



Technical Report: ALR-2005-017

## Proceedings



### International Workshop on Software Product Line Testing

September 26, 2005  
Rennes, France

Co-located with SPLC 2005 -  
9<sup>th</sup> International Software Product Line Conference

**Birgit Geppert, Charles Krueger, Tim Trew  
(Eds.)**

---

September 2005

The Avaya Labs Research Technical Report series is available online at:  
<http://www.research.avayalabs.com/techreport.html>

ALR-2005-017

Copyright © 2005 Avaya Inc., All rights reserved.

## Preface

One year ago, as we prepared for the first SPLiT workshop, we were not sure what to expect. We knew that testing played a critical role in software product line engineering and that it wasn't receiving nearly as much attention as the role of development. We knew that if development increased its production output by a factor of 10 but testing didn't, the bottleneck simply moved one step downstream in the engineering process. The organizers all had their opinions and experiences to draw on, but we weren't sure what message to expect from the broader community.

A clear theme emerged from SPLiT 2004. Indeed, it was one of the themes that reverberated in the halls of SPLC 2004 long after the workshops were over and engineers had turned their attention to the conference. It was one of the key contributions that emerged from the overall SPLC 2004 conference.

The clear message from SPLiT 2004 was that the inherent complexity and combinatorics associated with product lines can easily become intractable and overwhelm even a highly sophisticated and automated test organization. The problem of testing a software product line cannot be solved solely from inside the test organization, but rather must be considered well upstream in the definition, architecture, design, and implementation of a product line. While generality and broad scope may appear attractive to the development organization, they must be constrained to within practical limits for the test organization and the overall product line effort to be successful.

With SPLiT 2005, we carry forward the momentum and build on the foundation of SPLiT 2004. The core theme starting off will be on methods, tools, and techniques to effectively constrain the complexity and combinatorics of software product line testing. But beyond that, as we prepare for SPLiT 2005, we are not sure what to expect...

The SPLiT Organizers  
Birgit Geppert  
Charles Krueger  
Tim Trew

September 2005

## Organization

SPLiT is co-located with SPLC Europe 2005, the 9<sup>th</sup> International Software Product Line Conference, and held in Rennes, France, September 26.

### Workshop Chairs:

- Birgit Geppert  
Avaya Labs, Basking Ridge, NJ, USA  
[bgeppert@research.avayalabs.com](mailto:bgeppert@research.avayalabs.com)
- Charles Krueger  
BigLever Software, Austin, TX, USA  
[ckrueger@biglever.com](mailto:ckrueger@biglever.com)
- Tim Trew  
Philips Research, Eindhoven, The Netherlands  
[tim.trew@philips.com](mailto:tim.trew@philips.com)

### Program Committee:

- Hira Agrawal, Telcordia, USA
- Günter W. Böckle, Siemens AG, Germany
- Paul Clements, SEI, USA
- Krzysztof Czarnecki, University of Waterloo, Canada
- Claudia Fritsch, Bosch, Germany
- Jean Hartman, Microsoft, USA
- Peter Knauber, University of Applied Science Mannheim, Germany
- John Linn, Texas Instruments, USA
- John McGregor, Clemson University, USA
- Frank Röbler, Avaya Labs, USA

## Table of Contents

### Keynote

Reasoning about the Testability of Product Line Components .....	1
<i>John D. McGregor</i>	

### Accepted Papers

Executing Reusable System Tests for the Applications Derived from Software Product Lines .....	8
<i>Erika Olimpiew, Hassan Gomaa</i>	
Composing Unit Tests .....	16
<i>Markus Gälli, Orla Greevy, Oscar Nierstrasz</i>	
Towards Testing Response Time of Instances of a web-based Product Line .....	23
<i>Dharmalingam Ganesan, Ulrich Maurer, Michael Ochs, Björn Snoek, Martin Verlage</i>	
Product Line Testing and Product Line Development — Variations on a Common Theme .....	35
<i>Peter Knauber, William Hetrick</i>	

### Presentations

SPLiT - Introduction .....	41
<i>Birgit Geppert, Charles Krueger, Tim Trew</i>	
Composing Unit Tests .....	46
<i>Markus Gälli, Orla Greevy, Oscar Nierstrasz</i>	
Towards Testing Response Time of Instances of a web-based Product Line .....	50
<i>Dharmalingam Ganesan, Ulrich Maurer, Michael Ochs, Björn Snoek, Martin Verlage</i>	
Product Line Testing and Product Line Development — Variations on a Common Theme .....	58
<i>Peter Knauber, William Hetrick</i>	
Managing the complexity of your test space: challenges, ideas, solutions .....	61
<i>Birgit Geppert, Charles Krueger, Tim Trew</i>	
Reasoning about the Testability of Product Line Components .....	65
<i>John D. McGregor</i>	

### Supplementary Material

Guidelines for Discussion .....	76
---------------------------------	----



# Reasoning about the Testability of Product Line Components

John D. McGregor

Clemson University  
[johnmc@cs.clemson.edu](mailto:johnmc@cs.clemson.edu)

**Abstract.** The testability of a software component is the ability of the software to reveal its faults. In the development of high reliability systems, testability is an important quality attribute for guiding architecture decisions. The reuse of assets in a software product line propagates defects as readily as correct code. The strategic levels of reuse in a product line produce a high level of inter-dependency among the products in the product line that support this propagation. The increased frequency with which product line components are executed and the range of inputs over which they operate influence the amount of testing required to achieve specific levels of reliability. In this paper we begin the definition of a reasoning framework for testability by considering the characteristics of a product line that influence our view of how testable a component is. **Keywords:** testability, reachability, software product line

## Introduction

Many software product lines include *improved quality* as one of their goals, but improved with respect to what benchmark? The average customer's satisfaction with the product they purchased can be very different from the organization's satisfaction with the aggregate of all the products they sell. Even a small increase in product failures can result in increased help desk costs and increased liability particularly when a strategic level of reuse of core assets is achieved.

The testing activities in a product line can make an important contribution to achieving quality goals. The unique environment of a product line places equally unique requirements on the testing process. The strategic levels of reuse of assets achieved in a product line can counterbalance the desire for increased quality unless the quality requirements on the core assets are sufficiently stringent.

The testability of the product line constrains the degree to which defects can be identified using testing. Testability is often overlooked as a quality attribute of the architecture because it is not a quality directly visible to users. Testability, buildability and other production-related qualities are often critical to meeting the goals of the product line organization [Chastek 02]. We consider how to reason about architecture alternatives in terms of their testability.

Voas et al defines testability of a program P to be a prediction of the probability of software failure occurring if the software were to contain a fault, given that software execution is with respect to a particular input distribution [Voas, 95]. This definition relates testability to our ability to make the software fail and to the context within which the software is operated. Other definitions of testability [Kansomkeat 05] relate testability to having visibility into the software, being able to control the software, and being able to detect failures. Finally, testability is related to the size or complexity of the system as it relates to the effort needed to implement test criteria [Birgisson et al. 1999].

Testability is a requisite quality attribute in product line software if we intend to maintain or improve the level of aggregate quality. However, increases in testability can negatively affect other important qualities such as encapsulation. In this paper we argue that higher levels of testability for assets are required in a product line than in "one off" product development and we illustrate some of the trade offs when this is attempted.

The remainder of this paper is organized as follows. In the next section we put testability into the context of a product line to illustrate the problem we see. We then briefly present techniques for increasing testability. We then present a reasoning framework for the testability quality attribute for use in making architectural decisions. Finally we map out our future work in this area.

## Testability in the Product Line Context

There are two characteristics that illustrate the importance of testability of the software in a software product line. The strategic levels of reuse achieved in a product line result in a very large number of executions of a component relative to its use in a one-off system. The component is also exposed to more different contexts in a product line than in a one-off system where it is only placed in the context of the one system. In this section we explore the ramifications of these characteristics.

Consider that in custom development of certain types of systems, only one copy of the system ever exists so only one copy of its components exists. For example, the ground station designed to control a communication satellite. It is executed the number of times required by the algorithm. When executed, it processes a set of inputs. The component is executed over some set of data that is a subset of the total possible input. Although the data varies over time, it has a single user on a single hardware configuration; the data has a fairly consistent variation. After the initial plateau of finding defects, new defects are usually discovered only when the hardware or operating system is upgraded. Testing is a relatively straight-forward process with a fairly narrow range of test cases.

In consumer product development, multiple copies of a product exist and therefore multiple copies of a given component. For example, tax preparation software. A given component will see a somewhat different profile of input data from one copy of the system to another because of differences between users. Both the hardware and system software may vary. We will expect to see a wider range of defects exposed than in the custom product. Testers must select data over a wider range or run tests with multiple configurations.

In product line development, a component is used, not only in multiple copies of the same product on different hardware, but also in multiple different products. For example, the core asset base for a product line of cell phones. If we make the logical assumption that the total sales of products in the product line exceeds the sales for the single consumer product, the number of executions that all the instantiations of a component accumulate over time is probably much higher.

For a given component, if it is implemented in a single-product, custom development effort, we will assume that a component C is executed  $x$  times for a specified time period. In development where multiple copies of the product are deployed the same component will now be executed  $nc * x$  times in a specified time period,  $nc$  is the number of copies. In product line development, the same component will now be executed  $\sum_{i=1}^{np} (nc_i * x_i)$  times in the same time period, where  $np$  is the number of products,  $nc_i$  is the number of copies of each product, and  $x_i$  is the number of executions for a given product.

In the above scenario, assume that the probability of a defect in the component causing a failure is  $P(d)$ . Obviously the number of failures observed in the product line scenario will likely be greater than the other two scenarios as long as  $P(d)$  remains constant. The expected number of failures<sup>1</sup> can be stated as:

$$expectedNumFailures = P(d) * \sum_{i=1}^{np} (nc_i * x_i)$$

In product line development, a component is used in multiple products. These products may have different levels of certain quality attributes and different types of users. We expect that the range of input data presented to a component will vary from one product context to another. Therefore,  $P(d)$  does not remain constant, it varies from product to product. If we assume that when a failure occurs in one product, its failure is known to the organization, the number of failures can now be stated as:

$$expectedNumFailures = \sum_{i=1}^{np} P_i(d) * (nc_i * x_i)$$

This argument leads to the conclusion that traditional rules of thumb used by testers and developers about the levels of testing to which a component should be subjected will not be adequate for a software

---

<sup>1</sup> We assume that execution of a defect leads to a failure.

product line environment. The increased number of executions raises the likelihood that defects will be exposed in the field unless the test coverage levels for in-house testing are raised correspondingly.

This is not to say that individual users will see a decline in reliability. Rather, the increased failures will be experienced as an aggregate over the product line. Help desks and bug reporting facilities will feel the effects. The weight of this increase in total failures may result in pressures, if not orders, to recall and fix products.

$P_i(d)$  can be lowered by removing more defects before system deployment. Test coverage may be improved by a judicious increase in the selection and number of test cases that are executed. This may not be sufficient for some circumstances. If the products have low testability it may not be possible to greatly increase the defect finding power of our testing. The defects may result from minor variations that are not externally visible. Therefore, we will focus on achieving higher levels of testability.

## **Increasing testability**

Before presenting the reasoning framework, we consider ways to increase the testability of software. The techniques that can be used depend on the form of the software. Since it affords the widest scope for discussion, we will assume that we have access to the source code and can build the software. We will briefly discuss the case where object code only is available.

The definitions for testability presented earlier relate to the ability to observe the internal state of the software under test and to control that state in order to begin tests at various points in the software's state machine. Therefore, to increase testability, the observability of the software and/or its controllability must be improved. Some definitions also discuss being able to detect the failures. We will address this by assuming that the test suite for a piece of software is selected using a specific criteria. This criterion will be selected to be compatible with the form of the software, source or object, and the techniques used to provide access [Nikora 03].

An architect has several choices for enhancing the testability of components.

1. The architect may allow general access from outside the module to specific elements within the component via either direct memory access or via accessor methods.
2. The architect can provide a test interface to a component that provides accessor methods for variables within the module. The interface makes it easier to check for unauthorized use of the test interface by any but the test harness or to set permissions for access to the interface. The test interface allows outside code a means to reach certain variables. This increases the visibility of the software under test to the test code.
3. Another approach is to build the test cases into the software under test. This has been practiced by some object-oriented software designers. This, however, is not a good practice particularly when the software is operating in a resource constrained context.

In the next section we discuss a reasoning framework for making testability decisions at the architecture level.

## **Testability reasoning framework**

Bass et al have outlined the requirements for a framework for reasoning about quality attributes of architectures [Bass 05]. Each framework is based on an analytic theory that allows exacting comparisons between design alternatives. We will use their outline as the basis for sketching a possible reasoning framework for the testability of components defined by a product line architecture. The headings used in this section come from the Bass outline.

We will focus on the common ground among the definitions of testability by focusing on visibility and controllability. Even more basically, we will be concerned with whether the structure of the software allows the successful implementation of whatever test criteria is selected.

## **Problem description**

The proportionately larger number of executions to which much of software product line code is subjected requires improved testing to maintain the same level of aggregate quality. This need for improved testing can be partially addressed by setting more complete test criteria that add test cases to cover more values within the expected range of inputs. However, significant portions of the component's behavior may be untestable if high levels of encapsulation are achieved. There is a need to have highly testable products.

Unlike most other quality attributes, the architect must consider software that is not part of the product being architected when evaluating for testability. A component is usually harnessed into a test framework for unit testing. The architect must plan how that harness will interact with the software under test.

The following general scenarios, which also follow a structure specified by [Bass 05], partially describe the problem:

### **S1:**

Stimulus – A component is checked in for unit testing.

Source of stimulus – The component developer

Environment – In the component development phase with limited amount of component integration occurring.

Artifact – The component under test, the test harness, and the test cases

Response – Test cases are selected to the limit allowed by the testability of the component

Response measure – The extent to which chosen test criteria can be achieved

### **S2:**

Stimulus – A subsystem is successfully built prior to integration testing.

Source of stimulus – An integration team member

Environment – sufficient components have passed unit test and have been integrated

Artifact – The subsystem under test, the test harness, and the test cases

Response – test cases are selected to the limit allowed by the testability of the component

Response measure - The extent to which chosen test criteria can be achieved

## **Analytic theory**

Testability is based on two attributes: visibility and controllability. To validate test results, it must be possible to see the current value of a variable in order to compare it with the expected value. To conduct the widest range of tests possible it must be possible to set the value of a variable to establish a starting point for the test. The analytic theory relates the degree to which these two attributes can be accessed to the testability of the software.

There are several theories that have been developed for testability. Most require completed code before evaluating the testability and many require measurements taken during execution of the code. Voas has perhaps the most comprehensive approach that computes probabilities of a statement being reached by a test case, the probability of a fault being introduced by mutants, the probability of that fault causing a failure. However, this technique, like many others, can not be applied at the architecture level. Voas also has defined an architectural level measure Domain to Range Ratio (DRR) [Voas and Miller 95]. There are some problems with fault masking with this measure.

An alternative theory for estimating testability is to compute the average module size [Hatton 97]. Fenton et al show that this relationship is very complex [Fenton et al 99]. Fenton et al argue that there is a relationship between testability and the amount of testing needed. This relationship is not sufficiently well understood to include in our analytic theory.

We propose an approach that, while less exact, uses a form of reachability analysis to obtain direct measures of visibility and controllability. This analysis uses module specifications and their state machine specification to provide measures of visibility and control. Like code-level reachability analysis this analysis constructs graphs (actually Labeled Transition Systems) that consider all possible paths from an origin through the state space defined by the architecture specification. This theory lends itself to

component-based development since it is a compositional technique. The compositionality reduces the state space and thereby increases the size of components that can be analyzed.

We are interested in finding those counter examples where a state can not be reached rather than the ones identified by the analysis. Those states which can not be reached indicate areas that would be hidden from test cases.

The architect must consider that some defect types will defy discovery more than others. This is translated into a concern about how easy it is to fulfill test criteria. For example, event-based test criteria are more difficult to apply than time-based [Lindstrom 00]. This makes the software inherently less testable. The architect makes choices, e.g. to use events or not, that make one test criteria more desirable than others. While this is difficult to quantify at this point, it is nevertheless an important result of the architecture analysis process.

### **Analytic constraints**

The major constraint on the reachability analysis is the potential for state explosion in the reachability graph. There are several techniques, some of which are being investigated in model checking research, which can be used to address this issue based on a number of factors. Giannakopoulou et al discuss techniques for reducing the state explosion problem.

The state explosion problem varies as the level of software under test moves from component-level testing to system-level. The larger the piece of software under test, the larger the state space. By focusing on testability of components rather than whole systems, we minimize the affect of this constraint as well as testing earlier in the development process.

A second constraint on the reachability analysis, is that most techniques for this type of analysis are static. The theory is not effective for dynamic designs such as reflective, or self-modifying, code. Some work is seeking to extend the techniques to dynamic actions such as dynamic class loading [Rountev 01].

The major constraint in a software product line the major constraint is the need to accommodate variability. Most of the techniques by which variability is provided in an architecture break up the architecture. It is across interfaces such as this that most problems arise with static analysis techniques.

### **Model representation**

There is a variety of reachability tools available depending upon the form of the system to be analyzed. There are tools for many programming languages and some architecture description languages. For example, Tracta provides a compositional reachability analysis for the Darwin ADL [Giannakopoulou, 99].

Most of the tools use a graph-based representation. The graphs are directed with edges running from a program point to a specified statement such as an attribute definition or use. Each node represents a program state. The Tracta implementation uses Labeled Transition Systems (LTS) as the representation for the reachability output [Cheung 99].

### **Interpretation**

The components and connectors in the architecture description of the component are translated into the links and nodes of a reachability graph. Each node represents a particular state configuration for the component. Each link is the method invocation that would transform the system into that state. Each node is connected to possible states of the components to which they are attached.

### **Evaluation procedure**

For the purposes of evaluating testability, a reachability analysis is conducted on the component under test. The analysis is conducted between the point at which test cases will be applied to the component – some interface - and the interior structure of the component where sub-components are defined. The analysis will identify all attributes, the substructure of the component, which can be reached from the component's

interfaces. The analysis will then identify those states that can not be reached due to attributes that can not be accessed.

The analysis can be conducted for each architecture design option for a component. Then the analysis results for the design alternatives are used to rank each design as to testability. These results are integrated with the other quality attribute analyses to make the architecture decision.

## Designing for Testability

Testability can be viewed as having a negative impact of design quality overall. For example, increasing visibility of attributes decreases the information hiding quality of the design and may threaten the modularity of the design. Reporting 100% visibility for the attributes in a piece of software under test would seem to completely defeat information hiding. However, if that level of testability is gained by adding a separate test interface that gives explicit access to attributes, the impact on information hiding is controlled and focused in a single interface. Tool support can be provided to ensure that inappropriate – any production code using the test interface – uses of the test interface are identified and removed. However, the test interface, and its implementation, increases the size of the executable. This has a negative impact on resource-constrained devices such as embedded systems or wireless sensor networks.

There may be process-oriented solutions to this problem such as having the test interface present in the code during testing but removed during product building. There are obvious problems with this proposal as well.

## Summary

This brief introduction is a request for input. We welcome suggestions for alternative analytic theories about testability and for additional architectural tactics for ensuring high levels of testability in a product line architecture. Our research will progress into specific techniques for specified domains. One particular target is resource constrained real-time, embedded devices such as braking assemblies for vehicles.

We have argued that components developed in a software product line need increased testability because they need increased test coverage. Otherwise the product line organization will experience an increase in problem reports and product failures. Beginning with the testability of the architecture is an essential first step.

## References

Birgisson, R., Mellin, J. and Andler, S. Bounds on Test Effort for Event-Triggered Real-Time Systems, The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99), 1999.

Gary J. Chastek and John D. McGregor. Guidelines for Developing a Product Line Production Plan, CMU/SEI-2002-TR-006.

Shing Chi Cheung and Jeff Kramer. Checking safety properties using compositional reachability analysis, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 8 , Issue 1 (January 1999).

Norman E. Fenton and Martin Neil. A Critique of Software Defect Prediction Models, IEEE Transactions on Software Engineering, VOL. 25, NO. 5, SEPTEMBER/OCTOBER 1999.

D. Giannakopoulou, J. Kramer, and S.C. Cheung, "[Analysing the Behaviour of Distributed Systems using Tracta.](#)" Journal of Automated Software Engineering, special issue on Automated Analysis of Software, vol. 6(1), pp. 7-35, January 1999. R. Cleaveland and D. Jackson, Eds.

L. Hatton, "Re-examining the Fault Density-Component Size Connection," *IEEE Software*, vol. 14, no. 2, pp. 89-98, Mar./Apr. 1997.

Birgitta Lindstrom. *Methods for Increasing Software Testability*. MS Thesis, University of Skovde, 2000.

Supaporn Kansomkeat, Jeff Offutt, Wanchai Rivepiboon. INCREASING CLASS COMPONENT TESTABILITY, Proceedings of the 23<sup>rd</sup> IASTED, 2005.

Allen P. Nikora, Raphael R. Some, and Yuval Tamir. *Increasing Software Testability with Standard Access and Control Interfaces*, ISSRE 2003.

A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java based on annotated constraints. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, Oct. 2001.

J. M. Voas. Software Testability Measurement for Assertion Placement and Fault Localization, in M. Ducasse (ed.), (AADEBUG) 2<sup>nd</sup> International Workshop on Automated and Algorithmic Debugging, IRISA-CNRS, pp. 133 – 144, 1995.

J. M. Voas and Keith W. Miller. Software testability: the new verification. *IEEE Software*, 12(3): 17-28, May 1995.

# Reusable System Tests for Applications Derived from Software Product Lines

Erika Mir Olimpiew and Hassan Gomaa

Dept. of Information and Software Engineering  
George Mason University  
4400 University Dr. MS 4A4  
Fairfax, VA, 22030  
[colimpie@gmu.edu](mailto:colimpie@gmu.edu), [hgomaa@gmu.edu](mailto:hgomaa@gmu.edu)

**Abstract.** This paper describes a system testing process for software product lines and applications derived from these product lines. System testing concepts introduced previously, such as the creation of test item, test design and test procedure specification are fitted into the SPL engineering and application engineering processes of an SPL. By this means, both the application and reusable system tests are derived by selecting the required features for the application.

## 1 Introduction

This paper describes a system testing process for software product lines and applications derived from these product lines. This research builds on previous research on SPL testing but differs because it describes how a testing process fits into the SPL engineering and application engineering processes of an SPL. Model-based testing for SPLs was introduced in [1]. This paper describes the role of the system test item, test design, and test procedure specification documents in the SPL engineering processes.

Section 2 describes related work on SPLs and SPL-based testing techniques. Section 3 describes a testing process for single systems, and section 4 describes how that process is extended for a SPL.

## 2 Related Work

### 2.1 Software Product Lines

Software product line development consists of SPL engineering and application engineering (Figure 1). *SPL engineering* is the development of requirements, analysis and design models for a family of systems that comprise the application domain. A *family of*

systems [2], or product line, is a collection of systems that have so much in common that it is worthwhile to study and analyze the common features before analyzing the features that differentiate the systems. Several software product line development methods have been investigated by [3-9].

During *application engineering*, the software product line models are adapted to derive a given software application, which includes all the common features and selected optional and alternative features. An executable application may also be constructed, by generating, or selecting implemented components associated with a selected feature, or by customizing a product line framework to enable or disable functions depending on whether or not a feature is selected.

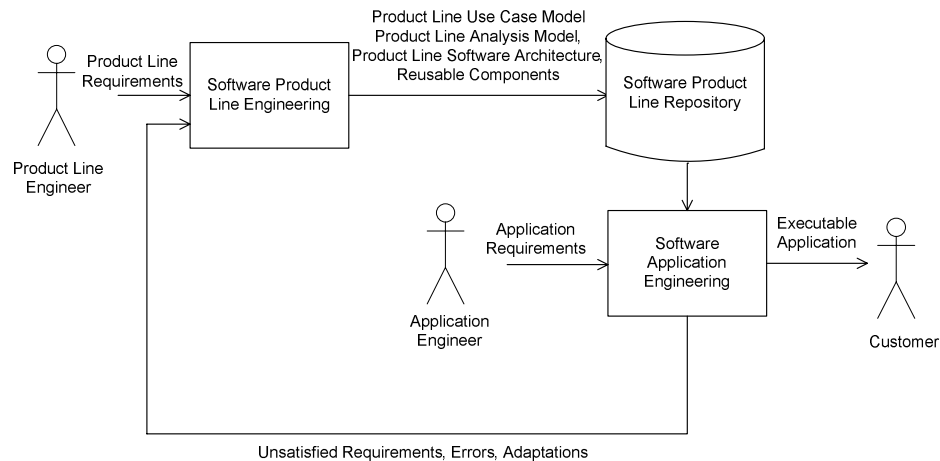


Fig. 1 SPL Engineering and Application Engineering Processes

## 2.2 SPL-based Testing Techniques

Using SPL based testing techniques, reusable tests are created from the SPL functional requirements, which can be customized to test the applications derived from SPLs. McGregor [10] discussed abstract and combinatorial tests for SPLs. Kamsties et al [11] described three methods to create tests from SPL use case requirements using parameterization, fragmentation and segmentation techniques, and described how to apply these tests with a use case based coverage criterion. Bertolino et al [12] extended the category partition testing strategy for product line use cases. Nebut et al [13] described how to instantiate use case contracts for an application derived from a SPL, how to apply some testing coverage criteria based on these contracts, and also described a robustness

testing strategy. Geppert et al [14] investigated the problem of defining and managing the relationship between a feature model and tests using a decision model.

### 3 Testing Process for Single Systems

IEEE standard 829-1998 [15] describes test item, test design and test procedure specification documents for system testing a single system. A *test item specification* describes a test case, which contains a test objective, inputs, outputs and test case dependencies. A *test design specification* describes a testing strategy, the functions that will be tested, and the relationship between the functions and tests. A *test procedure specification* describes the procedure for executing the tests.

Model-based testing for single systems is described in [1] and summarized here. Model-based testing is an approach whereby test cases for a system are created from the models of the software system rather than from the software system itself.

A system test template is constructed from an interaction diagram that describes a use case scenario. The input and output messages in the sequence diagram become test steps in the system test template. The message parameters and system state variables become test parameters. The use case precondition becomes part of the template, and describes part of the system state. A test condition and postcondition are also added to the template to specify constraints on the values of input variables and to describe expected output values. A test template can be used to generate the specific data values for a test item specification, by substituting the data values that satisfy a test template's test condition into the test parameters.

The test design specification describes one or more functional testing strategies and a traceability matrix. A testing strategy is a rule that guides test template selection and / or the generation of test items from these templates. A traceability matrix shows the relationships between use cases and test templates.

The test procedure specification includes descriptions of a test execution model and a test harness. A test execution model describes the order in which tests can be executed. This order is determined by examining the use case scenario dependencies. Use case scenario A depends on use case scenario B if the postcondition of B is part of a precondition of A. The precondition and postcondition constraints are described in the test template associated with the use case scenario.

A test harness executes test items generated from the test template. A test item is executed if its test condition can be enabled, given the current system state. If multiple test conditions are enabled during test selection, only one of the test items is chosen for execution. The actual results are then compared with the expected results to determine if the test passed or failed.

## 4 SPL testing process

This section describes how the testing process for single systems is expanded to address the software product line processes of a SPL. Product line test cases are developed during product line engineering. The test cases are feature based, such that there are common test cases, which are used to test all members of the product line, and variable test cases, which are used to test some of the applications. Application derivation is feature-driven, so that the application is derived from the product line architecture and components by selecting the appropriate features. The same feature selection is used to derive and reuse the application test cases from the product line test suite. The test derivation process is shown in Figure 2.

The product line testing approach is model-based [1], such that the product line test cases are determined from models of the product line. In this research, the models are developed using the PLUS (Product Line UML based Software engineering) method, which depicts multiple-view models using the UML notation [6]. However the testing approach could be used with any model based product line method.

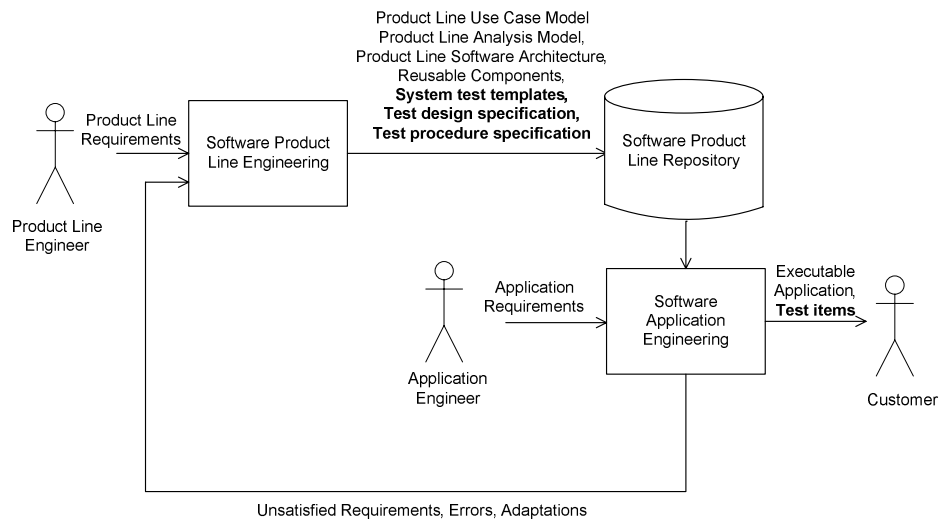


Fig. 2 Test derivation process

### 4.1 SPL engineering

#### 4.1.1 Testing process

The test templates, test item, test design and test procedure specifications are developed during SPL engineering, then selected and customized during application engineering. Some system testing of the system core may also be carried out during SPL engineering. Depending on the SPL development method, it may be necessary to build stubs to simulate the components that correspond to variable features.

#### **4.1.2 Test templates and test item specifications**

The use case model is the starting point for the creation of test item specifications. With the use case modeling approach, the functional requirements of the system are described in terms of actors and use cases. A use case in a single system contains one or more scenarios, a main scenario and usually several alternative scenarios. A main scenario describes a sequence of interactions between one or more actors and the system. The alternative scenarios describe a sequence of inputs and actions that differ from a typical use of the system, such as for error handling. Each use case scenario can in turn be described with an interaction diagram that shows the sequence of interactions between the use case actors and the system.

The use cases of a product line are categorized as kernel, optional or alternative, and correspond to common, optional and alternative features in the SPL feature model [6]. These use cases may also contain variation points, which describe locations in the use case description where optional and alternative parameters, inputs and actions may be inserted or enabled when an optional or alternative feature is selected for an application derived from the product line [6].

As in the single system testing method, each use case scenario is refined into a test template that contains a precondition, test condition and a postcondition. The test templates of a SPL contain an additional section for feature conditions. A feature condition is a Boolean variable that is set to true when the corresponding feature is selected for an application derived from the SPL. Test items are generated from test templates that are selected for an application derived from an SPL.

#### **4.1.3 Test design**

The test design specification for a software product line includes an additional section for a SPL-based testing strategy, which describes feature combinations, or specific target applications to test. Single system testing strategies are applied to each target application. Further, the traceability matrix section is extended to describe the relationships between the features in the feature model to the use cases and test templates.

#### **4.1.4 Test execution model**

As in the single system testing method, an interaction reference is created for each test template. Test templates are grouped by use cases into other interaction references. A test

template that is associated with one or more variable features in the feature model contains feature conditions. In the interaction overview diagram, these feature conditions are depicted in a constraint of the decision node guarding the test template.

#### **4.1.5 Test harness**

The test harness for an SPL is similar to that of a single system. It can be implemented as the driver program described in [16] and customized for each application derived from the SPL.

### **4.2 Application engineering**

#### **4.2.1 Testing process**

Application derivation and application test case generation are both feature driven. A target application is configured during application engineering and, at the same time, the test templates for this application are selected. The test design and test procedure specifications are customized, and test item specifications are generated from the selected templates according to the single-system testing strategies described in the test design specification.

#### **4.2.2 Test templates and test item specifications**

A system test template with no feature conditions is always selected for each application derived from the SPL. If a system test template contains one or more feature conditions, it is selected when all of the features associated with the feature conditions are also selected for the target application. The selected test templates become part of the application's test template suite.

#### **4.2.3 Test design**

Test items are generated from each test template of the application's test template suite according to the single-system testing strategies described in the test design specification.

#### **4.2.4 Test execution model**

The test execution model is also customized for a target application, by enabling the feature conditions that correspond to the selected features, and disabling feature conditions for the unselected features. Disabling a feature condition in a test template prevents the selection and execution of test items associated with the test template.

#### 4.2.5 Test harness

The test harness is also customized for the target application, by disabling references to test templates that are not included in the application's test template suite.

### 5. Conclusions

This paper has described how the testing process for a single system can be expanded to address the SPL engineering and application engineering processes of a SPL. The test item, test design and test procedure documents of an IEEE standard are extended for SPLs during SPL engineering, and then customized for a target application during application engineering.

Future work will map test templates to an executable specification language and test generator tool such as described in [17] in order to automatically generate test items and test sequences for the target applications of a SPL. Then, test items will be mapped to tests that can be executed against the target applications of the SPL. Future work will also experiment with separation of concerns techniques for certain types of variable features. Separation of concerns techniques can be used to reduce redundancy and maintenance effort of the test template suite of a SPL.

### 6. References

1. Olimpiew, E.M. and H. Gomaa. *Model-based Testing For Applications Derived from Software Product Lines*. in *Advances in Model-based Testing*. 2005. St. Louis, Missouri.
2. Parnas, D.L. *Designing Software for Ease of Extension and Contraction*. in *3rd International Conference on Software Engineering*. 1978. Atlanta, Georgia, United States.
3. Kang, K.C., et al., *FORM: A feature-oriented reuse method with domain-specific reference architectures*. *Annals of Software Engineering*, 1998. **5**: p. 143-168.
4. Weiss, D.M. and C.T.R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*. 1999, Reading, MA: Addison-Wesley.
5. Clements, P. and L. Northrop, *Software Product Lines Practices and Patterns*. SEI Series in Software Engineering. 2002, Boston, MA: Addison-Wesley.
6. Gomaa, H., *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*. The Addison-Wesley Object Technology Series. 2005: Addison-Wesley Professional.

7. Gomaa, H. and D.L. Webber. *Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model*. in *Hawaii International Conference on System Sciences*. 2004. Big Island, Hawaii.
8. Gomaa, H. and M. Saleh. *Software Product Line Engineering for Web Services and UML*. in *IEEE International Conference on Computer Systems and Applications*. 2005. Cairo, Egypt.
9. Saleh, M. and H. Gomaa. *Separation of Concerns in Software Product Line Engineering*. in *Workshop on the Modeling and Analysis of Concerns in Software Product Line Engineering*. 2005. St. Louis, Missouri.
10. McGregor, J.D., *Testing a Software Product Line*. 2001, SEI.
11. Kamsties, E., et al. *Testing Variabilities in Use Case Models*. in *Software Product-Family Engineering: 5th International Workshop*. 2003. Siena, Italy.
12. Bertolino, A. and S. Gnesi. *PLUTO: A Test Methodology for Product Families*. in *Software Product-Family Engineering: 5th International Workshop*. 2003. Siena, Italy.
13. Nebut, C., et al. *A Requirement-Based Approach to Test Product Families*. in *Software Product-Family Engineering: 5th International Workshop*. 2003. Siena, Italy.
14. Geppert, B., J. Li, F. Robler, and D. M. Weiss. *Towards Generating Acceptance Tests for Product Lines*. in *8th International Conference on Software Reuse*. 2004. Madrid, Spain: Springer-Verlag.
15. IEEE, *IEEE standard for software test documentation*, in *IEEE Std 829-1998*. 1998.
16. Ben Potter, Jane Sinclair, and D. Till, *Chapter 11: From specification to program: operation decomposition*, in *An Introduction to Formal Specification and Z*. 1996, Prentice Hall. p. 282-284.
17. Wolfgang Grieskamp, Nikolai Tillmann, and M. Veanes, *Instrumenting scenarios in a model-driven development environment*. *Information and Software Technology*, 2004. **46**: p. 1027-1036.

# Composing Unit Tests<sup>\*</sup>

Markus Gälli, Orla Greevy, and Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland  
{gaelli,greevy,oscar}@iam.unibe.ch

**Abstract.** If we were to apply the testing techniques of object-oriented systems prescribed by the XUnit framework to a car factory, the result would be an inefficient process: A tire would be created, quality assured and then thrown away, only to be recreated later to test the functionality of the whole car.

XUnit makes it difficult to reuse intermediate results of low level unit tests. As a consequence a higher level unit test is forced to recreate test scenarios which were already created by lower level unit tests. This duplicated testing effort is time-consuming both for setting up new scenarios and for running the tests. To address this problem we suggest a semi-automatic approach to compose tests. First we describe how we can detect candidates of composable test cases by partially ordering their sets of covered method signatures, then we present techniques to refactor unit tests accordingly.

**Keywords:** Unit testing, factories, XUnit, composition

## 1 Introduction

A software product line is defined as a 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.

XUnit [BG98] in its various forms is a widely-used open-source unit testing framework. It has been ported to most object-oriented programming languages and is integrated in many common IDEs such as Eclipse.

We claim that units under test can not only be single methods or classes but also whole software components of a software product line. By allowing the unit test to deliver the tested core asset as a return value, we can reuse the tested core asset in assembling a test for a composed asset in order to facilitate scenario creation and to reduce testing time.

The XUnit framework does not allow low level unit tests themselves to be composed into higher level unit tests - whereas low level functionality is composed out of lower level functionality.

---

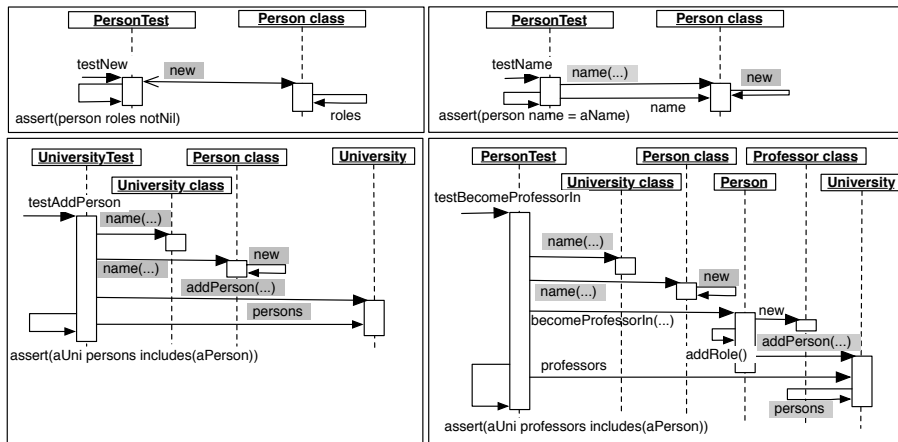
<sup>\*</sup> We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “A Unified Approach to Composition and Extensibility” (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006)

This makes the set up of test scenarios an unnecessary tedious task and leads to unnecessary long testing times. Our hypothesis is that a majority of unit tests can be refactored into composed test cases.

We will explain our approach with an illustrating example of a simplified university administration system, which consists of the following four XUnit test cases:

- `PersonTest`»`testNew` tests if the roles of a person are defined.
- `PersonTest`»`testName` tests if the name of a person was assigned correctly.
- `UniversityTest`»`testAddPerson` tests if the university knows a person after the person has been added to it.
- `PersonTest`»`testBecomeProfessorIn` tests if some person, after having been added as a professor, also has this role.

In Figure 1 one can see, that all methods called in `UniversityTest`»`testAddPerson` are also called in `PersonTest`»`testBecomeProfessorIn`, but that neither the methods called in test case `PersonTest`»`testNew` nor in test case `PersonTest`»`testName` are also called completely in any other test case.



**Fig. 1.** The test for `#becomeProfessorIn`: covers the test for `#addPerson`:. The test for `#new` overlaps with the test for `#name`. Intersecting signatures are displayed gray.

## 2 Approach

We first present a technique to identify comparable test cases of existing test suites by sorting their sets of covered method signatures and then introduce two refactorings to tune these comparable tests.

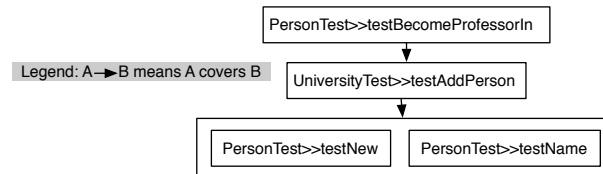
## 2.1 Identifying Redundant Test Cases with Coverage Sets

We say that unit test A *partially covers* unit test B ( $A \supseteq B$ ), if only up to *tolerance* method signatures covered by B are not included in the set of method signatures covered by A (1). We say that unit test A *overlaps* B ( $A \equiv B$ ), if A *partially covers* B and B *partially covers* A (2). We say that two unit tests A and B are *comparable* if at least either one *partially covers* the other.

$$A \supseteq B \Leftrightarrow |\text{Signatures}(B) \setminus \text{Signatures}(A)| \leq \text{tolerance} | \text{tolerance} \in \mathbb{N}$$

$$A \equiv B \Leftrightarrow A \supseteq B \wedge B \supseteq A \quad (\mathcal{A})$$

Based on this partial order we developed the following algorithm to identify comparable test cases, which are candidates for refactoring: First we instrument the code and obtain traces of method calls that are invoked during the execution of the tests. Then we extract and store the set of method signatures of each test into an `InstrumentedTestCase` object. We sort all this `InstrumentedTestCase` objects according to the cardinality of the sets starting with the smallest. For each `InstrumentedTestCase` we detect the first covered test, that is both bigger than it and includes its method signatures tolerating *tolerance* methods not to be included in the bigger one. If we find one, we move the partially covered one into the covering one. If we find that both partially cover each other, we merge these two tests, building an equivalence relation between them. For our example using a *tolerance* 2 of we end up with a partial order of our tests depicted in Figure 2.

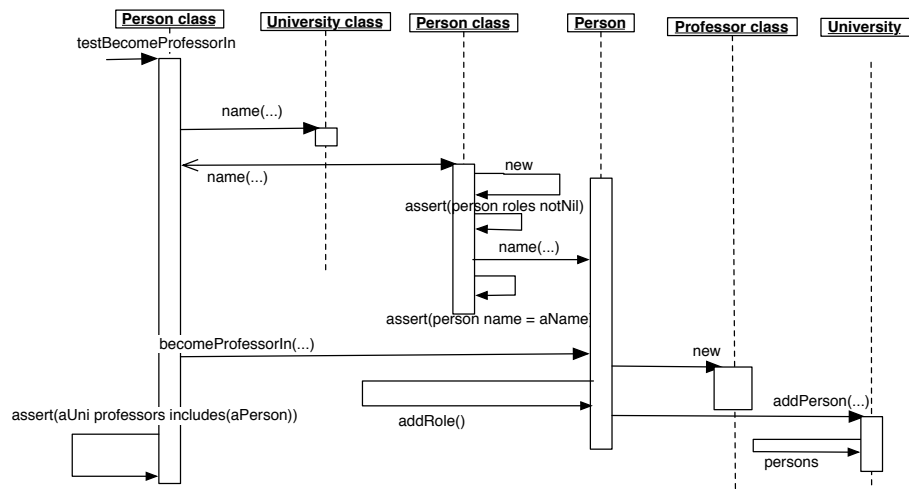


**Fig. 2.** A sample test hierarchy based on coverage sets.

## 2.2 Refactoring comparable test cases

Having identified comparable tests forming a partial order, we can refactor them: We start again with the smallest test case and try to include it into its next smallest comparable test case with either of the following two refactorings:

- *Abstract assertions:* Move the assertion from the test into a post condition of the method under test. In our toy example the assertions are already abstract enough to work directly as post conditions of the method under test. We thus could move the includes assertion of the `UniversityTest`  $\gg$  `testAddPerson` into a post condition of `University`  $\gg$  `addPerson` itself. Otherwise one can try to convert the concrete assertion of the unit test into an abstract assertion serving then as the post condition.
- *Publish Test Result:* If an object created by a low level test can be immediately used as parameter or receiver for the method under test of a higher level test, we can directly call the low level test from our higher level test, eliminating the need to run the low level test standalone and to recreate the scenario. This is certainly only possible, if the test framework allows us to let the tests return objects. In JUnit Version 4.0 all tests have to be void, thus this kind of easy test composition would not be possible there.

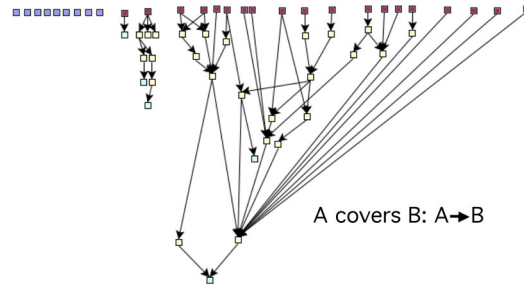


**Fig. 3.** The tests refactored. Only one test has to be run now, instead of four, as it captures all former subtests in post-conditions.

### 3 Status and Future Work

Having analyzed [GLNW04] the partial order of the unit tests of Code-Crawler, a code visualization tool, [Lan03] using a *tolerance=0*, the resulting graph looked like seen in Figure 4. Observing the three right subgraphs formed by the test covering relationship one can see that a

surprising high number of these tests are comparable, but we have not yet refactored them.



**Fig. 4.** The coverage hierarchy of the Code Crawler tests visualized with Code Crawler.

In [GLNW04] we automatically detected several covering relationships between unit tests like in the following interesting two examples (the former is always covering the later):

- LoaderTest»testConvertXMItoCDIF  
(LoaderTest»testLoadXMI)
- SystemHistoryTest»testAddVersionNamedCollection  
(SystemHistoryTest»testAddVersionNamed)

A possible refactoring suggested by the first covering relationship is to test the loader of some XMI data structure and, letting the test giving back the XMI structure, reusing the structure to convert it to another structure called CDIF in the covering test.

The second covering relationship indicates an n-to-1 relationship between addVersionNamedCollection and addVersionNamed, where reusing the result of addVersionNamed or changing the concrete assertion of addVersionNamed into some post condition can lead to a composed unit test.

We plan to refactor a big case study after having analysed it with our approach and show that we can efficiently reduce testing time and provide reusable tests. With this case study we want to answer the following questions:

- How much can we speed up the execution of the test suite?
- Can we decide automatically between several refactorings?
- Can we use our partial ordering of coverage to give us a hint, if a refactoring was successful and all former tests are still run?
- How can we prioritize our tests in an efficient way so that atoms are only run within their calling tests but not stand alone?

## 4 Related Work

In previous work we showed that failing unit tests are presented in a random order, whereas they could be presented in a meaningful order using

the partial order of covered method signatures. [GLNW04] We also defined a taxonomy of unit tests [GLN05], where we manually categorized more than 1000 unit tests of Squeak, an open source object oriented development system. Our results from this large case study show that most unit tests are either atoms, which we call one-method tests, or composable out of these one-method-tests. In [GND04] we suggested a Smalltalk browser where one can integrate tests with the methods under test and where tests are stored as factory methods on the class side of the returned object. Liebermann and Hewitt also tightly integrate testing and programming in [LH80] and reuse tests.

McGregor [McG01] suggested a way to compose partial tests along variation points.

Edwards also underlined the importance of examples [Edw04].

Test case prioritization [RUCH99] has been successfully used in the past to increase the likelihood that failures will occur early in test runs. The tests are prioritized using different criteria, the criterion which most closely matched our approach was *total function coverage* [EMR00]. Here a program is instrumented, and, for any test case, the number of functions in the program that were exercised by this test case is determined. The test cases are then prioritized according to the total number of functions they cover by sorting them in order of total function coverage achieved, starting with the highest.

## 5 Conclusion

We have presented a partial order using sets of covered method signatures to detect comparable test cases. We have introduced two refactorings which can be applied to some of these comparable test cases in order to reduce testing time and increase reuse of test scenarios. We have given first evidence that relevant portions of test cases do partially cover each other, and that results obtained by the partial order are semantically meaningful. We have not yet applied our approach to a big case study.

## References

- [BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.
- [Edw04] Jonathan Edwards. Example centric programming. In *OOPSLA 04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 124–124. ACM Press, 2004.
- [EMR00] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, 2000.

- [GLN05] Markus Gälli, Michele Lanza, and Oscar Nierstrasz. Towards a Taxonomy of SUnit Tests. In *Proceedings of ESUG Research Track 2005*, September 2005. To appear.
- [GLNW04] Markus Gälli, Michele Lanza, Oscar Nierstrasz, and Roel Wuyts. Ordering broken unit tests for focused debugging. In *20th International Conference on Software Maintenance (ICSM 2004)*, pages 114–123, 2004.
- [GND04] Markus Gälli, Oscar Nierstrasz, and Stéphane Ducasse. One-method commands: Linking methods and their tests, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.
- [Lan03] Michele Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.
- [LH80] Henry Lieberman and Carl Hewitt. A session with tinker: Interleaving program testing with program writing. In *LISP Conference*, pages 80–99, 1980.
- [McG01] John D. McGregor. Testing a software product line. Technical report, Carnegie Mellon University, 2001.
- [RUCH99] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings ICSM 1999*, pages 179–188, September 1999.

# Towards Testing Response Time of Instances of a web-based Product Line

Dharmalingam Ganesan<sup>1</sup>, Ulrich Maurer<sup>2</sup>, Michael Ochs<sup>1</sup>, Björn Snoek<sup>1</sup>,  
Martin Verlage<sup>2</sup>

<sup>1</sup> Fraunhofer Institute for Experimental Software Engineering  
Sauerwiesen 6

67661 Kaiserslautern, Germany

{ganesan, ochs, snoek}@iese.fhg.de

<sup>2</sup> market maker Software AG

Karl-Marx-Str. 13

67655 Kaiserslautern, Germany

{u.maurer, m.verlage}@market-maker.de

**Abstract.** Instances of a product line share code, but might differ in the non-functional quality attributes (e.g., response time and load). This difference is either because of the requirements on the instances of a product line or being influenced by the architecture decision to achieve high reuse. Hence, testing these non-functional qualities attributes is equally important as their functional counterparts. This paper describes our on-going work aimed to realize an environment for testing the response time and load of a product line, in the domain of stock market. The major components of this environment are a) selective injection of code (using byte code instrumentation technology), b) generation of load and c) support for visualization of response time. We discuss the challenges and the need for such an environment in the context of product lines and then discuss the current results of using the proposed approach on an instance of a product line.

## 1 Introduction

The instances of a product line may differ in the non-functional properties. For example, in case of a portal framework for stock market information, they vary with respect to load and response time. The load is measured in number of accesses to a web server within a given time frame; here, some instances must fulfill hundreds of requests per minute, others are developed for only a few accesses per hour. Response time to a request lies between a required minimum latency of 250 milliseconds to a few seconds. While there has been substantial advancement in methods and tools for functional testing (e.g., [6], [11], [12], [14], [15]) the testing of non-functional quality attributes (e.g., load and response time) is often still ad-hoc [5] [13].

Product instances share code which, in our opinion, has to be evaluated in certain contexts. The reference architecture is not a manifest. It contains definition of generic and flexible structures. For example, in our product line, one may instantiate multiple instances of one component in order to address issues of throughput or availability. Hence, non-functional quality attributes, especially performance, cannot be tested without instantiation of the generic components in the reference architecture. The main idea presented here is, that for testing non-functional aspects of a component, they have to be instantiated given a specific context and measuring non-functional properties of the product instance has to take into account the configured and tailored components. In this paper, we report about an approach developed for and validated in a specific context, namely the testing of performance of components within a web product line for displaying stock market data (see [1] and [2] for details). The system is developed mostly in Java and consists of a number of components that are integrated by a standard framework.

Web-based applications must respond to the user requests within acceptable time limits even under heavy load. A typical limit for an information portal determined by usability studies is six seconds within the system has to give response. This demands an environment which helps developers to test quality attributes systematically. In our case, the major problem is located in the fact that there is no architectural bottleneck; the architecture was designed scalable and fast. Asynchronous communication by message sending is employed to achieve high performance (see [2] for details). Nevertheless, from time to time one observes an unusual long response time to requests. General profiling tools do not fit because they slow down the whole system as they measure each object or class. A more focused approach is needed which concentrates on hotspots identified and does not bias the overall performance, hence allowing for performance tests even in a system running in production. During the initial design of architecture, such an approach was not considered. It was added after the need for non-functional attribute testing was identified.

Determining non-functional behavior of configurable components is not meaningful when just analyzing the generic component itself. Some configuration parameters do impact timing, for example and most obvious, the log level. Also, the instantiation of the reference architecture for a particular product instance does have an impact, too. For example, it does matter, whether a small scale service runs on a single computer, or whether a large-scale product instance runs on a networked set of computers adding communication latency to each request processed. We believe that testing non-functional behavior of a product line has to be performed by testing product instances in specific contexts, which might be at runtime, too.

The performance measurement has to conform to the following principles:

- (1) The approach must not affect the runtime behavior above a limit of a few milliseconds per request.
- (2) The instrumentation of the code must be performed after the build of a version has finished; it is not acceptable to include measurement code in the system source code.
- (3) The subject to measurement might change, because a) new problem areas are identified, or b) more detailed analysis of system parts is necessary.
- (4) The method of measuring performance is allowed to change.
- (5) In addition to pure

timing information context data should be provided for interpreting performance numbers; the actual configuration of the component must be captured.

Early in the project an approach was identified which fulfilled the requirements. The general term for this approach is “byte code injection”: an existing Java archive (i.e., JAR file) is taken and code is added. Existing methods are moved and new methods substitute them. The main advantage is that the external JARs can still use the injected JARs as in the past without any changes. For that we used the system developed in the open-source project JRat [3], which is based on BCEL [4]. We gained the following benefits: (1) The application code and its test code can be clearly separated. (2) Third party applications whose source code is not available can be tested as well. (3) Existing open source solutions like the tool JRat or the framework BCEL could be used.

In principal, byte code instrumentation is similar to aspect-oriented programming. But at the implementation-level, byte code instrumentation offers more flexibility than aspects. On top of JRat we developed a framework which allows performance measurement of asynchronously communicating components. The approach consists of the following elements: (1) Load generation. (2) Enhancements of the code to make requests traceable. (3) Selective code injection.

System load is produced with our prototype called i\*Test. Using i\*Test, the testers apply load to the system and then measure response time automatically using our proposed environment. For each product instance a number of scenarios is defined which represent patterns of usage. A scenario is a test plan which consists of several threads to drive the tests, a list of requests (“click path”) as well as methods to check the response from a functional perspective. Variability is not addressed in the test plans.

Some of the challenges in testing the response time of instances of product lines are as follows:

1. How to collect response time of multi-threaded programs under different loads?
2. How to locate the bottleneck quickly?
3. How to monitor changes to a single product instance in order to check whether the change does not decrease performance of this particular product instance and all other product instances?
4. How to establish the relationship between product line architecture patterns and its influence on the response time and load?

In this work, we discuss the above challenges by employing an existing approach of byte code injection. Consequently, this paper illustrates the benefits of byte code injection as well as discusses a validation of the approach in a larger setting.

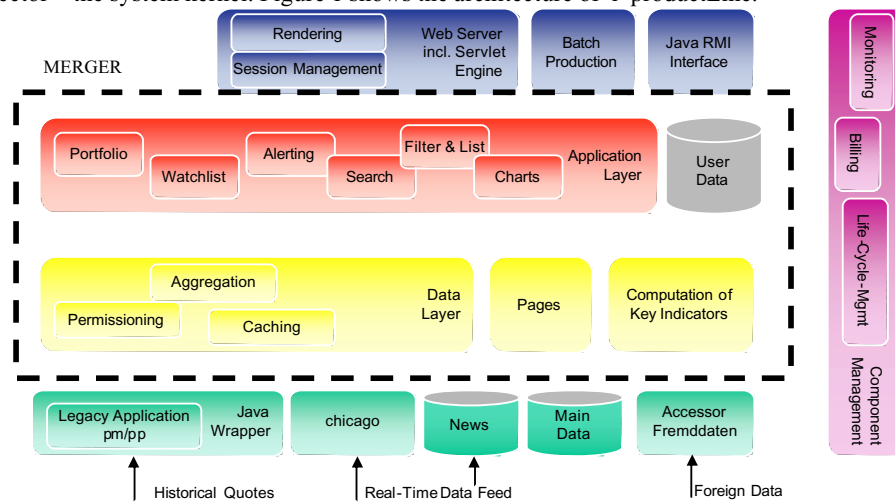
The remaining paper is structured as follows: section 2 describes the context of the work. In section 3 our process model for collecting response time is described. Afterwards, the load testing is briefly explained in section 4. Finally, in section 5, the current results and next steps are outlined.

## 2. Context of the Work

### market maker and i\*ProductLine

market maker Software AG [7] is a small software and service company based in Kaiserslautern, Germany. Starting from the early 1990s, it has developed software for the management and visualization of stock market data. In 1999, market maker decided to establish a new product family, which uses Internet technology to deliver services to their customers. The first product instances were delivered in 2000.

i\*ProductLine consolidates a number of different sources of financial data content and delivers this data in various output formats to diverse interfaces, including Web, Web Services and specialized interfaces to client systems. This leads straight forward to a three layered architecture: The data sources layer, the front end and – as a connector – the system kernel. Figure 1 shows the architecture of i\*productLine.



**Figure 1: Logical View of i\*productLine**

The system kernel is called MERGER (see dashed line in Figure 1). It consists of several components. They can be instantiated multiple times to achieve redundancy and scalability. Communication between the kernel components follows a standardized pattern and uses request flow mechanisms provided by the kernel. The kernel receives requests from the Front Ends. It sends parallel subrequests to the involved data sources and returns the compound response. It makes sure that the user permissions are obeyed. For example only those markets are displayed that are defined in the client contract.

We have chosen the MERGER kernel as the location for testing the response time of `i*ProductLine`, because all requests initiated by a user from the web-interface must enter and leave this kernel.

### **Performance Issues**

Servlets are expected to respond to a request within a range of a few hundred milliseconds, excluding overhead caused by slow lines. Consequently, performance is measured first at the interface to the merger kernel. During black-box tests of the production system one could observe occasionally that some requests had an unusual long response time (e.g., several seconds). The standard system parameters like CPU load, paging activity, or network interface load did not show reasons for that behavior. A more detailed look into the system and the performance of single methods was needed to identify the performance bottleneck(s).

At best, performance tests running the scenarios would run on the production system. Performing the tests in a laboratory setting would have the following drawbacks: (1) The system configuration comprises a number of distributed components running on expensive hardware. (2) A laboratory setting seldom exactly rebuilds the real environment. Line speed, network capacity, exact order of requests, or concurrent use of the CPU are hard to control.

Difficulty to performance analysis is added by the fact, that the product instances are configured and call flow is dynamically arranged by a middleware which bypasses inactive copies of identical components (in the case of component failure) and distributes load to equal services. In our opinion, it is hard to put measurement routines explicitly into the code because due to the configuration of the component one cannot be sure whether the code is executed in the sense one wants it; moreover configuration may disable essential parts of the measurement activities.

### **Terminology**

In the remainder of the paper we will use the following terms for explaining our approach:

*Product line* implements a generic set of product instances.. It consists of a reference architecture and core assets (or components).

*Product instance* is a specific service, tailored and configured for a customer.

*Scenario* is a set of usage patterns which describe the way how users interact with a product instance. A usage pattern is described by request/response pairs which are sent between user and web server.

*Test plan* is an implementation of a scenario.

### 3. Process Model

Traditional performance analysis steps include: (1) Instrumentation: The object code of the product instance is instrumented and during execution the instrumentation hooks record the performance data. (2) Data Extraction: After performance data is recorded from one or more scenario executions, data relevant to specific code elements (classes, methods) is extracted. (3) Data Analysis: Data is analyzed to extract bottlenecks. (4) Optimization: Areas of poor performance are identified and plans are made for improvements.

Here, we focus on the problem of obtaining measurement data for the purpose of locating bottlenecks. Figure 2 shows the process model for performance analysis.

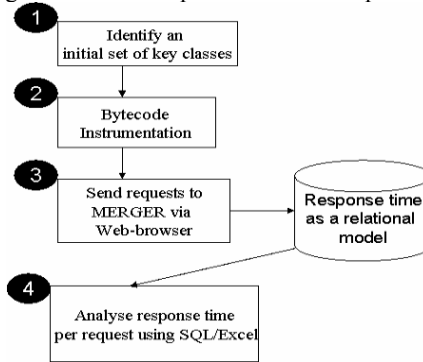


Figure 2: Different steps in performance analysis

#### Step 1: Key class identification

Since our approach uses byte code injection for collecting response time, it is important that we reduce the overhead associated with additional byte code. Therefore, a few classes called key classes are instrumented instead of the complete system. These key classes are identified by the developers of the system. In fact, due to product line development and build-in abstractions the number of key classes is fairly low. In the following example, it is just one key class that is target to instrumentation.

In the case of MERGER, an expert selected a Java class called TemplateComponent responsible for handling various types of requests from the users. The class TemplateComponent contains two methods namely the sendRequest and the getAnswer. sendRequest is called whenever a new request is initiated by the user. In MERGER, every request is identified by request\_id, which is a unique identifier. The sendRequest method delegates a request to other classes of MERGER based on its arguments. When a request is processed, the getAnswer method will be called. Because of the request\_id within the arguments of getAnswer we know the connection to the corresponding sendRequest method.

```

public class TemplateComponent {
    public void sendRequest(Arg1, ..., ArgN) {
        /* Analyse arguments and delegate requests to
other classes */
    }
    public Answer getAnswer(Arg1, ..., ArgN) {
        /* When the answer is ready for a request, this
method will be called */
    }
}

```

## Step 2: Byte code Instrumentation

In this step, the key classes from step 1 are instrumented using JRat. For every method, JRat introduces a wrapper method that first invokes the JRat components and then calls the “real” method. This way, the rest of the system is not affected when a set of particular classes is instrumented. Basically, JRat records the starting and finishing time of a method. Every method has an associated handler object that implements the *MethodHandler* interface:

```

public interface MethodHandler {
    void onMethodStart (Object obj, Object[] params);
    void onMethodFinish (Object obj, Object[]
        params, Object ret, long dur, boolean suc);
    void onMethodError (Object obj, Object[] params,
        Throwable thro);
}

```

In the earlier phase of this work, we found out that the existing logging mechanism within JRat is not sufficient for logging arguments of methods. To support this functionality we implemented a new component which implements the JRat interface called *MethodHandler* and integrated it. This enhancement neither affects the existing architecture nor the API of other components. Unfortunately, we found out that the *params* argument within the above methods is never set to the expected value. To fix this, we enhanced the instrumentation component of JRat to log the arguments of the methods. To introduce this new feature into JRat, the following tasks were performed: (1) Locate the instrumentation component within the JRat architecture. (2) Identify the important interfaces of JRat.

Using our new component, we logged the arguments of the methods *sendRequest* and *getAnswer* (see *TemplateComponent* class) together with the current system time in milliseconds to a database. Based on the *request\_id*, it is possible to find the two method calls of *sendRequest* and *getAnswer* that belong to the particular request.

### Step 3: Send Requests to MERGER

When the code runs productive, the requests are web traffic. Additional requests are generated by running test plans which are also used during system integration. These test plans implement scenarios for the specific product instance and typically put load on the system parts configured or implemented for that particular product instance. In the case the tester already has an idea about the problem area, he even may put additional requests by own scripts or by hand. The System expert has knowledge about the scenarios that invoke the instrumented classes (see Step 2), and hence, the response time was logged into the database as shown in Figure 3. The first column denotes the name of the service that the user has requested with the corresponding request id (column number five), whereas the fourth column tells us the class that has been used to provide the service.

Service	Method	StartingTime in Millis	Class that provides the Service	ID	Duration [millisec]
demon_chart	sendRequest	1101812293865	WpidRequest	113002	
demon_chart	answer	1101812294166	WpidRequest	113002	301
demon_chart	sendRequest	1101812294176	LivedatenAIRequest	114002	
demon_chart	answer	1101812294807	LivedatenAIRequest	114002	631
demon_chart	sendRequest	1101812294827	MMResultRequest	115002	
demon_chart	answer	1101812295177	MMResultRequest	115002	350
demon_chart	sendRequest	1101812296089	WpidRequest	118002	
demon_chart	answer	1101812296109	WpidRequest	118002	20

Figure 3: Collected response time data

### Step 4: Analyzing the response time

Figure 4 shows the average of all requests out of one service, that calls the same class will be calculated and visualized as one bar within a chart. If, for example, a class xy-Class is used by a service xyService several times then the response time of each usage will be used to calculate the average. The average value is shown as a bar of the chart. This provides the expert the needed Information to find bottlenecks or other performance problems. The red line within the chart shows the average duration of all requests that the system has answered within the testing-period.

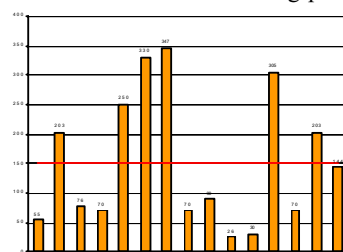


Figure 4: Average response time per class and service for each request

In the beginning of our work, the response time for various requests was collected with only one user at a time. The reason was, to identify those requests whose response time is not satisfactory without any load. Before we started tracing the portion of source code for potential bottlenecks, we also had to identify the requests whose

response time is affected by load. The next section explains how we applied load to MERGER.

#### 4. Load Testing

In order to stress the production system, load testing was performed to detect unwanted behavior of the i\*ProductLine using our prototype i\*Test. The demands on the load testing utility are as follows: (1) The utility should focus on the Web Front End. (2) The specification of a load test should be very simple. (3) Single tests should be performed multiple times in sequence as well as in parallel. (4) The parallel execution of tests must be parameterizable, so that parallel tests can be performed (e.g., usage of different user logins). The HTML response to a test request should be evaluated to make sure that the system returns the requested data. If the system does not return the requested data but an error message, the test should be marked as failed.

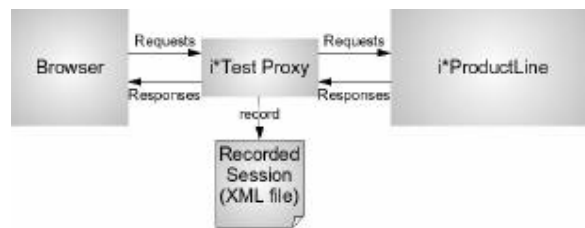


Figure 5: Architecture of load testing tool i\*Test

i\*Test meets the given demands. It is based on a number of standard technologies: Apache Ant [8], Jakarta JMeter [9], usage of XML as data and configuration format and usage of XPath [10] for result testing. i\*Test provides a proxy that can be configured in the browser (see Figure 5). The tester uses the i\*ProductLine Web Front End for that specific product instance in the normal way and navigates through the pages that should be called during the load test. The proxy records each request of the session and stores them in an XML file.

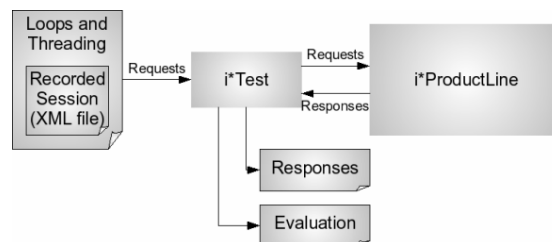


Figure 6: Modify and replay the recorded session

The recorded XML file can directly be used as input for i\*Test to replay the complete test (see Figure 6). All responses are stored in a repository and finally an evaluation summary is generated, containing textual evaluations of the response times as well as graphical evaluations. A collection of XML files makes up a test plan which implements a scenario for a specific product instance.

## 5. Current Results

The above described tool set was validated in the context of i\*ProductLine. In order to identify performance issues in the production system, code was injected into JAR files and measurement data was collected. The results so far are:

1. The injected code does not affect system behavior above a critical limit. Due to the small number of classes and methods injected per test run there was no visible performance degradation observed. JRat perfectly integrates the code and keeps overhead to a minimum.
2. Scenarios encompass usages of a product instance. The degree of variability which can be expressed in the scenarios does not comply with the one of the components to be tested, i.e. one ends up with more scenarios than product instances.
3. It is unlikely that test plans get reused. Even in the case product instances are build similarly and reuse of code is high each product instance may focus on very differing aspects of the system. Scenarios stress these aspects, and may result in very different test plans. Currently, for a tester it looks easier to write a new test plan instead of analyzing commonalities and variabilities with respect to an existing one.
4. Code injection is a step which may take place after the build process of a JAR has finished. This is very important since the regular production process does not change and developers do not have to take into account that the performance measurement takes place afterwards. A JAR file could be copied from the production environment, instrumented, placed back, and taken effective by restarting the system. The source code of the original system does not change. In our understanding, this is the very meaning of aspect-oriented programming.
5. The JRat approach had to be changed in order to fulfill the requirements of our project. The JRat development team was responsive and helpful. The change proposals are accepted for inclusion in a later version of JRat.

In summary, we have established an environment for testing response time and load of a product line with in market maker. Currently, this approach is customized for our product line, in principal this idea can be customized for other domains as well. The obvious challenge lies in the instrumentation; for resource-constraint systems the overhead associated with instrumentation has to be kept as low as possible.

## Next steps

With the encouraging results we got by applying the proposed approach in our test environment, the immediate next steps involve collecting response time under different load for many instances of i\*productline. Using these data, building models of non-functional behavior for our product lines will be one direction for us to proceed. Another interesting aspect to be addressed in future is the influence of architecture decision with respect to response time and load of instances of a product line. We believe such an automatic support for testing response time and load is necessary for evolving the product line architecture towards better performance.

## Acknowledgement

This work was performed within the project CBTesten (BMBF 01ISC29), which is funded by the Federal Ministry of Education and Research, Germany. Special thanks go to T. Schmitt, formerly ICTeam, to point to the JRat approach. We are overjoyed by the responses given by the JRat developers whenever there are some issues or feature requests or bugs in their tool. Last, but not least, we thank the anonymous reviewers for their feedback.

## References

1. J.-F. Girard, M. Verlage, and D. Ganesan: Monitoring the Evolution of an OO System with Metrics: an Experience from the Stock Market Software Domain. Proc. of the 20th International Conference on Software Maintenance, Chicago, September 11 14th 2004
2. M. Verlage and T. Kiesgen: Five Years of Product Line Engineering in a Small Company, Proceedings of the 27th International Conference on Software Engineering ICSE05, St. Louis, USA, May 2005
3. JRat Homepage, <http://jrat.sourceforge.net>, February 2005
4. BCEL Homepage, <http://jakarta.apache.org/bcel/>, February 2005
5. CBTesten Homepage, <http://www.cbtesten.org>, February 2005
6. JUnit Homepage, <http://www.junit.org>, February 2005
7. market maker Software AG Homepage, <http://www.market-maker.de>, February 2005
8. Apache ANT Homepage, <http://ant.apache.org/>, February 2005
9. Jakarta JMeter Homepage, <http://jakarta.apache.org/jmeter/>, February 2005
10. XPath Homepage, <http://www.w3.org/TR/xpath>, February 2005
11. H-G, Gross: Testing and the UML – A Perfect Fit, IESE Report 110/03E, October 2003
12. B. Beizer: Software Testing Techniques. Thomson Computer Press, 1990
13. H-G. Groß, C. Peper, and M. Ochs, A. Kalenborn und M. Verlage: Vorgehensweise Planung & Generierung Testartefakte, CBTesten Konsortium, Kaiserslautern, 2004

14. J.D. McGregor: Parallel Architecture for Component Testing,  
<http://www.cs.clemson.edu/~johnmc>
15. J.D. McGregor: Testing a Software Product Line, Technical Report, CMU/SEI-2001-TR-022

# Product Line Testing and Product Line Development — Variations on a Common Theme

Peter Knauber  
Mannheim University of Applied Sciences  
Windeckstraße 110, 68163 Mannheim, Germany  
p.knauber@fh-mannheim.de

William Hetrick  
Engenio Information Technologies, 3718 N. Rock Road  
Wichita, KS 67226 USA  
bill.hetrick@engenio.com

**Abstract.** The production capability of a software product line development organization can overwhelm traditional test practices. Test organizations have to employ product line engineering principles to yield the same improved throughput efficiency and reduced costs benefits as when these principles are applied to software development. This position paper sketches an approach for strategic development and reuse of test assets at different levels together with their management as part of a product line infrastructure.

## 1 Introduction

Software validation is widely recognized as a critical function of the software development cycle. An organization can transition its development group<sup>1</sup> to product line engineering (cf. Figure 2) to improve the product development capability. However, product line engineering is capable of dramatically increasing the product development efficiencies, and thus can outpace if not overwhelm traditional test practices.

Traditional test practices often begin with unit tests owned by the development organization. These tests emphasize functional verification, exception verification, and code coverage of a subset of the software assembly. When a feature or product is sufficiently developed and validated with unit-level testing, the software is passed to a dedicated test organization to validate as a software solution. Mature test organizations will have substantial investments in an infrastructure of databases and automation tools generally based on one-at-a-time product validation requirements. Product validation test suites are assemblies of feature test cases. The test environment infrastructure commonly facilitates re-use at the feature validation level based on the assumption that future products may or may not include a feature.

---

1. For the purpose of this paper we distinct between *development* and *validation*: *development* comprises all activities from requirements analysis to implementation. *Validation* starts after (at least) rudimentary unit tests have been performed successfully and is carried out by an explicit quality assurance group.



Test costs are comprised of test case and test infrastructure development, test execution, and test execution equipment. A test organization could attempt to increase its capacity by increasing staff and adding equipment, but this strategy can be very expensive, insufficient, and quite likely a physical impossibility. Business needs and schedule pressures can coerce the organization to reduce test coverage, which can introduce substantial risks in product maintenance costs and customer satisfaction. One possible way out is to employ product line engineering principles to yield the same improved throughput efficiency and reduced costs benefits as when these principles are applied to software development.

### 3 A Product Line Testing Infrastructure

In order to let testing benefit from the characteristics of product line engineering in the same manner as development, product line principles should be applied to testing practices as well as to development practices (see Figure 2):

- During domain engineering, tests for product line assets should be developed simultaneously to the product assets themselves, feeding a *test infrastructure*. This infrastructure should comprise test assets for component tests<sup>1</sup>, feature tests, and product (or system) tests.
- During application engineering, assets from this test infrastructure should be reused to speed up testing and thus overall product development. Like with development assets, it should be possible to customize test assets at certain variability points [4] using a decision model as described in [6]. The expectation here is that test assets are customized by using *the same resolutions in the same decision model* that are used for customization of the other product line assets.

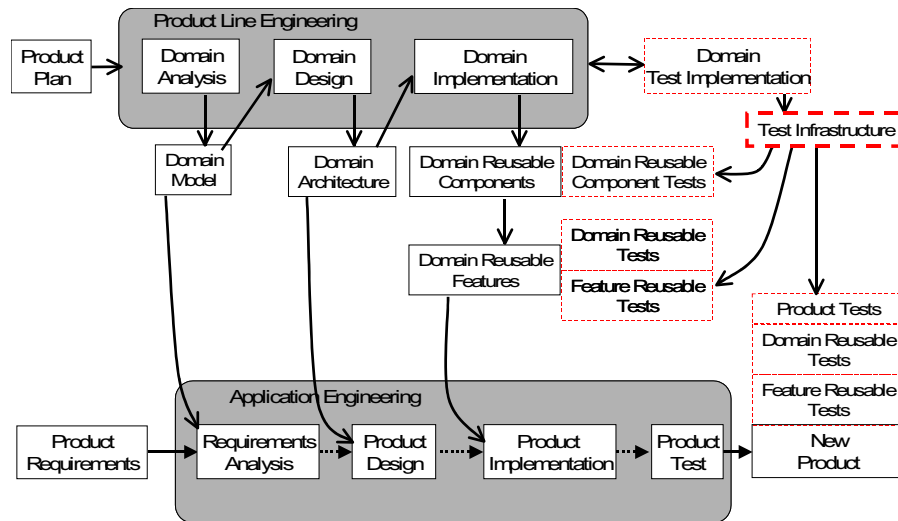


Figure 2: Test Implementation in Product Line Engineering

1. We prefer to talk about *components* instead of *units* because the term *reusable components* is used frequently in product line engineering (see Figure 2).

Product line testing then can be done at three levels: at component level, at feature level, and at system level. These are described in more detail in the following sub sections.

### **3.1 Product Line Testing at Component Level**

At component level, traditional and well-established test development practices can be applied to develop component tests, nevertheless, *test-driven development* at component level offers a specific advantage. In test-driven development, test cases are developed first, that is, calls of a component's routines are written before coding of the component has started. Among other advantages, this gives early feedback about the usability of the component's interfaces. While this is already an advantage in single system development, interface usability is even more crucial for product line components because these are reused often in several systems and by several users.

Whatever test approach is chosen, in a product line context at component level *tests should be automated* because these tests are likely to be reused very often and as part of potentially many products. On the other hand, components that are reused over a long time very likely need to be revised for (mostly adaptive or preventive) maintenance purposes. Automated tests cannot guarantee the correctness of a component but provide a very useful basis for assuring its reliably high quality.

To validate a generic or parameterizable component, structural tests are essential to make sure that all *product-specific paths* through the component have been tested. Again, automated tests for the common (that is, non-variant) parts of the component are very useful in that they provide basic coverage. On top of these common tests, variant tests have to be developed. If these are automated then the same variability mechanisms should be used as in the component implementation (cf. [5]).

### **3.2 Product Line Testing at Feature Level**

As for single system development, product integration with feature-level validation should start only after the components used have passed their component tests. Using features as integration units towards the final product does make sense for product line development because the members of a product line often differ in availability or comprehensiveness of their features. This implies that tests at this level are customized using the decision model in exactly the same way as the feature implementation itself. Whether the tests are automated or not, they have to be treated as core product line assets and managed consistently with the respective feature code.

Any defects discovered at this level must be fed back into the component tests because errors have to be corrected at component level. Depending on the kind of error detected at feature level it should be carefully checked if there is a general problem with test generation at component level and, if so, the test generation procedure should be improved and the tests developed so far should be completed.

### **3.3 Product Line Testing at Product Level**

Validation at product level (*system tests*) requires that the validation of components and of the partially integrated features was successful. Again, tests at this level have to be treated as product line assets and their product-specifics have to be managed using the decision model.

As for errors detected at feature-level, any defects discovered at this level must be fed back into the previous test levels, i.e., component-level tests and/or feature-level tests. Errors have to be corrected at component level, test practices at component test-level and feature test-level may have to be revised in order to make sure that this kinds of errors can be avoided or at least detected earlier in future product developments.

### **3.4 About Automated Tests**

To some extent we have discussed the topic of automated tests in the previous sections but would like to elaborate some more on this issue. For single system development, automated tests are mostly used if (parts of) the system to be tested is frequently revised, adapted to another environment aso., or in the context of test-driven development [1]. These situations have in common that the tests are run frequently / very often.

For software product lines the situation is similar: even if the core components are not revised as often as it may be necessary for components of a single system, they are developed as being generic to allow that they can be parameterized (slightly) different for each product they are used in. Even if, while developing one product, the tests are not run as often as in test-driven development they are run again and again for each product derived from the same infrastructure.

Automated tests for common component parts can be executed very efficiently during validation of each product line member, thus saving lots of effort and time. For testing the variant parts the situation is similar as for tests at feature or product level: for some tests the investment in their development will pay while the effort to develop some other tests may exceed the potential savings from their later application. In any case, automated tests for variant components, features, and systems need to be connected to a common decision model for the product line in order to make sure that tests are used consistently with the code they are testing.

## **4 Summary**

The amount of products produced using product line engineering may exceed the capacity of traditional testing resources: the amount of variability contained in product line assets cannot be dealt with using traditional testing methods. Since exhaustive testing is expensive, a testing strategy is needed to do product line testing effectively and efficiently. Strategic development and reuse of test assets allows testing to keep pace with product line development. To achieve this, test assets have to be developed and managed as core assets of their respective product line. Therefore product line product development and product line test development follow exactly the same structure: reusable and customizable tests are created during domain engineering and reused during application engineering. This paper describes how to test product lines at three levels: at component level, at feature level, and at system level.

## **5 Acknowledgements**

Many of the ideas presented in this paper were discussed in a break-out group of SPLiT in 2004. We thank our colleagues Chris Condron and Yuan Zhan for sharing their ideas with us and for a fruitful discussion.

## References

- [1] K. Beck: *Test-Driven Development By Example*, Addison-Wesley, 2002
- [2] BigLever Software Case Study: Engenio, Report # 2005-06-14-1, 2005, available at <http://www.biglever.com/papers/EngenioCaseStudy.pdf> (July 2005)
- [3] B. Geppert, Ch. Krueger, J. J. Li (Eds.): *Proceedings of SPLiT 2004: International Workshop on Software Product Line Testing*, Technical Report: ALR-2004-031, Boston, Massachusetts, USA, 2004
- [4] G. van Gorp, J. Bosch, M. Svahnberg: On the Notion of Variability in Software Product Lines. In *Proceedings of Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2001
- [5] P. Knauber, J. Schneider: Tracing Variability from Implementation to Test Using Aspect-Oriented Programming, in: [3]
- [6] K. Schmid, U. Becker-Kornstaedt, P. Knauber, and F. Bernauer: *Introducing a software modeling concept in a medium-sized company*, in: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, 2000

# Welcome to

**SPLiT**  
**2005**

Workshop on  
Software Product Line Testing

Rennes, France, September 26, 2005

**Chairs:**

Birgit Geppert, Avaya Labs

Charlie Krueger, BigLever Software

Tim Trew, Philips Research

## SPLiT — Why we do it (again 😊)

### Goals

- Exchange forum for practice and research
- Meeting forum for testing and product-line engineering (PLE)
- We want to understand better:
  - PLE specific challenges in testing
  - Industrial needs
  - Technology gap
- Work on building a community

### To Do

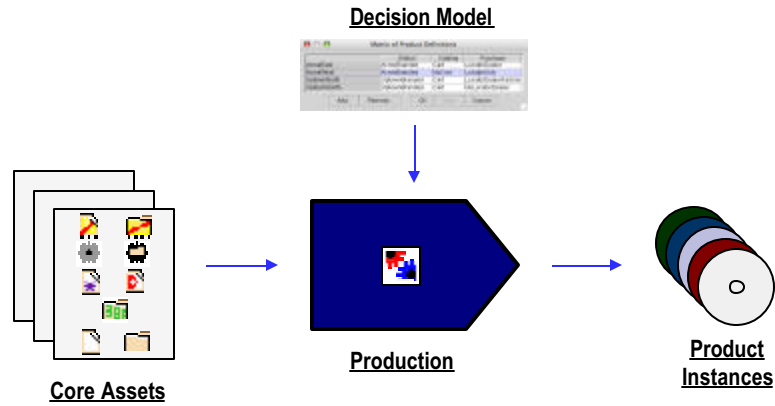
- Current contributions
- Exchange ideas and produce new ones
- Research agenda

**SPLiT**  
**2005**

Rennes, September 26, 2005

2

# Product Line Engineering

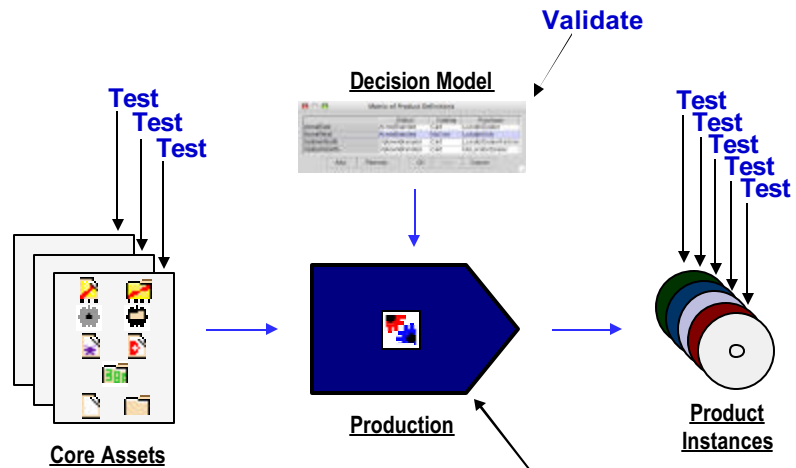


**SPLIT**  
2005

Rennes, September 26, 2005

3

# Product Line Engineering & Testing



**SPLIT**  
2005

Rennes, September 26, 2005

4

## Lots and Lots of Open Issues ...


- How can we keep pace with development productivity gains?
  - How to leverage core asset testing to minimize product instance testing?
  - How can we leverage the commonality among the product instances to minimize redundant testing?
- How can we manage the complexity of the test space?
  - Additional complexity due to (a) variation points and (b) large number of products to test
  - How to deal with the combinatoric explosion?
- Are there PLE techniques that can provide similar efficiency gains for testing as is possible for development?
- Can we leverage our established testing tools and procedures?
- Techniques for strategic reduction in test time, test cost, and test flaws
- Definition and measurement of test coverage and test effectiveness in the context of software product lines
- ...



Rennes, September 26, 2005

5

## Some Results from Last Year


- **Managing the complexity of the test space:** Constrain its combinatorics by being reactive and conservative on the scope of variability in the core assets and decision model. 
  - » BUT ....
- Not one testing technique is sufficient, we need a combination of them. Optimizing test effort means choosing the appropriate methodology and technology for unit, integration, system testing in PLE.
  - » BUT ....
- Techniques and methods for development and testing should match. Co-design of software and test infrastructure/cases very important.
  - » BUT ....



Rennes, September 26, 2005

6

## Some Results from Last Year, cont.


- **Design for Testability** is an important quality attribute and should be among the top-priority abilities. 
  - » BUT ....
- Inspection – on design as well as code level - is a powerful tool for reaching PL quality.
  - » BUT ....
- PL should not be implemented until we get to a point where we have test design or plan.
  - » BUT ....

**SPLIT**  
2005

Rennes, September 26, 2005

7

## Agenda - Morning

- **09:00 - 09:30 Introduction**
- **09:30 - 10:50 Paper presentations**
  - *Executing Reusable System Tests for the Applications Derived from Software Product Lines* (E. Olimpiew, H. Gomma)
  - *Composing Unit Tests* (M. Gälli, O. Greevy, O. Nierstrasz)
  - *Towards Testing Response Time of Instances of a web-based Product Line* (D. Ganesan, U. Maurer, M. Ochs, B. Snoek, M. Verlage)
  - *Product Line Testing and Product Line Development — Variations on a Common Theme* (P. Knauber, W. Hetrick)
- **Refreshment break (10:50–11:10)**
- **11:10 - 12:30 Discussion round 1**
  - Topic: "Managing the complexity of your test space: challenges, ideas, solutions." 
- **Lunch break (12:30 – 14:30)**

**SPLIT**  
2005

Rennes, September 26, 2005

8

## Agenda - Afternoon

- Lunch break (12:30 – 02:30)
- 02:30 - 03:15 **Invited Talk**
  - **John D. McGregor**: *Reasoning about the Testability of Product Line Components* 
- 03:15 - 04:30 **Discussion round 2**
  - Break-out group(s) – topic(s) determined by audience
  - Refreshment Break included ☺
- 04:30 – 06:00 **Results/Group discussion and Wrap-up**

## One last thing ...

- Don't forget the **Sticky Notes**: please write down ideas/topics/open issues you think are worth discussing in the afternoon!

And now ...

Let's start working!



**SPLIT**  
2005

Rennes, September 26, 2005

11

## Paper Presentations

- **09:30 - 09:50**  
*Executing Reusable System Tests for the Applications Derived from Software Product Lines*  
(E. Olimpiew, H. Gooma)
- **09:50 - 10:10**  
*Composing Unit Tests*  
(M. Gälli, O. Greevy, O. Nierstrasz)
- **10:10 - 10:30**  
*Towards Testing Response Time of Instances of a web-based Product Line*  
(D. Ganesan, U. Maurer, M. Ochs, B. Snoek, M. Verlage)
- **10:30 - 10:50**  
*Product Line Testing and Product Line Development —Variations on a Common Theme*  
(P. Knauber, W. Hetrick)

**SPLIT**  
2005

Rennes, September 26, 2005

12

# Composing Unit Tests

Markus Gälli

Orla Greevy

Oscar Nierstrasz

Software Composition Group Bern

## Roadmap

- Example product line: Two different types of Bank Accounts
- What is the problem with tests and product lines?
- Our Approach: Analysis with PO and Refactoring
- Conclusion

## A simple Product Family: Account with or without credit

- Common Assets:
  - Create Account, Deposit money...
- Individual Assets:
  - Withdraw Money....
- Common Tests
  - testCreateAccount, testDepositMoney
- Individual Tests
  - testWithdrawMoney

## Problems of not being able to compose tests

To withdraw one needs to deposit.

So deposit is once tested and once used to create scenario.

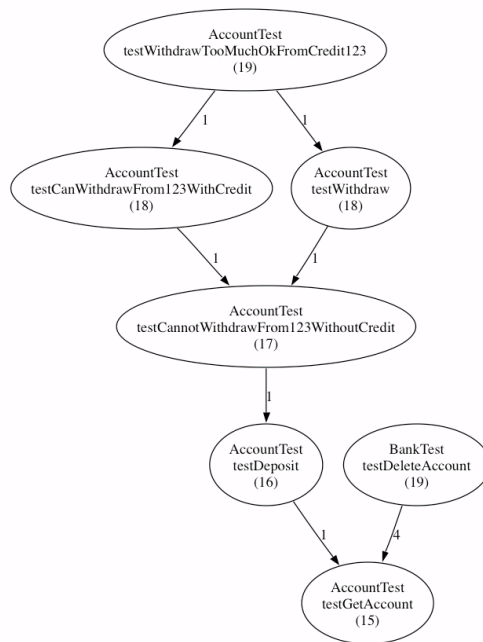
--> Difficult scenario setup

Mocks can be complex too.

--> Long testing times.

Automatic test reduction is not safe.

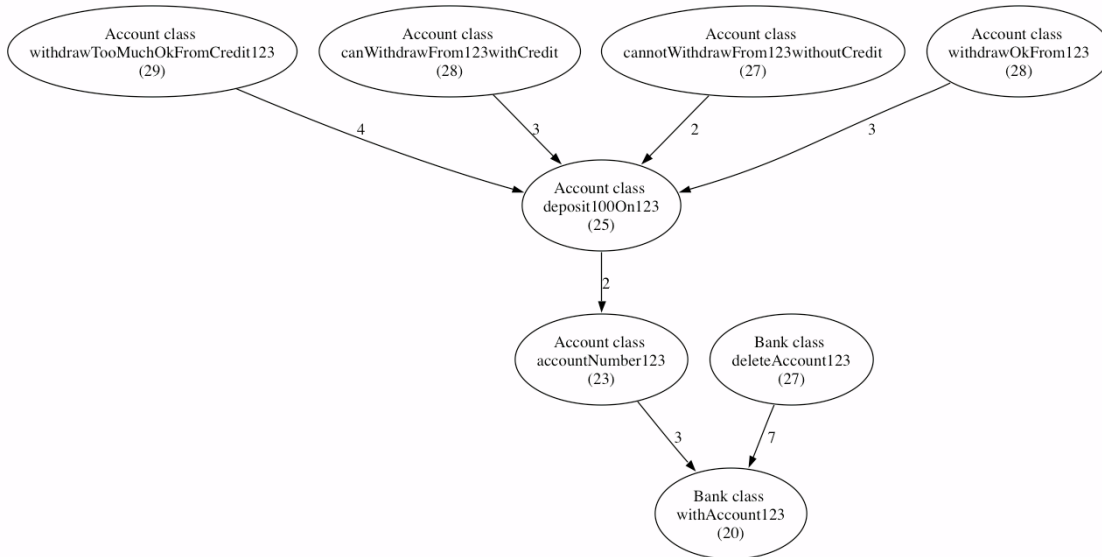
# Our Approach (I): Analysis with PO-Set of covered signatures



## Our Approach (II): Refactoring

- Publish Results!  
Each test can return the value of the changed object - move into factories
  - in our case the accounts
- Move concrete assertions into abstract pre-/postconditions
  - “withdraw: anAmount” asserts
  - “canWithdraw: anAmount” as precondition

# Our refactored Bank Example: Only have to run the root-tests



## Conclusion

- We can detect composable test cases and reuse them in depending assets by refactoring them and thus:
  - reduce the code significantly
  - reduce testing time
  - detect errors more on the spot
- Demo / Questions...

# Towards Testing Response Time of Instances of a web-based Product Line



Martin Verlage

Ulrich Maurer

Fraunhofer  
r



Institut  
Experimentelles  
Software Engineering

Dharmalingam Ganesan

Micheal Ochs

Bjoern Snoek



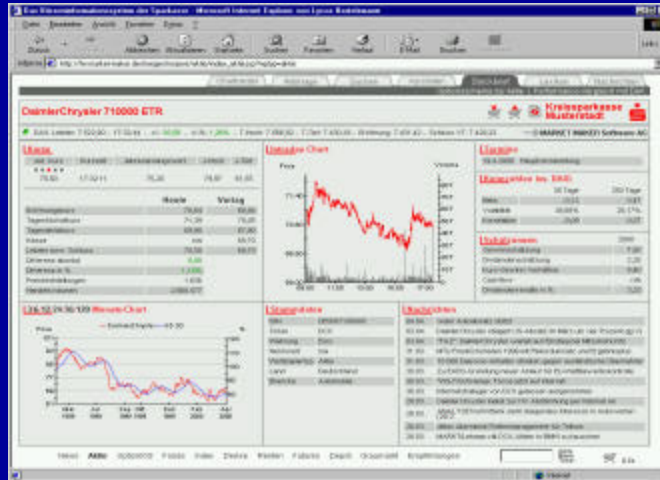
## Outline

- Testing Problem
- Introduction to i\*ProductLine
- Testing Response Time of Requests
- Conclusion and Discussion

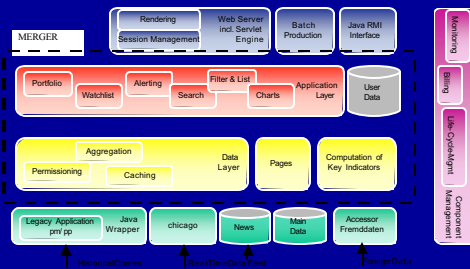
- Instances of product line share code, but occasionally **performance problems (response time)** are encountered in the production environment
- It is not clear whether this problem is due to product-line architecture design where code reuse is the major focus
- Customers do not care about product line – they just want services that satisfies their need –
- Hence, an environment for testing response time is a must

### Challenges

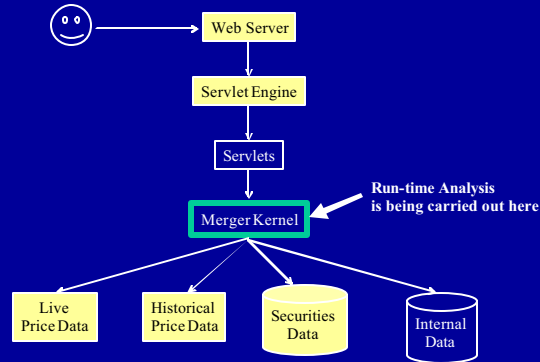
- How to collect response time of the instances of a product line under different load ?
- How to locate the bottleneck quickly (Debugging issue) ?
- How to monitor changes to a single instance in order to check whether change does not decrease performance of other instances of product line ?
- How to establish the relationship between product line architecture patterns and its influence on the response time and load?



Copyright © Fraunhofer IESE 2005



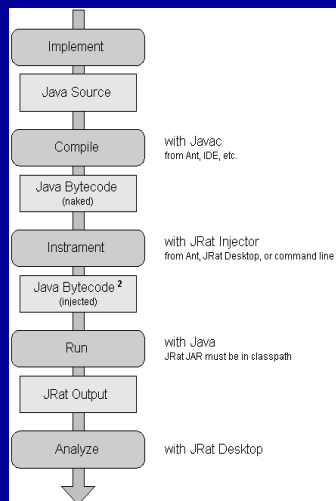
Copyright © Fraunhofer IESE 2005



Users access web pages over a web server. The web server provides either static html pages or pages generated by servlets. Servlets are run in a servlet engine.

Servlets access Merger services to get data and return this data in html pages.

The Merger services get their data from various data sources. The sources can be either internal to Merger (that means they are also updated by Merger) or external.



JRat Process Flow

## Different Steps

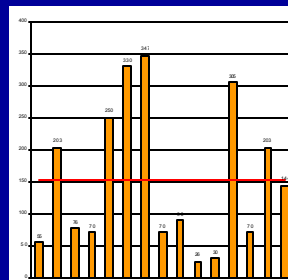
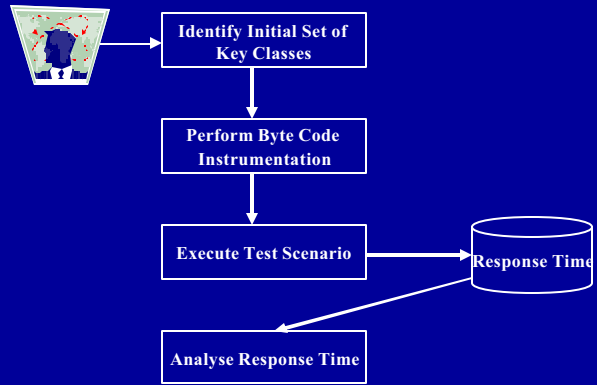


Figure 8. Average response time per request.

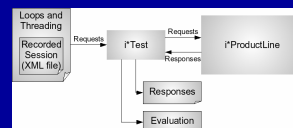
### Load Generation

- Response Time should also be analysed under different load
- Load Generation infrastructure is needed for i\*ProductLine with the following requirements:
  - The utility should focus on the Web Front End
  - The specification of a load test should be very simple
  - Single tests should be performed multiple times in sequence as well as in parallel
  - The parallel execution of tests must be parameterizable, so that parallel tests can be performed (e.g., usage of different user logins)
- A prototype i\*Test using the open-source tools (Apache Ant, JMeter, XPath) was developed

### Load Generation Environment



Architecture of load testing tool i\*Test



Modify and replay the recorded session

- Product line instances might differ in their non-functional quality attributes
  - An environment for testing them is needed.
- A practical approach to measure the response time of a request is presented
- Expert role is necessary to identify the key classes for instrumentation, and also to reason about the response time
- In summary, an environment for testing non-functional quality attributes (Response time and load) of instances of a product line is created at Market Maker.
- Open-source tools can be used in industrial projects through experimentation

### Reviewer's comments

- It is not quite clear how this approach can be generalized for other kinds of product lines
- A software architecture determines the qualities (non-functional properties) of a software. If product instances are derived from a common software product line architecture, how can they expose different qualities? Or why should they? Or, if instances do not share the same architecture, why are they members of a product line ?
- It would be interesting to learn whether this approach was considered in the initial design of the i\*ProductLine or was it added after the need for non-functional attributes testing was identified. If not considered early, what might be done differently in future designs given this non-functional testing requirement ?
- Would it help to develop a little language for scenario specification that expresses the variabilities among the product instances and use it for test script generation ?



**THANK YOU**

## PL Testing and PL Development –

### Variations on a Common Theme

William Hetrick

Peter Knauber

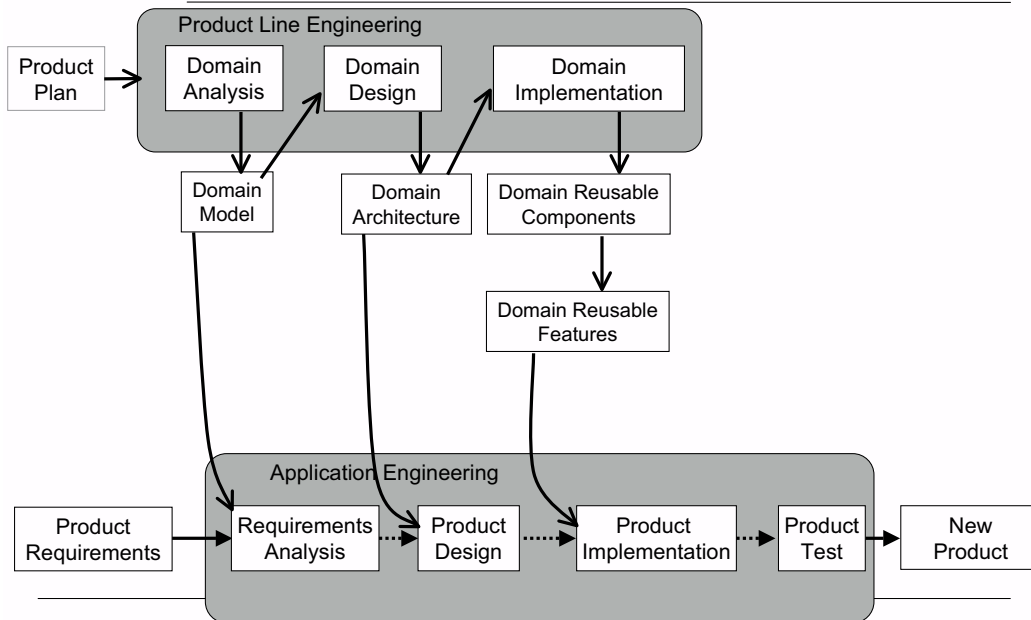


### Challenges

---

- Product line development practices may outpace traditional test practices
- Consequences: must...
  - Increase product validation throughput:  
total product throughput := effort available / effort necessary per product
  - Limit product validation costs:  
total costs := costs per product \* number of products
- Example from Engenio Information Technologies
  - Implementation effort: change **100 LoC**
  - QA effort: **3 PM**

## PLE Process with explicit Testing Step

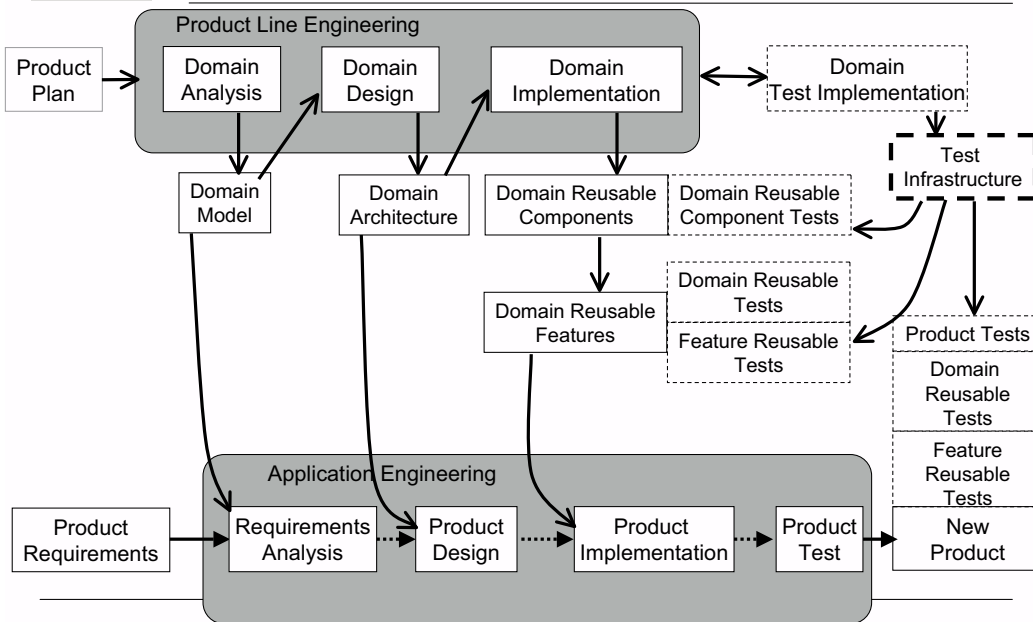


SPLIT, with SPLC9  
September 26, 2005, Rennes, France

Slide 2

Peter Knauber, William Hetrick

## Test Implementation in PLE



SPLIT, with SPLC9  
September 26, 2005, Rennes, France

Slide 3

Peter Knauber, William Hetrick



- At component level
  - Test-driven development increases interface reusability, fostering reuse
  - Automated tests increases test productivity (reuse in multiple product line members)
  - Structural tests help covering product-specific paths
- At feature level
  - Integration tests at feature-level correspond to product line variants
  - Product-specific test assets are derived using the (existing) decision model
- At product level
  - System (-specific) tests are derived using the (existing) decision model

- How can (existing) methods of regression testing (e.g., test case prioritization, test coverage measurement) be re-used here?
- Are / why are the three levels (component level, feature level, and system/product level) appropriate?
- What portion of a product line development should be / needs to be dedicated to testing?
- Decisions to take a certain test strategy should be based on cost/benefits
  - Are there any quantitative data or case studies?
  - Are there any plans to do a case study?
  - Are there any “rules-of-thumb”?
  - Are we talking about short-term or long-term investment issues?
- Other issues?

## Discussion Round 1 (11:10 – 12:30)

- **The Topic:**  
**Managing the complexity of your test space: challenges, ideas, solutions**
- **The Plan:**
  - Elevator statement:
    - Short, clear, compelling description of the topic. What does it mean? What is the problem? Why is it important?
  - Technology gap:
    - Define the gap: Where are we (in industry and research)? Where would we like to be?
    - Decompose the gap in problems/challenges.
    - If possible, prioritize the problems/challenges.
    - Collect ideas and discuss possible solutions.
  - Related work:
    - Is there other work/areas/projects that might be relevant to look at/investigate/consult?

**SPLIT**  
2005

Rennes, September 26, 2005

1

## Elevator Statement

- **Managing the complexity of your test space: challenges, ideas, solutions**
- **Good morning, Mr. Wichtig. Let me use the opportunity of sharing the elevator with you for informing you about my latest project:**
- **Adopting a PL approach has increased developers' productivity by letting them configure and integrate existing assets. But we still have to completely retest each product, so that's become a real bottleneck and soon 90% of our staff will be testers.**
- **We can't test our core assets over all settings of their variation points and, anyway, we still couldn't be certain that they would work together after integration.**
- **We should try to architect our products to reduce the interactions between variation points and between core assets if we are to "divide and conquer" in testing.**



**SPLIT**  
2005

Rennes, September 26, 2005

2

## Technology Gap:

- **What is the current state of practice?**
  - Knowledge about commonalities and variabilities hardly leveraged.
  - Constraints on variation points not (sufficiently) specified.
  - Testing efforts focused on system testing.
  - Test space complexity managed by “adjusting test coverage criteria”.
- **What is the current state in research?**
  - PL techniques for test design – reusable test assets
  - Prioritizing core assets to be unit tested
  - Coverage criteria
  - Composing product tests from core asset tests
  - Regression testing of product variants
  - Architecture-specific test strategies, e.g. framework-based
  - Core technologies of testing, e.g. TTCN-3 language & tools
  - Architecting and design for testability
- **Where would we like to be?**
  - Only new features need to be tested. Only in the relevant configurations. We have tool that tells us when we need to do additional testing and where – maybe automatically generates and runs the tests.



**SPLiT**  
2005

Rennes, September 26, 2005

3

## Let's work together!



**SPLiT**  
2005

Rennes, September 26, 2005

4

## Technology Gap: Decompose Problem

- **Characterizing individual variation points**
  - What test coverage criteria are appropriate for different variation mechanisms?
  - How many test cases are needed for each configuration?
- **Reducing combinatorial complexity**
  - Variation points on one asset can interact so that combinations of configurations have to be tested
  - Can assets be designed to minimize these interactions?
  - Can the number of combinations be minimized through the product line decision model?
- **Test architectures and test case design for variability**
  - To what extent should we aim to test core assets to be confident that they can be configured for any product

vs.

being able to rapidly test configured core assets during product, prior to integration?



**SPLIT**  
2005

Rennes, September 26, 2005

5

## Technology Gap: Collecting Ideas

- **Manage complexity by decision/feature models**
  - Propagating from high-level models down to components and code
  - Architectural styles and relationship with other architectural concerns
  - Identifying crosscutting variabilities /aspects and what can be localized
- **Reduce complexity by constraining how variation points are implemented and instantiated**
  - E.g. at code-level, use binary switches/conditional compilation, etc.
  - E.g. at feature-level, specify constraints on instantiations
- **Adding further constraints to feature combinations and recording what has [not] been tested**
  - Possible sources of constraints: combinations that commercially don't make commercial sense, legally are not allowed, cultural norms, etc..
  - How does business model and development model impact your architecture and distribution test effort?



**SPLIT**  
2005

Rennes, September 26, 2005

6

## Technology Gap: Looking for Solutions



**SPLIT**  
2005

Rennes, September 26, 2005

7

## Related Work?



**SPLIT**  
2005

Rennes, September 26, 2005

8

# Testability

John D. McGregor  
Clemson University

## Quality Perspectives

- Individual user of a product from a product line sees no difference in number of problems
- The product line organization sees the aggregate of all problems

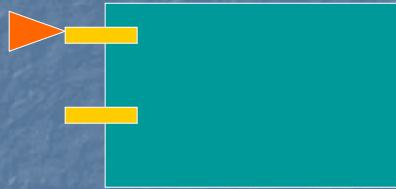
## Testability is

- The ability of software to reveal its faults
- The more difficult it is to apply test cases or validate results the more likely it is that defects will escape detection
- Voas defines testability of a program P to be a prediction of the probability of software failure occurring if the software were to contain a fault, given that software execution is with respect to a particular input distribution.

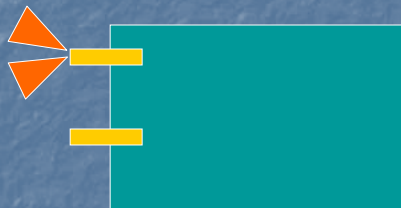
## Testability is

- Related to the size and complexity of the module

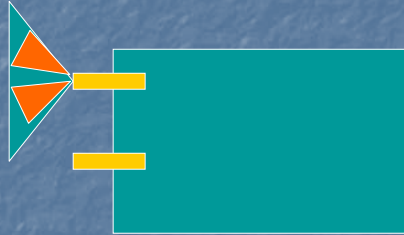
## Single Context



## Multiple Contexts



## Product Line of Contexts



## Product line issues

- Wider range of contexts
- More executions

$$\text{numDefectsExecuted} = \sum_{k=1}^{\text{numContexts}} (P_k(d_i) * np * \sum_{j=1}^{\text{numComponents}} (nc_j * x))$$

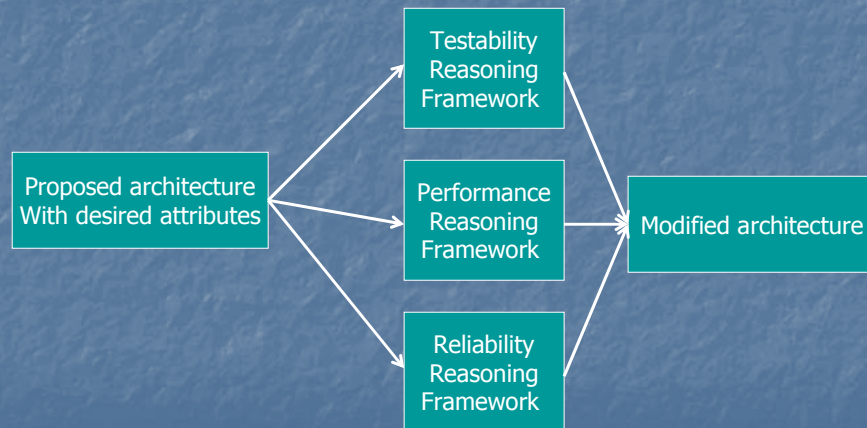
## Product line implications

- Defects will be revealed sooner in the field because of more executions across wider range of data
- More testing is needed to achieve same level of correctness

## Increasing testability

- Make attributes publicly accessible
- Provide a test interface
- Include the test cases inside the component/module

## Reasoning framework



## Testability Framework description

- Problem
- Analytic theory
- Analytic constraints
- Model representation
- Interpretation
- Evaluation procedure

## General scenario 1

- Stimulus – A component is checked in for unit testing.
- Source of stimulus – The component developer
- Environment – In the component development phase with limited amount of component integration occurring.
- Artifact – The component under test, the test harness, and the test cases
- Response – Test cases are selected to the limit allowed by the testability of the component
- Response measure – The extent to which chosen test criteria can be achieved

## General scenario 2

- Stimulus – A subsystem is successfully built prior to integration testing.
- Source of stimulus – An integration team member
- Environment – sufficient components have passed unit test and have been integrated
- Artifact – The subsystem under test, the test harness, and the test cases
- Response – test cases are selected to the limit allowed by the testability of the component
- Response measure - The extent to which chosen test criteria can be achieved

## Problem

- Need to be able to test at a variety of levels
- Need to provide higher testability

## Analytic theory

- Visibility
  - Must be able to read a variable to verify test results
- Controllability
  - Must be able to write a variable to setup tests
- Reachability analysis
  - Builds a graph of how each state is reached
  - Trivial use to determine that a variable definition can be accessed from the interface

## Analytic constraints

- State explosion problem can constrain the size of software that can be analyzed
- Reachability is static, some code is not

## Model representation

- Reachability builds a directed graph
- Tools for this are available but vary depending upon the representation

## Interpretation

- The accessibility of an attribute is binary.
- Either it is accessible or not.
- As an aggregate, a percentage of the attributes are accessible
- The higher the percentage that is accessible, the more testable

## Evaluation procedure

- Compare testability percentages
- Rank from smallest percentage to largest
- Incorporate into overall architecture analysis

## Designing for Testability

- Just making all attributes accessible is not a good design move
- Adding a test interface allows accessibility but controls the accessibility
- But this can impact the memory footprint
- For a real-time, embedded system that can be a problem

## Summary

- Aggregate quality is a problem, particularly in certain domains.
- It has implications for organizations that use product line marketing
- Just increasing test coverage may not be sufficient
- An appropriate level of testability needs to be provided in the architecture

## Guidelines for Discussion Rounds

### **Organization**

- Each discussion group should have one moderator for leading the discussion and one person (can be the same as the moderator) for documenting the results and presenting them later on during the wrap-up session to the rest of the workshop participants.
- Please prepare an *electronic* version of your results (ppt preferred – including names of participants).

### **Deliverables/Results**

Please consider the following points when discussing and preparing your results. Of course feel free to consider *additional* items ☺

- Prepare an **elevator statement** for your topic: How would you describe the topic in a few sentences? Why is it important? Why should we spend effort on it? The description should be clear and compelling.
- Define the **technology/practice gap**: what are the needs from industry, what is the current state-of-practice in industry, and what is the current state in research?
  - What do you think are the main problems/gaps?
  - Which of them are product-line specific?
  - Discuss possible solutions.
  - Prioritize/weight the problems/gaps - according to relevance, timeline (what would you like to see addressed next?), size of gap (how close are we to a solution?), and everything else that you think is relevant.
- Do you know about **related work** that is relevant for the discussed topic?

**- Notes -**

**- Notes -**

**- Notes -**

**- Notes -**