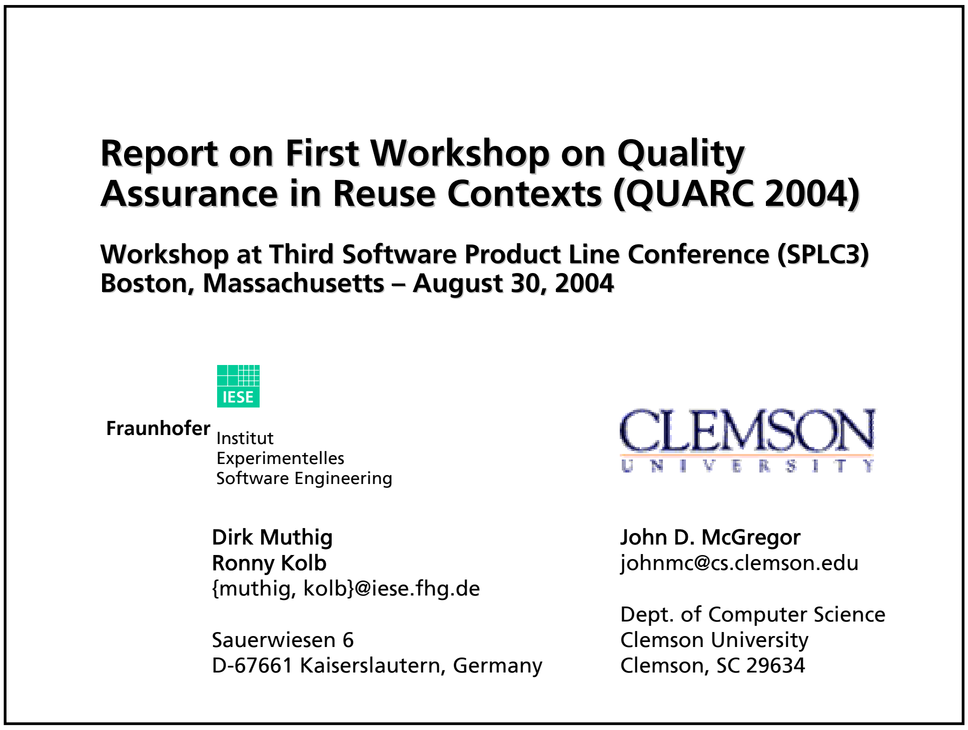


**Fig. 6. Product Line UML Based Software Engineering Environment (PLUSEE)**

## References

[CN02] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns, Addison Wesley, 2002.
[Gomaa00] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML," Addison-Wesley, 2000.
[Gomaa04] H. Gomaa, H. Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures, Addison-Wesley, July 2004.
[GomaaShin02] H. Gomaa and M. E. Shin, "Multiple-View Meta-Modeling of Software Product Lines" Proc. IEEE Intl Conf on Eng of Complex Computer Systems, MD, Dec 2002.
[GomaaShin04] H. Gomaa and M. E. Shin, "A Multiple-View Meta-Modeling Approach for Variability Management in Software Product Lines" Proc. IEEE Intl Conf on Software Reuse, Madrid, July 2004.
[NKF94] B. Nuseibeh, J. Kramer, A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification ", IEEE Trans. on Soft Engineering, 20(10): 760-773, 1994.
[Parnas79] Parnas D., "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979.
[Shin02] Michael E. Shin, "Evolution in Multiple-View Models in Software Product Families," Ph.D. dissertation, George Mason Univ., Fairfax, VA, 2002.
[Warmer99] Jos Warmer and Anneke Kleppe, "The Object Constraint Language: Precise Modeling with UML,"Addison Wesley, 1999.



# Report on First Workshop on Quality Assurance in Reuse Contexts (QUARC 2004)

**Workshop at Third Software Product Line Conference (SPLC3)**
**Boston, Massachusetts – August 30, 2004**

IESE
**Fraunhofer** Institut
Experimentelles
Software Engineering

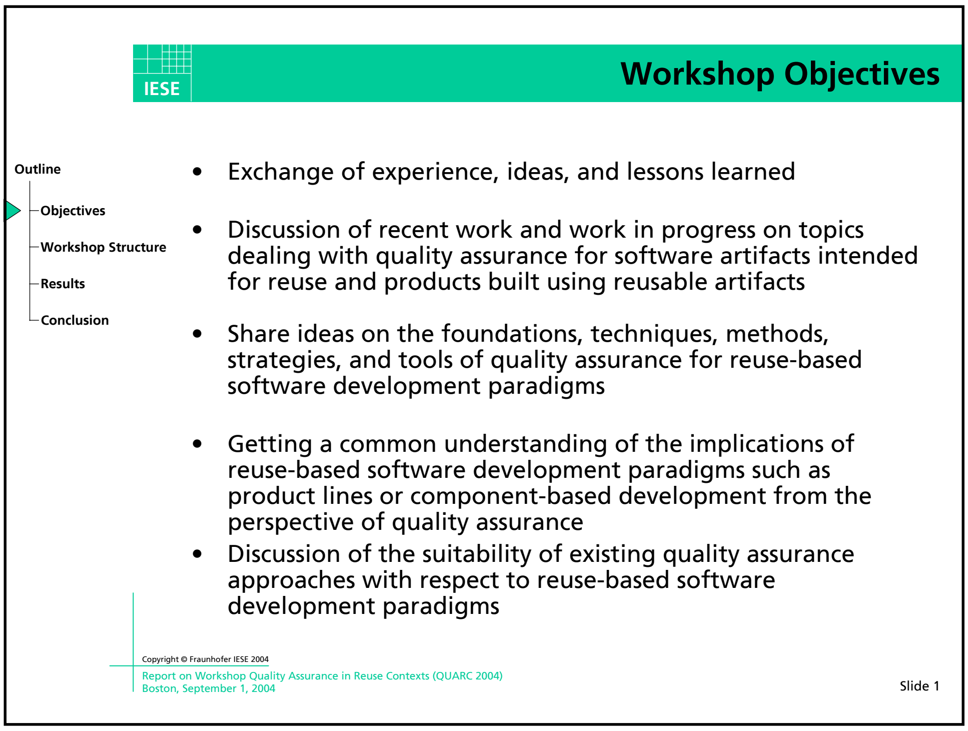
**Dirk Muthig**
**Ronny Kolb**
{muthig, kolb}@iese.fhg.de

Sauerwiesen 6
D-67661 Kaiserslautern, Germany

CLEMSON
UNIVERSITY

**John D. McGregor**
johnmc@cs.clemson.edu

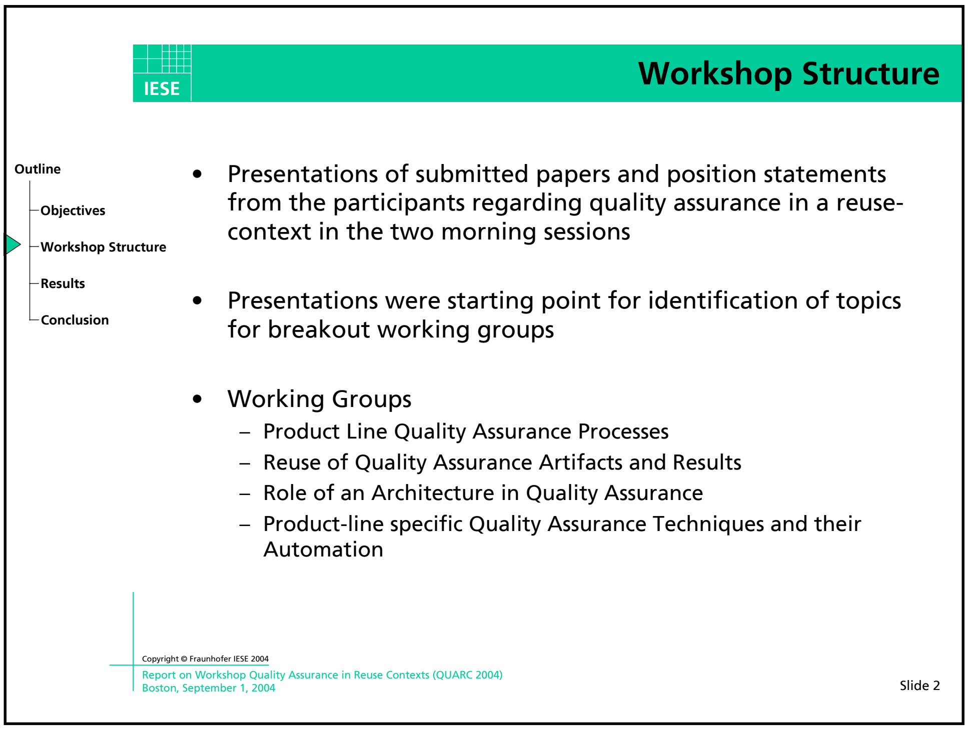
Dept. of Computer Science
Clemson University
Clemson, SC 29634



IESE

## Workshop Objectives

- Exchange of experience, ideas, and lessons learned
- Discussion of recent work and work in progress on topics dealing with quality assurance for software artifacts intended for reuse and products built using reusable artifacts
- Share ideas on the foundations, techniques, methods, strategies, and tools of quality assurance for reuse-based software development paradigms
- Getting a common understanding of the implications of reuse-based software development paradigms such as product lines or component-based development from the perspective of quality assurance
- Discussion of the suitability of existing quality assurance approaches with respect to reuse-based software development paradigms



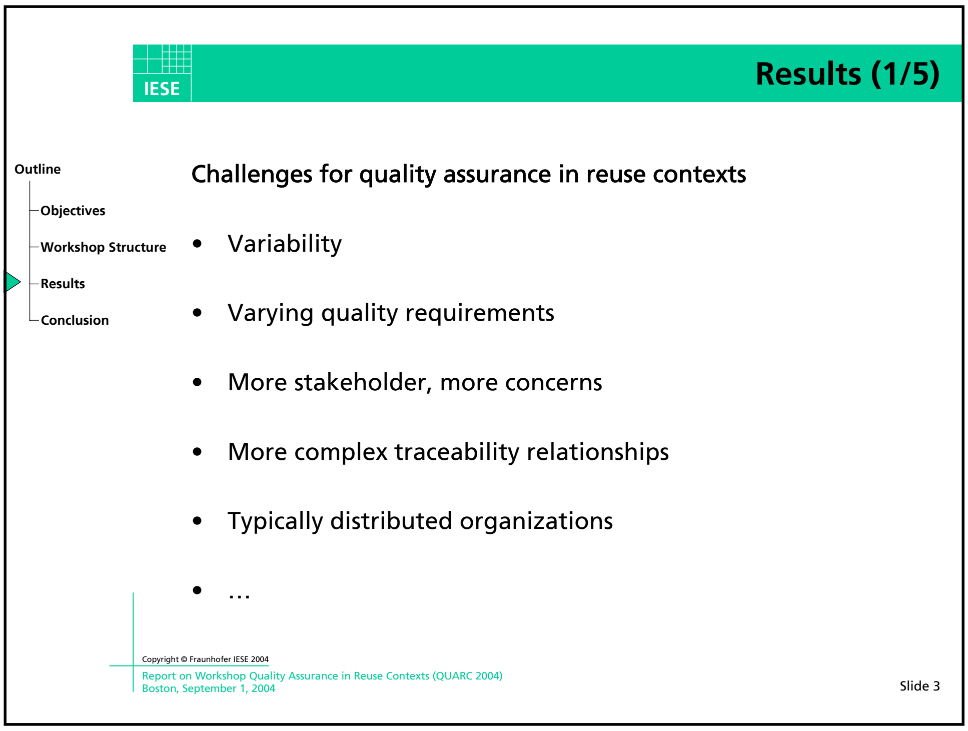
## Workshop Structure

Outline
- Objectives
- Workshop Structure
- Results
- Conclusion

- Presentations of submitted papers and position statements from the participants regarding quality assurance in a reuse-context in the two morning sessions
- Presentations were starting point for identification of topics for breakout working groups
- Working Groups
  - Product Line Quality Assurance Processes
  - Reuse of Quality Assurance Artifacts and Results
  - Role of an Architecture in Quality Assurance
  - Product-line specific Quality Assurance Techniques and their Automation

Copyright © Fraunhofer IESE 2004
Report on Workshop Quality Assurance in Reuse Contexts (QUARC 2004)
Boston, September 1, 2004

Slide 2



## Results (1/5)

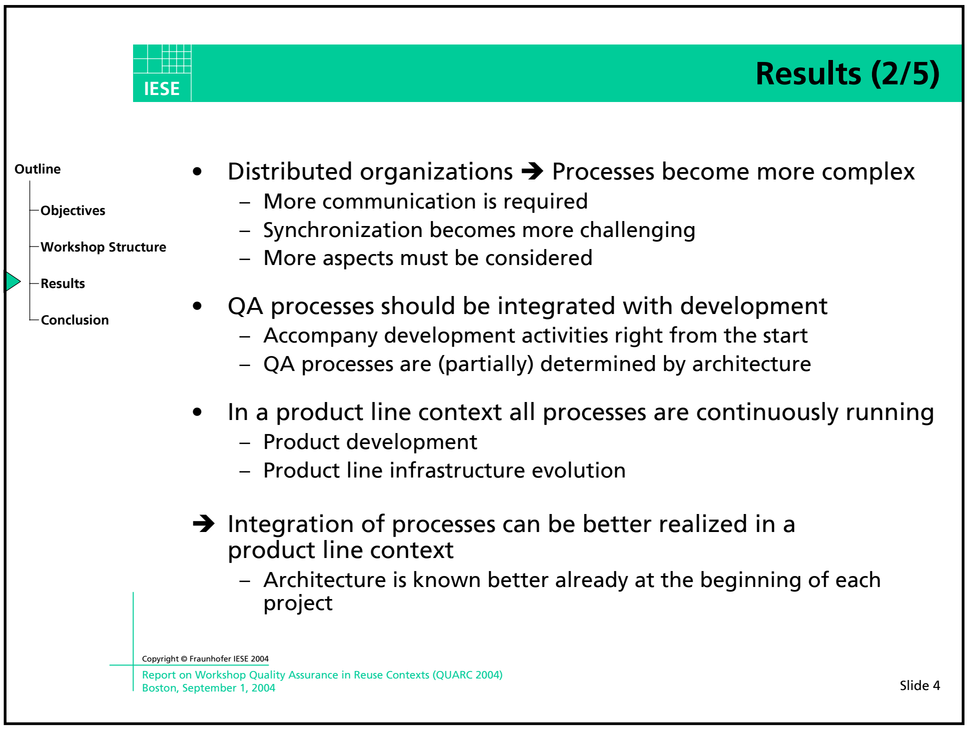
Outline
- Objectives
- Workshop Structure
- Results
- Conclusion

Challenges for quality assurance in reuse contexts

- Variability
- Varying quality requirements
- More stakeholder, more concerns
- More complex traceability relationships
- Typically distributed organizations
- …

Copyright © Fraunhofer IESE 2004
Report on Workshop Quality Assurance in Reuse Contexts (QUARC 2004)
Boston, September 1, 2004

Slide 3



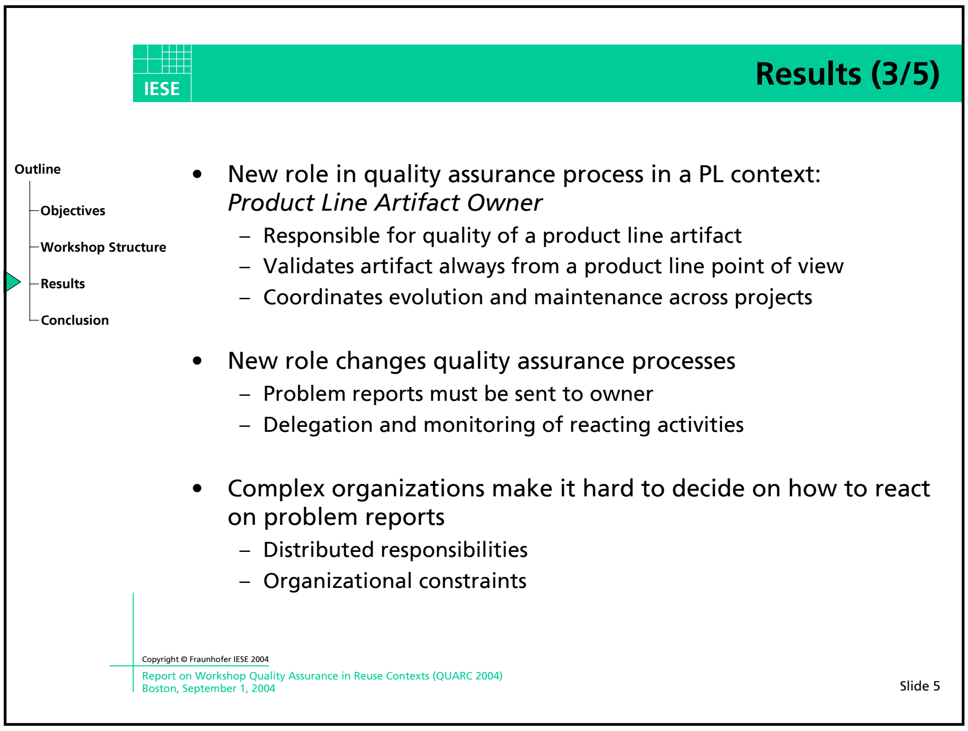
IESE

## Results (2/5)

Outline
- Objectives
- Workshop Structure
- Results
- Conclusion

- Distributed organizations ➔ Processes become more complex
  - More communication is required
  - Synchronization becomes more challenging
  - More aspects must be considered
- QA processes should be integrated with development
  - Accompany development activities right from the start
  - QA processes are (partially) determined by architecture
- In a product line context all processes are continuously running
  - Product development
  - Product line infrastructure evolution

➔ Integration of processes can be better realized in a product line context
  - Architecture is known better already at the beginning of each project

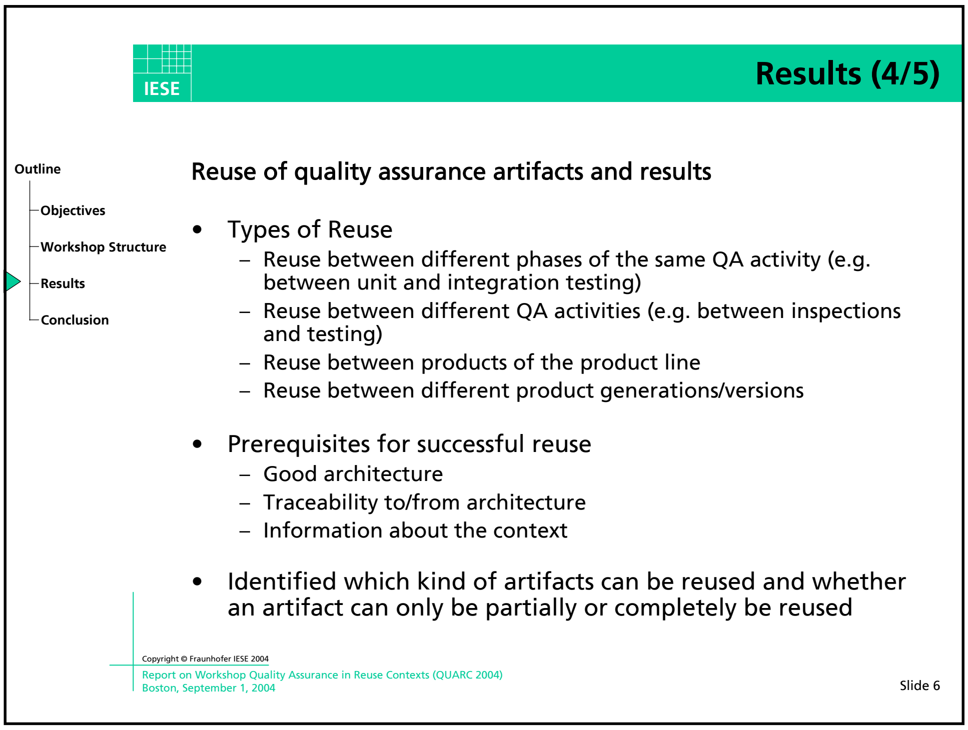


IESE

## Results (3/5)

Outline
- Objectives
- Workshop Structure
- Results
- Conclusion

- New role in quality assurance process in a PL context: *Product Line Artifact Owner*
  - Responsible for quality of a product line artifact
  - Validates artifact always from a product line point of view
  - Coordinates evolution and maintenance across projects
- New role changes quality assurance processes
  - Problem reports must be sent to owner
  - Delegation and monitoring of reacting activities
- Complex organizations make it hard to decide on how to react on problem reports
  - Distributed responsibilities
  - Organizational constraints



IESE

## Results (4/5)

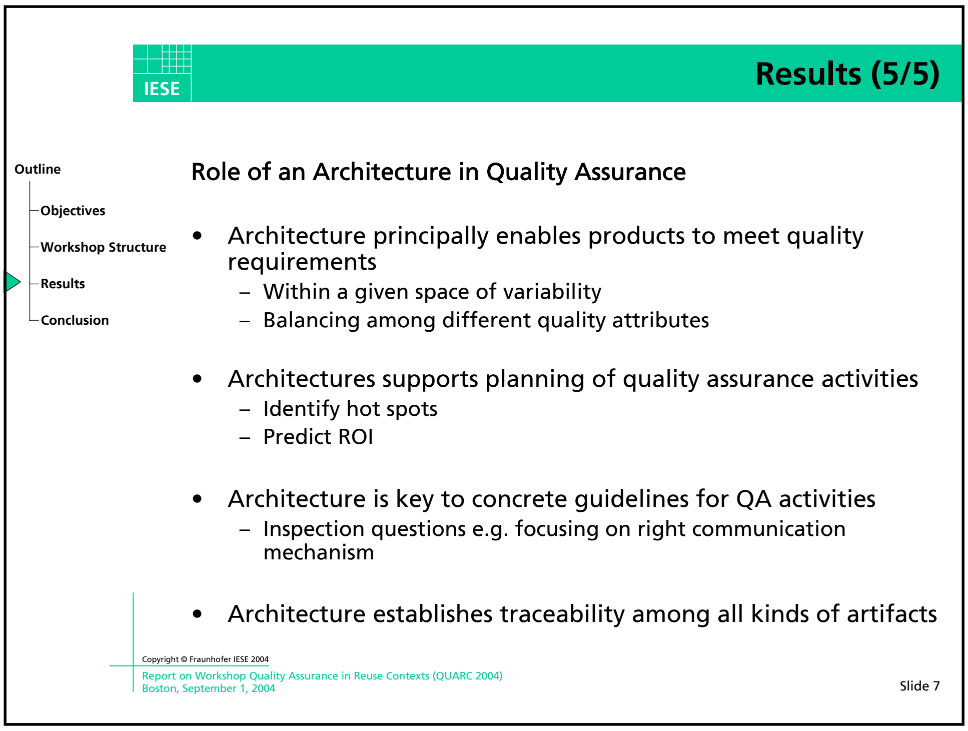
Outline
- Objectives
- Workshop Structure
- Results
- Conclusion

### Reuse of quality assurance artifacts and results

- Types of Reuse
  - Reuse between different phases of the same QA activity (e.g. between unit and integration testing)
  - Reuse between different QA activities (e.g. between inspections and testing)
  - Reuse between products of the product line
  - Reuse between different product generations/versions
- Prerequisites for successful reuse
  - Good architecture
  - Traceability to/from architecture
  - Information about the context
- Identified which kind of artifacts can be reused and whether an artifact can only be partially or completely be reused

Copyright © Fraunhofer IESE 2004
Report on Workshop Quality Assurance in Reuse Contexts (QUARC 2004)
Boston, September 1, 2004

Slide 6



IESE

## Results (5/5)

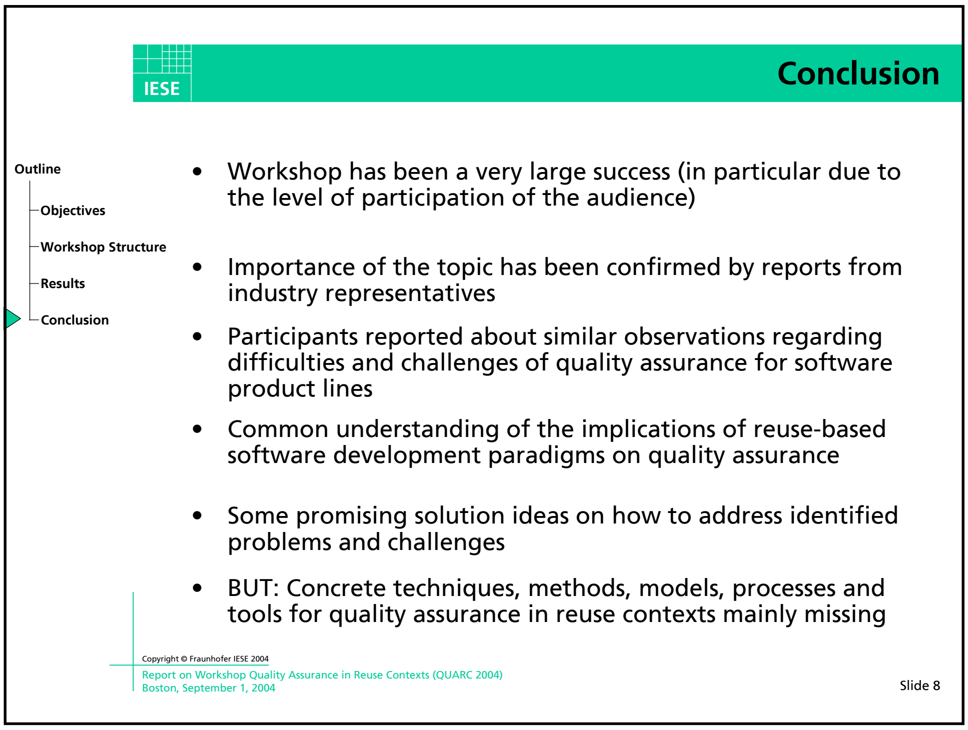
Outline
- Objectives
- Workshop Structure
- Results
- Conclusion

### Role of an Architecture in Quality Assurance

- Architecture principally enables products to meet quality requirements
  - Within a given space of variability
  - Balancing among different quality attributes
- Architectures supports planning of quality assurance activities
  - Identify hot spots
  - Predict ROI
- Architecture is key to concrete guidelines for QA activities
  - Inspection questions e.g. focusing on right communication mechanism
- Architecture establishes traceability among all kinds of artifacts

Copyright © Fraunhofer IESE 2004
Report on Workshop Quality Assurance in Reuse Contexts (QUARC 2004)
Boston, September 1, 2004

Slide 7



IESE

## Conclusion

- Workshop has been a very large success (in particular due to the level of participation of the audience)
- Importance of the topic has been confirmed by reports from industry representatives
- Participants reported about similar observations regarding difficulties and challenges of quality assurance for software product lines
- Common understanding of the implications of reuse-based software development paradigms on quality assurance
- Some promising solution ideas on how to address identified problems and challenges
- BUT: Concrete techniques, methods, models, processes and tools for quality assurance in reuse contexts mainly missing

# Document Information

| | |
|---|---|
| Title: | Proceedings of the First International Workshop on Quality Assurance in Reuse Contexts (QUARC 2004) |
| Date: | August 30, 2004 |
| Report: | IESE-096.04/E |
| Status: | Final |
| Distribution: | Public |