QUALITY ASSURANCE OF INTEGRATED BUSINESS SOFTWARE: AN APPROACH TO TESTING SOFTWARE PRODUCT LINES

Ulrike Dowie, Nicole Gellner, Sven Hanssen, Andreas Helferich, Georg Herzwurm, Sixten Schockert

Universität Stuttgart, Institute of Business Administration, Chair of Information Systems II, Business Software, Breitscheidstraße 2c, D-70174 Stuttgart

{dowie|gellner|hanssen|helferich|herzwurm|schockert@wi.uni-stuttgart.de}

Summary:

The use of integrated business software can be instrumental in reducing the difficulties occurring when various information systems have to be integrated. As a downside of this and due to the fact that these systems are designed to be used in all sorts of enterprises, the internal complexity of these systems increases exponentially. Software product lines on the one hand promise remedy by the conscious use of variability, on the other hand create new demands on quality assurance. The article on hand provides a theoretical framework for evaluating approaches to software testing, regarding their use in the development of software product lines. It turns out that only a practice-oriented approach emphasizing the buyer's view will be successful in the end.

Keywords:

Business software, software product line, software testing, quality assurance, domain engineering, application engineering.

1 INTRODUCTION

The demand for off-the-shelf or standards-based business software has increased steadily over the last few years. The simultaneous use of many software packages developed in-house as well as (often customized) off-the-shelf software has further intensified the existing integration problems of application systems. Microsoft guesses that the percentage of costs for application integration will rise from approx. 10-15 % today up to 60 % within the next years (cf. [Kind04]). Many enterprises therefore consider replacing the *"best of breed"* – strategy with the alternative *"single sourcing"*. In "*best of breed"* the alternative best fulfilling the requirements of the enterprise is selected, irrespective of the provider, and integrated into the enterprise application architecture. In "*single sourcing"* all application systems are purchased from one single supplier. The most important advantage expected with purchasing products from a single source seems to be the simplified integration of these systems.

Among manufacturers of integrated business software, such as Enterprise Resource Planning (ERP) systems, this trend is reflected in an increasing performance range. Some years ago the market for Customer Relationship Management (CRM) and Supply Chain Management (SCM) systems was dominated by specialists. Nowadays ERP vendors and their CRM and SCM modules steadily gain market share, e.g. the CRM module of the mySAP business suite. The higher degree of process coverage that integrated application systems offer brings along a rise in the internal complexity of these systems. Additionally, the trend to differentiate products, e. g. by industry-specific solutions or special offers for small and medium-sized enterprises (SMEs), increases the challenge for software companies to offer high quality products. S*oftware product lines*, an approach used in research and industry for some years, promises remedy or at least relief here. The use of product lines holds both improvement potentials for the quality assurance of integrated business software as well as new challenges to quality assurance. This article introduces a framework for the evaluation of software testing approaches regarding their use in the quality assurance for software product lines.

2 SOFTWARE QUALITY, SOFTWARE PRODUCT LINES AND SOFTWARE TESTING

2.1 Software Quality

The International Standards Organisation norm 9126 (see [ISO01]) deals with software quality. Since the last revision in 2001, it distinguishes between *Internal Quality, External Quality* and *Quality in Use*. Quality in Use was not part of the previous version and is defined as "the user's view of the quality of the software product when it is used in a specific environment and a specific context of use. It measures the extent to which users can achieve their goals in a particular environment, rather than measuring the properties of the software itself…" ([ISO01], p. 5). Put another way, Quality in Use measures the fulfilment of user requirements. A comparison of 82 studies on the relationship between customer orientation and a company's success (see [Herz00]) shows a significant relationship between fulfilment of customer requirements and success in the marketplace.

Companies using integrated business software want the software to support a large degree of their business processes. At the same time business processes are usually different in different companies, therefore it is important for software companies to offer products that are flexible enough to be of value to a large number of different companies. This is where software product lines come in: they promise to make it possible to offer a large variety of products while still being able to manage this variety. Chapter 2.2 explains how this is supposed to work.

2.2 Software product lines

The term "software product line" implies that different products of one *domain* (also referred to as problem space or application range, e. g. operating systems for mobile telephones or software support of the sales department) are viewed as a family and not as single products anymore (while parts of the literature distinguish between product families and product lines, in the context of this article both are used synonymously). According to the Software Engineering Institute, software product lines are defined as "set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" (see [ClNo02], p. 5). The components of a software product line are the product line architecture and the individual products which are part of the product line. The *product line architecture* describes the individual products, their common components and - at least in outlines - the differences between the products of the family (cf. [Bosc00]).

The concept of software product lines supports the reuse of components: the common product line architecture is explicitly designed for reuse. Therefore a component does not have to be evaluated regarding its suitability for this use (cf. [Bosc00]). This is at the same time the most important difference compared to other reuse concepts of program parts or components: the multiple use is already planned *in advance.* This implies a high degree of reuse and therefore the possibility of fast, economical and high quality development of new products (systems). Both time-to-market and maintenance effort are expected to decrease, while customer satisfaction is expected to increase since software can be developed faster, in higher quality, and for more individual purposes (cf. [Böll02]).

2.3 Software Testing

A *test case* comprises input values, preconditions and expected results. It serves a purpose, e. g. to verify a certain execution path (cf. [IEEE90]). The test case consists of a stimulus and the result that a correctly operating system would deliver in response to this stimulus (cf. [McGr01], p. 5).

Test procedure means a document that can give proper instructions on the execution of one or several test cases. *Test method* describes a procedure which is scheduled and built on a set of rules that derivate or select test cases (see [SpLi04], p. 214). The term *testing approaches* as used in this article summarizes test methods and test procedures as well as recommendations on how to go about testing software product lines.

A *test specification* is the documentation of the implementation of a component, its interfaces and a certain set of test cases related to the respective interface.

Test products are all documents and executable programs or program parts, which are required or are designed in the context of testing (e.g. test plan, test cases, and code for automatic tests). The proportional amount of source code which is tested by running the software tests is called *coverage*.

3 PRODUCT LINE DEVELOPMENT PROCESS

3.1 Overview

Different process models exist for the development process of product lines, e. g. those described in [Bay+99], [WeLa99] or [Muth02]. Common to them is that the product line development process is modelled along the structure of a product line. Just as the product line consists of product line architecture and product line members, the development process also consists of the process of the development of the product line architecture and the development process of product line members. The development of the product line architecture is called *domain engineering* and the development of product line members *application engineering*. Figure 1 shows the complete process. Testing is

highlighted due to the central importance for this work as part of the steps *implementation (Core Assets)* and *system implementation*.

A rough cost-benefit analysis and the so-called *scoping* (cf. [Bosc00]) precede domain and application engineering. During *scoping* the use of the product line or its products is planned (see [Böc+04], p. 44). One important aspect of this is the separation between requirements common to all products and variable requirements. Variable requirements are not demanded for all products of a product line in the same way. For example, all requirements which depend on the hardware platform used are variable requirements. In the context of business software, one example is the user interface. A user can alternatively access the system using a local client, web browser or UMTS mobile phone.

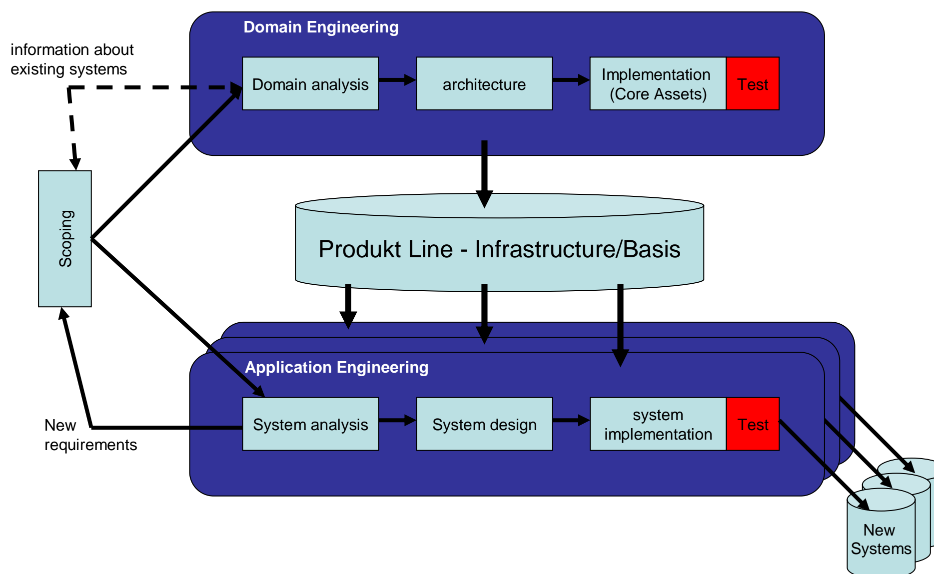

Figure1: The product line development process (modified from [Muth03])

3.2 Domain Engineering

Domain engineering consists of three steps: *domain analysis, architectural design* and *domain implementation*. During domain analysis, the analysis of the application scope of the product line that started with the scoping is continued and a requirements analysis is carried out for the complete product line. Common features among and differences between the products are defined and the so-called *variation points* are fixed. Variation points are those system parts where the products differ from one another (see [WeLa99], p. 20). A summary of variation points and their modeling and implementation is given in [Böc+04] (see ibid. p. 13 et sqq. and p. 109 et sqq.).

Following domain analysis, the product line architecture is designed. The product line architecture provides the framework for reusable components. This framework describes visible properties of the components and the relations between them (cf. [Bosc00]). Reusable components are designed in the last step of domain engineering, during domain implementation. These components represent the base for the products of the product line. Together with test cases or scenarios, documentation and models they form the so-called *core assets* (cf. [McGr01], p. 23).

3.3 Application Engineering

During application engineering, the individual products are implemented according to the results of scoping and domain engineering. Three phases can be distinguished: system analysis, system design and system implementation.

During system analysis the requirements on the respective product gathered during domain analysis are further particularized, especially focussing on differences between variable requirements on the individual products. For every single product, those requirements are disregarded which this product does not have to fulfil. Then, the architecture of this product is derived from the product line architecture. The following steps are carried out: architecture pruning, architecture extension, conflict resolution, and architecture assessment (see [Bosc00], p. 262 et sqq.). Next, product-specific components are implemented, using the possibilities of core asset varieties and all product specific components. Finally, the adapted core assets are tested and integrated into the designed product (cf. [WeLa99]).

3.4 Testing during the development process of software product lines

Testing software product lines is both part of domain engineering and application engineering. It builds upon well-known methods and principles of testing software components. It is common practice (cf. [Weyu98], [Reu+03]) in testing components, no matter whether these are made or bought (cf. [Szyp99]), that there are always the phases *unit test, integration test* and *system test*. During the unit test phase, the component is tested detached from any application. This is normally carried out by component developers and under consideration of aspects internal to the component. The subsystems which are composed of individual components are tested during the integration test. And finally the complete system, consisting of all subsystems, is checked for faultlessness in the system test. Integration and system tests depend on the context in which the respective component is used since by definition such tests include exactly the context of the component in the test case, thereby planning to validate the interactions of the component with other parts of the system. The problem is that all integration and system test results are therefore also context-dependent. Thus, a component can contain a serious error which, however, never comes to light in a certain system since the function of the component in question is not used at all. If the component is integrated into another system which uses this faulty function, the defect appears for the first time despite intensive testing in the previous system. A popular example of this is given by the accident of space rocket Ariane 5 (cf. [Weyu98]).

4 TESTING APPROACHES TO SOFTWARE PRODUCT LINES: STATE OF THE ART

4.1 Testing approaches to Domain Engineering

In the following, we will distinguish between testing approaches to Domain Engineering and testing approaches used for component testing since the testing approaches applicable for Application Engineering are the same that are usually used for component testing. As there is a multitude of testing approaches, a conscious decision was made to limit the number of approaches initially included in the framework presented in the following chapter. The approaches presented in chapter 4 and classified using the framework in chapter 5 were selected either because they were developed specifically for software product lines or because they are well-documented and researched and seem a good addition to the set of approaches covered.

One goal of domain engineering is the reuse of test products. In combination with early testing during development, it can lead to the same savings and advantages in product lines as the reuse of products

in development does (cf. [McGr01]). Test products like test plans and test cases should be usable for different products (cf. [ClNo02]). They are part of the reusable components of the product line.

A recommendation for testing product lines repeatedly given in literature is to map the program architecture to the test architecture (cf. [McGr02], [ClNo02]). The goal is to clarify which parts of the program code can be tested with which parts of the test code (cf. [Weyu98], [ClNo02]). Thus, the test code can be easily adapted in case of changes in the code to be tested. This is generally useful when tests are repeated (e. g. after error correction) and especially in case of software product lines, as planned tests are executed repeatedly in order to cover different configurations of the tested software. As in product development, it is important that specific test cases of the product remain compatible with the corresponding product line test case. The approach to requirements-based testing introduced in [Kam+04] keeps the variability in domain artefacts to support the reuse in test artefacts. In this approach called ScenTED (Scenario-based test derivation), test cases are already generated in domain engineering using Use-Case models and Use-Case specifications which are extended with variabilities. By keeping these variabilities, it is possible to derive test cases from test artefacts when creating an application, since all intended variants are already modeled in these test cases.

Due to the large amount of possibilities of combining and integrating components of a product line, research efforts are directed to finding metrics which measure test coverage of possible component interactions (cf. [WiPr01]). Furthermore methods were developed to determine the number of test cases required for a certain degree of coverage (cf. [Coh+97]). These approaches can be summarized as *combinatorial design*. In [Lulu02] a tree-based analysis technology (*FCTA Fault Contribution Tree Analysis)* is introduced. It uses the results from domain analysis when designing a new member of the product family. The method is based on Boolean logic and describes which elements (knots in the tree) may lead to a fault in the resulting system. Knots in the tree represent system functions. Starting at the root, a refinement of the functions shown is carried out. Test case generation for specifications in Boolean form can be automated. A detailed description of this method is given in [Wey+94].

4.2 Testing approaches to component testing

A software product line contains products sharing requirements, functions and concepts. It also contains program code, usually in the form of *components* (cf. [RiRo02]). A component is a unit with a specified interface and explicit context dependencies. A component can be used independently and is subject to composition by third parties (see [Szyp99], p. 34). Problems which arise in connection with testing component-based software therefore also affect products in product lines.

A first summary of the state of research in testing components can be found in [BeGr03a]. According to the authors, many problems with using components arise from the fact that there is no flow of information between the supplier of the component and the component user who integrates components into a system. As an example, source code of a component is not available to the user and the developer has no information about the context in which the component will be used. Approaches to testing components which address the problem of a lacking flow of information can be subdivided into two categories: those improving the flow of information between supplier and user and those which deal with the consequences of such a lacking flow of information.

The *metadata approach* by [Ors+00] is part of the first category. The basic idea is to augment components with additional information about the component itself (*metadata*). The information provided is determined by the component manufacturer and can be adapted specifically to the needs of the customer. In principle it can be any artefact created during component development, but it can also be any additionally generated information (e. g., control flow data). [Edwa01] suggests that the component developer provide the specification of the component. In [Ors+01], the authors expand this approach to regression testing of component-based software by providing test cases as metadata. These are derived from the source code of the component or from its specification. The provision of test data by the component developer is also addressed in [LiRi98]. In this approach, test cases already used by the component developer as well as test cases to be carried out later by the component user are

delivered as metadata. Additionally, the component is enhanced with additional features for collecting internal information during testing as well as for providing this information. A similar approach is found in [Bun+00], though it uses testing specifications rather than test cases.

Approaches which deal with the consequences of a lacking flow of information between the developer and a user of a component, aim at extending components with the functionality to test themselves. As opposed to the approaches already shown, components are extended by special testing functions which can be invoked by the component user. Approaches of this type can be found in [Yan+99], [Tra+99], [BeGr03b] and [HöEd02]. The testability of a component can also be improved by supporting test execution and observation. Such an approach can be found in [Gao+02]. Components are extended with functionalities showing the internal structure and the behaviour of the component. A special test interface can be provided to gain access to special test functionalities as well as to cooperate directly with external testing tools.

In the approaches mentioned previously, the reason for many problems using components is assumed to be the fact that component designer and user work separately and necessary information is not exchanged. *Reliability-based approaches* live by the fact that the component user selects or refuses components depending on their quality, e. g. accuracy. An approach which measures the quality of a component can be found in [Xiao03]. A component developer provides a metric *M*. The reliability of the component can be defined as a function *f*= {< s, M (s) >} for a defined set of possible input values *s*. Using this formula, the quality of the complete system can be calculated. In [Mena04] an advanced concept for web-based applications is outlined. According to the author, components are not evaluated by the component user with regard to their quality. Rather, components log into systems dynamically. Whether a component will be accepted or not is determined by its functionalities as well as the quality of its services. If the interaction between system and component is successful, the component becomes a registered part of the system.

4.3 Testing approaches: Summary

The following table gives a summarizing survey of the testing approaches introduced.

Testing approach	**Test object level**	**Source**
Testing architecture reflects program architecture	All products of a product line (PL), single products of the PL, single component	[McGr02], [ClNo02], [Weyu98]
Product line test case is adapted as product test case	All products of a PL, single products of the PL	[McGr01]
Requirements-based testing (ScenTED)	single products of the PL	[Kam+04]
Combinatorial Design	All products of a product line	[WiPr01], [Coh+97]
Self-testing	Single components	[Yan+99], [Tra+99], [BeGr03b], [HöEd02]
Metadata-Approach	Single components	[Ors+00], [Edwa01], [Ors+01], [LiRi98], [Bund+00]
"testable beans" (improving testability)	Single components	[Gao+02]
Quality of service-/ Reliability-based approaches (for web-based applications)	Single components	[Xiao03], [Mena04]
Automatic test case generation for specifications in Boolean form	All products of a PL, single products of the PL	[Wey+94]
FCTA (Fault Contribution Tree Analysis)	All products of a PL, single products of the PL	[Lulu02]

Table 1: Testing approaches to product lines and respective test object level

5 EVALUATION OF APPROACHES TO PRODUCT LINE TESTING

The variety of approaches indicates that a choice of testing approaches has to be made. A theoretical framework will be introduced which can serve as a decision basis (cf. for theoretical frameworks in general [Kubi75]). The selection of testing approaches depends on the objective pursued with testing. As most approaches are not mutually exclusive, several different approaches can be chosen when several objectives are pursued that do not contradict one another.

As any other entrepreneurial activity, testing is motivated by goals the organization intends to achieve (see [Hame92], p. 2634). Thus, the decision about how to test, i. e. which testing approach to choose, is influenced by the pursued goals (cf. [Hame92], p. 2635). As examples, test objectives that support efficiency, quality and competitive goals are used as criteria for evaluation of the testing approaches.

Table 2 summarizes which objectives are pursued by which approaches.

Testing approach	test objective					
	efficiency		quality objectives			Competitive goals
	minimize test efforts	minimize maintenance of test products	find many errors = correct implementation	high degree of coverage = complete implementation	minimize risks for the customer using the software	early market entry
test architec-ture reflects program architecture	after high initial efforts	Test architec-ture as stable as program architecture	- (test architecture does not reveal any faults)	x		not during first development but at updates
product line is adapted for indiv. products	product line test case is adaptable template	product line test case is adaptable template	- (only new test product line will reveal more faults)			
Combinatorial Design	in case of automatic test generation			x		x
ScenTED	accept test cases	accept test cases	- (no testing, but generation of test cases)	test cases of the specification	- (no approach)	? (indirectly by reducing test process)
self-testing	self-testing component	test cases included in components	- ("black-black-box test": test procedure AND test cases unknown)	? (depending on components manufacturer)	component detects errors itself and is able to react	tests postponed to later stages
meta data approach	less effort by more information	test products can be enclosed meta data	depending on component user, but e.g. source code is helpful	depending on component user, but e.g. source code is helpful	- (no approach)	? (indirectly by use of components)
testable beans (improved component testing)	improved testing = less efforts)	improved testing = less efforts)	depending on component user	depending on component user	- (no approach)	? (indirectly by use of components)
Quality-of-service / reliability based approach (web-based applications)	test effort avoided by dismissing or accepting component without testing	- (no relation to test products)	- (no faults can be found, because actually no testing)	- (no faults can be found, because actually no testing)	component can be accepted or dismissed according to field of application	? (indirectly by use of components)
automatic test case generation for specifications in Boolean form	effort avoidance by generation of test cases	automatic generation of test products	illustration of specification into test cases	coverage proportional to the number of test cases	-	? (indirectly by use of components)
FCTA (Fault Contribution Tree Analysis)	effort avoidance by analysis of possible error paths	? (no test product generation but basis for appropriate test cases)	- (faults are not found but sources of error excluded)	- (faults are not found but sources of error excluded)	-	x
regression test					x	
legend:		= approach suitable				
	-	= approach not suitable				
		= no clear statement possible				

Table 2: Testing approaches to product lines and test objectives

Not every testing approach introduced in chapter 4.1 and 4.2 is appropriate in every software product line development effort. Rather, the testing approach selection has to be carried out under

consideration of the framework conditions which need to be fulfilled so that a testing approach can be used and moreover can be useful (cf. [Herz00], p. 68ff). Considerable research has been done on the relevant conditions for successful software development (e.g., cf. [Herz00], p. 46ff, and [Boe+00]). For testing specifically, there's no established model to date capturing factors explicitly affecting testing success. Therefore, testing being part of software development efforts, conditions identified as influencing software development as a whole represent the basis of influencing factors for testing.

Transferred from development to testing, the conditions concern the product to be tested, the testing process followed, the organisation and the people carrying out testing tasks (cf. [Boe+00], [Herz00], p. 69). In the following, a selection of framework conditions is presented which suffices for illustration of the evaluation systematic and which has the strongest influence on the choice of testing approaches.

Testing approach	Framework and test parameters regarded as not alterable				
	enterprise characteristics		product characteristics		
	qualified test staff available in sufficient numbers	testing infrastructure: suitable soft- and hardware equipment	high complexity (many components)	well-known use profiles	product requirements frequently changing
test architecture reflects program architecture	highly qualified test staff necessary		with low complexity, test architecture might not be needed		test architecture changed in paralell to program architecture - not necessary, if program architecture does not change
test case for product line is adapted for indiv. products				only test cases actually needed for this product are adapted	x
Combinatorial Design	highly qualified test staff necessary		x		x
ScenTED	highly qualified test staff necessary	suitable testing infrastructure necessary	well suitable		results from use cases
self-testing	highly qualified test staff necessary		well suitable		
meta data approach	highly qualified test staff necessary	suitable testing infrastructure necessary	not suitable, too much effort for party reusing the component	? (maybe helpful depending on provided information)	? (depending on provided information)
testable beans (improved component testing)	highly qualified test staff necessary	suitable testing infrastructure necessary	x		
Quality-of-service / reliability based approach (web-based applications)			x		may be important for decision whether to accept a component
generation of specifications in Boole's form		x	x	important (but can be derived from specification)	well suitable (requirements arise from specification)
FCTA (Fault Contribution Tree Analysis)	x	x	x		
regression test		x	x	x	
legend:		= approach suitable			
	-	= approach not suitable			
		= no clear statement possible			

Table 3: Testing approaches to product lines and required framework conditions

Availability of resources is the most important organisational characteristic which influences the selection of the testing approach: on one hand the testing staff (number and qualification of testers), on

the other hand the test infrastructure (primarily hardware and software). Products can be distinguished by the following criteria, relevant for the choice of testing approach: complexity (e. g., number of combined components), availability of use profiles and frequency of changes to product requirements. A survey of conditions under which circumstances which approach is useful, is offered in table 3.

To make a clear decision regarding the testing approaches depending on the chosen test objective, all design parameters (e. g. the test resources) are considered to be fixed (in short term). If it turns out that the testing approaches suitable for the chosen test objective require certain framework conditions which are not given and cannot be established, the chosen aim must be changed. If for example the test objective is “to find many errors” and the approach "Self-testing" shall be followed up, this makes sense only in case of frequently changing requirements. If this is not given in the planned development situation, another approach should be chosen.

As a result of tables 2 and 3, different objectives are attainable under certain framework conditions with a choice of testing approaches. The economic aim "low test effort" is pursued by the clear majority of testing approaches. Also, reduction of costs for maintenance of test products is one focus. Quality objectives, however, seem to be of secondary importance. A high coverage range is pursued by 50 % of the presented approaches. A high measure of faultlessness is explicitly pursued only by 30%. The most important aim from the customers’ view is minimizing risks of using the software. This, however, is only the focus of 30% of the approaches. The aim "early market entry" that becomes achievable by early testing during development is explicitly sought by only a few approaches.

It is remarkable that many approaches were developed for tests of complex products or product lines and for development environments with frequently changing requirements. However, few testing approaches consult information about software usage in form of use profiles. According to one approach ("product line test case is adapted to the product test case"), a test architecture following the program architecture can be developed only if sufficiently qualified testing staff is available. The test outline by means of combinatorial design based on component interaction is not useful if the product to be tested consists of few components or if there are only few possible combinations of components.

All of these observations reveal that approaches to testing software product lines which take the customers’ perspective into account are still missing and that scientific research is incomplete. To be successful on the market for software products, the focus on customer interests is imperative (cf. [Herz00]). This applies to the development of software product lines in particular, since here scale effects take effect that can easily extend the advantages of customer orientation.

6 CONCLUSION

Software product lines promise reduced time to market as well as increases in technical and relative software quality. There are examples of successful application of this concept from various kinds of systems and enterprises developing the systems. Testing software product lines, however, seems to be a rather neglected activity. There are approaches to software testing that can be adopted or adapted for the use in software product line testing, but there does not seem to be one single approach covering all the aspects necessary. The framework developed and presented in this paper helps researchers and practitioners to decide which approaches to combine to effectively and efficiently test software product lines. Even more, by looking at the tables comprising the framework, researchers can determine which areas most urgently need additional approaches and direct their research efforts accordingly. For this to be most effective, the number of testing approaches classified the framework needs to be increased further, trying to make sure no approach that can be used in this context is being left out. Additionally, the framework needs to be other enhanced by incorporating additional testing objectives, the link between company goals and testing objectives should also be investigated.

Additional research effort should be directed at identifying the factors influencing the success of software testing initiatives. The results of this research could then be used to revise the framework,

replacing the conditions derived from research on the success of software development projects (as explained in chapter 5).

Bibliography

[Bay+99] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.-M., PuLSE: A methodology to develop software product lines, in Proceedings of the 5th Symposium on Software Reusability, pages 122-131, 1999.

[BeGr03a] Beydeda, S. und Gruhn, V., State of the art in testing components, in: Proceedings of the Third International Conference On Quality Software (QSIC'03), Dallas, Texas, USA, November 2003, p. 146

[BeGr03b] Beydeda, S., Gruhn, V., Merging components and testing tools: The Self-Testing COTS Components (STECC) Strategy, in: Proceedings of the 29th EUROMICRO Conference „New Waves in System Architecture" (EUROMICRO'03), Belek-Antalya, Türkei, September 2003, p. 107.

[Boe+00] Boehm, B.; Abts, C.; Horowitz, E.; Madachy, R.; Reifer, D.; Clark, B.K.; Steece, B.; Brown, A.W:; Chulani, S.: Software cost estimation with Cocomo II. Upper Saddle River, NJ: Prentice Hall, 2000.

[Böc+04] Böckle, G., Knauber, P., Pohl, K., Schmid, K. (Eds.), Software-Produktlinien: Methoden, Einführung und Praxis. Dpunkt: Heidelberg, 2004.

[Böll02] Böllert, K., Objektorientierte Entwicklung von Software-Produktlinien zur Serienfertigung von Software-Systemen. Dissertation, TU Illmenau, 2002.

[Bosc00] Bosch, J., Design and use of software architectures. Addison-Wesley: Harlow, 2000.

[Bun+00] Bundell, G., Lee, G., Morris, J., Parker, K., Lam, P., A software component verification tool, in: International Conference on Software Methods and Tools (SMT'00), Wollongong, Australien, November 2000, p. 137.

[ClNo02] Clements, P., Northrop, L., Software product lines: practices and patterns. Addison-Wesley: Boston, MA, London, 2002.

[Coh+97] Cohen, D.M., Dalal, S., Fredmann, M., Patton, G., The AETG System: An Approach to testing based on Combinatorial Design, in: Transactions on Software Engineering, 23,7 (Jul. 1997), p. 437-444

[Edwa01] Edwards, S., Toward reflective metadata wrappers for formally specified software components, in: OOPSLA Workshop Specification and Verification of Component-Based Systems (SAVCBS), 2001.

[Gao+02] Gao, J., Zhu, Y., Shim, S., On building testable software components, in: COTS-Based Software Systems (ICCBCC), volume 2255 of LNCS, Springer 2002., p. 108-121.

[Hame92] Hamel, Winfried: Zielsysteme. In: Frese, E. (Eds.) Handwörterbuch der Organisation. 3. Aufl., Stuttgart 1992, p. 2634 – 2652.

[Herz00] Herzwurm, G., Kundenorientierte Softwareproduktentwicklung. Teubner: Stuttgart, 2000.

[HöEd02] Hörnstein, J., Edler, H., Test reuse in cbse using built-in tests, in: Workshop on Component-based Software Engineering, Composing Systems from components, 2002.

[IEEE90] Institute of Electrical and Electronics Engineers, IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.121990, New York 1990.

[ISO01] International Organisation for Standardization: ISO/IEC 9126-1:2001, Geneva 2001.

[Kam+04] Kamsties, E., Pohl, K., Reuys, A., Anforderungsbasiertes Testen, in: Böckle, G., Knauber, P., Pohl, K., Schmid,. K. (Eds.), Softwareproduktlinien, Methoden, Einführung und Praxis, Heidelberg 2004, p. 119 – 137

[Kind04] Kindzorra, O., Software development at Microsoft. Lecture script, Stuttgart Institute of Management and Technology (SIMT), fall semester 2004.

[Kubi75] Kubicek, H., Empirische Organisationsforschung. Konzeption und Methodik, Stuttgart 1975.

[LiRi98] Liu, C., Richardson, D., Software Components with retrospectors, in: International Workshop on the Role of Software Architecture in Testing and Analysis, 1998, p. 63-68.

[Lulu02] Lu, D., Lutz, R., Fault Contribution Trees for Product Families, in: Proceedings of the 13th International Symposium on Software Reliablity Engineering (ISSRE'02), Annapolis, Maryland, USA, November 2002, p. 231.

[McGr01] McGregor, John D., Testing a Software Product Line, Technical Report, CMU/SEI-2001-TR-022, Software Engineering Institute, Carnegie Mellon University, Hanscom AFB, MA, USA 2001.

[McGr02] McGregor, John D., Building Reusable Test Assets for a Product Line, in: 7th International Conference on Software Reuse, "Software Reuse: Methods, Techniques, and Tools", Austin, TX, USA, April 2002, p. 345-346.

[Mena02] Menasce, D., Load Testing of Web Sites, in: IEEE Internet Computing, Juli/August 2002, p. 70 - 74, 2002.

[Mena04] Menasce, D., QoS-Aware Software Components, in: IEEE Internet Computing, März/April 2004, p. 91 - 93, 2004.

[Muth02] Muthig, D., A light-weight approach facilitating an evolutionary transition towards software product lines. Dissertation, Universität Kaiserslautern, Fraunhofer-IRB Verlag, Stuttgart, 2002.

[Muth03] Muthig, D., Produktlinien – Einstieg. Webseite des Kompetenzzentrums Software Engineering: http://www.software-kompetenz.de/?2246, Abruf am 15.6. 2004

[Ors+00] Orso, A., Harrold, J., Rosenblum, D., Component metadata for software engineering tasks, in: Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000), LNCS Vol. 1999,Springer November 2000, p. 129.

[Ors+01] Orso, A., Harrold, J., Rosenblum, D., Rothermel, G., Do, H., Soffa, M., Using Component Metacontent to Support the Regression Testing of Component-Based Software, in: IEEE International Conference on Software Maintenance (ICSM'01), Florenz, Italien, November 2001, p. 716.

[Reu+03] Reuys, A., Kamsties, E., Pohl, K., Götz, H., Neumann, J., Weingärtner, J., Testen von Software-Produktvarianten – ein Erfahrungsbericht, in: 2. Deutscher Software-Produktlinien Workshop (DSPW-2), Multikonferenz Wirtschaftsinformatik (MKWI 2004), März 2004, p. 244-259.

[RiRo02] Riva, C., Del Rosso, C., Experiences with Software Product Family Evolution, in: Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03), Helsinki, Finnland, September 2003, p. 161.

[SpLi04] Spillner, A., Linz, T., Basiswissen Softwaretest, 2. Aufl., d.punkt-Verlag, Heidelberg 2004.

[Szyp99] Szyperski, C., Component Software, Beyond Object-Oriented Programming, ACM Press, Addison-Wesley, 1999.

[Tra+99] Traon, Y., Deveaux, D., Jezequel, J.-M., Self-testable components: from pgramatic tests to design-to-testability methodology, in: Technology of Object-Oriented Languages and Systems (TOOLS), S. 96-107, IEEE Computer Society Press, 1999.

[WeLa99] Weiss, D.M., Lai, C.T.R., Software product-line engineering: a family-based software development process. Addison-Wesley: Reading, MA, Bonn, 1999.

[Wey+94] Weyuker, E., Goradia, T., Singh, A., Automatically Generating Test Data from a Boolean Specification, in: IEEE Transactions on Software Engineering, Vol. 20, No. 5, May 1994, p. 353 – 363.

[Weyu98] Weyuker, Elaine J., Testing component-based software – a cautionary tale, in: IEEE Software, September/Oktober 1998, p. 54-59.

[WiPr01] Williams, Alan W.; Probert, Robert L., A Measure for Component Interaction Test Coverage, In: ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'01), Beirut, Libanon, Juni 2001, p. 304.

[Xiao03] Xiaoguang, M., A General Model for Component-Based Software Reliability, in: Proceedings of the 29th EUROMICRO Conference "New Waves in System Architecture" (EUROMICRO'03), Belek-Antalya, Türkei, September 2003, p. 395.

[Yan+99] Yang, Y., King, G ., Wickburg, H., A method for built-in tests in component-based software maintenance, in: Third European Conference on Software Maintenance and Reengineering (CSMR), Amsterdam, Niederlande, März 1999, p. 186.