Proceedings of

SPLiT 2006 – Third International Workshop on Software Product Line Testing

Editors: Peter Knauber, Charles Krueger, Tim Trew

Informatik-Berichte
Hochschule Mannheim – Fakultät für Informatik

Computer Science Reports
Mannheim University of Applied Sciences – Computer Science Department

CSR 003.06
August 2006

URL: http://www.informatik.hs-mannheim.de/reports

hochschule mannheim

FAKULTÄT FÜR INFORMATIK

Organization

SPLiT is co-located with 10th International Software Product Line Conference (SPLC) 2006.

Workshop Chairs:

- Peter Knauber
  Mannheim University of Applied Sciences, Germany
  p.knauber@hs-mannheim.de
- Charles Krueger
  BigLever Software, USA
  ckrueger@biglever.com
- Tim Trew
  Philips Research, The Netherlands
  tim.trew@philips.com

Program Committee:

- Guenter Böckle, Siemens AG, Germany
- Krzysztof Czarnecki, University of Waterloo, Canada
- Hassan Gomaa, George Mason University, USA
- Georg Grütter, Robert Bosch, Germany
- Frank Roessler, Avaya Labs, USA
- Stefan Jungmayr, Teradyne Diagnostic Solutions, Germany
- Ronny Kolb, Fraunhofer IESE, Germany
- John McGregor, Clemson University, USA

Table Of Contents

Keynote

Challenges for Testing in Software Product Lines
Georg Grütter 1

Accepted Papers

A Reuse Technique for Performance Testing of Software Product Lines
Sacha Reis, Andreas Metzger, Klaus Pohl. 5

Data Driven Test Environment Considerations when Developing Controls Product Line Test Architecture
Jamie J. Williams 11

Customizable Requirements-based Test Models for Software Product Lines
Erika Mir Olimpiew and Hassan Gomaa 17

A Verification Approach for Crosscutting Features Based on Extension Join Points
Roberta Coelho, Vander Alves, Uirá Kulesza, Alberto Costa Neto, Alessandro Garcia, Arndt von Staa, Carlos Lucena, Paulo Borba 23

Invited Talk

ODC crystallizes Test Effectiveness
Ram Chillarege. 31

Paper Presentations

Workshop Introduction
Tim Trew, Peter Knauber 33

Keynote: Challenges for Testing in Software Product Line Testing
Georg Grütter 38

A Reuse Technique for Performance Testing of Software Product Lines
Andreas Metzger 47

Considerations in Developing a Controls Product Line Test Architecture
Jamie J. Williams 61

Customizable Requirements-based Test Models for Software Product Lines
Erika Mir Olimpiew 67

A Verification Approach for Crosscutting Features Based on Extension Join Points
Vander Alves79 79

Invited Talk: ODC crystallizes Test Effectiveness
Ram Chillarege. 91

Breakout Session

SPLiT 2006 Breakout Session Introduction
Peter Knauber, Tim Trew 1

Breakout Group on Design for Testability
Jamie Williams, Kathrin Scheidemann, Christian Tischer, Peter Knauber 98

Breakout Group on Test Strategy
Ram Chillarege, Georg Grütter, Ronny Kolb, Tim Trew, Kentaro Yoshimura . . . 100

Breakout Group on Adapting Techniques and Methods for Testing Software Product Lines
Erika Olimpiew, Hotae Kim, Rob Reckzine, Vander Alves 104

Workshop Summary

Summary of SPLiT 2006 – Third Workshop on Software Product Line Testing
Tim Trew. 107

Keynote Outline - Challenges for testing in software product lines

Georg Grütter, Robert Bosch GmbH, CR/AEA3-Fr
georg.gruetter@de.bosch.com

1 Motivation

The importance of SW for Bosch products has been and will be continuously growing. Bosch manufactures hundreds of thousands of electronic control units per year for the automotive and other industries worldwide. The number of software developers Bosch employs worldwide rivals that of major players in the software industry. The majority of software intensive products are developed in software product lines.

A lot of Bosch's software is safety relevant, like e.g. the ESP, engine and airbag controller software. It is also continuously growing more complex. The software for engine control units typically contain up to 840.000 lines of code (before compile time and including the feature superset for a high number of variants), processes the input of up to 25 sensors, controls up to 31 actuators, has up to 5200 features and about 7200 calibration parameters. Some of the Bosch product lines yield up to 1000 products in one year.

Since 2000, Bosch successfully applies the Software Product Line Practices Framework (SPLP) of the SEI in various business units. The new value motronic product line of the Gasoline Systems business unit has been successfully introduced into the market. The same is true for the new AB10 generation of airbag controllers produced by the Automotive Electronics division. The development of both these product lines has been guided by the SPLP Framework from the outset. A thorough scoping combined with a solid business case, as well a good system and software architecture, have been the deciding success factors for both product lines.

In the light of the growing complexity and the safety criticality, Bosch regards testing as one of the major challenges for its future software development. The software product line testing community has addressed many of the technical challenges and proposed processes and practices during the last few years. In our experience, design for testability is both the most fundamental and best understood of those. However, there are a number of important nontechnical challenges for product line development that have, so far, not been addressed. These are:

1. Meeting the product developers' quality expectations for core assets
2. Establishing testing as a practice well regarded by managers and developers
3. Controlling the growth of variability
4. Making design decisions for testability

In [1], Barry Boehm argues that in the software industry, there is currently no good understanding of the business impact that software design decisions have. We might add that this is also true for testing. For the software product line paradigm, being business-centric, it is paramount. With business impact we refer to everything that has an impact on the achievement of the business goals of an organization. For a company trying to achieve the business goal "Quality leadership" e.g., a design decision to apply the latest and immature technology potentially has a negative business impact. It is our hypothesis that the lack of

understanding of this business impact is the root cause for the last three challenges not being solved yet. But first things first.

2 Meeting the product developers' quality expectations for core assets

One of the promises of product lines is increased quality. This is a very appealing argument for companies producing safety critical software products. The assumption behind the quality promise is, that a component, if reused very often, will mature in the process of reusing it. However, this assumption is usually quickly forgotten. What remains is: "With product lines, we get better quality than before". Furthermore, in the market of safety critical applications there really is no margin for "maturing in the field" - "quality by quantities" does not work here.

The expectations in management and project development are high and so is the pressure on the core asset developers. The common expectation is that core assets will meet higher quality standards from the outset than before product line development was employed. Additionally, the core asset developers are expected to fill up the core asset base with a complete set of core assets early in order to keep the time-to-market promise. This, certainly, doesn't make things easier.

In our experience, when product developers find out that the quality promise is not kept, the faith in product lines can be seriously shaken pretty quickly. Product developers tend to revert to the old ways of clone and own and the whole idea of product line may die.

All this puts a high pressure on testing. Although quality cannot be "tested into" core assets, the testers must make sure that even less defects slip through to product development than before. At the same time, variability in the core assets makes testing more difficult.

To deal with these challenges, we need to convey a realistic and clear picture of what the effects of product line development are and when they will take place. We see two, complementary means to achieve this. The means is to create a mind share in the organization. Have key persons temporarily participate in the work of other teams. That is, a core asset developer works with the product development team and vice versa; a manager spends a day with the developers to get a feel. The second means is to measure and try to quantify the experiences made during this exchange.

3 Establishing testing as well regarded practice by management and developers

The problem is not exactly new. The only surprising thing is that it is still around. Testing is an inherently destructive task. Most developers prefer creating software to trying to break it. That is why software developing organizations operate test centers. Management often views testing as unproductive and does not give any or not enough incentives for proper testing. When the pressure is on, the decision "Do we start working on the next feature our customer wanted or do we finish testing the ones already realized?" is easily made. There is an inherent vicious circle to this: the more pressure there is, the less testing is done, the less stable core assets are, the more rework is needed, the less productivity there is, the more pressure there is, and so on.

Before product line testing can lift off, we need to break that vicious circle. We need to acknowledge that, although inherently destructive, testing actually creates value. Automated suites of tests are valuable, reusable core assets. As with other development efforts for product lines, a rather huge up-front investment into testing is necessary. The problem is this: we are usually not able to specify the positive business impact of testing and tests. How exactly does testing help us to attain our business goals and how could we weigh this against the costs. The costs, however, are clear, which makes it hard to break the vicious circle.

4 Controlling the growth of variability

The major source of complexity within product lines stems from the number of variants. Variation points are often a source of faults. Testing all variants of core assets a priori is usually impossible for all but the simple cases. So far, research in product line testing has focused on making variant testing more effective and efficient.

We propose to additionally investigate how we can identify or reject those variants that have a negative overall business impact. We'd need to be able to specify the business impact of those design decisions and tradeoffs that would need to be altered for those new variants. We would need to be able to specify, how much additional testing effort a variant would cause. In our experience it is very hard to reject requests for unprofitable variants without a justifiable estimate of their negative business impact. The impact on testing and the overall health of the product line is obvious.

5 Making design decisions for testability

In the automotive embedded software domain, the dominating qualities of software are currently reliability and resource consumption. It has been recognized that, for product lines to fulfill their promise of increased quality, decreased cost and shorter time to market, testability needs to be added to this list. Design for testability is what is needed.

A lot of appropriate design decisions are known, such as "Separate variant from invariant code" or "Separate controller software from information processing software". In our experience, the impact of design decisions for testability on the system architecture can be substantial. Changing the architecture of a system might affect its reliability in the short run.

Hence, a precondition for these design decisions to be put into practice is, that we understand and communicate their business impact. Barry Boehm stated in [1]: "The links between value and software design are tight but still not understood well enough today. ... The connection between technical parameters and value creation are understood vaguely, if at all.". The questions to be answered are: "How will design for testability impact other qualities?" "How much more effective and efficient will testing be when design decision x is implemented?".

Without being able to answer these questions, design decisions for testability will likely not be implemented and the testing challenge will remain. Obviously, further research in this area is needed.

6 Conclusion

The product line paradigm is business centric. In order to achieve economical success, we need to quantitatively understand the business value of the technical decisions that we can make. The absence of such knowledge constitutes a serious problem. This is especially true for testing, which is not yet widely viewed as creating value in the first place.

So we need to find adequate models. George Box said "All models are wrong, some are useful". We need to focus on the useful models that help us to understand and communicate what we do in product line testing.

The Personal Software Process (PSP), e.g., has demonstrated how building and applying models can be put to good use. We should also learn how other engineering disciplines align themselves to business goals. But in the end, we need to measure, measure, measure, build and validate our own models, which might be different for each company and domain.

From all this also stems a requirement for the future technically oriented research in our field: it should quantitatively describe the possible business impact of its results. Although laborious, controlled software engineering experiments will help to verify and communicate the utility of the proposed new testing processes and practices.

References

[1] Barry W. Boehm, Kevin J. Sullivan: *Software Economics*, University of Southern California and University of Virginia, 1999

A Reuse Technique for Performance Testing of Software Product Lines

Sacha Reis[*], Andreas Metzger[*], Klaus Pohl[*,+]
[*]Software Systems Engineering, ICB
University of Duisburg-Essen, Germany
(reis|metzger|pohl)@sse.uni-due.de
[+]Lero
University of Limerick, Ireland
pohl@lero.ie

Abstract

Testing that the applications of a software product line comply with their functional as well as with their non-functional requirements (for example performance) is important for achieving the desired product quality. Existing approaches for software product line testing only deal with testing an application against its functional requirements. In this paper we present a technique that supports the development of reusable performance test case scenarios in domain engineering and the reuse of these scenarios in application engineering. The technique is an extension of the ScenTED technique for system testing from our previous work. The technique focuses on load testing and performance profiling, two types of performance testing, and it has been validated in a case study at Siemens Medical Solutions HS IM.

1. Introduction

Software product line engineering (SPLE) is based on the planned, systematic, pro-active reuse of development artifacts, which is reflected by the following two characteristics of SPLE [9]:

1. The division of the development process into domain and application engineering.
2. The explicit modeling of variability, i.e. the documentation of variation points and variants.

Like in the development of single systems, making sure that an application complies with its functional requirements as well as with its non-functional (or quality) requirements is also important in software product line engineering.

Several approaches for the development of reusable test artifacts in domain engineering and the reuse of these artifacts in application engineering exist (e.g., [2][6][7][10]). However, these approaches focus on testing functional requirements only. There are currently no product line specific approaches that deal with testing performance or other non-functional requirements.

Performance test approaches from the development of single systems (e.g., [11][12]) do not support product line variability and therefore no pro-active reuse of performance test artifacts is possible with these approaches.

In this contribution, we present the ScenTED-PT technique that supports the development of reusable performance test case scenarios in domain engineering and the reuse of these scenarios in application engineering. ScenTED-PT is an extension of ScenTED, a requirements- and model-based system testing technique for product line engineering [10]. We have chosen performance, because it is one important kind of non-functional requirements and also because performance was of major interest to our validation partner Siemens.

The basic idea of ScenTED-PT is to map the variability in the functional as well as in the performance requirements (documented by use cases and scenarios) to domain performance test artifacts. In application engineering, the variability in the domain performance test artifacts is then bound to create application-specific performance test artifacts. The binding can be performed in different ways, depending on the kind of performance test that should be performed.

The remainder of the paper is structured as follows. After an overview of how variability in performance requirements is documented (Section 2), the activities of ScenTED-PT are explained in Section 3. Finally, the case study and its results are presented in Section 4.

2. Variability in Performance

Performance is defined as conditions or constraints which are added to the functional requirements concerning their accomplishment such as speed, accuracy, or memory usage [5].

For ScenTED-PT we have used the OMG's *UML Profile for Schedulability, Performance, and Time* (*SPT*, see [8]) as a basis to model performance aspects. To document variation points and variants, we have introduced specific annotations, because the original profile did not allow expressing product line variability.

Figure 1 shows a sequence diagram including variable performance requirements, which are expressed with our

extension of the SPT profile. The variation point VP1 specifies variability in the load of the scenario (SPT stereotype <<PAclosedLoad>>). In the example, the possible variants are expressed by the parameter $n.

Depending on the population of the load (SPT tag PA-population), the response time of the last scenario step can vary. Therefore, VP4 is specified and a dependency to VP1 is expressed by quantifying the response time by referring to the parameter $n.

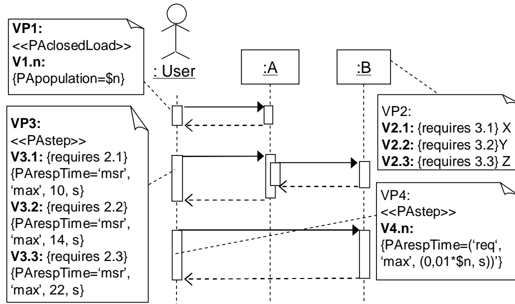


Figure 1: Example of variability modeling

The response time of the second step of the scenario is modeled by three different variants of VP3. The response time depends on the realization of the component B, where B is a placeholder for three different realizations X, Y, and Z. Depending on the selection of the concrete component, the maximum response time is allowed to vary. This dependency is modeled by adding a requires dependency to the variants of VP3 and VP2.

3. ScenTED for Performance Testing

The ScenTED-PT technique is model-based, i.e. based on the requirements and the architecture of the system, test engineers develop a test model from which performance test case scenarios are derived.

ScenTED-PT, like ScenTED, employs use cases as an input to the testing process. In more detail, a use case diagram and textual descriptions of each use case that describe the use case scenarios are used as input.

ScenTED-PT consists of five activities (four in domain engineering, and one in application engineering):

1. The use case scenarios of the system are manually supplemented in domain engineering with performance requirements. The use cases and their scenarios represent domain requirements and therefore they can contain variability.
2. A basic test model is manually created from the extended use case scenarios.
3. The test model is manually supplemented with architectural information, i.e. the system-user interactions are refined to interactions between the different components. This architectural information is needed for specific kinds of performance tests (see Section 3.3).
4. Based on the supplemented test model, domain test case scenarios (DTCS) are automatically derived from the test model using a coverage criterion.
5. Application test case scenarios (ATCS) are automatically derived by binding the variability of the DTCS for a customer-specific application.

3.1. Supplementing UC Scenarios (Activity 1)

Input to the first activity of ScenTED-PT is the textual and template-based description of domain use cases. These use cases can contain variability caused by different functionalities that should be available for the product line applications. In the first activity of ScenTED-PT, a requirements engineer has to supplement these use cases with performance requirements. These performance requirements can cause additional variability, which is also documented. The output is a set of use case scenarios extended with performance requirements.

Figure 2 shows the use case diagram of a simple E-Shop product line that we will be using as a running example. Variation points are represented by black triangles (see [4]). The E-Shop product line consists of five use cases. The common use cases define how customers can search and buy items. For payment, E-Shop applications can provide payment with invoice, with card or both. For card payment, an E-Shop application can provide either credit card or debit card payment.

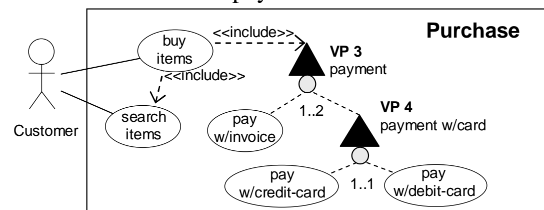


Figure 2: UC diagram of an E-Shop example

We have chosen the use case buy items to illustrate the supplementation of a use case with performance aspects. In the textual description of the use case, variation points and variants are explicitly documented in addition to the individual scenario steps. As an example, Step 6 of the scenario contains variability caused by performance:

Step 6: System calculates and shows the invoice {VP6}

VP6 defines variability in the maximum calculation time of Step 6:

VP6 <<PAstep>>
V6.1: Has to be accomplished within 50 ms
V6.2: Has to be accomplished within 100 ms …

Variation points as well as dependencies between different variation points or variants are specified within

curly brackets. VP5 defines the variability in the load of the system:

VP5 <<PAclosedLoad>>
- **V5.1:** The number of active users in the system is limited to 500 {requires V6.1};
- **V5.2:** The number of active users in the system is limited to 1000 {requires V6.2} …

3.2. Creation of the Basic Test Model (Activity 2)

The result of the second activity of ScenTED-PT is a basic test model. Based on the use case descriptions that have been supplemented with performance requirements, test engineers develop a test model based on functional and performance requirements. In ScenTED, this basic test model is represented by a UML activity diagram.

In a first step, for each use case, an individual activity diagram is created. Each activity diagram describes all possible scenarios of the associated use case as well as the variability and performance aspects.

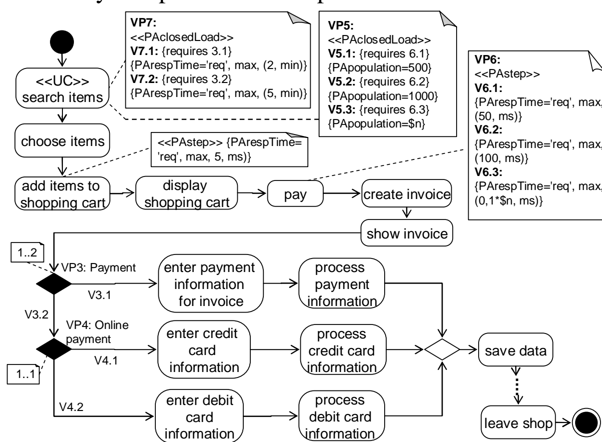


Figure 3: Basic test model of *buy items*

Figure 3 shows the activity diagram of the use case buy items. In this activity diagram, variation points in the control flow are shown as black diamonds. Variants are transitions that emerge from such variation points.

The test model contains the action search items, which refers to the activity diagram of another use case (specified by the stereotype <<UC>>). This reference reflects the inclusion of the use case search items by the use case buy items (see Figure 2).

The information about variation points 5 and 7 is attached to the action search item, because – following the SPT profile – all information that affects the whole scenario has to be attached to the first action in the diagram. (see [8]). The distinction whether the information affects the whole scenario or only the first action can be extracted from the stereotype. The stereotype <<PAclosedLoad>> describes information of the whole scenario. The stereotype <<PAstep>> is used if only the information for the first step is described.

In a second step, all activity diagrams are merged into one overall activity diagram. The overall activity diagram specifies the order in which the individual use cases are executed. This order is determined by analyzing the pre- and post-conditions of the use cases. Activities in this overall activity diagram are stereotyped <<UC>> to denote that they are refined by the activity diagrams of the use cases. This procedure is equivalent to the one in the basic ScenTED technique [10]. The overall activity diagram is used to derive test case scenarios in the subsequent activities of ScenTED-PT. To keep the remaining examples short, we will only use the activity diagram of buy items and not the overall activity diagram.

3.3. Supplementing the Test Model (Activity 3)

ScenTED-PT supports two different types of performance tests: load testing and performance profiling. During load testing, performance requirements are tested for a given configuration. During performance profiling – in contrast – one configuration is identified which fulfils the performance requirements in the best possible way.

For both types of tests, the basic test model is not sufficient for deriving test case scenarios, because information about the possible configuration of the system is missing. Depending on the selection of components, different variants of performance requirements may be selected for load testing. For performance profiling, the information about potential configurations is essential, because one specific configuration is determined.

Therefore, in the third activity of ScenTED-PT, the basic test model is supplemented with architectural information. Information about the components of the system is added to the test model which may include additional architectural variability. The components of the system are reflected by activity partitions in the activity diagram. Each action is associated to one activity partition. Additional actions might have to be added if the basic test model was not detailed enough.

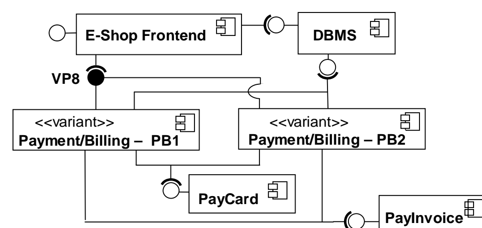


Figure 4: Architecture of the E-Shop

In Figure 4 the architecture of the E-Shop example is shown as a UML component diagram. The architecture consists of six components. The user of the system inter-

acts with the E-Shop Frontend. All data are managed by the data base component DBMS. The payment is managed by the Payment/Billing component. Two different implementations of Payment/Billing exist (PB1 and PB2). The components differ in the complexity of the implementation and therefore vary in their performance, i.e. in the memory usage as well as in response time. Each Payment/Billing component uses the components PayCard and PayInvoice. These two components implement the functional variants for the payment by invoice and the payment by credit or debit card.

Assigning actions to components is done manually and requires the knowledge of the detailed design or implementation. In Figure 5 the extended test model for the use case buy items is shown. Because of architectural variability, the variation point VP8 from Figure 4 has been added.

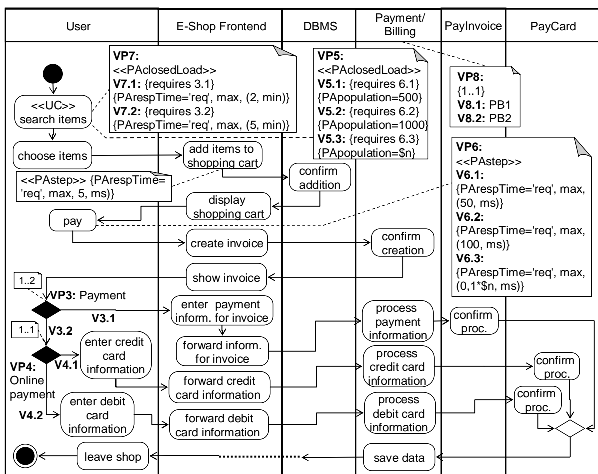


Figure 5: Extended test model of *buy items*

3.4. Derivation of DTCS (Activity 4)

In the fourth activity of ScenTED-PT, domain performance test case scenarios are derived on the basis of the extended test model. The derivation is performed in the same way as in the original ScenTED technique. Under consideration of an adapted branch coverage criterion, test case scenarios are derived that preserve variability (see [10]). First, the variability in the control flow is handled as a black box and the coverage criterion is applied to all invariant parts of the model including the black boxes for the variability. Then, the black boxes are replaced with the variable control flow. The result is a set of performance test case scenarios that allow 100% branch coverage for each customer-specific application.

The resulting domain performance test case scenarios are represented by UML sequence diagrams. For each activity partition of the activity diagram, one role with a life-line exists. For the use case buy items one domain performance test case scenario already satisfies our adapted branch coverage criterion (see Figure 6).

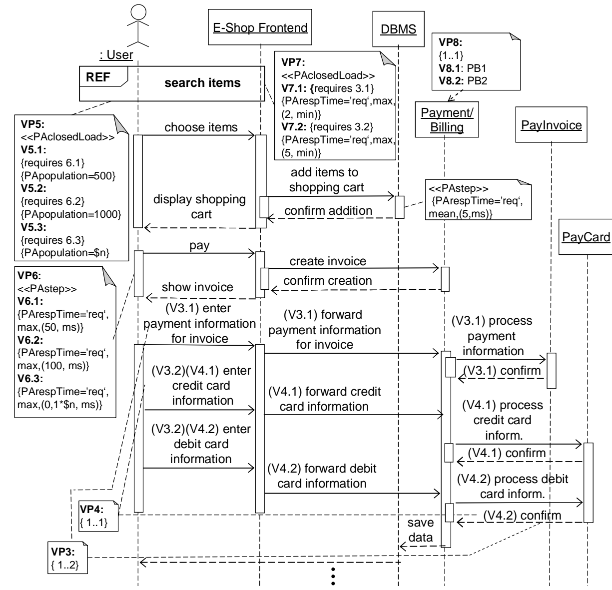


Figure 6: Domain performance test case scenario for *buy items*

3.5. Derivation of ATCS (Activity 5)

The final activity of ScenTED-PT is to derive test case scenarios in application engineering.

For load testing, the complete variability of the domain performance test case scenarios is bound. The binding depends on the binding information that is defined both by customers and the designers. The selection of the variants by the customers is reflected in the binding of the domain requirements. The variants that are selected by the designers are reflected in the binding of the domain architecture. When deriving the DTCS, the variability was handled as a black box. Because of this abstraction, it is possible that more than one ATCS has to be derived from one DTCS to guarantee 100% branch coverage for a customer-specific application. If more than one variant has been selected for a variation point, more than one ATCS is necessary to guarantee coverage.

In the E-Shop example, the selection of the variants V3.1, V5.1, V6.1, V7.1, and V8.1 results in the derivation of one application performance test case scenario (see Figure 7). The component PayCard is not part of the scenario, because the variant V3.2 is not selected. If also the variant V3.2 were part of the application, two ATCS would be necessary to achieve 100% branch coverage.

During performance profiling, the configuration of the application is not fixed, because the best possible configu-

ration should be identified. For this reason, initially the variability in the DTCS is not bound completely. All variability is bound except for the variability in the architecture that is performance relevant. Then, for the remaining architectural variability, test case scenarios for all potential bindings (i.e., configurations) of this variability are derived. The Test cases for all these test case scenarios are executed and evaluated. The scenario that fulfils the performance requirements in the best possible way defines the configuration that will be chosen for the application. If many performance relevant variation points and variants in the architecture of the product line exist, the number of possible scenarios can become very high. In that case, one could rely on the experience of domain experts to select a representative set of test case scenarios.

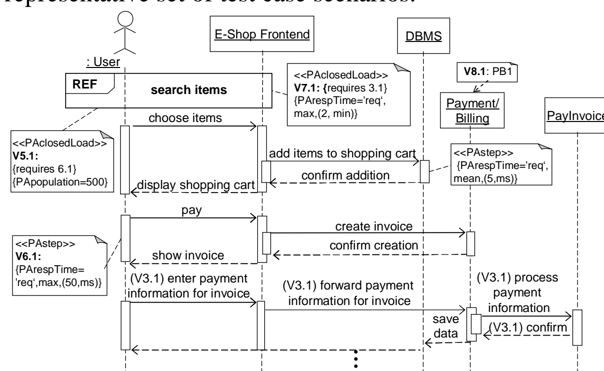


Figure 7: Application performance test case scenario for *buy items*

In the E-Shop example, the variation point VP8 represents variability in the architecture. Two different components implement the same functionality in a different – and performance relevant – way. For this reason this variability is initially not bound for performance profiling. All possible application scenarios are derived out of the partially bound scenario. As a result, two different scenarios were derived. One includes the variant V8.1 (component PB1) and the other one includes variant V8.2 (see component PB2).

4. Case Study

ScenTED-PT was partially validated in a case study at Siemens Medical Solutions HS IM. Siemens AG Medical Solutions HS IM develops software-systems for workstations in the radiology domain. The tasks of those radiology systems cover the complete clinical workflow. The radiology systems vary in their hardware, like CT or X-Ray scanners, or high-end or low-end hardware for the data server, as well as in different functionally, like workstations with 3-D or only with 2-D functionality.

In the testing process of Siemens Medical Solutions HS IM, performance testing plays an important role. Traditionally, performance requirements have been separately defined in the specifications of the product line and performance test cases have been derived under consideration of a Siemens specific performance framework.

Because of specific needs of Siemens, ScenTED-PT was adapted. The most important adaptation was the use of existing DTCS for system testing that have been created in previous work [3].

The most important observations of the case study were as follows:

Early validation of performance requirements and variability. To supplement the DTCS with the performance requirements, the test engineers have to analyze and understand the performance requirements. This analysis leads to a validation of the performance requirements concerning completeness, inconsistencies, ambiguities, and testability. Test managers at Siemens have estimated the early validation of the performance requirements as one of the most important benefits of the application of ScenTED-PT. In this case study, 35 performance requirements out of a total of 51 were associated to the domain performance test case scenario. The other 16 requirements affected other parts of the SPL that were not within the scope of our case study. From the 35 selected requirements, 5 requirements (14.3%) have been identified that could not be tested, because of a too abstract description. Also, two additional requirements were identified (5.7%). Altogether, 20 performance requirements (57.1%) were supplemented with variability for further reuse.

Additional tags are necessary to model loads. During the supplementation of the DTCS with performance requirements, it was not possible to completely specify specific loads with the SPT profile. Therefore, we have introduced two additional tags to support the modeling of the size and the number of objects.

Traceability. Traceability from the requirements specification, over the domain performance test case scenarios, to the application performance test case scenarios is enabled by annotating performance aspects with identification numbers. Tools (Mercury TestDirector for this case study) efficiently support the automated tracing.

Synergy Effects. ScenTED-PT has realized synergy effects with system testing, because ScenTED-PT was based on ScenTED for system testing. In this case study, the overall activity diagram and the use case scenarios were reused from the system test. Moreover, the domain system test case scenarios were reused and supplemented with architecture information, performance requirements, and additional variability. Therefore, effort was reduced, because testers did not have to create use case scenarios and the main workflow from scratch.

Selection of Configurations. The variability of the domain performance test case enables the identification of an optimized customer-specific configuration. In the case study, application performance test case scenarios were derived for two hypothetical customers. The performance requirements of the customers were different, because of different basic conditions (e.g., number of examinations per year, type and size of pictures). For the customers two different configurations were identified that fulfill the performance requirements in the best possible way.

Customer-specific Performance Tests. On the basis of the domain performance test case scenario, the derivation of customer-specific performance test case scenarios is possible. The variability in the DTCS enables the customer-specific selection of variants and therefore the adaptation of the scenario to the customer-specific needs.

5. Conclusion and Future Work

In this paper, we have presented the ScenTED-PT technique for the derivation and reuse of performance test case scenarios in software product line engineering. ScenTED-PT bases on ScenTED for system testing that was developed in our previous work [10][3].

The ScenTED-PT technique consists of five main activities. In all five main activities performance and variability is considered and preserved. The UML Profile for Schedulability, Performance, and Time of the OMG was extended to model the variability in performance aspects.

In cooperation with Siemens AG Medical Solutions HS IM ScenTED-PT was partially validated in a case study. All goals that have been defined by Siemens were achieved in this case study. One significant benefit was the early validation of the performance requirements, i.e. the early identification of defects in the requirements specification.

In our future work, we will approach the complexity problem in performance profiling. When the variability in performance is bound the number of possible scenarios can be very high depending on the architectural variability (see Section 3.5). We plan to develop a technique without relying on a domain expert. Moreover, we focus on the extension of ScenTED concerning integration testing. In addition, the prioritization of test case scenarios will be examined. We will develop a risk-based prioritization technique, i.e. we want to prioritize the test case scenarios to enable the execution of the most critical ones first.

6. Acknowledgements

We thank the employees of Siemens, in particular Helmut Götz, Frank Rometsch, and Josef Weingärtner, for the opportunity of validating our approach.

7. References

[1] Avritzer, A.; Kodek, J.; Liu, D.; Weyuker, E.J.: Software Performance Testing Based on Workload Characterization. In: *Proc. of the 3rd Workshop on Software and Performance – WOSP'02 (Rome, Italy, July 2002)*, ACM Press, 2002.

[2] Bertolino, A.; Gnesi, S.: PLUTO: A Test Methodology for Product Families. In: *Proc. of the 5th Intl. Workshop on Software Product-Family Engineering (Siena, Italy, Nov. 2003)*, LNCS 3014, pp. 181-197, Springer, 2004.

[3] Götz, H.; Kamsties, E.; Neumann, J.; Pohl, K.; Reis, S.; Reuys, A.; Weingärtner, J.: Testing a Product Line of Radiology Systems at Siemens. In: *Proc. of the 5th Conf. on Software Validation for Healthcare*, Düsseldorf, Germany, April 2005.

[4] Halmans, G.; Pohl, K.: Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2(1), Springer, pp.15-36, March 2003.

[5] IEEE Std. 610.12-1990; *IEEE Standard Glossary of Software Engineering Terminology*, New York, 1990.

[6] McGregor, J.D.: Testing a Software Product Line. *Technical Report CMU/SEI-2001-TR-022*, SEI, Carnegie Mellon University, December 2001.

[7] Nebut, C.; Pickin, S.; Le Traon, Y.; Jezequel, J.-M.; Reusable Test Requirements for UML-Modeled Product Lines. In: *Proc. of the Intl. Workshop on Requirements Engineering for Product Lines*, pp. 51-56, September 2002.

[8] Object Management Group: UML Profile for Schedulability, Performance, and Time. *OMG Specification*, ptc/02-03-02, July 2002.

[9] Pohl, K.; Böckle, G.; van der Linden, F.: *Software Product Line Engineering - Foundations, Principles, and Techniques.* Springer, 2005.

[10] Reuys, A.; Kamsties, E.; Pohl, K.; Reis, S.: Model-based System Testing, In: *Proc. of the 17th Conf. on Advanced Information Systems Engineering (Porto, Portugal, June 2005)*, pp.519-534, Springer, 2005.

[11] Vokolos, F.I.; Weyuker, E.J.: Performance Testing of Software Systems. In: *Proc. of the 1st Intl. Workshop on Software and Performance (WOSP'98)*, pp. 80-87, 1998.

[12] Weyuker, E.J.; Vokolos, F.I.: Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. *IEEE Transaction on Software Engineering,* Vol. 26, No.12, pp. 1147-1156, 2000.

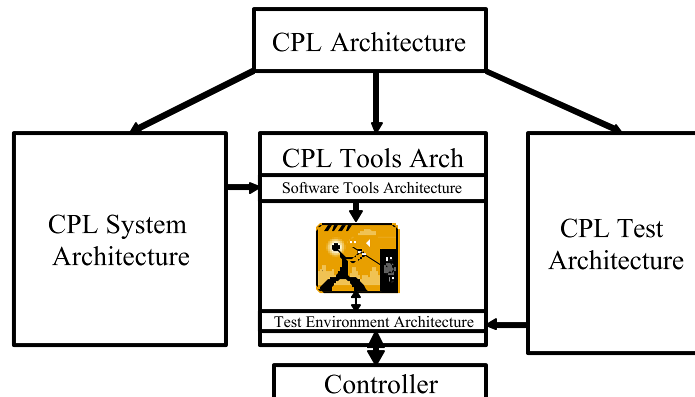
Data Driven Test Environment Considerations when Developing Controls Product Line Test Architecture

By: Jamie J Williams

1460 National Road, Columbus IN 47201
jamie.j.williams@cummins..com

Abstract. Cummins Inc. is currently developing it 2nd generation Controls Product Line Architecture. Controls Product Line (CPL) Architecture is comprised of a System Architecture, a Test Architecture and a Tools Architecture. The Tools Architecture provides the ability to verify and validate the CPL Architecture. The CPL Test Architecture is comprised of domains, taxonomy, methodologies and the test environment. The defined domains, taxonomy and methodologies drive requirements into the test environment design. Therefore, within the CPL Test Architecture, there is a need for a CPL Test Environment Architecture that will facilitate implementation of the CPL Test Architecture and verification of the CPL System Architecture. The CPL Tools Architecture includes, but is not limited to the CPL Test Environment Architecture. This paper's primary focus is the considerations in the design of a data driven CPL Test Environment Architecture and its implementation within a Controls Product Line application domain.

Controls Product Line Architecture Requirement Flow



Introduction

The Test Architecture is a design mechanism to insure that testing of the System Architecture is effective, efficient, reproducible, and repeatable and thus, produces a high confidence level in the product line quality. While the Test Architecture is comprised of test domains, test taxonomy, test methodologies and a test environment, the agreed upon test domains, taxonomy, and methodologies drive requirements into the test environment.

Likewise, the System Architecture is a design mechanism to determine the system infrastructure, requirements and requirement allocations. Just as the Test Architecture is dependent upon tools to facilitate implementation of the test architecture, the System Architecture is dependent upon tools to facilitate software development.

The ideal situation for CPL Test Environment Architecture would be the ability to leverage the points of variability of the System Architecture, inclusive of the tools to support that architecture, as well as leveraging the points of variability of the controller software. A data driven CPL Test Environment Architecture facilitates leveraging the points of variability of the Controls Product Line Architecture.

This paper is written based upon the lessons learned and experiences obtained in the design, prototyping, project management, and implementation of the CORE II Test Architecture along with 20+ years of test experience.

Test Environment Architecture

Defining the Vision and Setting the Goals:

A primary consideration in CPL Test Environment Architecture design and implementation is the vision. The vision for Core II Product Line Test Architecture was to develop an architecture that facilitates automated testing of the Core II System Architecture. With the vision there were 3 primary goals. These goals included a test architecture that was in alignment with the existing tools architecture. Additionally, the test architecture had to be in alignment with the company objective to minimize toolsets while selecting a toolset that would facilitate testing across all test levels. The functional goal of the test architecture was to create a means to minimize test execution and development time while maximizing reuse and portability. The selection of tools and the design of the Test Environment Architecture based upon a data driven concept, provided the means to achieve the vision and reach the goals.

Realizing the vision and achieving the goals

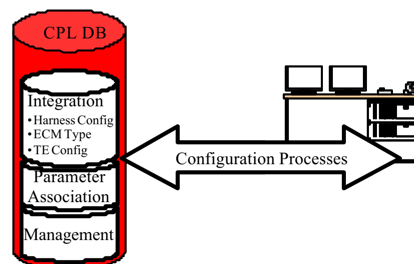
When developing a test environment that facilitates test automation, a Test Environment Architecture must be designed with consideration towards facilitating test automation in a portable, repeatable, reproducible, cost and time efficient and effective manner. With current technology and with the implementation of CPL System Architecture, the ability to meet these goals is achievable. Data driven tests provide the means to achieve these goals when data driven capabilities are designed into the Test Environment Architecture.

The CPL Test Environment Architecture can be designed and implemented to leverage build and software reconfigurability as well as leverage the System Architecture tools that control the reconfigurability. A data driven CPL Test Environment leverages the System Architecture points of variation in a manner that facilitates test environment reconfigurability. Data driven tests leverage the reconfigurability of the software within test vectors in a manner to support portability of the tests across the CPL.

Implementing the vision through test environment reconfigurability

In order to insure that data driven test capabilities are achievable, analysis led design is essential. **C**oncept, **D**esign, **O**ptimization and **C**ontrol steps need to be implemented to insure that the CPL Test Environment Architecture captures the voice of the customer, is extensible, configurable, and the lifecycle of the test environment will exceed the lifecycle of the Controls Product Line.

System architectural elements inclusive of build reconfigurability and software reconfigurability are leveraged through database interactions through automation processes that drive test environment configuration. Entries in the database are inclusive of the CPL points of variations necessary to configure the test environment to effectively interact with the controller. Database entries inclusive of I/O configuration, controller type, test environment channel, linearization capabilities, and I/O associations provide the data required to configure the test bench based upon the controller type, the I/O pin utilization on the controller, and the feature content based upon build configuration.



Implementing the vision through data driven test capability

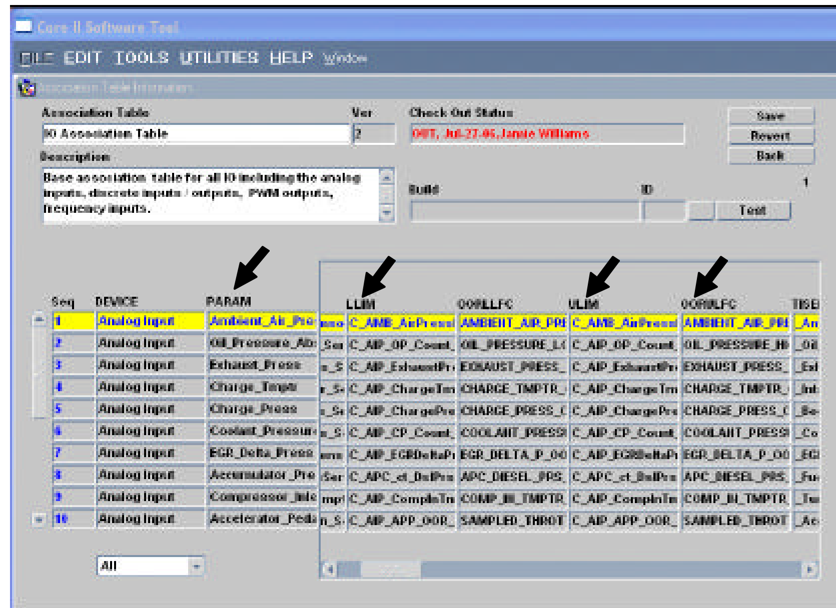
Implementation of data driven tests does not insure portability of tests across the CPL. Tests must be designed for portability by leveraging the points of variations in the software as well as in the System Architecture. This is achieved by utilizing the actual software values in lieu of constant values in test vectors. For example, Ambient Air Pressure High Limit provides the means to set the upper limit and Ambient Air Pressure Low Limit sets the lower limit of Ambient Air Pressure. The operational range of Ambient Air Pressure can be varied by redefining the upper and lower limits. Likewise, Ambient Air Pressure Low Limit and Ambient Air Pressure High Limit can be used to set the upper and lower limits on Ambient Air Pressure while maintaining a constant range, as well. The product line application's calibration of these parameters determines the operational range and implementation. However, despite the variation between CPL applications, testing the functionality of Ambient Air Pressure on all CPL applications can be achieved through one data driven test as indicated below:

```
{
Set Ambient Air Pressure = Ambient Air Pressure High Limit + tolerance.
Verify that AMBIENT_AIR_PRESSURE_HIGH_ERROR is set.
Ambient Air Pressure = Ambient Air Pressure High Limit - tolerance
Verify that that AMBIENT_AIR_PRESSURE_HIGH_ERROR is reset.
}
```

To incorporate data-driven test automation, the Test Environment Architecture must encapsulate the System Architecture points of variation into the test environment, capturing the range and variances of hardware, software, and modeling requirements while incorporating standardized end user interfaces that facilitate portability. A database provides a method to encapsulate the points of variation of the System Architecture as well as the Test Environment Architecture. An association database provides the ability to create data driven tests that are CPL generic, thus providing a portable test across the CPL. For example, the above test can be modified to the following using the below database entries:

Using the below data, the new test would appear as:

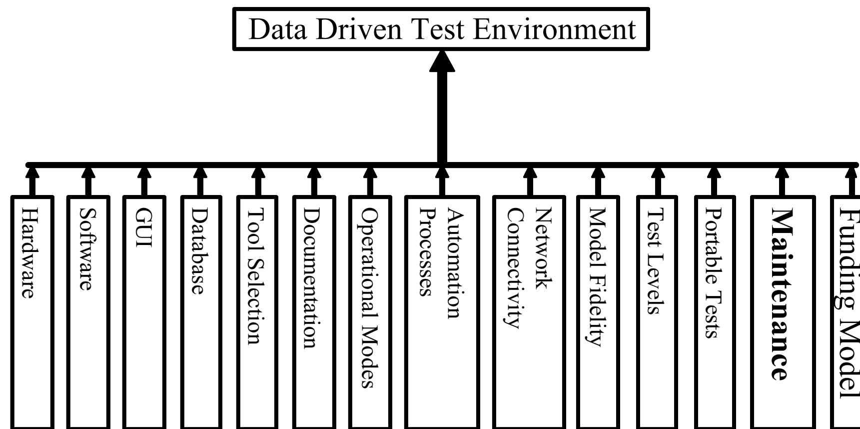
```
While PARAM /= NULL &&  ULIM /= NULL && OORULFC /= NULL;
{
Set PARAM  = ULIM + tolerance.
Verify that OORULFC error is set.
Set PARAM  = ULIM - tolerance.
Verify that OORULFC error is reset
}
```



This test completes the same verification as the original, with the exception that this test is not limited to just verification of Ambient Air Pressure. This test will verify all entries in the database that provide PARAM, ULIM, OORULFC entries. Further modification and reconfiguration of the data and a more complete test will result in a test that executes based upon the CPL application.

Considerations in implementation of a data driven test environment

Considerations that impact implementation of a data driven test environment include software and hardware selection, configuration, versioning control, and maintenance inclusive of toolset selection. Additionally, data management, test management, configuration, system and system interface documentation, toolset versioning control, data driven graphical user interface implementation, complete test automation processes and design, network connectivity in conjunction with corporate security issues, test environment operating modes in conjunction with model fidelity requirements and of course the test levels in which the environment is expected to support all are consideration in the design and implementation of a test environment architecture. Each of these considerations impact the concept of the project, the design of the architecture, the optimization capability of the architecture as well as the control process capability of the architecture.



Throughout the CDOC process, consideration to long term maintenance and support must be included in the design, otherwise, an unmanageable and non standardized test environment will surface.

Summary

Achieving a Test Environment Architecture that incorporates data driven test capabilities in conjunction with a Test Architecture that facilitates test portability requires early Concept, Design, Optimization and Control processes to address the numerous test environment input parameters. Additionally, establishing customer council that is consistently and regularly supported by customers, provides the means to capture unique customer requirements that drive the standardized test environment architecture design; provide training, roll out processes, as well as obtain buy-in to the Test Architecture, Test Environment Architecture and the test automation process that facilitates data driven testing across the product line.

Customizable Requirements-based Test Models for Software Product Lines

Erika Mir Olimpiew and Hassan Gomaa
Dept. of Information and Software Engineering,
George Mason University
eolimpie@gmu.edu, hgomaa@gmu.edu

Abstract

Customizable requirements-based test models help to manage the large number of possible feature combinations of a Software Product Line. This test design method is applied during software product line engineering to create the following customizable test models: activity diagrams from the use case and feature models, decision tables from the activity diagrams, and test templates from the decision tables. During application engineering these test models are customized for an application of the Software Product Line. This test design method was applied by five graduate students to the software product line engineering phase of an automated highway toll system case study.

1. Introduction

This paper describes how customizable test models for an SPL (Software Product Line) help to deal with the following problem: How to represent and manage a large number of possible feature combinations in the test models of an SPL. The test design method creates customizable test models from the feature and use case requirement models of an SPL. These models are developed during software product line engineering, and then customized during application engineering for the individual applications of the software product line.

Section 2 describes related work. Section 3 gives an overview of the test design method, with examples from an automated highway toll system SPL. Section 4 describes the application of this method to the automated highway toll system SPL case study by five graduate students.

2. Related work

2.1. Software product lines

A *family of systems* [1], or software 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 [2-7]. Software product line development consists of SPL engineering and application engineering. *SPL engineering* is the development of core assets for a family of systems that comprise the application domain. *Core assets* are the requirement, design, implementation, documentation, tests and any other artifacts used in the development of the software product line. *Application engineering* is the selection and customization of these assets for an application of the family. Fig. 1 describes how the development of the customizable test models fits within a software line development process. A Customizable test model is developed along with the Requirements models during Software Product Line Engineering. Feature-based test derivation is used to customize the models for an application of the SPL.

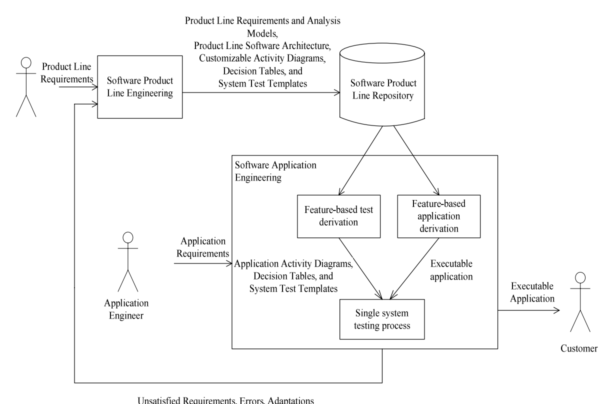


Figure 1 Software Product Line Development Processes

In some software product line development methods feature and use case requirement models are created during software product line engineering. The *feature model*, first introduced by Kang in [8] describes common, optional and alternative features, and relationships between these features. The *use case model*, first introduced by Jacobson in [9] describes a behavioral approach to defining software requirements. A use case groups sequences of interactions that provide a service of value to an outside user (actor) of the system.

2.2. Model based testing for single systems

Mayrhauser et al described an automated testing environment for command-based systems [10]. Object and command-language definitions that were useful for testing purposes were extracted from domain models. Poston [11] described a method to automatically derive test cases from test-ready object, dynamic, and functional OMT models. Binder [12] introduced the extended use case pattern. This pattern was used by Briand and Labiche [13] to list test paths and conditions traced from the interaction diagrams of a use case. Grieskamp et al [14] described how use case specifications could be transformed into a program written in an executable specification language, which can be used to automatically generate test data and test sequences with a test generator tool.

2.3. Requirements based testing for software product lines

McGregor [15] introduced a process and requirements-based test models for software testing of software product lines. Kamsties et al [16], Bertolino et al [17] and Nebut et al [18] expanded on this work by describing and applying three use-case based testing techniques. Kamsties et al created activity diagrams from SPL use cases, and then applied parameterization, fragmentation and segmentation techniques to these activity diagrams in order to create reusable test cases. Reuys et al [19] further developed the creation and customization of use case activity diagrams [16] with the ScenTED technique (Scenario-based Test case Derivation) and applied it to an industrial case study at Siemens.

Bertolino et al [19] used tags to extend use cases into product line use cases (PLUCs), and then applied category partition testing strategy on these product line use cases. Nebut et al [20] created customizable use case contracts for an SPL, customized these contracts for an application derived from a SPL, and then applied predicate-based testing coverage criteria on these contracts.

Geppert et al [20] investigated the problem of test reuse in a software product line. System tests were parameterized to make them reusable, and a decision model was used to guide test selection and customization for an application of the SPL.

This research builds on previous work. An initial version of a model-based testing method for software product lines was introduced in [21] and applied to a hotel management system case study. This model-based testing method was incorporated into a standardized software testing process in [22]. However, the initial version of the method did not scale up to software product lines with a large amount of feature combinations. This paper introduces a second version of the model-based testing method which uses customizable test models to address the scalability problem.

3. Requirements-based Test Models for an SPL

In this paper the PLUS (Product Line UML based Software engineering) feature and use case models [5] are used to design the test models for a software product line. In PLUS, the features in the feature model are mapped to the use cases and use case variation points using a feature to use case relationship table. The customizable test models are created during software product line engineering in three phases, which are described in more detail below:

- Create activity diagrams from the use cases,
- Create decision tables from the activity diagrams, and
- Create test templates from the decision tables.

3.1. Create activity diagrams from use cases

The first phase involves creating activity diagrams from the use cases and use case descriptions. This concept has been used in single system testing to help formalize the use case model [13]. Hierarchical activity diagrams have been adapted for an SPL in the ScenTED approach [19]. In ScenTED, decision nodes and branches associated with a variation point are stereotyped as variant. A modified branch coverage criterion is applied during software product line engineering. The variability is temporarily abstracted away from the activity diagrams. Then, paths are traced from these diagrams that satisfy the branch coverage testing criterion.

The first phase in this paper also uses activity diagrams. However, it differs from the ScenTED approach because it explicitly considers the feature model by associating features with the variable elements in the activity diagrams [7]. This aids the management of variability as follows: (a) An explicit description of the relationship between a feature and its related activities aids traceability; (b) defining the activities associated with a feature in one place aids maintenance and reuse.

The steps of the first phase are as follows. At first, variability is ignored and an activity diagram is created for each use case. Each activity node corresponds to a step in the use case, and alternative paths are added using decision nodes and execution conditions. An *execution condition* is a controlled variable that affects the control flow of a path in an activity diagram during the actual execution of the use case.

Next, the activity diagrams are adapted for an SPL by considering the impact of each feature. A feature is associated with one or more use cases or a use case variation point using the feature to use case relationship table [5]. If a feature is associated with one or more use cases, a feature condition is added to the start node of the activity diagram of the use cases associated with the feature. A feature condition is a Boolean variable used to customize the control flow of an activity diagram. It is set to True when the corresponding feature is selected for an application of the SPL, and False otherwise.

A feature associated with a variation point is parameterized and tagged as <generic>. A generic activity node is a placeholder, or location of insertion for the activity nodes and / or conditions associated with a variation point value. The activities associated with a variation point value are described separately in a feature to variation point relationship table.

An activity diagram with generic activity nodes is shown in Fig. 4 for the “Enter toll road” use case of the automated toll system case study. The “Enter toll road” use case describes how a vehicle enters a toll road through a transponder-enabled or ticket-issuing entry toll booth in Fig. 3. These two types of tollbooth configurations are represented by the features “Transponder-enabled Booth” and “Ticket Booth” in the automated toll system feature model in Fig. 2.

The “Enter toll road” use case has two variation points: vpBarrier, which corresponds to a barrier feature which can be raised or lowered in low speed entry booths, and vpLight, which corresponds to a traffic light feature which turns amber, or green depending on whether the vehicle has been authorized to enter the toll road.

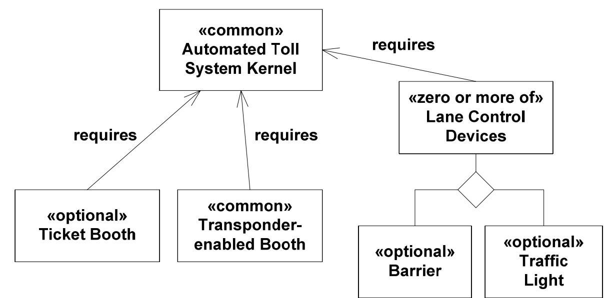


Figure 2 Automated toll system feature model

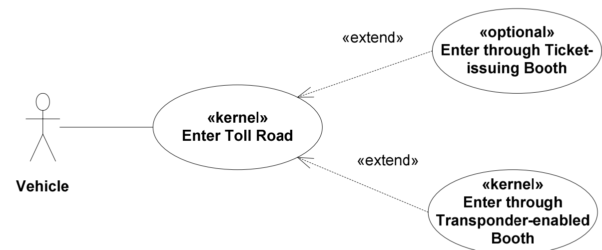


Figure 3 Enter toll road use case

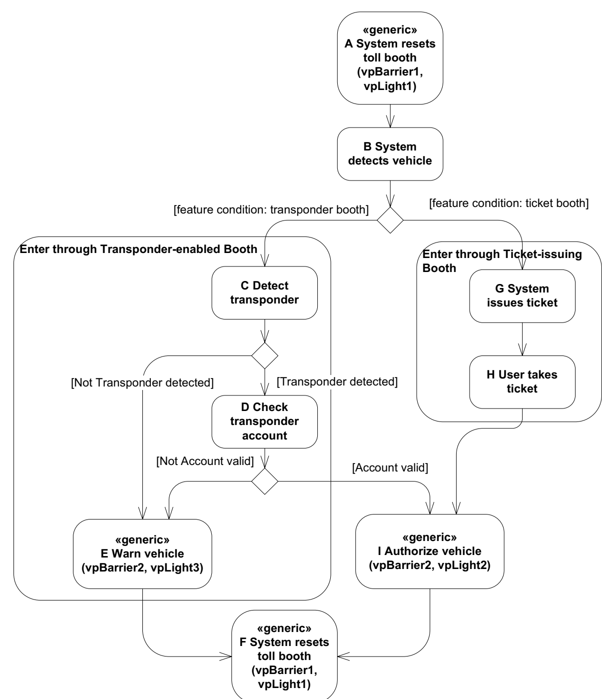


Figure 4 Enter toll road use case activity diagram

An example of a feature to variation point relationship table is shown in Table 1 for the automated toll system case study. The barrier feature is associated with a vpBarrier variation point, which is present in the Enter toll road and Exit toll road use cases. This variation point is associated to two

parameters, which correspond to “Lower barrier” and “Raise barrier” variant activities.

Table 1 Feature to variation point relationship table

Feature	Variation point	Parameter	Parameter value
Barrier	vpBarrier	vpBarrier1	Lower barrier
		vpBarrier2	Raise barrier
Traffic Light	vpLight	vpLight1	Turn light red
		vpLight2	Turn light green
		vpLight3	Turn light amber

3.2. Create decision tables from activity diagrams

The second phase involves creating decision tables from the activity diagrams. The decision tables are based on the extended use case pattern described in [12], with some changes incorporating the separation of concerns ideas described in the previous section.

The steps of the second phase are as follows: A decision table is created for each use case activity diagram. Each execution condition in the activity diagram is added to a column in the execution conditions section of the table. If the use case activity diagram contains feature conditions, each feature condition is added to a column in the feature conditions section of the table. Post conditions are deduced by the tester from the use case description, and refer to observable system states. Each post condition is added to a column in the post conditions section of the table.

Each activity node in the activity diagram is mapped to a sequence of test steps. Generic activity nodes are mapped to parameterized test step sequences. The parameters in a parameterized test step sequence indicate a location where a variable test step may be inserted or replaced.

Paths are traced from the use case activity diagram until all edges are covered. A path is a sequence of activity nodes. If the path traverses a generic activity node then the path is tagged as <generic>. Each path is added to a row in the decision table. If the path traverses a condition, the value of the condition (True or False) is entered in the row, column intersection of the table.

A decision table for the “Enter toll road” use case is shown in Table 2, for four paths traced from the activity diagram in Fig. 4. Path 1 (B-C-D-I-F) is the main scenario of the Enter through transponder-enabled toll booth use case. Path 1 has the feature condition fc: transponder booth, and the execution conditions: transponder detected and account valid set to true.

Table 2 Decision table for Enter toll road use case

Path	Fc: Transp booth	Fc: Ticket booth	Ec: Transp detected	Ec: Acct valid	Post: Valid entry
1	T	-	T	T	T
2	T	-	F	-	F
3	T	-	T	F	F
4	-	T	-	--	T

3.3. Create test templates from decision tables

The third phase involves creating test templates from the decision tables and paths traced from the activity diagram. Each path (row) in the decision table is mapped to one test template. The execution conditions, feature conditions, path traces, and post conditions become the initial conditions, test steps, and post conditions of the test template.

If a path (row) in the decision table is tagged as generic, a customizable (generic) test template is created for that row. The generic test template contains a parameterized test step sequence. Table 3 shows an example of a generic test template for Path 1, Table 2.

Table 3 Generic test template for main scenario of Enter toll road use case

Name	1. Enter Toll Road use case_Transponder - Main <generic>
Feature conditions	TransponderBooth = True TicketBooth = True or False
Execution conditions	TransponderDetected = True AccountValid = True
Test steps	
B	<Input> System detects customer approach
C	<Input> System detects transponder
	<Input> System writes entry data to transponder (in location, in time)
D	System checks transponder account
I	<Output> System authorizes vehicle to pass
	$vpBarrier2
	$vpLight2
F	<Output> System detects that vehicle has passed and resets booth
	$vpBarrier1

	$vpLight1
Postconditions	VehicleEnteredRoad = True VehicleAuthorized = True

The generic activity nodes I and F of Fig. 4 are mapped to parameterized test sequences in Table 3. A $ in front of a test step indicates that this step will be replaced by the value of a variation point parameter in Table 1, if the feature associated with the variation point is selected for an application of the SPL. Otherwise, this test step is ignored.

3.4. Customization of Test Models

The activity diagram, decision tables and test templates are customized by selecting a set of features for an application of the SPL. Test data is generated to satisfy the execution conditions of each test template. For example, suppose that the barrier and the transponder-enabled booth features are selected for an application. The "1. Enter Toll Road use case_Transponder - Main <generic>" test template would then be selected for the application, and the test steps corresponding to activity nodes I and F in that template would be customized as shown in Table 4.

Table 4 Section of customized test template

I	<Output> System authorizes vehicle to pass
	<Output> System raises barrier
F	<Output> System detects that vehicle has passed and resets booth
	<Output> System lowers barrier

Test data would then be generated to satisfy the execution conditions of the test template.

4. Application of Test Design Method to an Automated Toll System Case Study

Five graduate-level participants applied the three phases of this method to the feature and use case models of an automated toll system case study. The purpose of the study was to observe whether the method was usable and practical: Can a graduate student with some background in SPL modeling methods correctly apply the test design method on a realistic problem within a reasonable amount of time?

Each participant had created requirements models for an automated highway toll system software product line as part of a group project. The use case models consisted of between eight and fourteen use cases, and the feature models consisted of between twenty and twenty-four optional and alternative features.

In the first phase, each participant was given detailed instructions describing the method to create activity diagrams. Each participant was then asked to create activity diagrams for all use cases in the use case model. In the second and third phases, the participant was given detailed instructions describing the method to create decision tables and test templates. Each participant was asked to create a decision table and test templates for the "Enter toll road" use case as well as its included and extension use cases.

Each phase was scheduled to take about two weeks. During that time, participants were encouraged to ask questions and to get help correcting the models. A researcher observed which instructions caused the most misunderstandings by responding to questions and evaluating the correctness of the test models.

The participants were able to create the test models within the allotted time. Most participants were able to correct their models after the researcher pointed out misunderstandings (ignoring or incorrectly interpreting an instruction). Most misunderstandings were on how to describe and map variation points to the test models. This mapping was done manually in all phases of the method.

5. Conclusions

This paper has described a method for developing customizable test models from the requirements models of an SPL, and the application of the first three phases of that method (creating activity diagrams from use cases, creating decision tables from activity diagrams, and creating test templates from decision tables) to an automated highway toll system case study. The fourth, fifth, and sixth phases are the customization of the test models for an SPL-based testing strategy, the generation of test data for the customized test templates, and the testing of the applications derived from the SPL, as described in previous papers [21, 22]. We plan to continue our research into customizable requirements-based test models of SPLs by investigating how separation of concerns could assist in the process.

6. Acknowledgements

Many thanks to Jung-woo Peter An, Dwight Donaldson, Hugo Kang, Frederic Kneisel and Chris Magrin for creating test models for the automated toll system case study using the second version of the model-based testing method, and to Ahmed Elkhodary

for building a prototype to support the initial version of this method.

7. References

[1] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," presented at Proceedings of the 3rd International Conference on Software Engineering, Atlanta, Georgia, United States, 1978.

[2] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A feature-oriented reuse method with domain-specific reference architectures," *Annals of Software Engineering*, vol. 5, pp. 143-168, 1998.

[3] D. M. Weiss and C. T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*. Reading, MA: Addison-Wesley, 1999.

[4] P. Clements and L. Northrop, *Software Product Lines Practices and Patterns*. Boston, MA: Addison-Wesley, 2002.

[5] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*: Addison-Wesley Object Technology Series, 2005.

[6] H. Gomaa and D. L. Webber, "Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model," presented at Hawaii International Conference on System Sciences, Big Island, Hawaii, 2004.

[7] M. Saleh and H. Gomaa, "Separation of Concerns in Software Product Line Engineering," presented at Workshop on the Modeling and Analysis of Concerns in Software Product Line Engineering, St. Louis, Missouri, 2005.

[8] K. Kang, "Feature Oriented Domain Analysis," Software Engineering Institute, Pittsburg, PA CMU/SEI-90-TR-21, 1990.

[9] I. Jacobson, M. Christerson, P. Jonson, and G. Overgaard, *Object-oriented Software Engineering: a Use Case Driven Approach*. Reading, MA: Addison-Wesley, 1992.

[10] A. v. Mayrhauser, R. T. Mraz and P. Ocken, "On Domain Models for System Testing," presented at Proceedings of the Fourth International Conference on Software Reuse, Orlando, Florida, 1996.

[11] R. M. Poston, "Automated Testing from Object Models," in *Automating Specification-Based Software Testing*, R. Poston, Ed. Los Alamitos, CA: IEEE Computer Society Press, 1996, pp. 24-35.

[12] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Reading, MA: Addison-Wesley, 2002.

[13] L. C. Briand and Y. Labiche, "A UML-Based Approach to System Testing," presented at Proc. 4th International Conference on the Unified Modeling Language (UML), Toronto (Ontario, Canada), 2001.

[14] Wolfgang Grieskamp, Nikolai Tillmann, and M. Veanes, "Instrumenting scenarios in a model-driven development environment," *Information and Software Technology*, vol. 46, pp. 1027-1036, 2004.

[15] J. D. McGregor, "Testing a Software Product Line," SEI CMU/SEI-2001-TR-022, 2001.

[16] E. Kamsties, K. Pohl, S. Reis, and A. Reuys, "Testing Variabilities in Use Case Models," presented at Software Product-Family Engineering: 5th International Workshop, Siena, Italy, 2003.

[17] A. Bertolino and S. Gnesi, "PLUTO: A Test Methodology for Product Families," presented at Software Product-Family Engineering: 5th International Workshop, Siena, Italy, 2003.

[18] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel, "A Requirement-Based Approach to Test Product Families," presented at Software Product-Family Engineering: 5th International Workshop, Siena, Italy, 2003.

[19] A. Reuys, Erik Kamsties, K. Pohl, and S. Reis, "Model-based Testing of Software Product Families," *Lecture Notes in Computer Science*, vol. 3520, pp. 519-534, 2005.

[20] B. Geppert, J. Li, F. Robler, and D. M. Weiss, "Towards Generating Acceptance Tests for Product Lines," presented at 8th International Conference on Software Reuse, Madrid, Spain, 2004.

[21] E. M. Olimpiew and H. Gomaa, "Model-based Testing For Applications Derived from Software Product Lines," presented at Advances in Model-based Testing, St. Louis, Missouri, 2005.

[22] E. M. Olimpiew and H. Gomaa, "Executing Reusable System Tests for Applications Derived from Software Product Lines," presented at Software Product Line Testing Workshop (SPLiT '05), Rennes, France, 2005.

A Verification Approach for Crosscutting Features Based on Extension Join Points

Roberta Coelho[1], Vander Alves[2], Uirá Kulesza[1], Alberto Costa Neto[2], Alessandro Garcia[3], Arndt von Staa[1], Carlos Lucena[1], Paulo Borba[2]
[1]*Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)*
{roberta, uira,lucena,arndt}@inf.puc-rio.br
[2]*Centro de Informática – Universidade Federal de Pernambuco*
{vra, acn, phmb}@cin.ufpe.br
[3]*Lancaster University, Computing Department, Lancaster - United Kingdom*
garciaa@comp.lancs.ac.uk

Abstract

Recently, one arguing question in the context of product line development is how to improve the modularization and composition of crosscutting features. However, little attention has been paid to the closely related issue of testing the crosscutting features. This paper proposes a verification approach for the crosscutting features of a product line based on the use of a previously proposed concept called Extension Join Points.

1. Introduction

Framework technology has been widely used in the development of software product lines (PL) as a way of enabling systematic reuse-in-the-large. OO frameworks allow feature[1] modularization and composition, and offer extension options to target applications. Besides their advantages, some researchers [6, 20, 25] have recently described the inadequacy of OO mechanisms to address the modularization and composition of many framework features, such as, optional, alternative crosscutting features.

Crosscutting features represent concerns that are not well modularized in OO implementation. They are often spread over several modules of a software system and tangled with other features' implementation. Examples of such features are: security and transaction management. Hence, it is difficult to write a unit test for such features since there is no specific unit to be tested [8,17].

Aspect-oriented software development (AOSD) [11, 12] has emerged as a technology which aims at improving the modularization of crosscutting concerns. Recent work [2, 14, 15, 22] have been exploring the use of aspects to improve the modularization of crosscutting features in product lines.

While AOSD provides an effective way for modularizing crosscutting concerns and consequently providing a "unit" upon which a unit test can be defined, it brings new challenges to software testing. The new programming constructs provided by aspect-oriented languages are sources for new types of programming faults. Alexander et al [1] defined an initial candidate fault model for AOPs with new classes of AOP-specific faults, in addition to faults that can exist in object-oriented systems such as Java.

In a previous work [15], we have presented an approach to systematize the extension of OO frameworks by means of aspects. Aspects are used to modularize optional, alternative and integration crosscutting features encountered in the implementation of OO frameworks. According to this approach, the aspects introduce crosscutting features in the framework core by means of Extension Join Points (EJPs) [15].

This work proposes a verification approach for crosscutting features, complementary to the framework development approach proposed in [15]. A verification approach comprises techniques that aim at removing faults during the development phase [5]. Such techniques can be classified, according to whether they involve executing the system or not, as dynamic verification techniques (i.e testing) and static verification techniques (i.e code inspection), respectively [5]. The verification approach proposed in

this paper is structure in five steps, which comprises dynamic and static verification techniques.

The remainder of this paper is organized as follows. Section 2 presents background by showing the basic concepts of AOSD and revisiting some research work on product line testing. Section 3 discusses briefly our framework development approach based on AOP and extension join points (EJPs). Subsequently, Section 4 illustrates a case study in which EJPs were implemented using AspectJ. Section 5 presents our verification approach for crosscutting features that relies on the use of EJPs. Related works are presented in Section 6. Finally, Section 7 summarizes our contributions and provides directions for future work.

2. Background

This section briefly revisits the basic concepts of AOSD and research work on product line testing methodologies.

2.1 Aspect Oriented Software Development

Aspect-oriented software development (AOSD) [12,13] supports the modularization of crosscutting concerns by providing abstractions, called aspects, to extract these concerns and later compose them back when producing the overall system. Such abstraction is called aspect.

AspectJ [4] is an implementation of AOP for the Java programming language. The aspect abstraction in AspectJ is composed of: inter-type declarations, pointcuts and advices. Inter-type declarations specify new attributes or methods to be introduced in specific classes. Joinpoints are well-defined locations within the base code where a concern can crosscut the application. Examples of join points are method calls and method executions. AspectJ pointcuts are expressions that match collections of join points. Finally, advices are a special method-like construction of aspects which are used to attach new crosscutting behaviors along the aspect pointcuts.

2.2. Software Product Line Testing

According to product line testing methodologies proposed so far [21, 23, 13], product line testing should be done at three main levels: at unit (or component) level, at integration (or feature) level, and at system level. Features are a suitable integration criteria since the instances of a product line often differ basically in the availability of product line features.

Current approaches propose general guidelines to structure the whole process of product line testing. For instance, they state that the tests defined for the core assets, at any one of these levels, should be treated as core product line assets and managed consistently; and that such tests may be fully or partially reused across versions of the product line and at specific products. However, there is still a lack of techniques to help developers in the low level design of tests at each level.

In this work, we propose a feature-level testing technique and complementary manual and automatic inspections techniques for crosscutting features. Such techniques rely on the use of the extension join points detailed in the next section, and are structured in a verification approach detailed in Section 5.

3. A Framework Extension Approach

In a previous work [15], we have proposed a systematic approach for framework extension by means of aspects. In our approach, we defined the concept of Extension Join Point (EJP). The EJP consists on a unified way of designing and documenting existing crosscutting extension points. It provides new means for extending framework core functionality, introducing optional and alternative crosscutting features. EJP represents a new kind of framework hotspot, different from the well-known object-oriented extension points. Figure 1 illustrates these two kinds of framework hotspots.

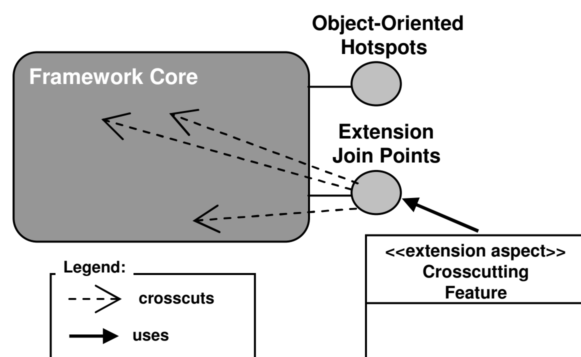


Figure 1: OO hotspots x Extension Join Points.

The object-oriented hotspots are usually represented as abstract classes that should be extended, or interfaces that should be implemented during framework instantiation. On the other hand, the EJPs represent framework hotspots that will be used by aspects that will implement a crosscutting feature in the framework. We call extension aspects, the aspects that address the implementation of a crosscutting feature, as shown in Figure 1.

The EJPs are inspired by a recent study performed by Sullivan et al [26] which proposes the use of an interface between the base code and the aspects, called crosscutting interfaces (XPIs). EJP extends the concept of XPI to the context of framework development. The EJP comprises different attributes from the ones proposed by Sullivan in the XPI specification. It also defines a set of internal and extension contracts which regulates the relationships between the framework and the extension aspects (Figure 1).

Each EJP is composed of the following elements: (i) a name that is represented by the aspect's name; (ii) a scope which defines all the framework elements that are "encapsulated" by the EJP; (iii) a set of crosscutting extension points, which specifies the framework join points that represent relevant events or transition states occurring during the execution of the framework functionalities; and (iv) a set of internal and extension contracts.

The framework internal contracts define constraints whose purpose is to assure that framework refactorings and evolution do not affect the functionality of its extension aspects. They are classified in the following categories:

- *Structural*: which aims to guarantee the framework implements specific interfaces defined by the EJPs; and
- *Behavioral*: which assures the framework EJPs comprises all and only the framework events (or states) that the EJP is intended to expose.

The framework extension contracts are used to assure that each extension aspect respects constraints and invariants of the framework. The following categories were defined:

- *Structural*: these contracts assure that aspects only extend the framework join points exposed by the EJPs, and specify the framework classes methods that can be invoked by the extension aspects; and
- *Behavioral*: define specific pre- and pos-conditions that must be preserved before and after the execution of extension aspect advices.

Tables 1 and 2 show different AspectJ mechanisms that we have used to implement these contracts.

Contract Type	AspectJ Implementation
Structural	Specification of interfaces that must be implemented by framework classes. The obligation to implement these interfaces is assigned by the EJPs using the `declare parents` inter-type construction of AspectJ. The interfaces are also declared inside the aspects that represent the EJPs.
Behavioral	Implementation of enforcement policies guaranteeing that the extension join points are called only and in all appropriate places inside the framework. This contract can be specified using `declare warning` and `declare error` AspectJ statements.

Table 1. Framework Internal Contracts.

Contract Type	AspectJ Implementation
Structural	It is possible to define AspectJ contract to restrict the framework classes' methods that can be accessed inside the extension aspects. There are two different ways to specify it: (i) using `declare warning` and `declare error` AspectJ statements, which allow the static verification of policies; and (ii) by defining advices which intercept every advice execution that realizes calls to the framework classes' methods. The `adviceexecution()` pointcut designator is used to intercept the advices execution.
Behavioral	This contract defines pre- and pos-conditions that must be assured before and after the advice execution. These contracts are also defined using `adviceexecution()` pointcut designator to intercept the advices execution.

Table 2. Framework Extension Contracts

Due to a current limitation of the AspectJ it is not possible to automatically assure that aspects only extend the framework join points exposed by the EJPs. Hence, to assure it, the developers must follow the programming practice of using only pointcuts specified in the EJP[1], which will be checked during manual inspections.

4. Case Study: Game SPL

In this case study, we used EJPs to support the implementation and test of the crosscutting features of a software product line (SPL) for games in cell phones [2]. Due to space limitation, this section briefly describes the implementation of EJPs. For a complete description of the EJPs implementation and extensions aspects please refer to [15].

The overall structure and behavior of this product line are defined by a framework known in this domain as the *game engine*. Essentially, the *game engine* consists of a state machine whose state changes according to the elapsed time and user input - through

[1] Larochelle et al [16] have proposed a mechanism, called join point encapsulation, which aims to prevent selected join points from being modified by aspects. Since this mechanism was implemented only to previous versions of AspectJ, we did not have the chance to experiment it in our case studies.

the device keypad. The state changes affect the state of various *images* on the game screen, which, as a consequence, should be re-drawn.

An important variability issue in this SPL is flipping images of game object images (such as enemies, dragons, and weapons) on the game screen. How an image can be flipped varies according to the device in which the game is executing. Some devices have built-in flip API and others have not. For those devices that do not have a built-in API, specific flipping algorithms had to be defined.

The flipping feature is crosscutting since it depends on image drawing events which are spread over a set of framework modules. We implemented this crosscutting feature according to the framework development approach proposed in [15] and detailed in Section 3.

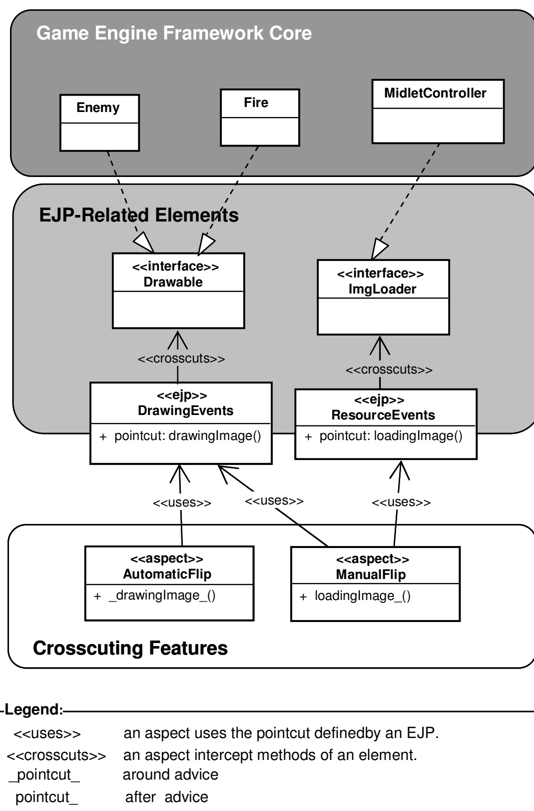


Figure 2: Crosscutting features implementation according an EJP-based approach.

According to this approach, before implementing the crosscutting features a set of EJPs are defined to the framework. The EJPs comprises a set of framework events that will be of interest when implementing the crosscutting features. The crosscutting feature is then represented by an aspect that reuses (and may specialize) a set of events exposed by the EJPs.

Figure 2 illustrates a subset of the EJPs and crosscutting features defined for the *game engine* framework. The DrawingEvents EJP comprises all the drawing events defined across the framework, and the ResourceEvents EJP comprises the image loading events – the image is loaded at the beginning of the game. The `ManualFlip and the AutomaticFlip` aspects implement the crosscutting features of automatically and manually flipping an image, for devices that have and have not built-in flip API, respectively. Those aspects depend on the `DrawingEvents` and `ResourceEvents` EJPs, which exposes the image drawing and image loading events.

According to the EJP-based development approach, besides exposing a set of framework events, the EJPs also define the set of interfaces that should be implemented by framework elements. Such interfaces are responsible for firing the events exposed by EJPs. As a consequence, every framework component whose events should be intercepted by crosscutting features should implement one (ore more) of these interfaces. Hence, the EJPs do not directly access the framework elements, but the interface that these elements implement. Such architectural decision improves the testability of the crosscutting features since it supports the unit test of such features through the definition of mock objects as detailed at step 3 of next section.

Figure 3 illustrates the partial code of the *Drawable* interface and the DrawingEvents EJP. In lines 8-9, the DrawingEvents EJP defines the set of elements inside the framework that should implement the *Drawable* interface - via inter-type declaration of AspectJ [15].

```
1. public interface Drawable {
2.    public int getWidth();
3.    public void drawImg(Graphics g, int ofsX);
4.
5. }


6.public abstract aspect DrawingEvents {
7.
8. declare parents: Enemy implements Drawable;
9. declare parents: Fire implements Drawable;
10.
11./**
12.* The purpose of the drawingImage PCD is to
13.* expose all calls to methods that draw images
14.* of game itens that move around the game screen
15.*/
16. public pointcut drawingImage (Drawable d, int
17.  offSetX, Graphics g) :
18.  execution (public void Drawable.drawImg(
19.     Graphics,int)) && this (d) &&
20.     args(g, offSetX);
21.   …
   }
```

Figure 3: Partial code of the DrawingEvents EJP and *Drawable* interface.

The game engine framework detailed in this section is used in the next section in order to exemplify the steps of our verification approach for crosscutting features.

5. The Approach

A crosscutting feature can be broken down in two major parts: what the feature does and where it is applied. The faults discovered in a crosscutting feature can arise from one of the following sources[2]: (i) bugs in the crosscutting feature logic; (ii) inaccurate pointcut designator which intercepts a wrong set of join points; (iii) an emergent property created by interactions between the crosscutting features and the base code; and (iv) faults in the core components themselves.

This section presents a verification approach for the crosscutting features of a product line composed by five steps, as illustrated in Figure 4.

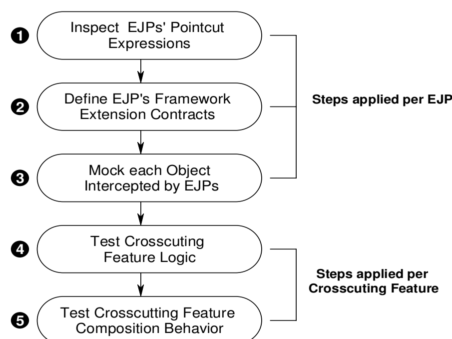


Figure 4: The crosscutting feature verification approach steps.

One of the hardest issues about assuring the quality of a crosscutting feature is to check whether the places where the crosscutting features apply are correct. Checking all the places affected by a crosscutting feature can be a daunting task, specially if we need to check its negative scope (i.e check whether there is any accidental inclusion of a point to be intercepted). To help developers in finding faults on pointcuts, we supplemented our approach with steps of manual and automatic inspections based on the use of EJPs (Steps 1 and 2)

This approach aims at finding faults of types (i) (ii) and (iii) defined previously. Steps 1 and 2 aim at finding faults of type (ii). Step 3 implements a test infrastructure to be used in Step 4 which detects faults of type (i). Finally, Step 5 looks for bugs of kind (iii). In order to detect faults of type (iv) traditional OO testing techniques can be applied to the PL context.

Step 1: Inspect EJPs' Pointcut Expressions

For each EJP defined for the system, the developer should check whether the EJP pointcuts intercept the correct places and only them. In order to accomplish this, the developer can use the crosscutting visualization tools available at AJDT (the most used AOP IDE). It can also use string matching algorithms to calculate the distance between the pointcuts and the join points in the system and verify whether there is any accidental inclusion of a point to be intercepted [17, 3].

This step is very time consuming since there is not a fully automatic way to detect this kind of faults [17]. According to our strategy, however, only the EJPs have to be inspected and the crosscutting features only need to reuse them. Figure 5 illustrates the code of `AutomaticFlip` aspect that reuses the pointcut descriptors defined in the DrawingEvents EJP (line 5)[3].

```
1. public privileged aspect AutomaticFlip {
2.
3. void around (DrawingEvents.Drawable d,
4.      int offSetX, Graphics g):
5.     DrawingEvents.drawingImage(d, offSetX, g) {
6.
7.  //If the speed is negative this means that
8.  //the image must be re-drawn in the opposite
9.  //directions - it must be flipped.
10.  if (d.getXspeed() < 0) {
11.     //code to flip the image using device API
12.     ...
13.  }
14. //Otherwise,it is re-drawn without flipping
15.  else {
16.       proceed(d, offSetX, g);
17.  }
18.}
```

Figure 5: Partial code of a crosscutting feature.

If the EJPs were not used to mediate the relationship between the core and the extension aspects, every extension aspect would have to be inspected thus compromising the scalability of the approach. The techniques presented at this step are not sufficient to verify pointcut expressions that involve complex dynamic conditions, which depend on the execution stack. Such expressions can only be verified after weaving. Steps 4 and 5, detailed next, can help developers in detecting such kind of faults.

Step 2: Define EJPs' Framework Extension Contracts

[2] There are faults that emerge from a property created when more than one aspect affects the same element in the base code. However, this kind of fault can not happen in a feature unit test scenario, in which one feature is tested at a time.

[3] The proceed() command that appears in line 17 is an AspectJ specific command that executes the intercepted method.

Since we have inspected the set of pointcut expressions defined by the EJPs, now we need to assure that each extension aspect will respect the constraints and invariants of the framework. In order to do it the developer should define a set of EJP's framework extension contracts.

As detailed in Section 3, these contracts can be checked during manual inspections, verified at compilation time or runtime. The EJP's contracts evaluated in runtime act as test oracles, since they will alert the developer when a contract is violated during feature-test executions.

Figure 6 illustrates a contract associated to the `DrawingEvents` EJP shown in Figure 3. This contract states that the crosscutting extension features, represented in this case study by `ManualFlip` and `AutomaticFlip` aspects, are not allowed to access framework elements besides *`Drawable`*, *ImgLoader*, Graphics types - which are EJP-related interfaces and is a parameter of an intercepted method, respectively (Figure 3, lines 16-20).

```
1. public aspect DrawingExternalContractChecker {
2. // Framework Scope - Calls Not Allowed
3. public pointcut FWScopeNotAllowed():
4. call(* !(Drawable||Graphics||ImgLoader).*(..))
5.    && call (* gameenginecore.*.*(..));
6.
7.  public pointcut aspectsPackages():
8.                 within(extensionaspects..*);
9.
10. declare warning:
11.     FWScopeNotAllowed() && aspectsPackages():
12.     "Extension aspects are accessing \
13.      internal framework details";
14. }
```

Figure 6: Extension contract checked at compilation time.

The `FWScopeNotAllowed()` pointcut (line 3) denotes calls to framework internal types that should not be visible to the extension aspects. The `aspectsPackages()` denotes calls within such aspects. Both `FWScopeNotAllowed()` and `aspectsPackages()` pointcuts are composed in a declare warning AspectJ command (lines 10-14). This command warns at compilation time if any extension aspect tries to use a framework class or interface different from those defined in the `FWScopeNotAllowed()` pointcut.

The PL developers can define extension contracts as complex as needed. We are currently investigating simpler ways of defining interaction rules, thereby not requiring the developer to learn too complex AspectJ constructs.

Step 3: Mock each object intercepted by the EJPs

Mackinnon et al [19] proposed the Mock Object test design pattern [7], and since then, Mock Objects have been recognized as a useful approach to the unit test and design of object-oriented software. A Mock Object is a regular object that acts as a stub, but also includes assertions to instrument the interactions of the fake object with its neighbors.

The Mock Object allows the unit test of a component that depends on others which may not be implemented yet. This is exactly the scenario that we find when testing a crosscutting feature that affects hotspots that will only be implemented by PL products.

This third step states that the PL developer should define mock objects of the code intercepted by the EJPs. Both the real object and its mock version should implement the same interface. Since the EJP only refers to an object by its interface, it can remain ignorant of whether it is intercepting the real object or the mock object. The affected code can be a core asset or a framework object-oriented hotspot.

One mock object should be created for each interface intercepted by the EJP. In the case study we implemented mock objects for the *`Drawable`*, `and` *`ImgLoader`* interfaces. These mock objects can be very simple: classes which contain empty implementations of the methods specified by an interface. Or more complex components which can be automatically generated and also include assertions to improve tests diagnoses. Figure 7 illustrates a simple implementation of a mock object created in this case study [4].

```
1. public class MockDrawable implements Drawable {
2.  public int call_getWidth(){
3.        getWidth();
4.  }
5.  public void call_drawImg(Graphics g,int ofsX){
6.         drawImg();
7.  }
8.  public int getWidth(){}
9.  public void drawImg(Graphics g, int ofsX){}
10.}
```

Figure 7: Mock object code.

Step 4: Test the Crosscutting Feature Logic

The set of *Mock Objects* defined in the previous step is used at this step to enable the testing of crosscutting features logic. At this step the crosscutting feature is weaved with one (or more) mock objects and the methods of the resultant component (*weaved mock object*) are unit tested using an OO testing framework (such as JUnit).

---

[4] The MockDrawable class includes one extra method to each method specified by the Drawable interface. These methods were necessary due to specific characteristics of AspectJ language: these methods capture the crosscutting behavior included through advices associated with call pointcuts – for more details about call pointcuts the reader should refer to [4]. Only these extra methods are unit tested in Step 4, since the others are called by them.

Moreover, the Mock Object can also simulate some error conditions (i.e throw of an exception) on the base code [19]. As a consequence, it allows the developer to test the crosscutting features under abnormal conditions. Some crosscutting features, when exposed to abnormal conditions, execute statements that throw exceptions, and as a consequence might cause undesired modifications in the system control flow.

At this step, the developer can use well known OO testing criteria to test crosscutting features' logic – embedded in each method of the *weaved mock object*: statement coverage, branch coverage, condition coverage, and dataflow coverage [24].

Since the testing criterion specifies the conditions that must be covered during tests, helping the developer select the test cases and decide whether the software has been adequately tested, it would be very useful to use EJP-specific test criteria at this step. However, we are still investigating the definition of possible EJP-based criteria.

Step 5: Testing Crosscutting Feature Composition Behavior

The crosscutting feature composition behavior results from the interaction between the crosscutting feature and the base code (arises after weaving) [18]. This step aims at checking the behavior of functionalities affected by the crosscutting feature against their specification - as if the developer would do if the crosscutting feature code were scattered among affected features. Thus, the test should fail if the crosscutting feature misbehaves or does not apply at the specified points.

This kind of tests reveals faults that just occur when the features interact. However, it is difficult to diagnose a failure detected in such tests, since the cause can be in the crosscutting code, in the base code, or the crosscutting code not being applied in the appropriate place.

According to Colyer et al. [9] the crosscutting concerns should be classified as: orthogonal, altering, and stateful. Orthogonal aspects do not change control or data dependencies in the system (i.e logging). Altering aspects change control flow or data flow of a system. (i.e aspects using around advice). A stateful aspect has behavior that depends on an aspect attribute or introduced object attributes. When a non-orthogonal aspect is weaved with the base code, existing test suites may be missing in covering the feature resulting behavior. Thus, a new set of test cases needs to be defined to each effected feature in order to cover such behavior.

This process is costly; however, the test suites will be used during the tests of each PL product. We are currently investigating ways of reducing the test cases must be re-run when a crosscutting feature is added to the base code.

6. Related Work

Research on testing aspect-oriented programs [27, 28, 3] has been focused on code-based unit and integration testing, automated test case generation, and the definition of an AOSD fault model. Some of these works can be used in our testing approach. For instance, Xie et al have proposed a framework for generating test inputs for AspectJ programs [27], Step 4 could be extended in order to incorporate the test generation solution proposed by them. Zhou et al. [28] have proposed an algorithm based on control flow analysis for selecting relevant test cases, this technique could be applied on Step 4 and 5 in order to select the test cases to be executed.

7. Conclusions and Future Work

In this work, we have proposed a systematic approach for detecting faults in crosscutting features implemented by means of aspects. In particular, our approach is complementary to a framework development approach proposed previously, which addresses the modularization of optional, alternative, and integration framework crosscutting features by using AO techniques. Our verification approach is composed of five complementary steps (Section 5) ranging from aspect pointcuts inspections to the unit test of every crosscutting feature using mock objects. Moreover, a set of contracts can be defined to guarantee an adequate interaction between the framework core, the EJPs and the extensions aspects. Thus, different types of faults can be detected by using these different mechanisms of software testing.

As our approach is still under development, we intend to refine it by addressing the testing of different software product lines or software family architectures implemented using aspects. Also, the development of a testing tool supporting the generation of many elements (mocks, aspect unit testing) from the approach is under investigation.

8. References

[1] Alexander, R., Bieman, J. and Andrews., A. Towards the Systematic Testing of Aspect-Oriented Programs. Technical Report CS-4-105, Department of

Computer Science, Colorado State University, Fort Collins, Colorado, 2004.
[2] Alves, V., Matos, P., Cole, L., Borba, P., Ramalho, G., Extracting and Evolving Mobile Games Product Lines. Proceedings of SPLC'05, LNCS 3714, pp. 70-81, September 2005
[3] Anbalagan, P. and Xie. T., APTE: Automated Pointcut Testing for AspectJ Programs. Proc. of the 2nd Workshop on Testing Aspect-Oriented Programs (WTAOP 2006) (to appear).
[4] AspectJ Team. The AspectJ Programming Guide. http://eclipse.org/aspectj/.Batory, D., Cardone, R. and Smaragdakis , Y., Object-Oriented Frameworks and Product-Lines. 1st Software Product-Line Conference (SPLC),1999.
[5] Avizienis, A., Laprie, J-C, Randell, B., Landwehr, C., Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1), pp. 11-33, 2004.
[6] Batory, D., Cardone, R. and Smaragdakis , Y., Object-Oriented Frameworks and Product-Lines. 1st Software Product-Line Conference (SPLC),1999.
[7] Binder, R. Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Longman Publishing Co., Inc., 1999
[8] Ceccato, M., Tonella, P., e Ricca, F. Is aop code easier or harder to testthan OOP code? Workshop on Testing Aspect Oriented Programs, 2005.
[9] Colyer, A. Rashid, G. Blair, On the Separation of Concerns in Program Families. Technical Report COMP-001-2004,
[10] Czarnecki, K., Eisenecker, U., Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[11] Filman, R., Elrad, T., Clarke, S., Aksit, M., Aspect-Oriented Software Development. Addison-Wesley, 2005.
[12] Kiczales, G., et al. Aspect-Oriented Programming. Proc. of ECOOP'97, 1997.
[13] Knauber, P. , Product Line Testing and Product Line Development - Variations on a Common Theme, Proceedings of International Workshop on Software Product Line Testing (SPLiT 2005)
[14] Kulesza, U., Alves, V., Garcia, A., Lucena, C., Borba, P. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming, Proceedings of ICSR'2006, June 2006.
[15] Kulesza, U., Coelho, R., Alves, V., Garcia, A., Lucena, C., Borba, P, Implementing Framework Crosscutting Extensions with EJPs and AspectJ, Proc. of Brazilian Symposium on Software Engineering , 2006 (to appear).
[16] Larochelle, D. et al., Join Point Encapsulation, Proc. Workshop Software Eng. Properties of Languages for Aspect Technologies, 2003.
[17] Lesiecki, N., AOP@Work: Unit test your aspects, developerWorks, November 2005.
[18] Lopes, C. and Ngo, T.. Unit testing aspectual behavior. In Proc. Workshop on Testing Aspect-Oriented Programs, 2005.
[19] Mackinnon, T., Freeman, S., and Craig, P. EndoTesting: Unit Testing with Mock Objects. Proc. of XP'2000, 2000.
[20] Mattson, M., Bosch, J., Fayad, M., Framework Integration: Problems, Causes, Solutions. Communications of the ACM, 42(10):80–87, 1999.
[21] McGregor, J.D., Testing a Software Product Line, Technical Report, CMU/SEI-2001-TR022.
[22] Mortensen, M., Ghosh, S. Using Aspects with Object-Oriented Frameworks, Proc. of AOSD'2006, Industry Track, 2006.
[23] Muccini, H.,and Hoek, A., Towards Testing Product Line Architectures, ETAPS2003 workshop.
[24] Myers, G. J. The Art of Software Testing. Wiley, 2nd Ed. 2004.
[25] Riehle, D., Gross, T. "Role Model Based Framework Design and Integration". Proceedings of OOPSLA'1998, pp. 117-133.
[26] Sullivan, K. et al. Information Hiding Interfaces for Aspect-Oriented Design, Proceedings of ESEC/FSE´2005, pp.166-175, 2005.
[27] Xie, T., Zhao, J., Marinov, D., and Notkin, D. Automated test generation for AspectJ programs, AOSD 2005 Workshop on Testing Aspect-Oriented Programs, Chicago, 2005.
[28] Zhou, Y., Richardson, D., and Ziv, H. Towards a practical approach to test aspect-oriented software. In Proc Workshop on Testing Component-Based Systems (TECOS), 2004.

ODC Crystallizes Test Effectiveness

Ram Chillarege
Chillarege Inc.
August 2006
ram@chillarege.com, +1 (917) 790 9390, www.chillarege.com

Abstract -- Orthogonal Defect Classification (ODC) extracts two key measurements from defects: the defect Type and the defect Trigger. The cross product of the defect Type and the defect Trigger tell us what force (Trigger) made which fault (Type) surface. This provides the core to a measurement on the effectiveness of test. These insights and quantification allow us to assess the tools and methods used in test, and guide us in improving the test process.

Introduction

Software test productivity and its measurement remains blurred. While there are a number of measurements used in the practice, none are definitive or complete, and this remains an area of research. A few of the popular measures used are: number of test cases run, percent regression, coverage, defects found, software reliability, etc. While all these measures provide some measure of goodness, they all suffer from a common weakness, namely they are incidental to the fault detection process itself.

This paper delves into the activation process of faults and uses two attributes from orthogonal defect classification to create a measure of the effectiveness of test.

What's ODC?

Orthogonal Defect Classification (ODC) uses the defect data as an information source on the development process [1]. ODC extracts the semantic information from the defect and turns it into a measurement on the process. The different attributes in ODC measure different aspects of the development process. While each defect is categorized by a pre-defined attribute value set, the distributions of these attributes yield measures on the product and the process.

Figure 1. shows a subset of the major attributes in ODC Version 5.30 [5]. The defect Type measures the advancement of the product through the development process. The seven values for design and code are: Assignment, Checking, Timing, Relationship, Interface, Algorithm, and Function. Trigger measures the nature of testing being conducted and Table 1 lists the Triggers. These values have been carefully designed to provide a measurement on the process through their distribution as a function of time or process phase.

While the Type and Trigger can be associated with "cause", the attributes of Impact, Severity, Source and Age are associated with "effect". Collectively these attributes capture a significant amount of information from individual defects. When these data are aggregated over several

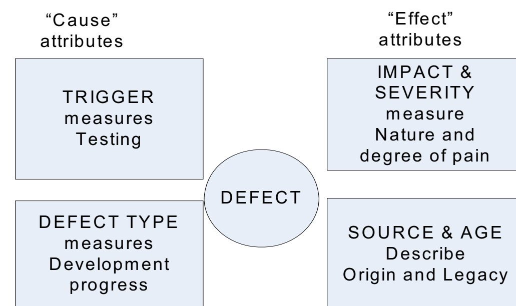


Figure 1: ODC attributes extracted from each defect collectively yield relationships and measures across key areas of the development and test processes.

defects they not only provide measurements of their respective areas, but also create a relationship between cause and effect that can provide a process diagnostic. One of the popular uses of ODC is to yield a 10x reduction in the cost and time of doing process level root cause analysis. [4]

Test Effectiveness

How does ODC measure test effectiveness? It does this through the cross product of the defect Type and Trigger distributions as shown in Table 1. To understand this, we need to recognize a couple aspects of what the Type and Trigger measure.

Defect Types tell us the kind of faults uncovered by test, and the Triggers tell us the kind of force necessary to surface the fault. Since the Type distribution tells us how the product is evolving in the development process, we can access whether we are finding the right faults at the right time. However, that is also partly dependent on the nature and quality of inspection and test. The Trigger distributions tell us what part of the testing process we are in, as opposed to what we think we are in. Just as the Type measures the progress of a product through the development process, the Triggers tell us of the advancement in the testing space. For instance, if the Trigger distributions show a preponderance of hand Triggers during the system test phase we know that testing has not really advanced to the system test phase. The case studies in [2] and [3] provide an insight in the use of Triggers for real world problem solving.

There is a relationship that can be evaluated between Types and Triggers which tells us of the likelihood of certain Triggers surfacing certain Types. Once we recognize the Type-Trigger cross product as a measure of the effectiveness of test, there are a family of measures that can be

computed for test effectiveness. We can develop measures either by phase, lifecycle, product, product line, release, or nature of legacy. When these data are further conditioned by an impact category, the measures of effectiveness can be extended to factor in risk and customer satisfaction. For instance, if there is a concern about reliability in a release, these data allow us to evaluate the most effective test methods that find faults that cause a reliability impact. Similarly, this notion can be used to access the efficacy of a new test method - such as, say, static analysis.

TABLE 1 - The cross product of defect Type and Trigger yields insight into the effectiveness of test and verification in activating faults. Triggers group into subsets of “head”, “hand”, and “foot” which are associated with inspection, function test, and environmental test respectively. These subset help measure phase containment. When further divided by specific Triggers they measure the effectiveness of the phases in surfacing specific types of faults.

Columns: Defect Type Rows: Triggers	Trigger Group	Assignment	Checking	Timing	Relationship	Interface	Algorithm	Function
Design Conformance	Head							
Logic Flow	Head							
Backward Compatibility	Head							
Lateral Compatibility	Head							
Concurrency	Head							
Side Effects	Head							
Rare Situations	Head							
Internal Documents	Head							
Language Dependency	Head							
Simple Path	Hand							
Complex Path	Hand							
Coverage	Hand							
Variation	Hand							
Sequencing	Hand							
Interaction	Hand							
Workload Stress	Foot							
Recovery, Exception	Foot							
Startup, Restart	Foot							
Hardware Configuration	Foot							
Software Configuration	Foot							
Blocked Tests	Foot							

Empirical data helps us build a model of the likelihood of a particular Trigger activating faults with specific defect Types. And given the design of the development and test process, we learn the likelihood of Triggers supplementing other Triggers. Thus, we go beyond phase containment into prediction and management.

References

[1] *“Orthogonal Defect Classification - A Concept for In-Process Measurements*”, Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, Man-Yuen Wong, IEEE Transactions on Software Engineering, Vol 18, No. 11, Nov 1992.

[2] *“Test and Development Process Retrospective - A case study using ODC Triggers”,* Ram Chillarege and Kothanda Ram Prasad, Proceedings International Conference on Dependable Systems and Networks, IEEE, 2002

[3] *“Improving Software Testing using ODC - Three Case Studies”,* M. Buther, M. Murino, T. Kratcher, IBM Systems Journal, Vol 41, No. 1, 2002.

[4] “*ODC: a 10x for Root Cause Analysis”*, Ram Chillarege, Prof. Ram2006 Symposium, Berkeley, CA 2006.

[5] odc.chillarege.com, www.research.ibm.com/softeng

Author

Ram Chillarege is the inventor of Orthogonal Defect Classification and his consulting firm specializes in software engineering optimization. He was executive vice president of Software at Opus360. He headed IBM’s center for software engineering, and was instrumental in creating IBM’s corporate wide software testing center of excellence. He is an IEEE Fellow, author of ~50 technical articles, and is currently the chair of the steering committee for the IEEE Symposium on Software Reliability Engineering.

Welcome to

SPLiT 2006

Workshop on Software Product Line Testing

Baltimore, MD, August 22, 2006

Chairs:

Peter Knauber, Mannheim University of Applied Sciences

Charlie Krueger, BigLever Software

Tim Trew, Philips Research

If you thought that SPLiT was Birgit's baby …

SPLiT — Why we do it

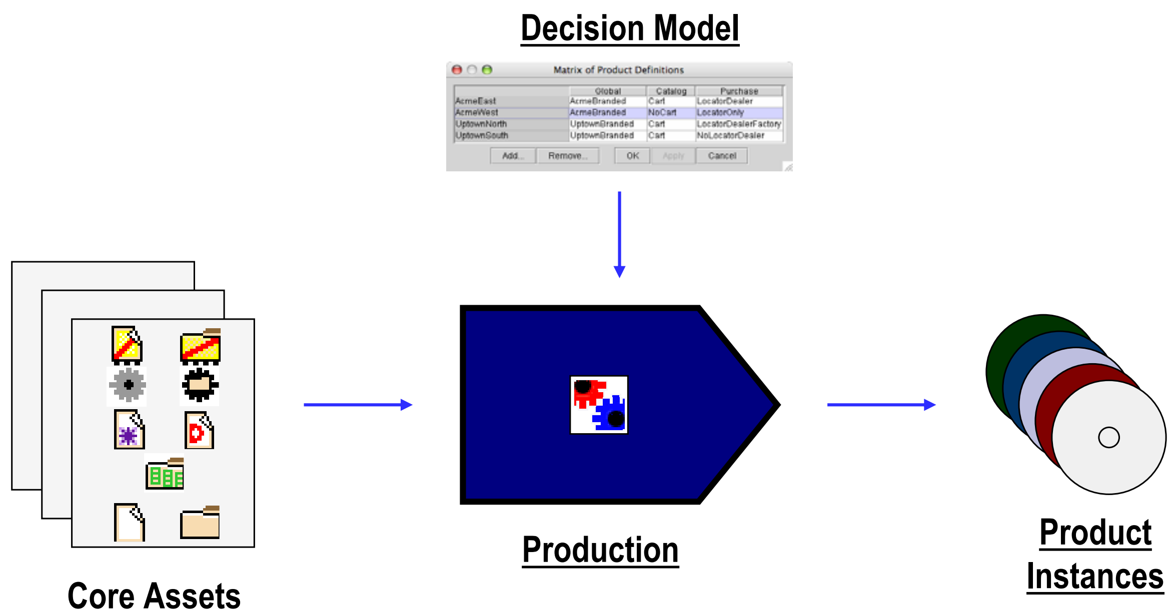
Goals

- 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
  - www.softwareproductlines.com/forums
    - Product Line Quality Assurance

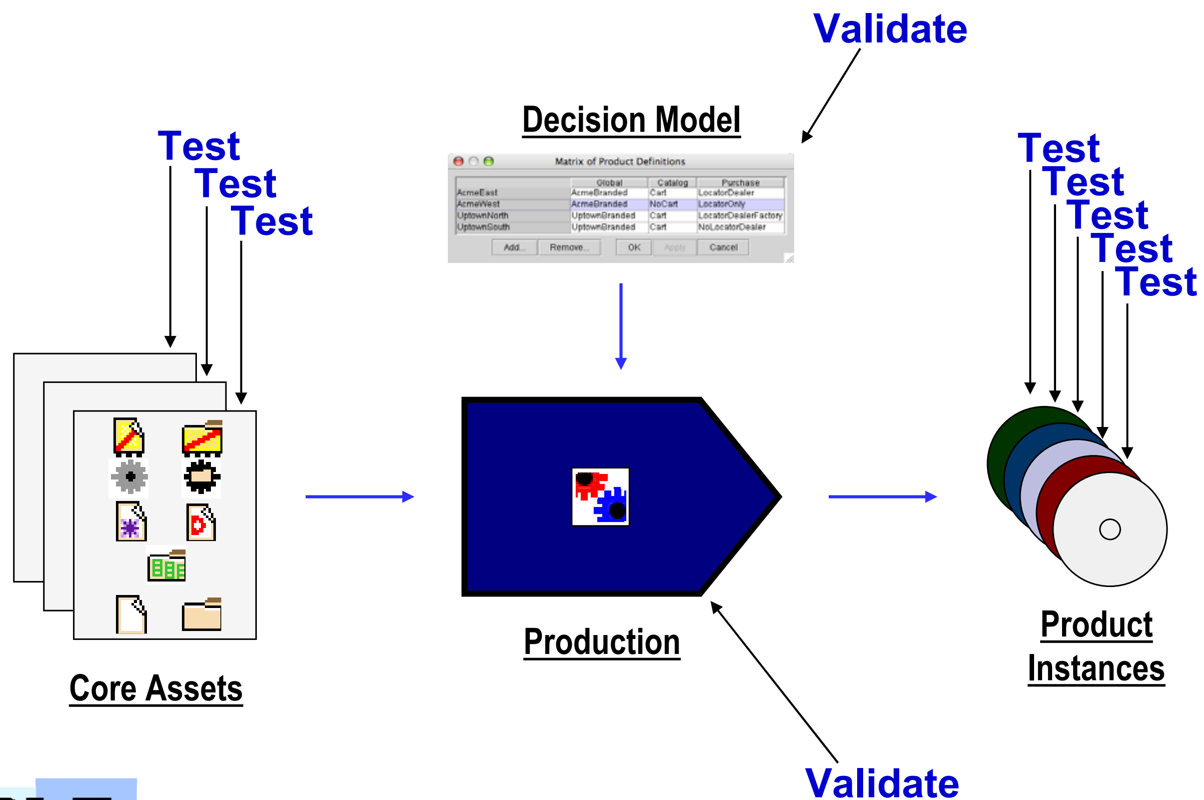
To Do

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

Product Line Engineering



Product Line Engineering and Testing



Lots and Lots of Open Issues …

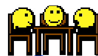
- How can we keep pace with development productivity gains?
  - How to leverage core asset testing to minimize product instance testing?
  - How can we leverage the commonality among the product instances to minimize redundant testing?
- How can we manage the complexity of the test space?
  - Additional 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?
  - Techniques for strategic reduction in test time, test cost, and test flaws
- How do choices during architecture and design impact product line testing?
- Definition and measurement of test coverage and test effectiveness in the context of software product lines
- How to make it work in real companies with real developers and testers

Agenda - Morning

- **08:30 - 09:00 Introduction**
- **09:00 - 10:00 Keynote**
  - *Georg Gruetter (Robert Bosch GmbH): Challenges for Testing in Software Product Lines*
- **Coffee (10:00–10:30)**
- **10:30 - 12:00 Paper Presentations**
  - *S. Reis, A. Metzger, K. Pohl: A Reuse Technique for Performance Testing of Software Product Lines*
  - *J. J. Williams: Considerations in Developing a Controls Product Line Test Architecture Implementation*
  - *E. Olimpiew, H. Gomaa: Customizable Requirements-based Test Models for Software Product Lines*
  - *R. Coelho, V. Alves, U. Kulesza, A. Costa Neto, A. Garcia, A. v. Staa, C. Lucena, P. Borba: On Testing Crosscutting Features using Extension Join Points*

Agenda - Afternoon

- **Lunch break (12:00 – 13:00)**
- **13:00 – 13:30 Invited speaker**
  - *Ram Chillarege (Chillarege Inc.): ODC crystallizes Test Effectiveness*
- **13:30 – 14:00 Breakout session topic selection**
- **14:00 – 16:15 Breakout session**
  - 15:00 Refreshment Break
- **16:15 – 17:00 Results/group discussion and wrap-up**



- **17:00 – 19:00 SPLC Conference Reception**

One last thing …

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

SPLiT 2006

Paper Presentations

- **10:30 - 10:50**
  S. Reis, A. Metzger, K. Pohl: A Reuse Technique for Performance Testing of Software Product Lines
- **10:50 - 11:10**
  J. J. Williams: Considerations in Developing a Controls Product Line Test Architecture Implementation
- **11:10 - 11:30**
  E. Olimpiew, H. Gomaa: Customizable Requirements-based Test Models for Software Product Lines
- **11:30 - 11:50**
  R. Coelho, V. Alves, U. Kulesza, A. Costa Neto, A. Garcia, A. v. Staa, C. Lucena, P. Borba: On Testing Crosscutting Features using Extension Join Points

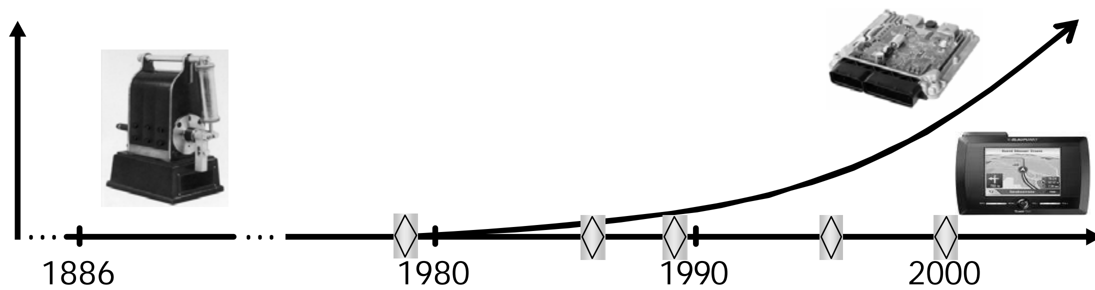
SPLiT 2006

Challenges for Software Product Line Testing

Georg Grütter
Corporate Research
Robert Bosch GmbH

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

Software by Bosch



➔ Software is a major business for Bosch

➔ Software allows the realization of

- complex systems
- specific quality goals

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

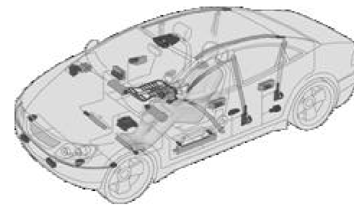
Software Product Lines @ Bosch

- Product line initiative started in 2000
- Launched several new successful product lines
- Deciding factor: scoping and architectural work

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

BOSCH

Automotive Software Specifics



- Safety relevance
- Complexity
- Market

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

BOSCH

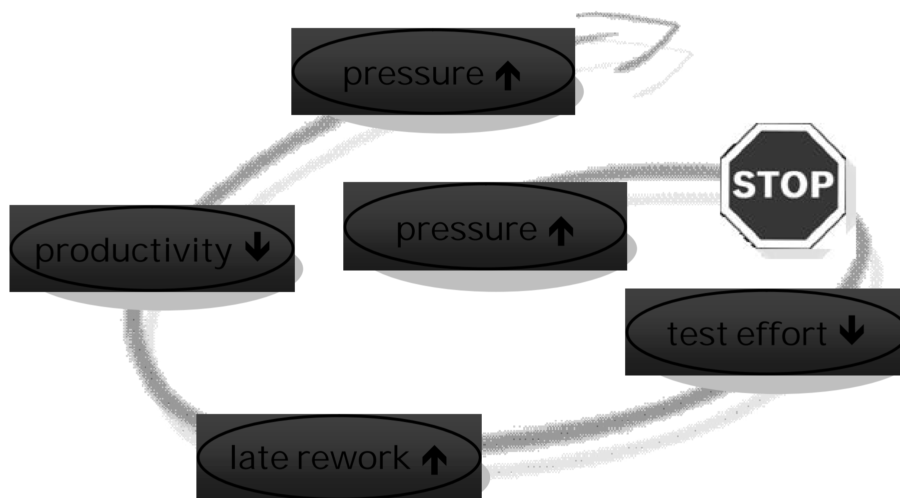
The testing challenges

- Testing is key future challenge
- Technical challenges vs. nontechnical challenges
- Hypothesis: Business impact of testing not well understood

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

BOSCH

Establishing Testing as a well regarded practice

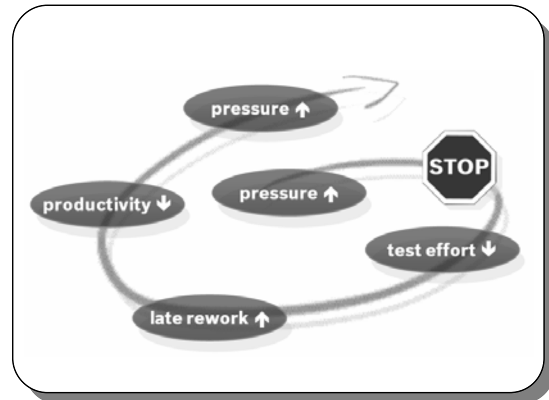


CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

BOSCH

Establishing Testing as a well regarded practice

- ➔ Developers don't like to break their software
- ➔ Testing considered unproductive by. Mgmt.

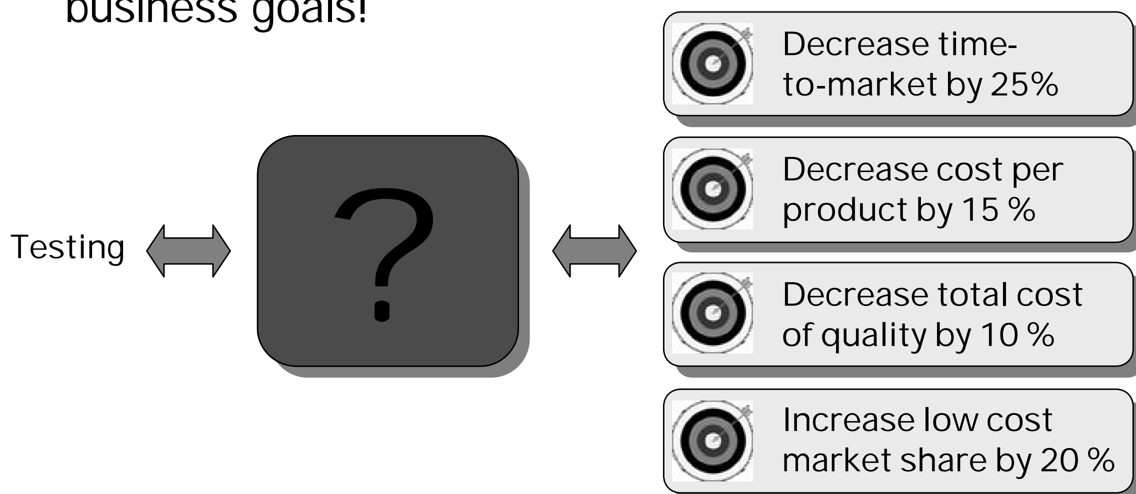


- ➔ We need to understand the business impact and appreciate the business value of testing

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.
BOSCH

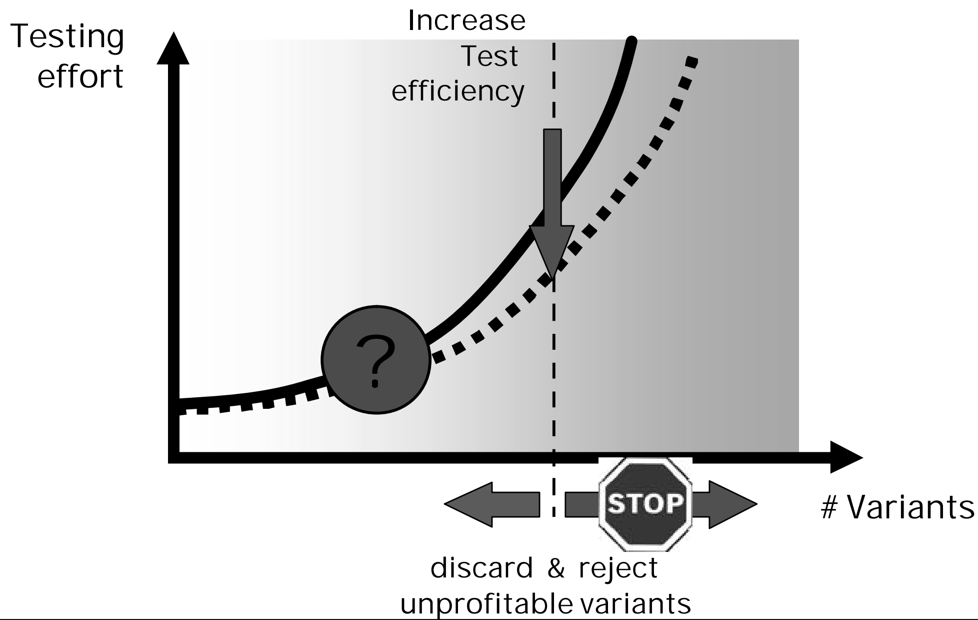
The business impact of testing

- ➔ Business impact: the impact on the attainment of business goals!



CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.
BOSCH

Controlling the growth of variability

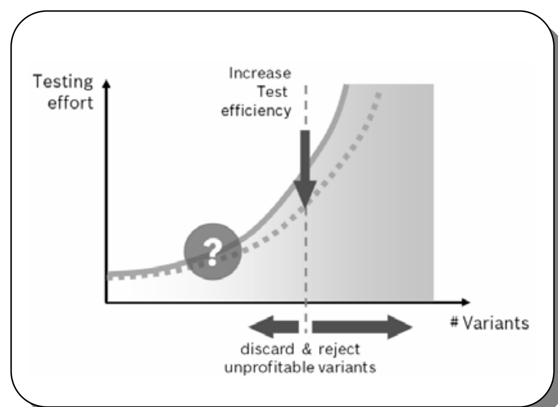


CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

SPLiT 2006 Challenges for Software Product Line Testing

Controlling the growth of variability

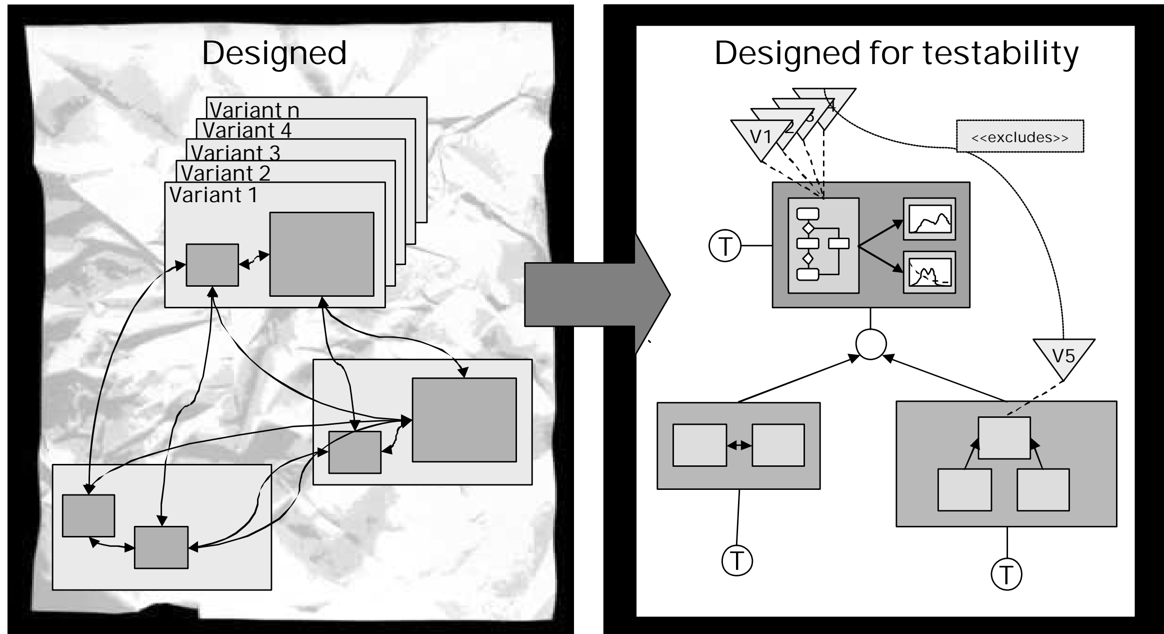
- ➔ Fun of reusing vs. fun of building !
- ➔ Value of new variants !
- ➔ Cost of new variants ?



- ➔ We need to understand the business impact of variability on testing

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

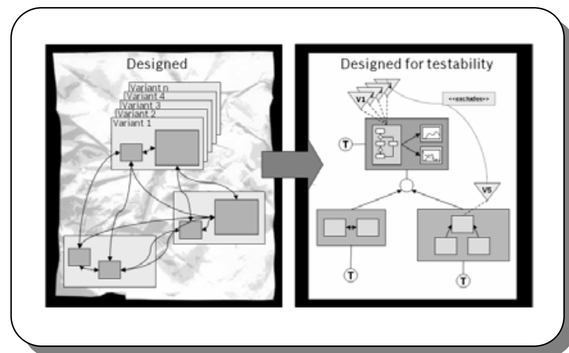
Acting on design decisions for testability



CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

Acting on design decisions for testability

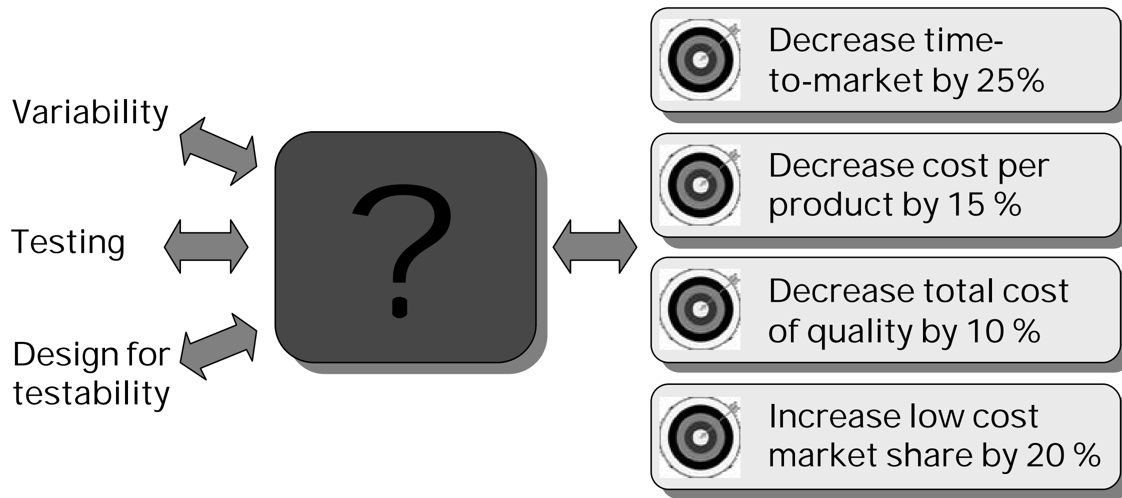
- ➔ Cost of changing architectural decisions !
- ➔ Impact of d4t on system qualities ?



- ➔ We need to understand the business impact of design decisions for testability.

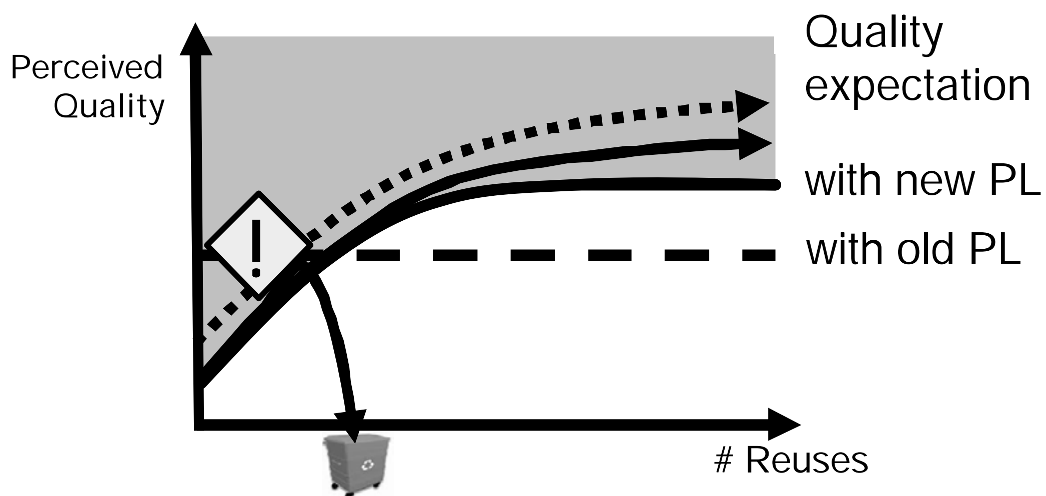
CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

Business impact?



CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

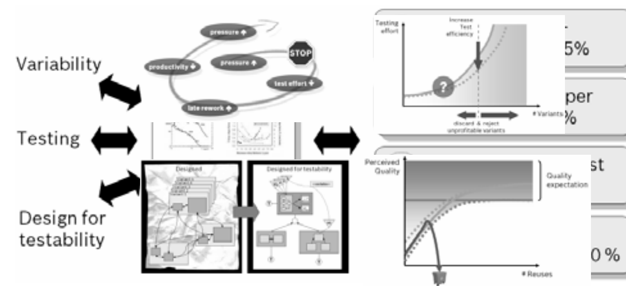
Meeting quality expectations



➔ We need to convey a realistic picture of early asset quality

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

Summary



- ➔ Nontechnical challenges need to be addressed
- ➔ Understand business impact of testing
- ➔ Appreciate business value of testing
- ➔ Models and empirical research are needed

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

Suggestions for workshop discussion topics

- ➔ Business value of testing?
- ➔ Why not commonpractice?
- ➔ Can there be generalized and yet useful models?
- ➔ D4T decisions ⇔ qualities and business impact?
- ➔ How to deal with early asset quality?
- ➔ How do other engineering disciplines do it?

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

The End

Contact me:

BOSCH

Georg Grütter

Corporate Research
Advance Engineering – Software

georg.gruetter@de.bosch.com

Robert Bosch GmbH, CR/AEA3-Fr
P.O. Box 94 03 59
60461 Frankfurt a. M.
Germany

Phone: +49 69 7909-514
Fax: +49 69 9540-295514

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

A Reuse Technique for Performance Testing of Software Product Lines

Sacha Reis, Andreas Metzger, Klaus Pohl

Software Systems Engineering

Institute for Computer Science and Business Information Systems (ICB)

University of Duisburg-Essen, Germany

www.sse.uni-due.de

Outline

- **Introduction**
- **Variability in Performance Aspects**
- **ScenTED for Performance Testing**
- **Case Study**
- **Open Issues**

Introduction
Motivation

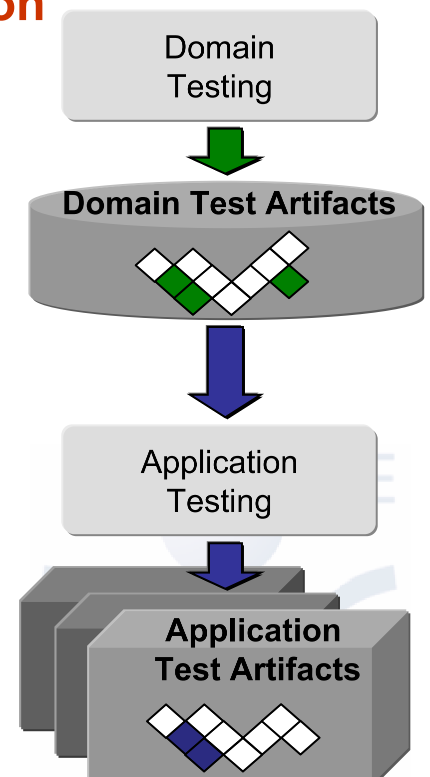
- Observation:
  – Even if software meets its **functional requirements**, **violating the quality requirements** will **diminish the software's value** to the user

  – **Examples:**
    - **Stock market data:**
      - no value if data arrives too late
    - **Emergency shutdown of nuclear reactor:**
      - catastrophic consequences if shutdown occurs too late

→ **Performance is an important quality property!**

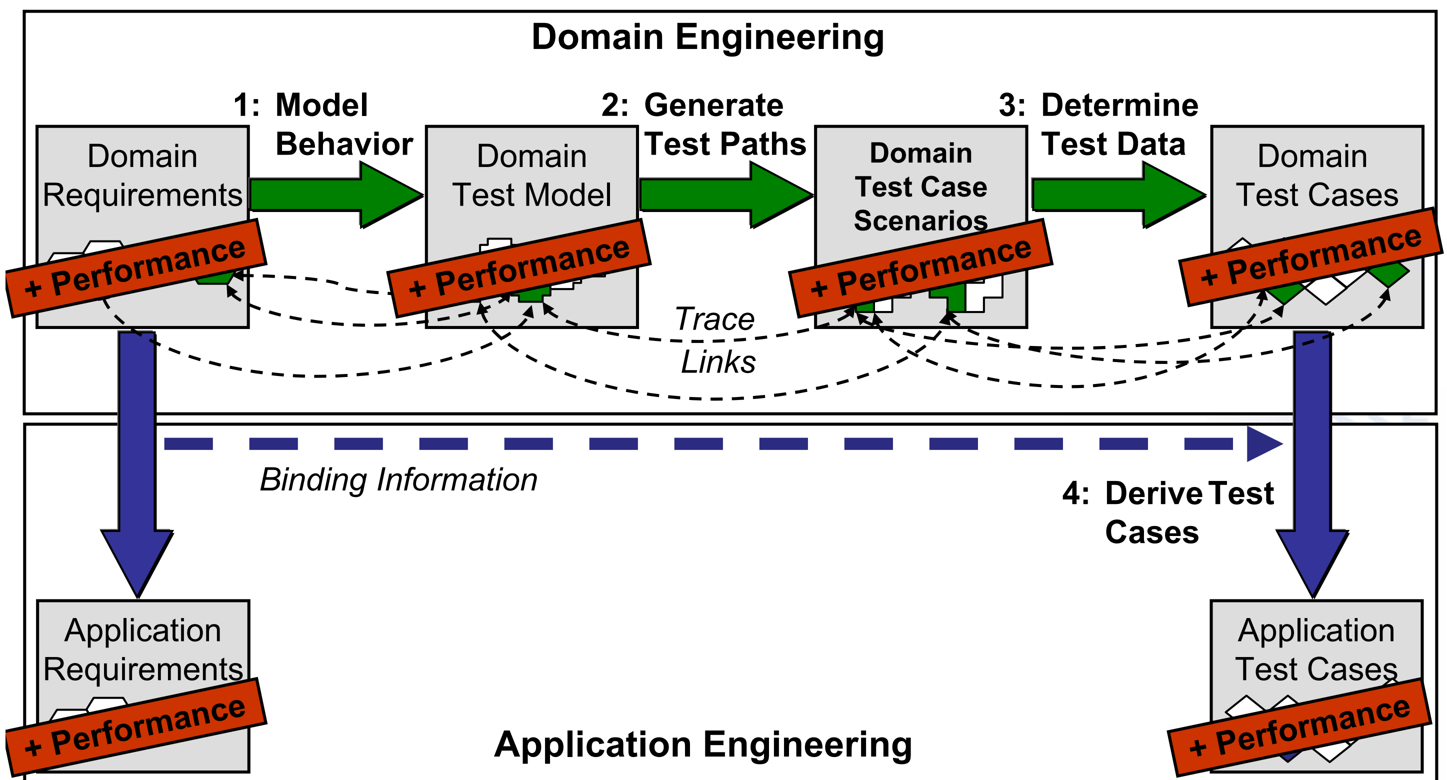
Introduction
Challenges in SPL Performance Testing

- How to support the **derivation of application performance test artifacts** from domain performance test artifacts?
  – **derivation of domain performance test artifacts**
  – **reuse of domain performance test artifacts**
  – **execution of application performance tests**



Introduction
Approach

- Extension of our **ScenTED technique** for system testing



Introduction
Approach

- **Questions to be answered** when extending ScenTED:
  - **Specification of performance requirements**
    - How can performance aspects be described in the relevant artifacts of ScenTED (i.e., use cases, scenarios, activity diagrams)?
    - Which additional variability in performance aspects has to be modelled?
  - **Support of different types of performance tests** [RUP, 2001]
    - Benchmark testing
    - Contention testing
    - **Performance profiling** } **Architectural information is needed**
    - **Load testing**
    - **Stress Testing**

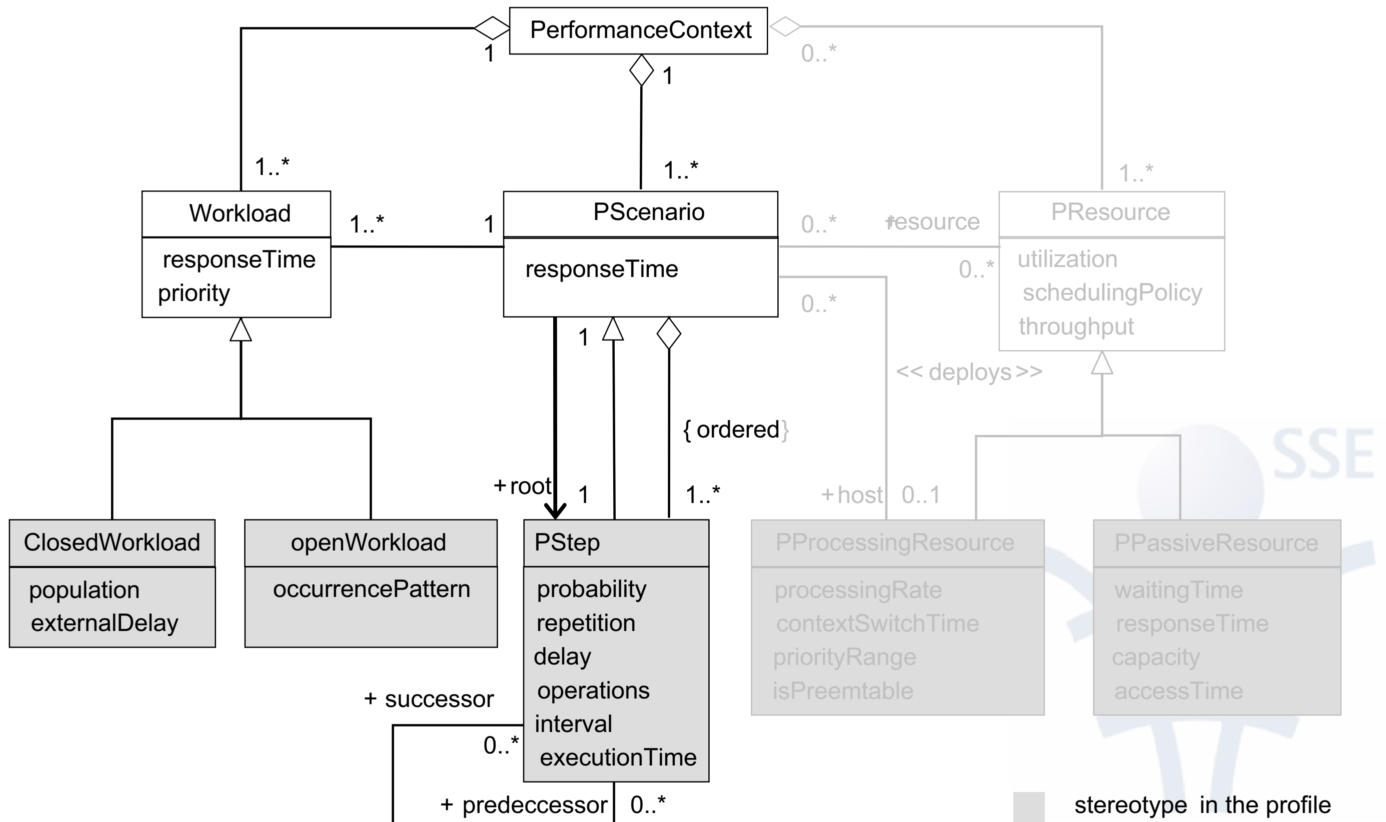
    } **Requirements are sufficient**

Outline

- Introduction
- **Variability in Performance Aspects**
- ScenTED for Performance Testing
- Case Study
- Open Issues

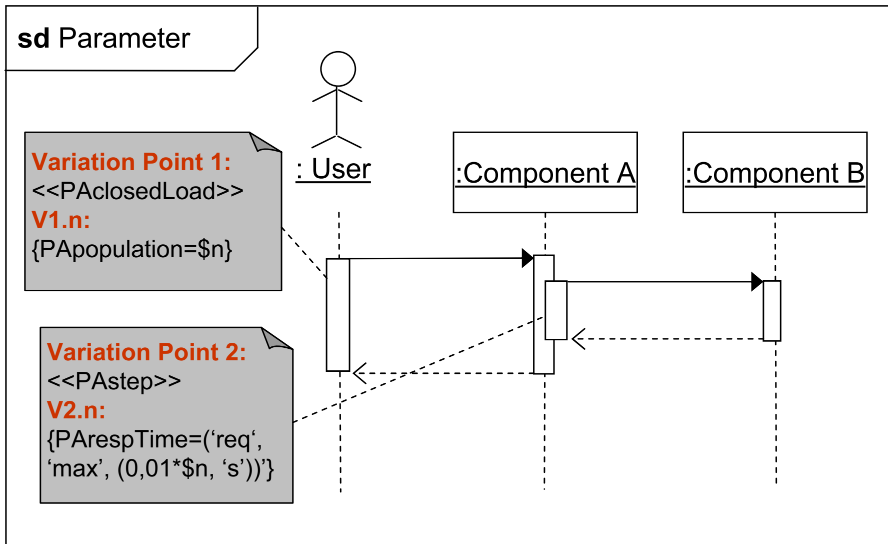
Variability in Performance Aspects
UML Profile for Schedul., Performance & Time

- **Conceptual model** [OMG, 2002]



Variability in Performance Aspects
Extensions of the SPT Profile (1)

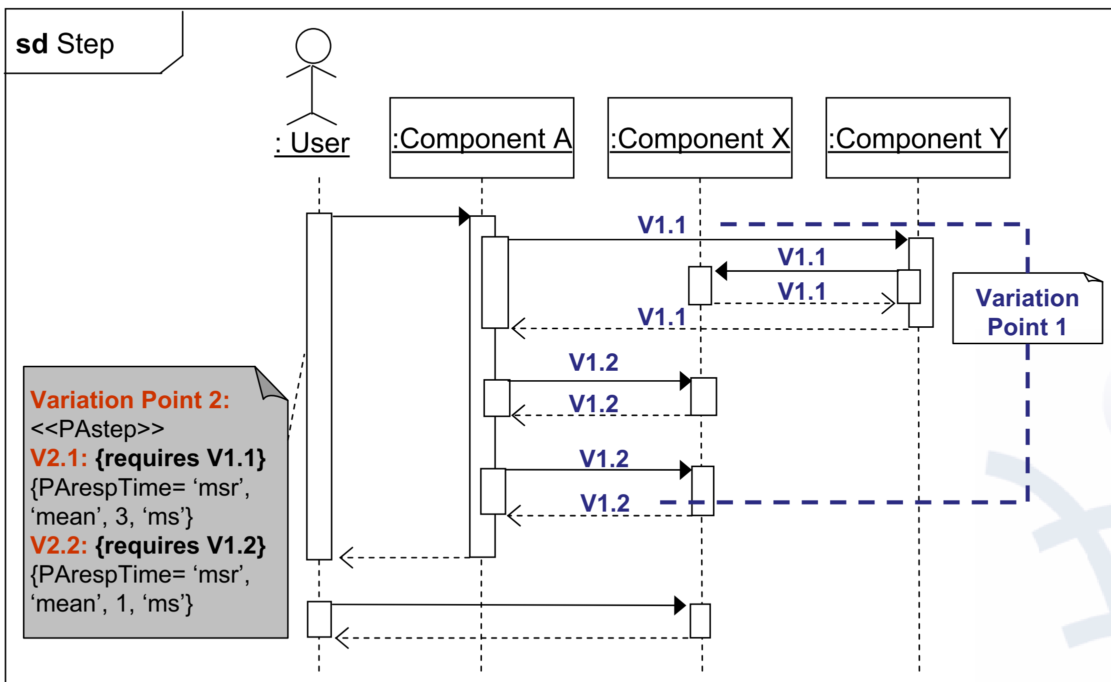
- Make **variation points (performance variability)** explicit



  – *Note:* **"red"** text in figures represents **performance aspects**

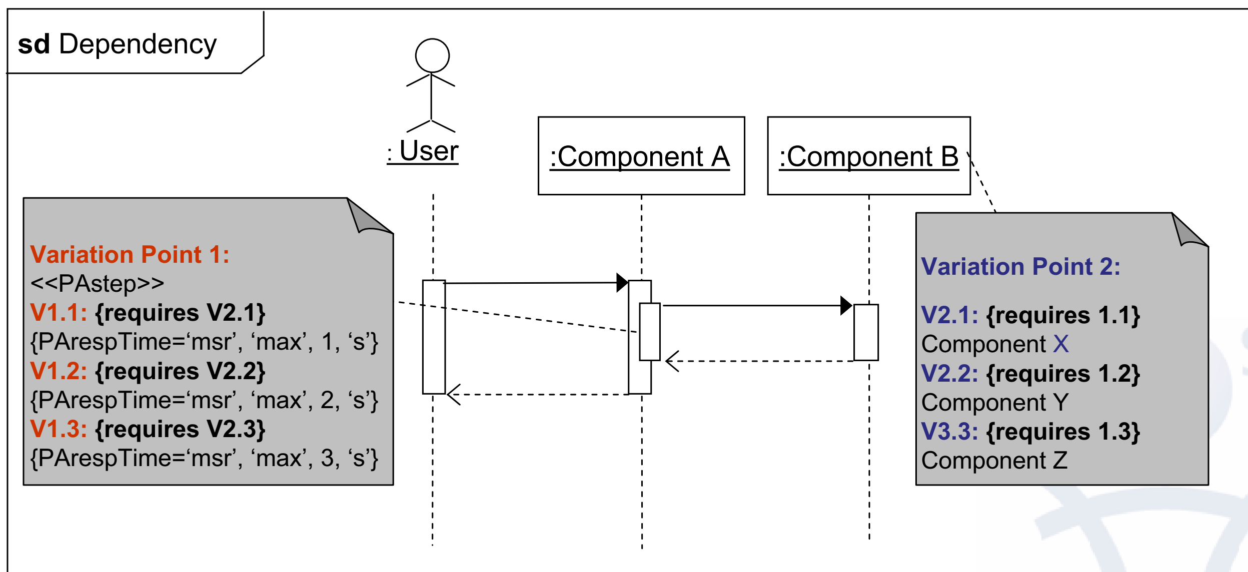
Variability in Performance Aspects
Extensions of the SPT Profile (2)

- Make **dependencies** between **variants in the control flow** and **variability in performance aspects** explicit



Variability in Performance Aspects
Extensions of the SPT Profile (3)

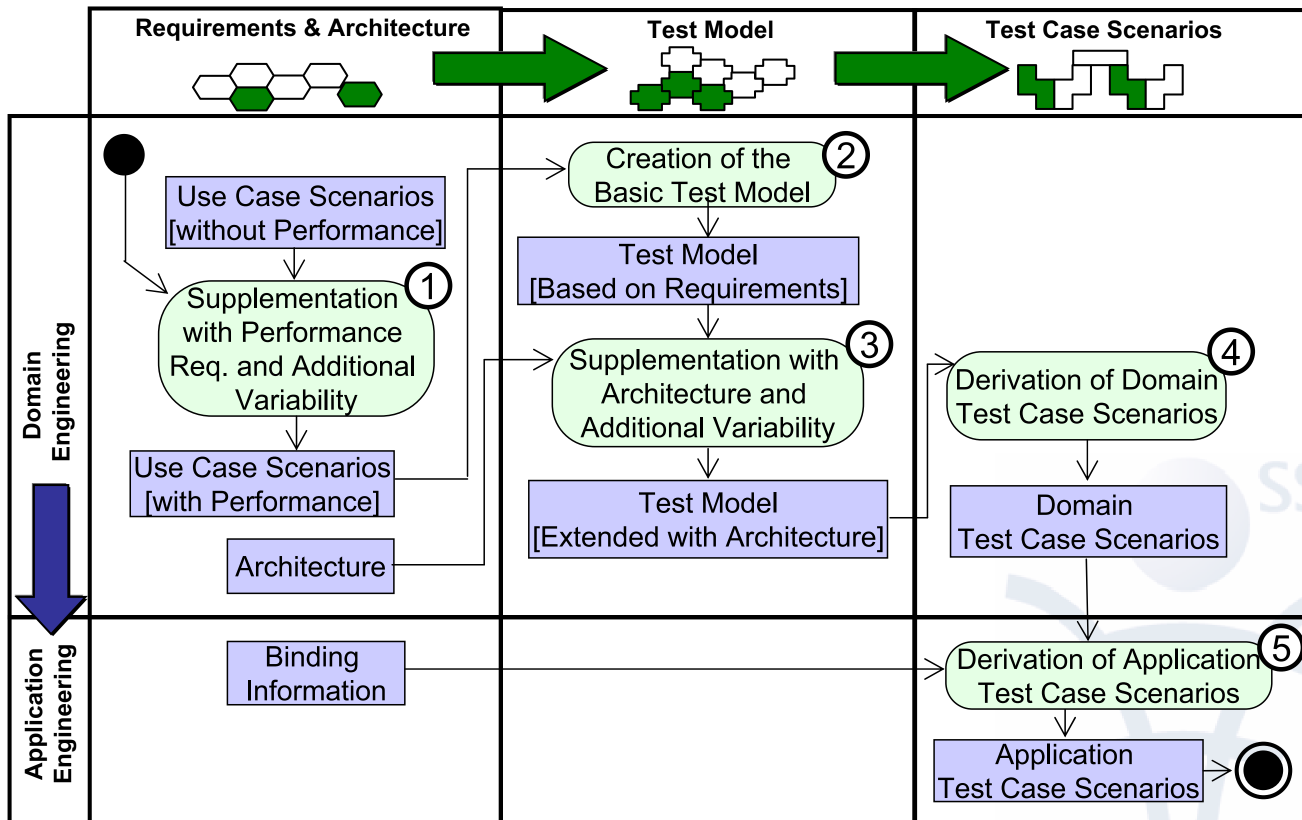
- Make **dependencies** between **architectural variability** and **performance variability** explicit



Outline

- Introduction
- Variability in Performance Aspects
- **ScenTED for Performance Testing**
- Case Study
- Open Issues

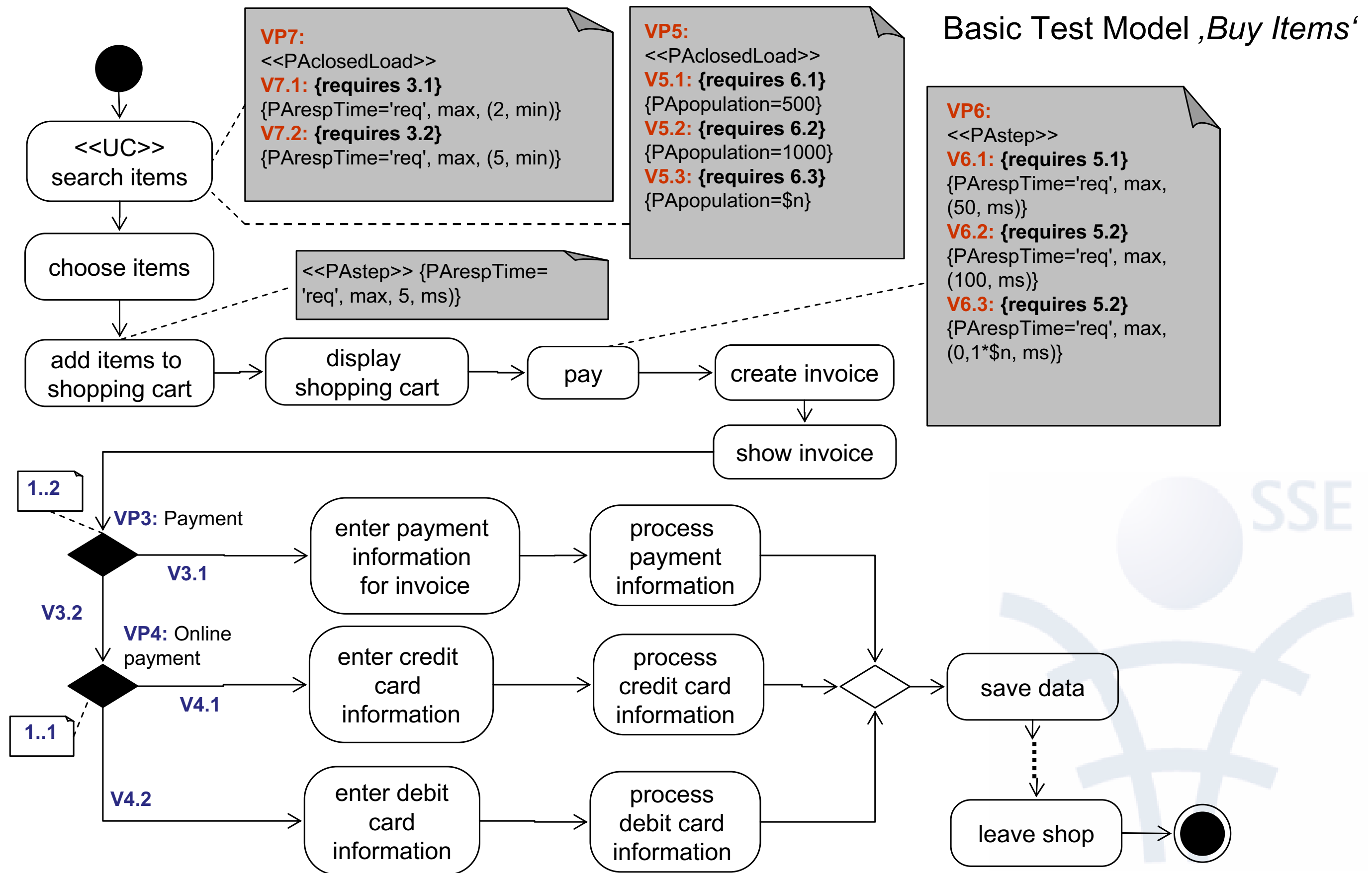
ScenTED for Performance Testing
Overview



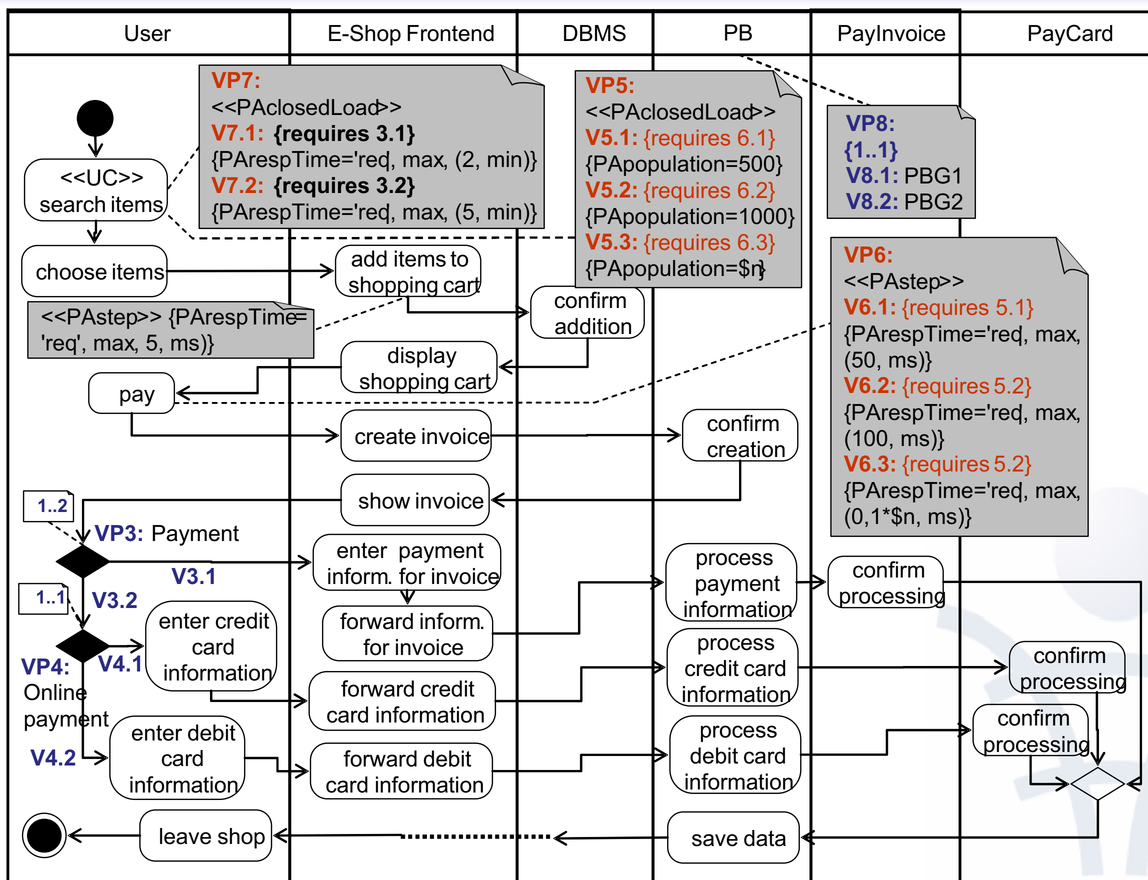
ScenTED for Performance Testing
Step 1: Suppl. with Performance and Variability

Use Case Name	Buy items	
...	...	
	Step	**Action Description**
Main Scenario	1 2 3 4 5 6 ...	(include UC: Search items) **{VP5} {VP7}** Customer chooses item(s) System adds item(s) to shopping cart (This activity has to be accomplished within 5 milliseconds) Customer wants to pay System calculates and shows the invoice **{VP6}** Systems asks for payment information **{VP3}** ...
Variation Points		**Variations**
VP3 (Payment, 1..2 out of 2)		**V3.1** (include UC: pay w/invoice) **(requires V7.1)** **V3.2** Payment w/card **(requires V7.2)** **{VP4}**
VP4 (pay w/card, 1..1 out of 2)		**V4.1** (include UC: pay w/credit-card) **V4.2** (include UC: pay w/debit-card)
VP5 <<PAclosedLoad>>		**V5.1** The number of users active in the system at one time is limited to 500 Users **(requires V6.1)** **V5.2** …
VP6 <<PAstep>>		**V6.1** This activity has to be accomplished within 50 milliseconds **(req. 5.1)** **V6.2 …**
VP7 <<PAclosedLoad>>		**V7.1** The scenario has to be accomplished within 2 minutes **(req.V3.1)** **V7.2** The scenario has to be accomplished within 5 minutes **(req.V3.2)**

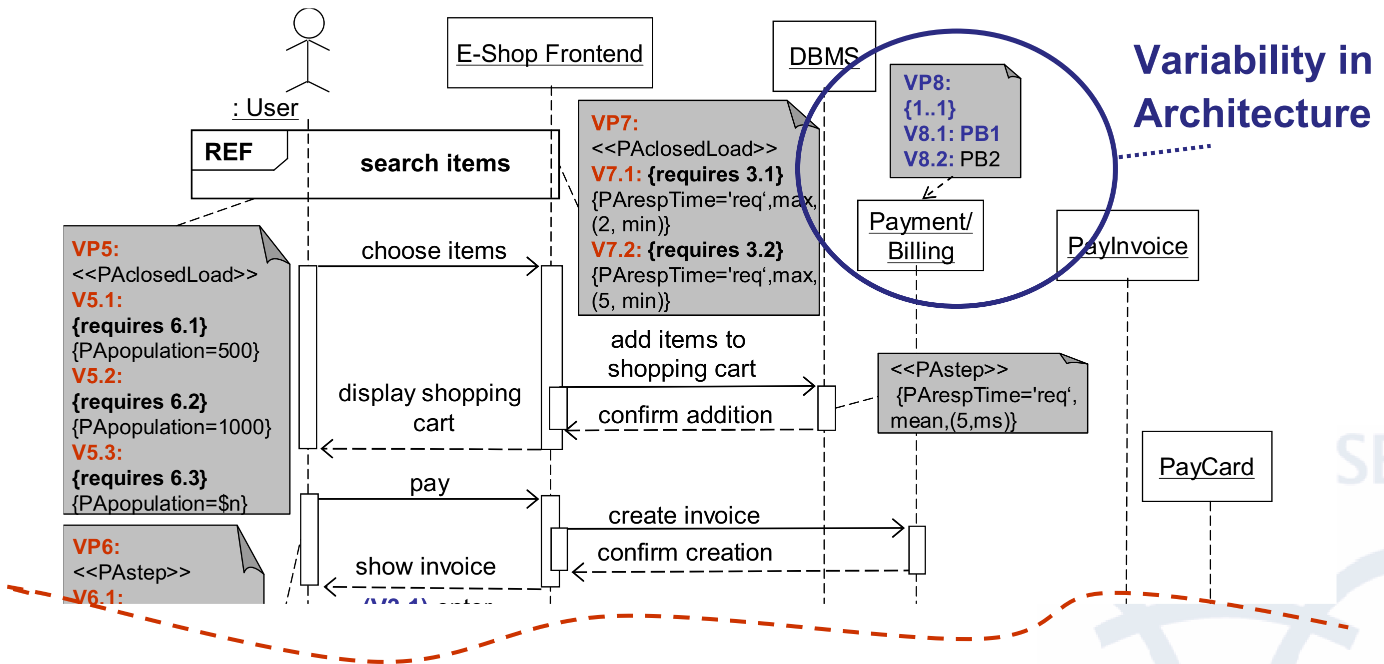
ScenTED for Performance Testing
Step 2: Creation of the Basic Test Model



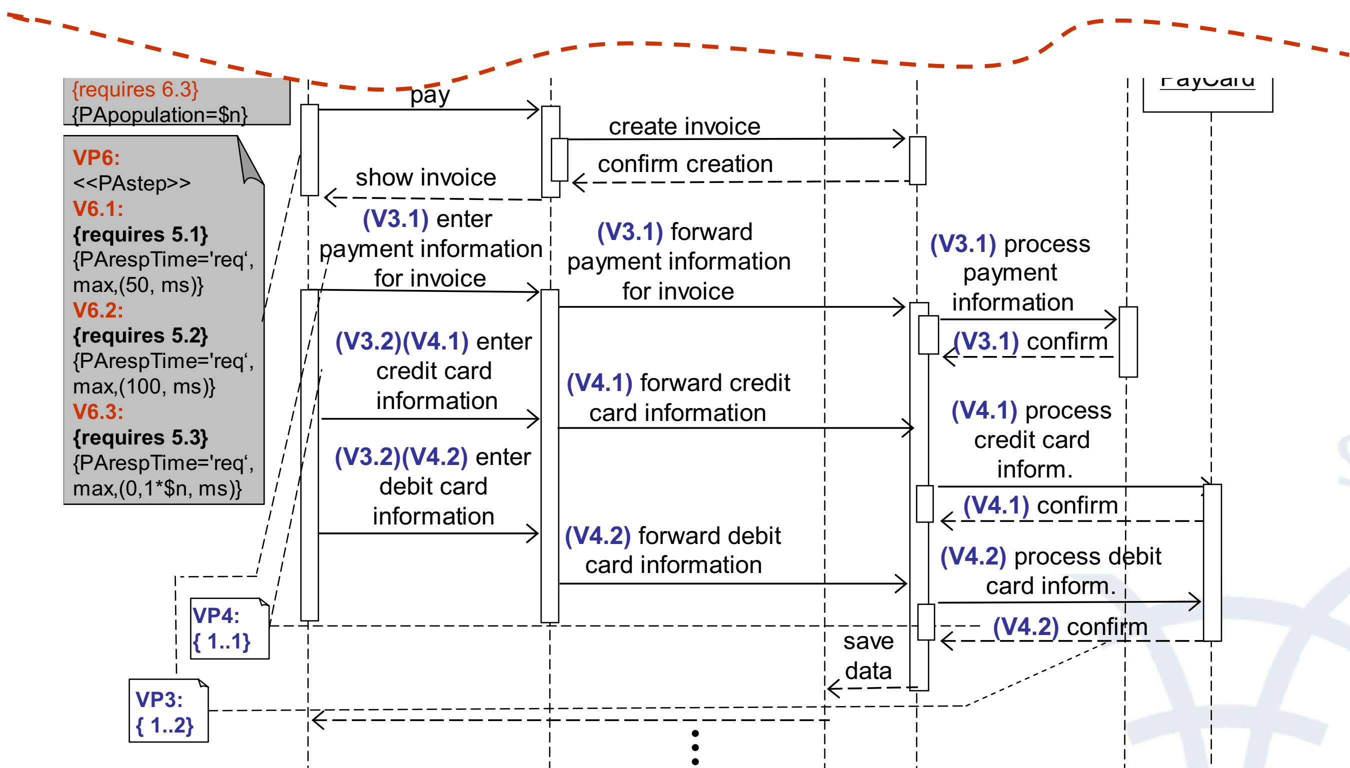
ScenTED for Performance Testing
Step 3: Suppl. with Architecture and Variability



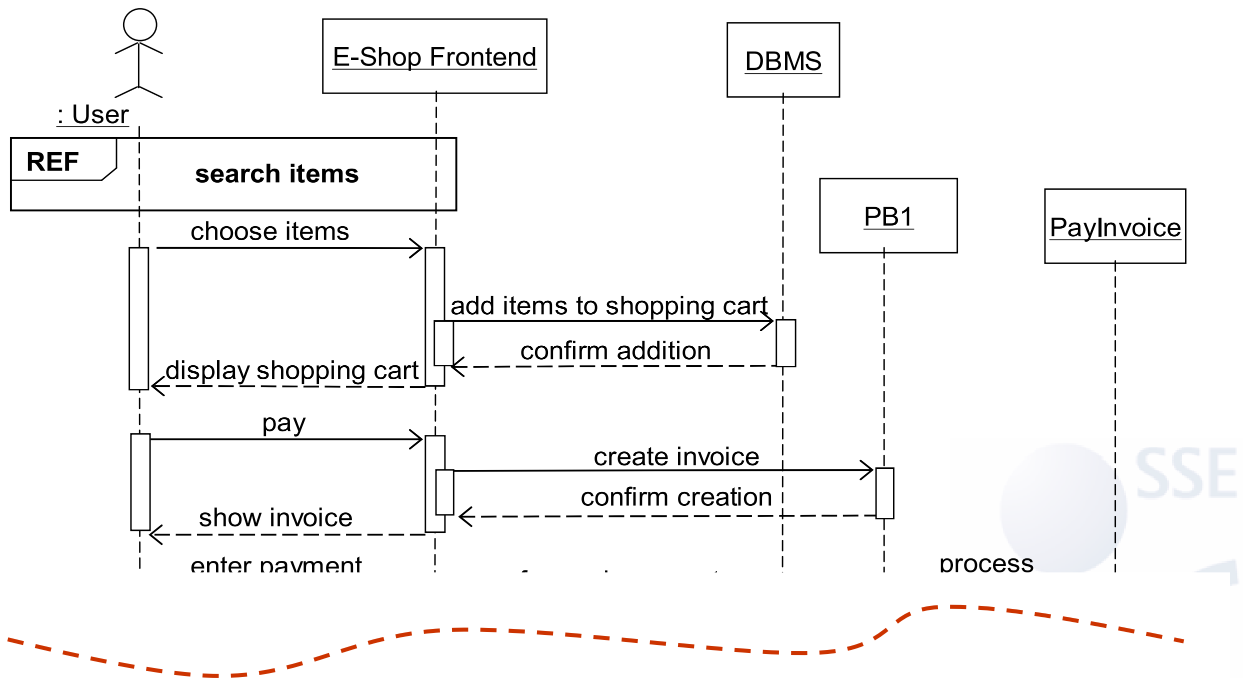
ScenTED for Performance Testing
Step 4: Derivation of Domain TC Scenarios (1)



ScenTED for Performance Testing
Step 4: Derivation of Domain TC Scenarios (2)



ScenTED for Performance Testing
Step 5: Derivation of Application TC Scenarios



Outline

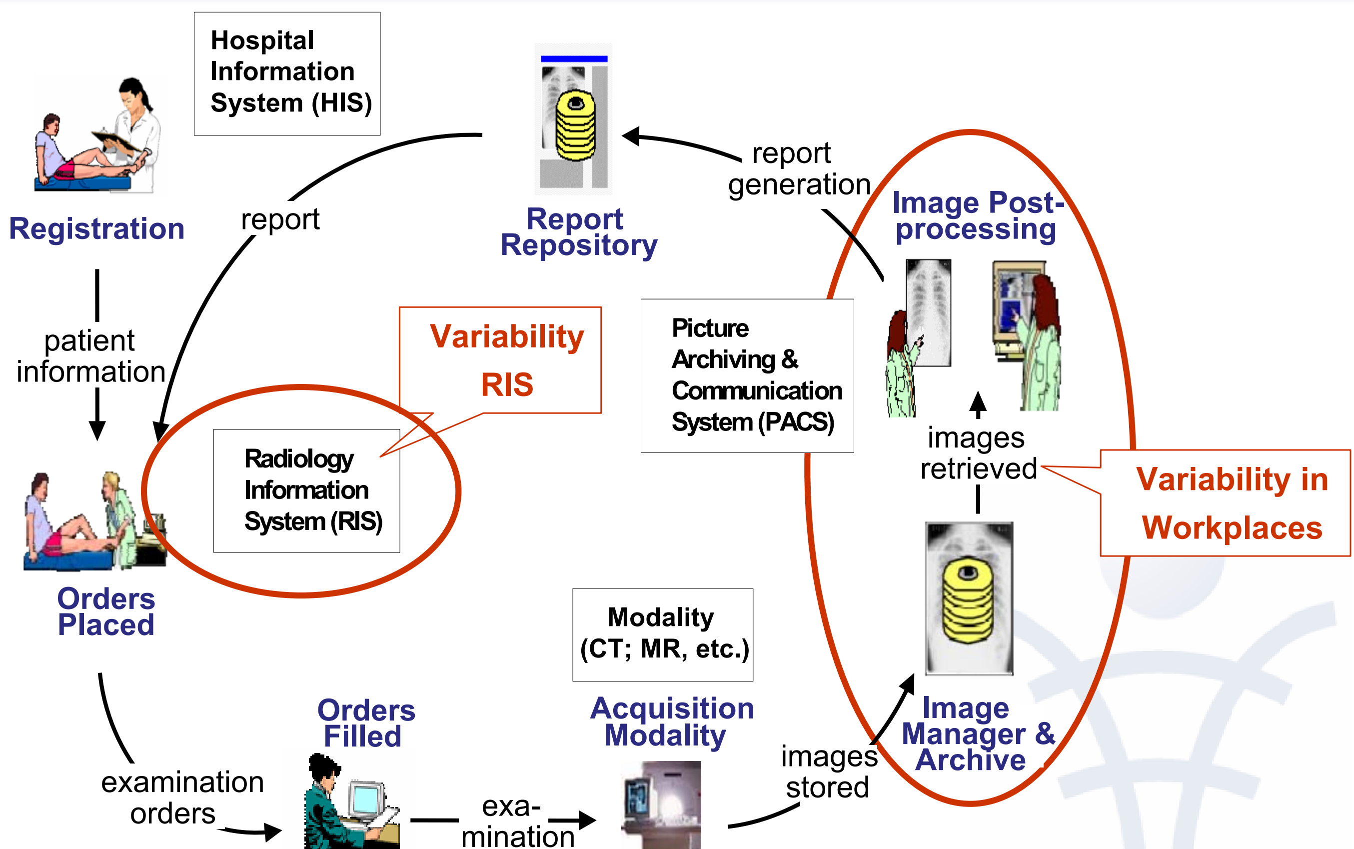
- Introduction
- Variability in Performance Aspects
- ScenTED for Performance Testing
- **Case Study**
- Open Issues

Case Study at Siemens Medical Solutions HS IM
Domain & Objectives

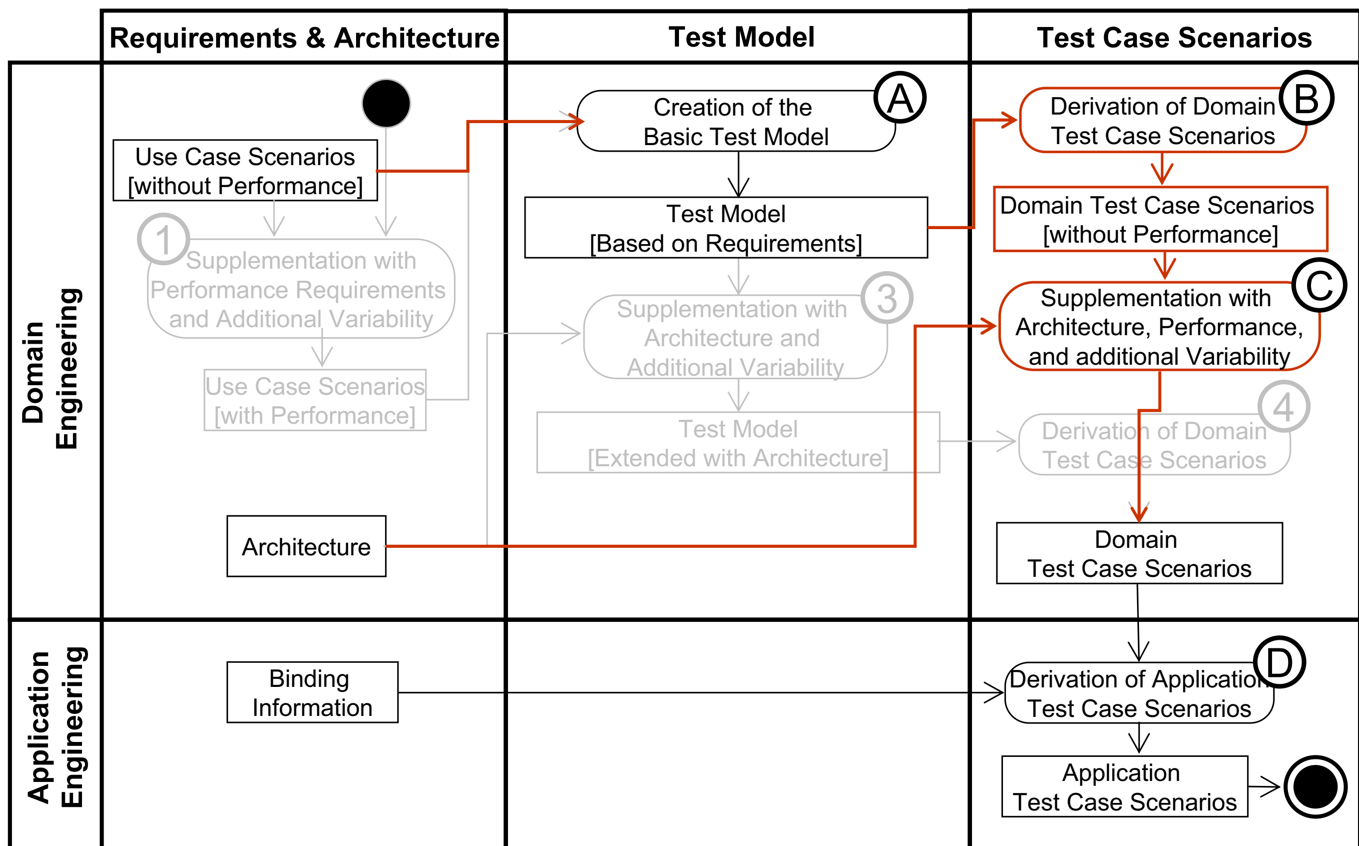
- **Domain**
  - **Integrated radiology suite**
    - Covers the whole basic clinical workflow
      - Examination
      - Imaging
      - Post-processing
      - Archiving

  - **Focus on variability in…**
    - …workplaces
    - …radiology information systems (RIS)

Case Study at Siemens Medical Solutions HS IM
Basic Clinical Workflow



Case Study at Siemens Medical Solutions HS IM
ScenTED Adaptation



Case Study at Siemens Medical Solutions HS IM
Results

- **Reduced performance testing effort**
  - Testers and test managers confirmed an effort reduction

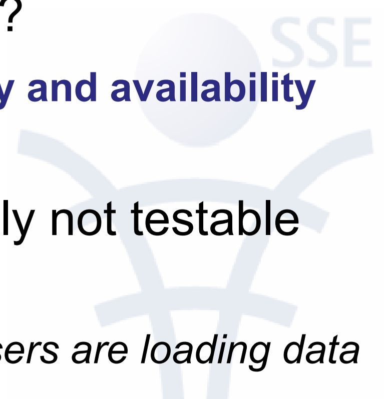
- **Early validation** of performance requirements (PR) and variability
  - **35** of **51** PR's were associated to *domain performance test case scenarios*
  - **16** did not affect the basic clinical workflow
  - **5** could not be tested
  - **2** new PR's were identified
  - **20** PR's were extended with additional variability

Outline

- Introduction
- Variability in Performance Aspects
- ScenTED for Performance Testing
- Case Study
- **Open Issues**

Open Issues

- Dealing with the problem of **combinatorial explosion**
  - A) During ***performance profiling*** **not all possible configurations** can be tested
  - B) **General problem** in SPL Testing
    - testing all potential combinations of the variants
    - →**also check the "Tesing in a Software Product Line" Panel om Wednesday!**
- How can **other quality properties** be tested?
  - →**current project at SSE that deals with reliability and availability (for single systems)**
- Influences of **parallel processes** are currently not testable (contention testing)
  - *"What is the influence on data saving, when 50 users are loading data simultaneously?"*



Further Information…

Questions ?

Contact:

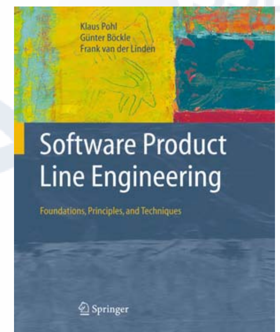
Dr. Andreas Metzger

Software Systems Engineering
Schützenbahn 70
University of Duisburg-Essen
45117 Essen, Germany

andreas.metzger@sse.uni-due.de

Text book:

Pohl, Böckle, van der Linden:
Software Product Line Engineering – Foundations, Principles and Techniques.
Springer, 2005

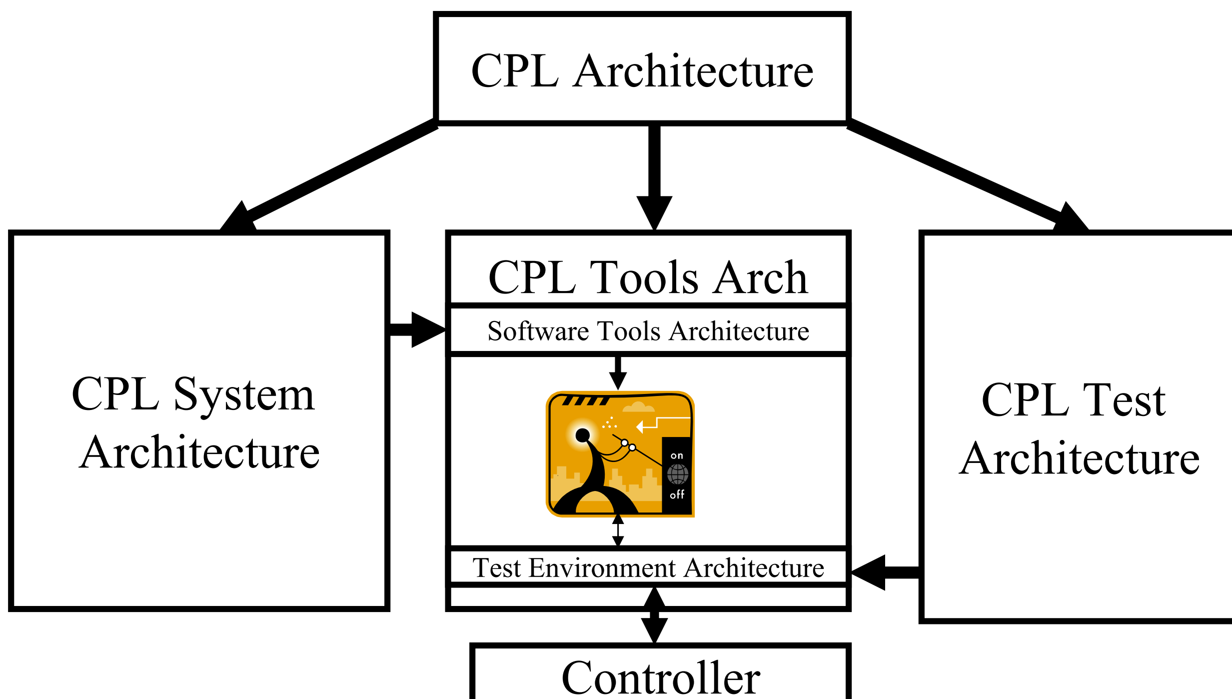


References

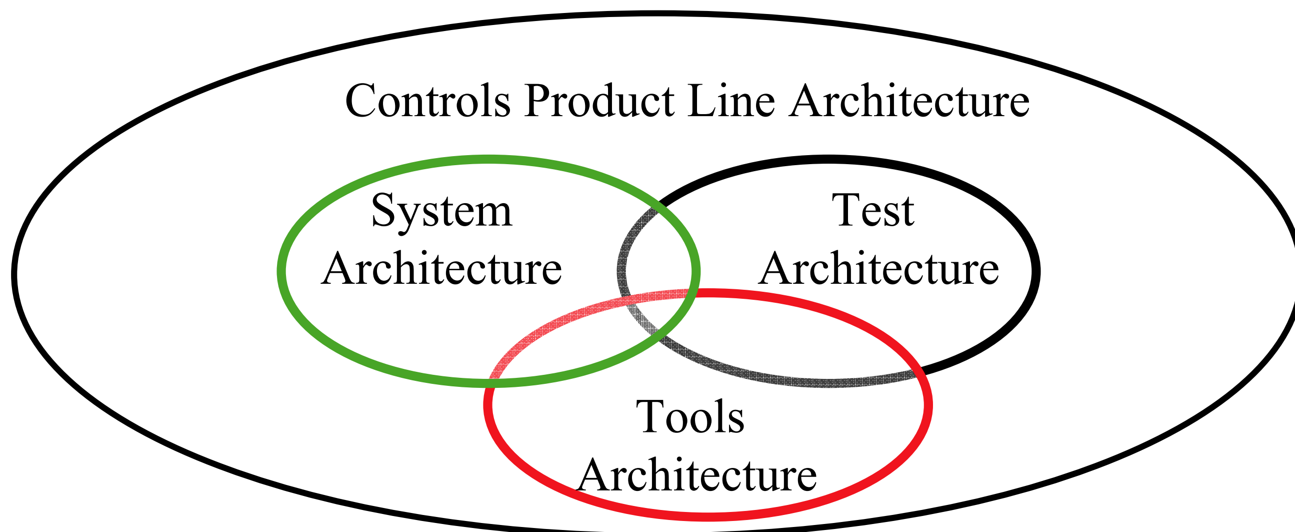
- [RUP, 2001] Rational Unified Cooperation; Concepts: Performance Testing; Rational Suite Enterprise; Rational Unified Process; 1987-2001
- [OMG, 2002] Object Management Group: UML Profile for Schedulability, Performance, and Time. OMG Specification, ptc/02-03-02, July 2002.

Considerations in Developing a Controls Product Line Test Architecture

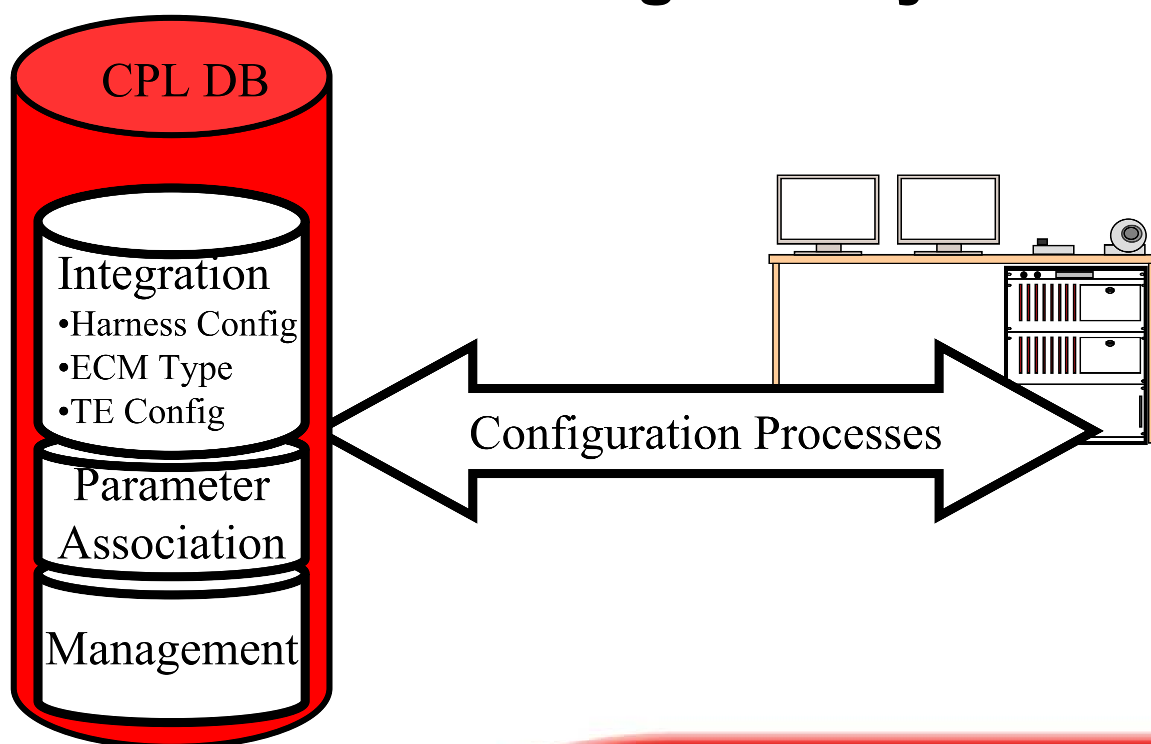
Controls Product Line Architecture Requirement Flow



Controls Product Line Architecture



Implementing the vision through test environment reconfigurability



Implementing the vision through test environment reconfigurability

Core II Software Tool

FILE EDIT TOOLS UTILITIES HELP Window

Harness

Show All
Show Latest
Filter

Save
Revert
Back

Controller Selectivity

ID	VID	Harness	RLOC Table	Checkout Status
3	2	CM2100MD - ACCEPTS CORE II	CM2100MD	OUT by Magdalena Purnama on 26
6	2	CM2100RC - ACCEPTS Core II	CM2100RAM	OUT by Magdalena Purnama on 26
10	1	CM2110 - ACCEPTS Core II	CM2110	OUT by Magdalena Purnama on 25
1	2	CM850 - ACCEPTS CORE II	CM850	OUT by Magdalena Purnama on 26
2	1	CM850 - CLOSED LOOP ACCEPTS P	CM850	IN by Scott Decker on 17-MAY-06
7	1	CM850 - Shockwave CoreII	CM850	IN by Scott Decker on 17-MAY-06
8	1	CM850 - Shockwave Industrial	CM850	IN by Scott Decker on 17-MAY-06
4	2	CM871 - ACCEPTS Core II	CM871	OUT by Magdalena Purnama on 26
5	2	CM876 - ACCEPTS Core II	CM876	OUT by Magdalena Purnama on 26

Search

Implementing the vision through test environment reconfigurability

Harness Information

Harness: CM850 - CLOSED LOOP ACCEPTS PROTO — Version: 1

Checkout Status: IN,Scott Decker,May-17-2006

Rloc Table Name: CM850 — Version: 1 — Status: IN

Harness Description: Initial CM850 harness for the prototype closed loop system in CORE II

Module: CM850

Save
Revert
Back

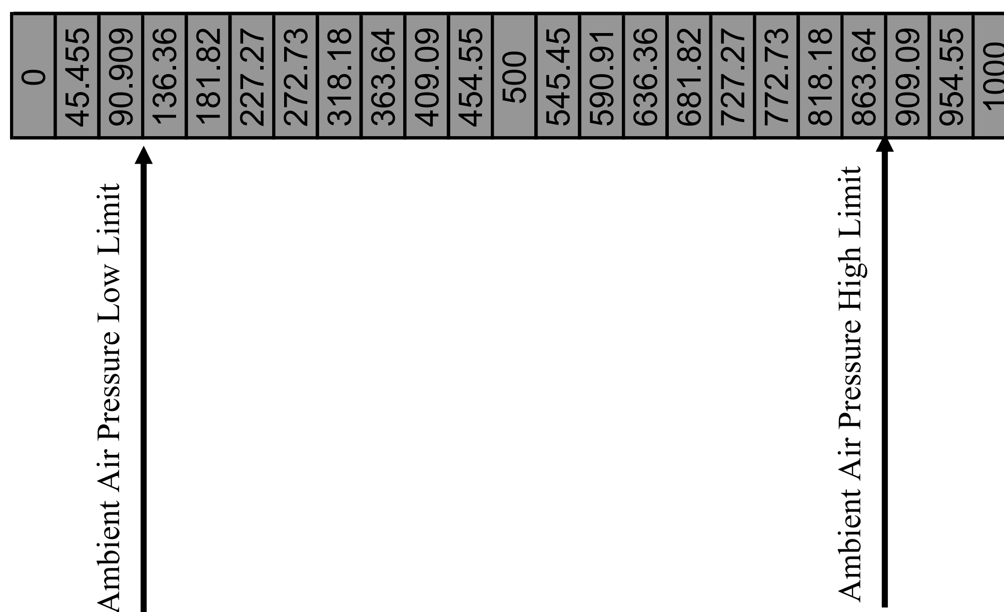
9

RLOC PIN Channel Relationship

Import

RLOC	PIN	Channel	Board	IO Type
4	J1-43	13		Analog Output
5	J1-15	16		Analog Output
6	J1-30	15		Analog Output
7	J1-23	4		Analog Output
8	J1-25	10		Analog Output
9	J1-36	6		Analog Output
10	J1-18	1		Analog Output
11	J1-50	21		Analog Output
12	J1-20	14		Analog Output
13	J1-12	9		Analog Output

Leveraging System Architecture Implementation on Ambient Air Pressure within Test



Ambient Air Pressure Upper Limit Test

{

Set Ambient Air Pressure = Ambient Air Pressure High Limit + tolerance.

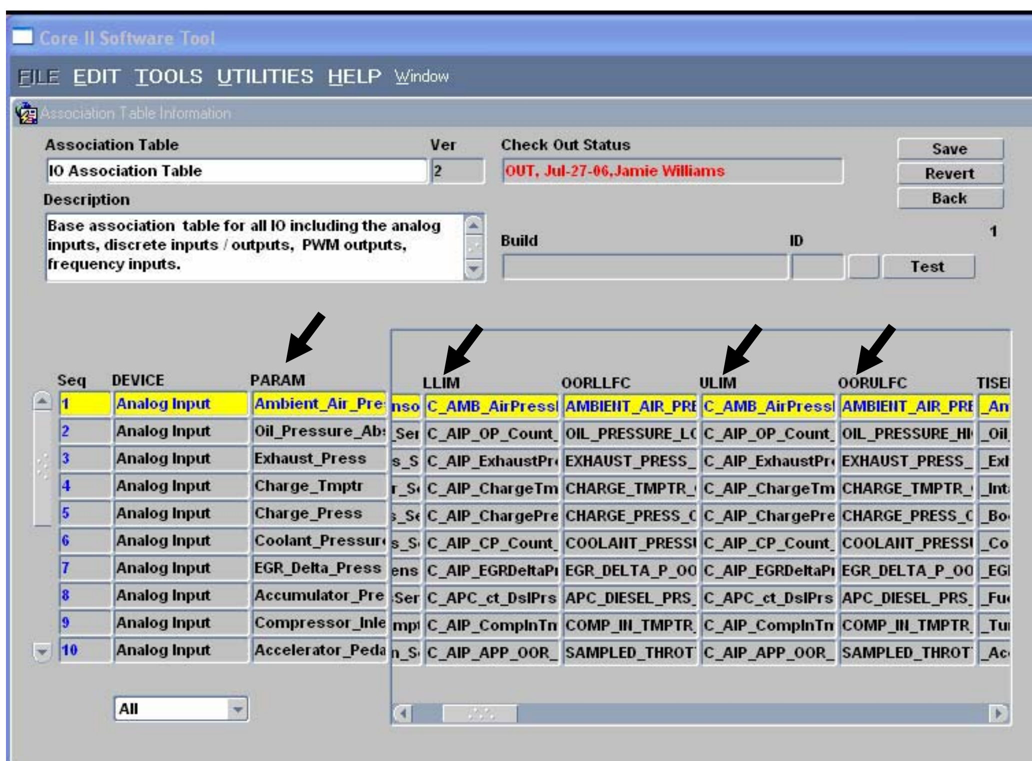
Verify that AMBIENT_AIR_PRESSURE_HIGH_ERROR is set.

Ambient Air Pressure = Ambient Air Pressure High Limit - tolerance

Verify that that AMBIENT_AIR_PRESSURE_HIGH_ERROR is reset.

}

Implementing the vision through data driven test capability



Generic Analog Upper Limit Test

While PARAM /= NULL && ULIM /= NULL && OORULFC /= NULL;

{

Set PARAM = ULIM + tolerance.

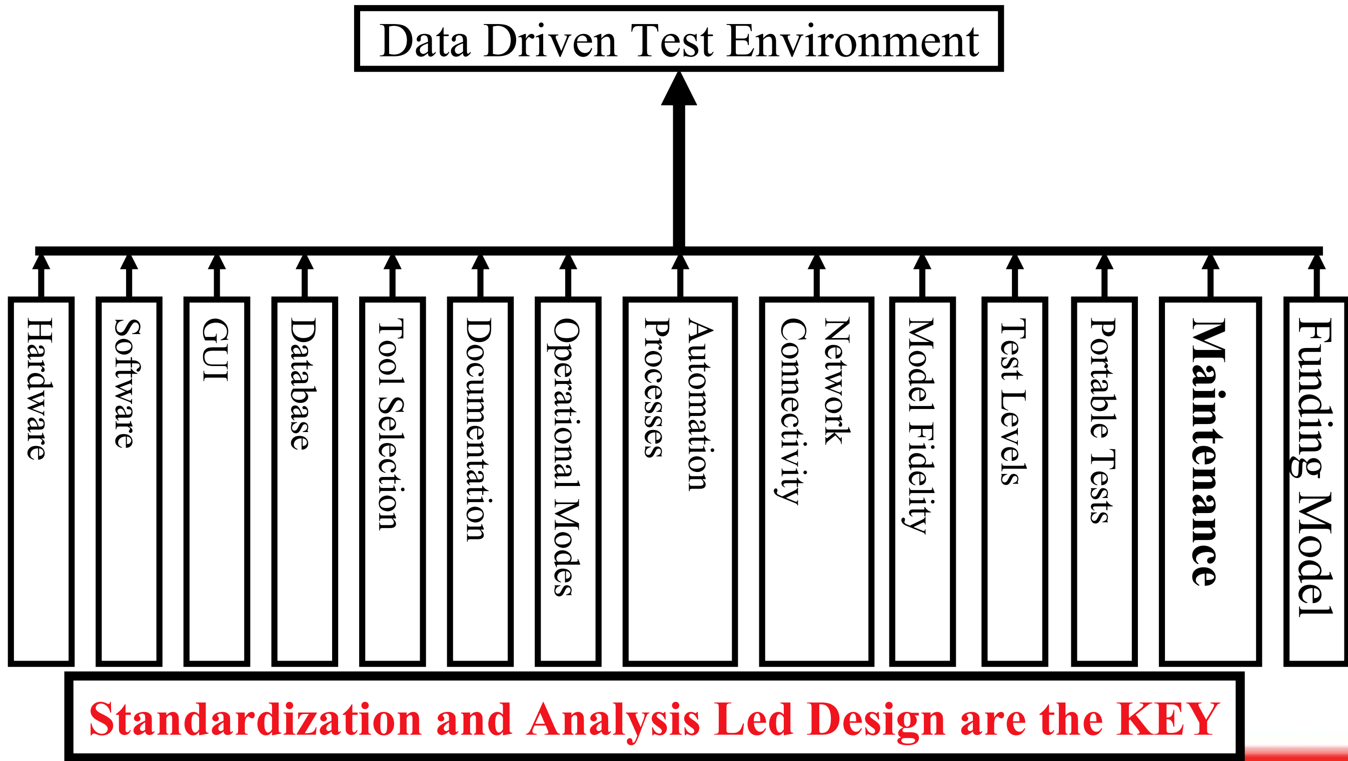
Verify that OORULFC error is set.

Set PARAM = ULIM - tolerance.

Verify that OORULFC error is reset

}

Considerations in implementation of a data driven test environment



Customizable Requirements-based Test Models for Software Product Lines

E. M. Olimpiew and H. Gomaa
Dept. of Information and Software Engineering
George Mason University
Presentation for SPLiT 2006 Workshop
Tuesday, August 22, 2006

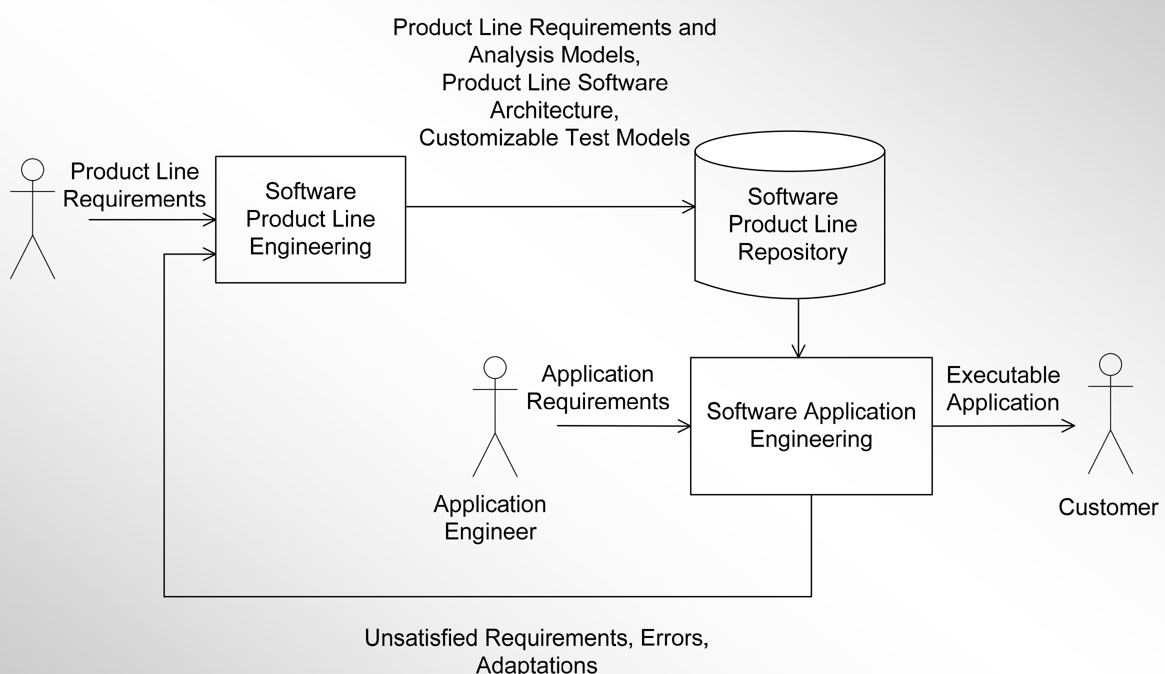
Outline

1. Introduction
2. Related work
3. Requirements-based Test Models for a Software Product Line (SPL)
4. Application of Testing Method to an Automated Toll System Case Study
5. Conclusions

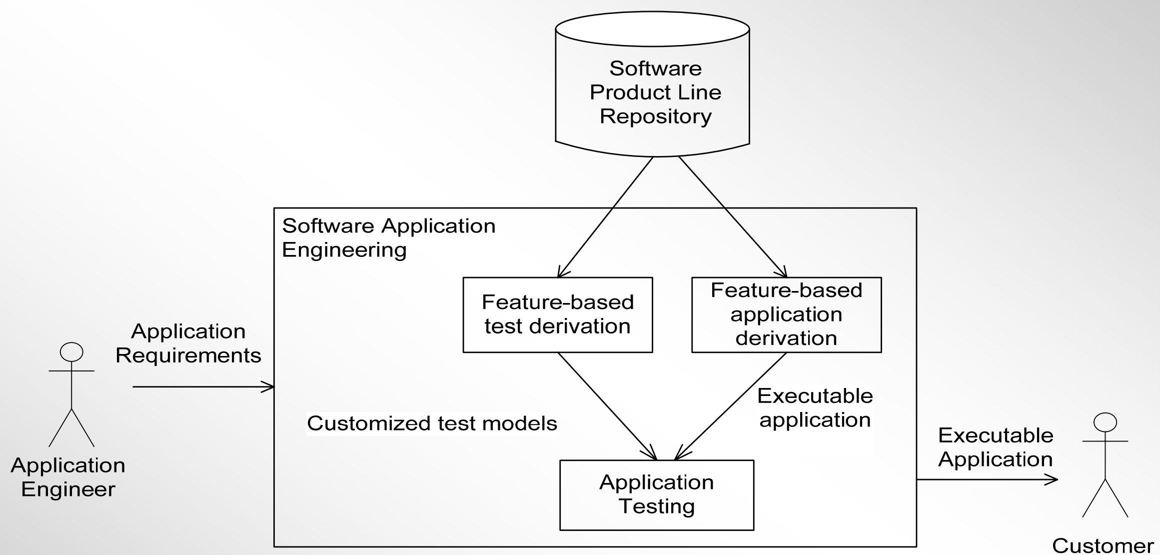
1. Introduction

- Software testing of SPL is difficult
  - Each application corresponds to a set of SPL features
  - Potentially many feature combinations
- Need method for developing test models
  - Manages a large number of feature combinations
  - Reuses test assets, such as test cases and test models

Software Product Line Development Processes



Detail of Application Engineering Process



2. Related Work: Requirements-based Testing for Software Product Lines

McGregor, 2001
Introduced process and requirements-based test models for an SPL

Bertolino et al, 2003
Applied category partition testing to product line use cases

Nebut et al, 2003
Developed an SPL testing strategy based on use case contracts

Kamsties et al, 2003
Developed an SPL testing strategy based on use case activity diagrams

Geppert et al, 2004
Investigated the use of decision trees to customize SPL tests

Reuys et al, 2005
Developed Kamsties et al work with ***Scen***ario Based ***TE***st case *D*erivation

Olimpiew, Gomaa, 2005
Described a testing process and test models for an SPL

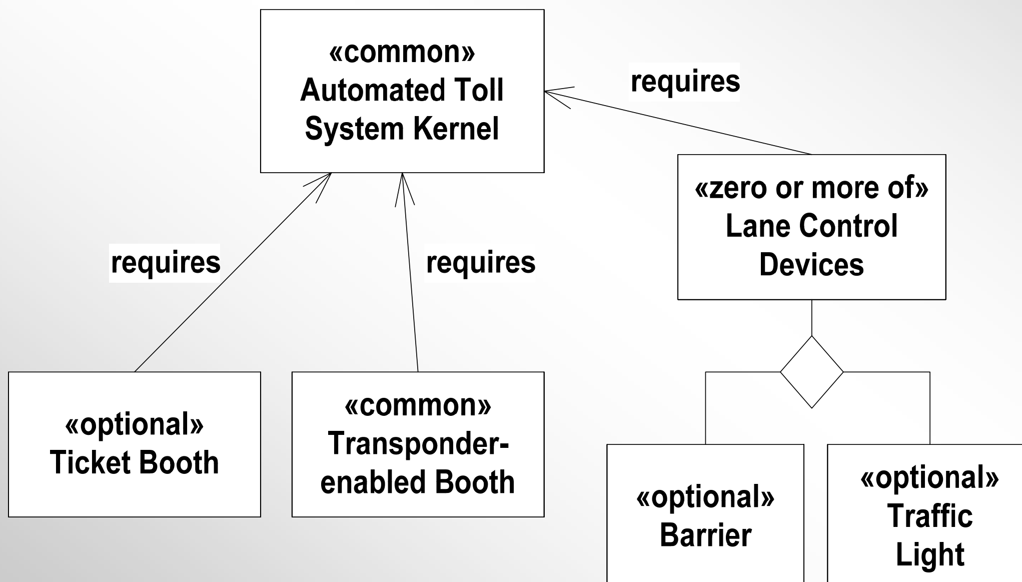
3. Requirements-based Test Models for an SPL

- During SPL engineering
  - Develop requirements models using PLUS (***P**roduct **L**ine **U**ML-Based **S**oftware Engineering*) (Gomaa, 2005)
    - Create feature model
    - Create use case model
    - Determine feature to use case relationships
  - Determine which target systems (or feature combinations) to test
  - Develop customizable test models
    - Phase I: Create activity diagrams from use cases and feature model
    - Phase II: Create decision tables from activity diagrams
    - Phase III: Create test templates from decision tables

Develop requirements models using PLUS

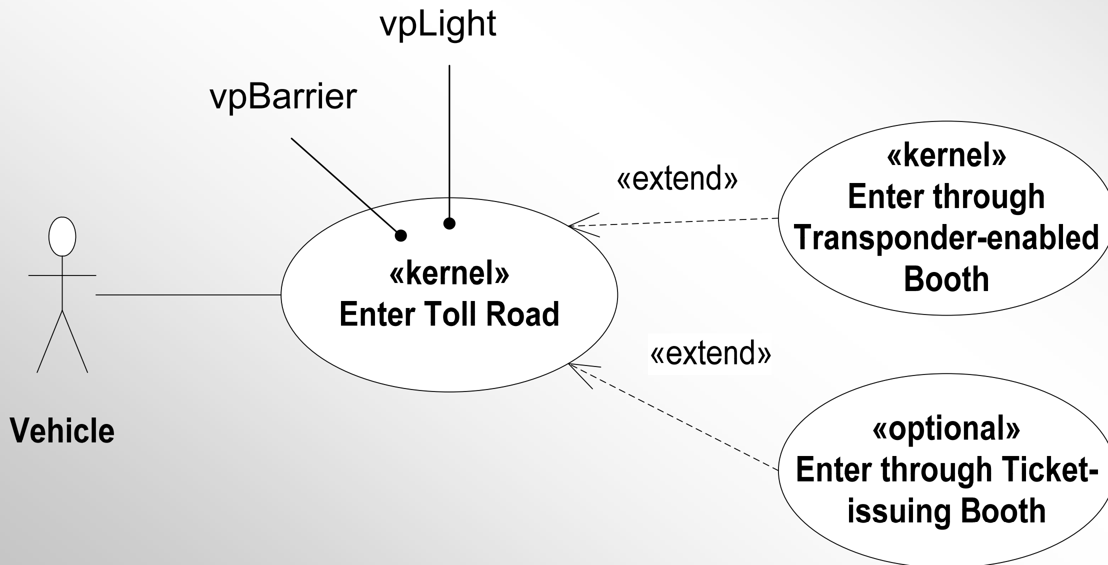
– Create feature model:

– Identify features, feature relationships, and feature reuse categories (common, optional, alternative)



Develop requirements models using PLUS

- Create use case model
- Identify use cases, use case relationships, and use case categories (kernel, optional, alternative)
- Identify use case variation points



Develop requirements models using PLUS

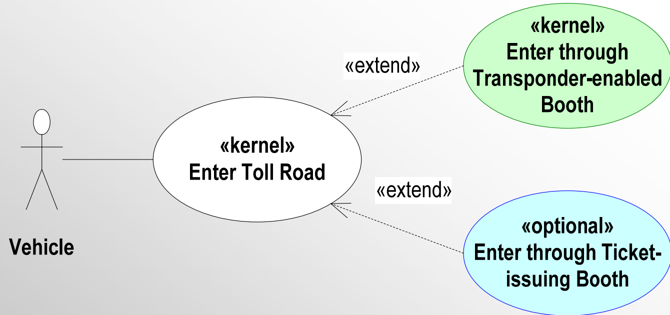
- Determine feature to use case relationships

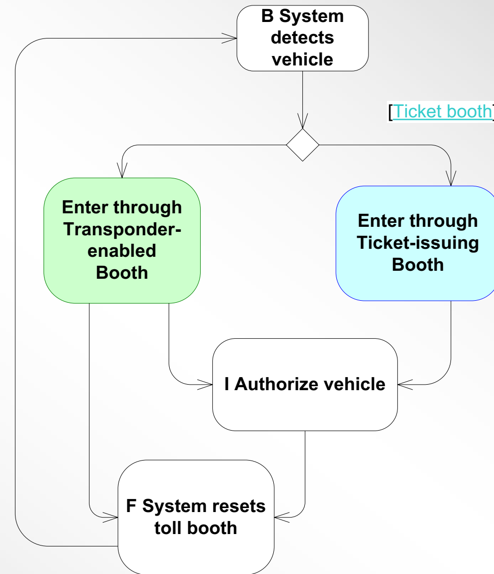
Feature name	Feature category	Use Case Name	Use Case Category / Variation point	Variation point name
Automated Toll System Kernel	Common	Enter Toll Road	Kernel	
Transponder-enabled Booth	Common	Enter through transponder-enabled booth	Kernel	
Ticket Booth	Optional	Enter through ticket issuing booth	Optional	
Barrier	Optional	Enter toll road	Vp	vpBarrier
Traffic Light	Optional	Enter toll road	Vp	vpLight

Develop customizable test models

Phase I: Create an activity diagram for each base use case

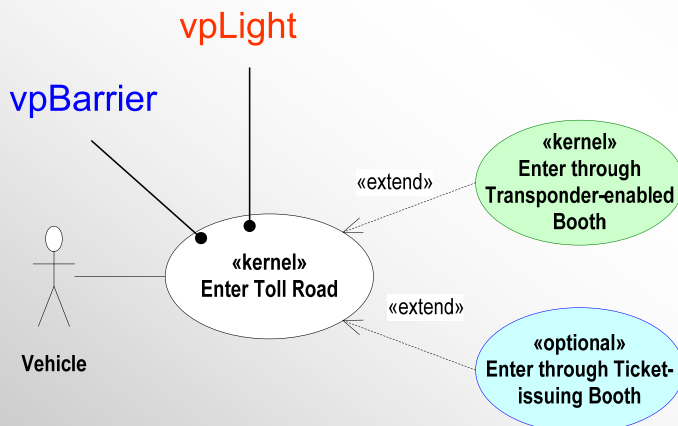
- Combine activities of extension use cases
- Map features to use case activity diagrams with feature conditions

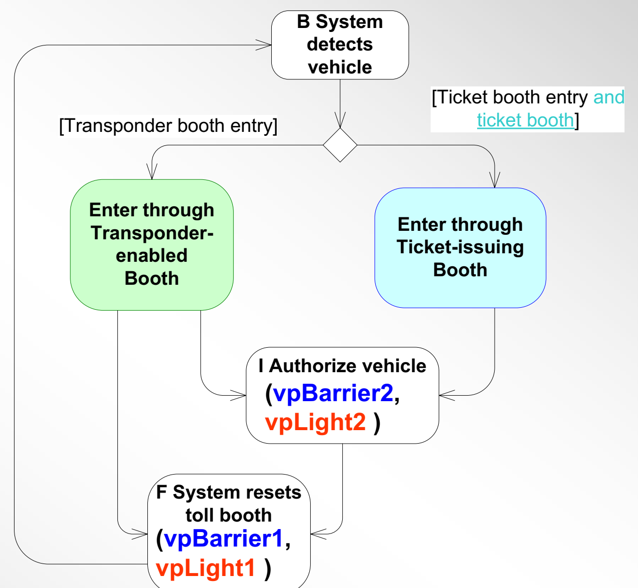




Develop customizable test models

– Map use case variation points to activity diagram
– Identify and name execution conditions





Develop customizable test models

- Map features to variation point parameters in activity diagram

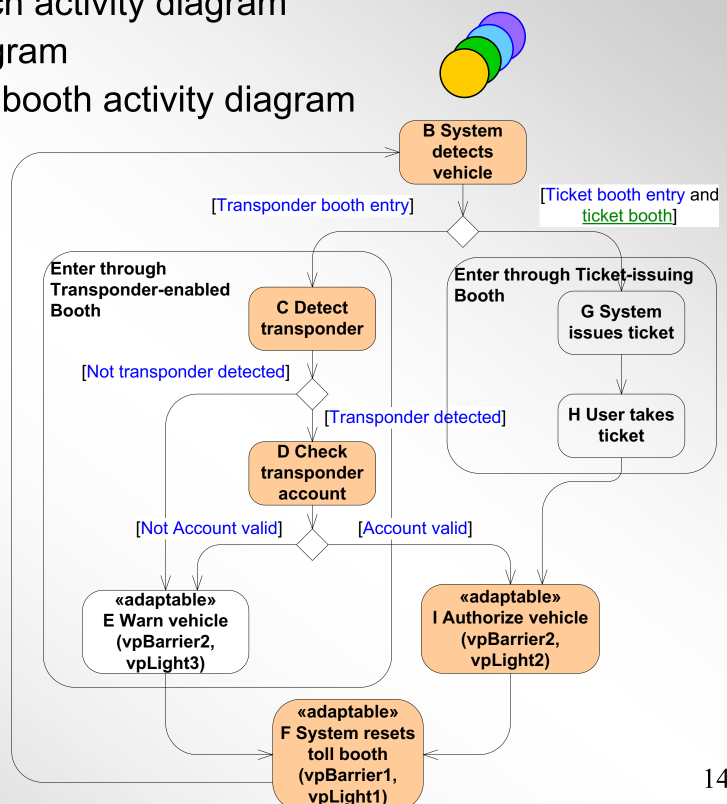
Feature	Variation point name	Parameter name	Parameter value
Barrier	vpBarrier	vpBarrier1	Lower barrier
		vpBarrier2	Raise barrier
Traffic Light	vpLight	vpLight1	Turn light red
		vpLight2	Turn light green
		vpLight3	Turn light amber

Develop customizable test models

Phase II: Create Decision Tables from Activity Diagrams

- Create one decision table for each activity diagram
- Trace paths from the activity diagram
- There are four paths in Enter toll booth activity diagram

Path → Conditions↓	1 BCDIF
Feature cond.	
barrier	{T, F}
traffic light	{T, F}
ticket booth	-
Execution cond.	
Trnsp booth entry	T
Ticket booth entry	-
Trnsp detect	T
Acct valid	T



Develop customizable test models

- {T, F} feature condition entries refer to adaptable paths
- Explicitly enumerating all feature condition values for path BCDIF would result in 2^2 paths
- Explicitly enumerating all feature condition values for all paths would result in 2^4 paths

Path → Conditions↓	1 BCDIF
Feature cond.	
barrier	{T, F}
traffic light	{T, F}
ticket booth	-

Adaptable path

vs.

Path → Conditions↓	1’ BCDIF	1’’ BCDIF	1’’’ BCDIF	1’’’’ BCDIF
Feature cond.				
barrier	T	T	F	F
traffic light	T	F	T	F
ticket booth	-	-	-	-

Explicitly enumerated feature condition entries

Develop customizable test models

Phase III: Create test templates from decision tables

- Each path becomes a test template
- A test template is a sequence of
  - Controlled actor inputs
  - Expected system actions
- Identify locations of variation in test template
- These locations refer to parameters in the feature to variation point table

Example of customizable test template

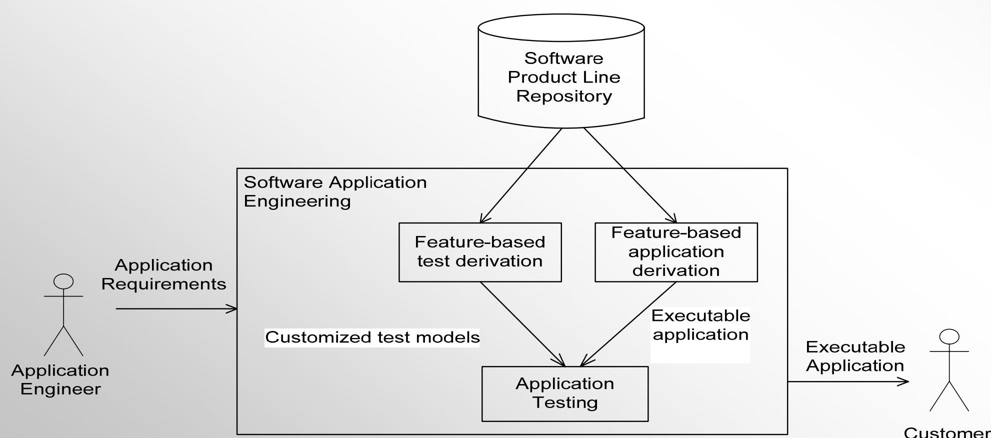
Name	1. Enter Toll Road via Transponder booth - Main
Feature conditions	$Barrier = {T, F} $TrafficLight = {T, F} TicketBooth = undefined
Execution conditions	TransponderDetected = True AccountValid = True
Test steps	
B	<Input> System detects customer approach
C	<Input> System detects and writes entry data to transponder (in location, in time)
D	System checks transponder account
I	<Output> System authorizes vehicle to pass $vpBarrier2 = {"<output> System raises barrier", null} $vpLight2 = {"<output> System turns light green", null}
F	<Output> System detects that vehicle has passed and resets booth $vpBarrier1 = {"<output> System lowers barrier", null} $vpLight1 = {"<output> System turns light red", null"}

Barrier Feature

Traffic Light Feature

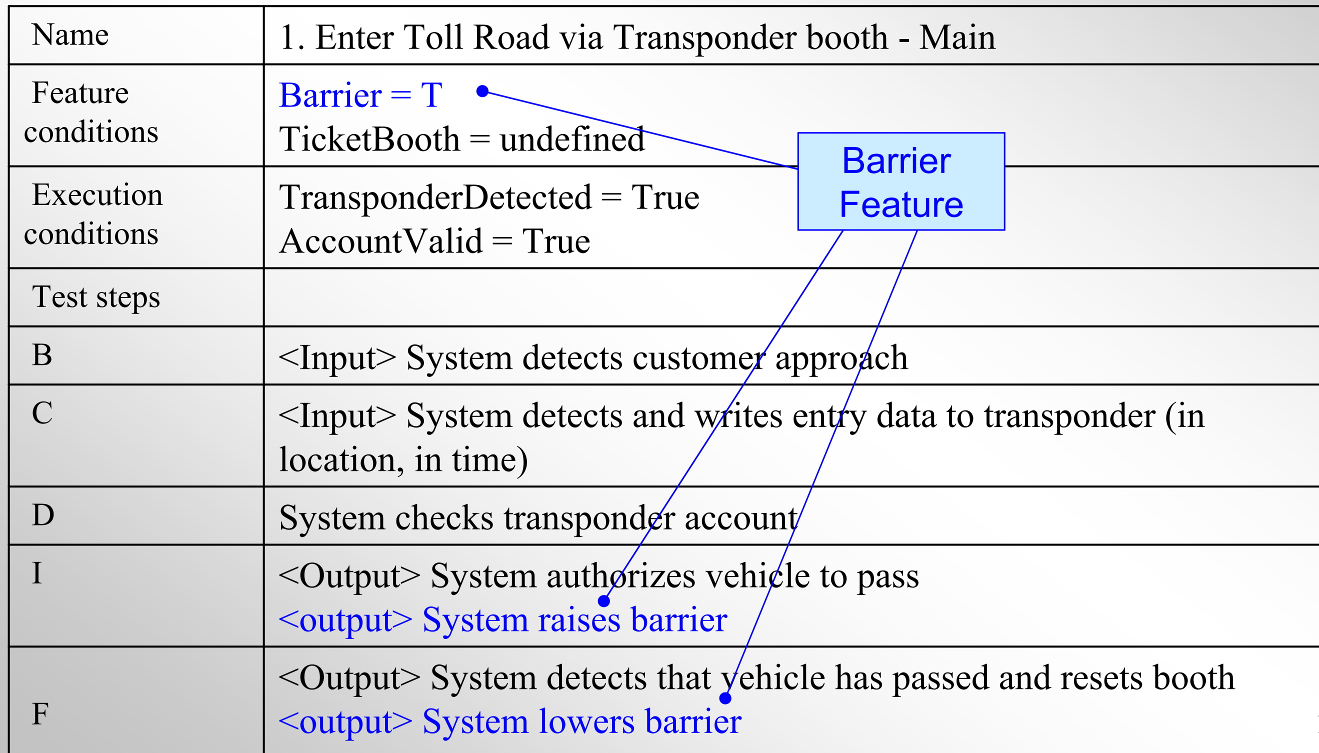
Application Engineering: Customize test template

- Select features for an application of the SPL
  - Customize requirements models
  - Customize application implementation
  - Customize test templates
- Select test data to satisfy the execution conditions of the templates
- Run test suite on application



Customized Test Template

- Barrier feature selected, Traffic light not selected
- Look up and replace parameters



Name	1. Enter Toll Road via Transponder booth - Main
Feature conditions	Barrier = T TicketBooth = undefined
Execution conditions	TransponderDetected = True AccountValid = True
Test steps	
B	<Input> System detects customer approach
C	<Input> System detects and writes entry data to transponder (in location, in time)
D	System checks transponder account
I	<Output> System authorizes vehicle to pass <output> System raises barrier
F	<Output> System detects that vehicle has passed and resets booth <output> System lowers barrier

4. Application of Testing Method to an Automated Toll System Case Study

- Object of study
  - First three phases of the test design method:
    - Phase I: Create activity diagrams
    - Phase II: Create decision tables
    - Phase III: Create test templates
- Purpose of the study
  - Is method practical?
- Participants
  - Five graduate students
- Models
  - Three different requirement models of the Automated Toll System

GEORGE MASON UNIVERSITY

Approach and Evaluation

- Give detailed instructions on each phase
- Allow two weeks for each phase
- Observe which instructions caused most misunderstandings
- Observe whether participants are able to
  - Complete the instructions correctly
  - Complete models within the allotted time.
- Survey participants

GEORGE MASON UNIVERSITY

Results

- Use case models consisted of 8-14 use cases
- Feature models consisted of 20-24 optional and alternative features
- Most participants were able to create and correct the models within the allotted time
- Most misunderstandings were on how to map variation points to the test models
- Method can be applied and understood with adequate training

GEORGE MASON UNIVERSITY

Conclusions

- Method of developing customizable test models for an SPL
  - Manages feature combinations
  - Creates reusable test templates
  - Traces features to test models
  - Benefits
    - Reduce # of test templates
    - Flexible to handle any feature combination
- Method applied to a realistic automated toll system case study
- Method needs more automation in order to make it more useable

A Verification Approach for Crosscutting Features based on Extension Join Points

Roberta Coelho[1], **Vander Alves[2]**, Uirá Kulesza[1], Alberto Costa Neto[2], Alessandro Garcia[3], Arndt von Staa[1], Carlos Lucena[1], Paulo Borba[2]

[1]*Informatics Department (PUC-RiO), Brazil*

[2]*Informatics Center –Federal University of Pernambuco, Brazil*

[3]*Lancaster University, Computing Department, Lancaster - United Kingdom*

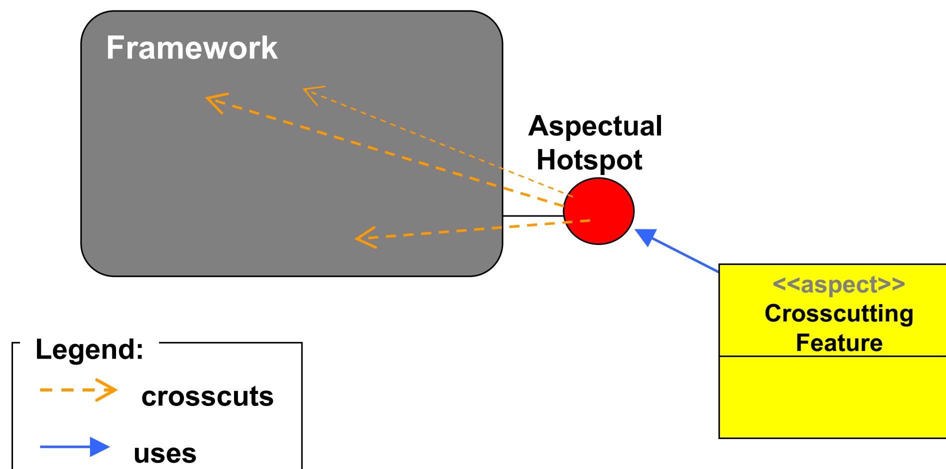
Motivation

- Crosscutting features in SPL;
- Modularization of crosscutting features;
- Testing crosscutting features;
- AOSD modularizes crosscutting concerns;
- AOSD: new challenges to software testing.
- New AO constructs → sources of new types of faults.

© LES/PUC-Rio

Key Idea

- Represent crosscutting features as aspects;
- Interfaces between aspects and the framework;



© LES/PUC-Rio

Key Idea

- Add information to such interfaces (i.e contracts)
- To support crosscutting features verification activities:
  - guarantee that framework invariants are preserved;
  - can be used in the definition of specific test criteria.
  - can be used as test oracles.

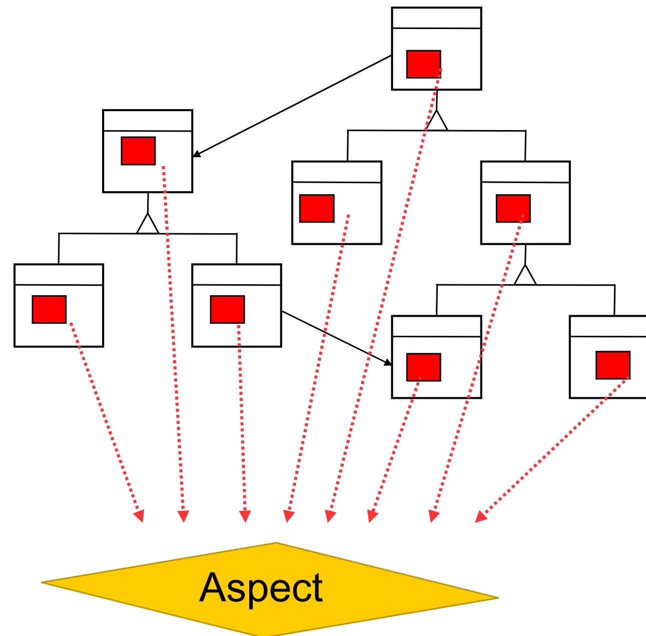
© LES/PUC-Rio

Back Ground: Aspects

LES

Laboratório de Engenharia de Software

- Aspects: a new abstraction to modularize tangled and spread concerns;



© LES/PUC-Rio

Back Ground: AspectJ Concepts

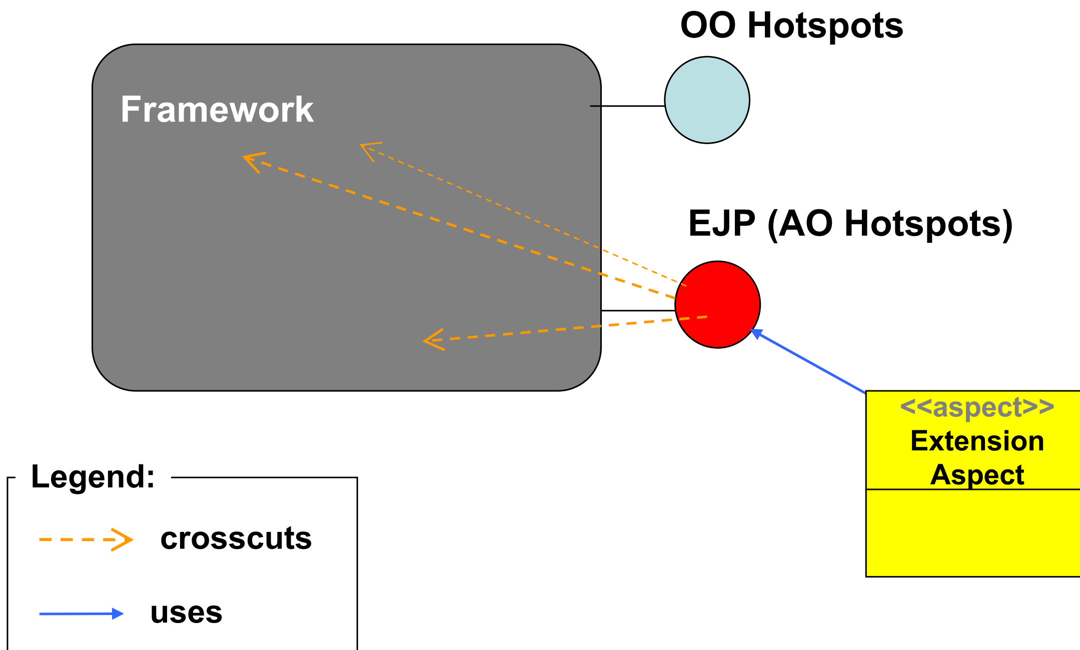
LES

Laboratório de Engenharia de Software

- The aspect abstraction in AspectJ is composed of:
  - **Inter-type declarations** specify new attributes or methods to be introduced in specific classes.
  - **Joinpoints** are well-defined locations within the base code where a concern will crosscut the application.
    - Ex: method calls and method executions.
  - **Pointcuts** are constructs that match the joinpoints, and performs specific actions performed defined in an advice.
  - **Advice** is a special method-like constructs defined in aspects.

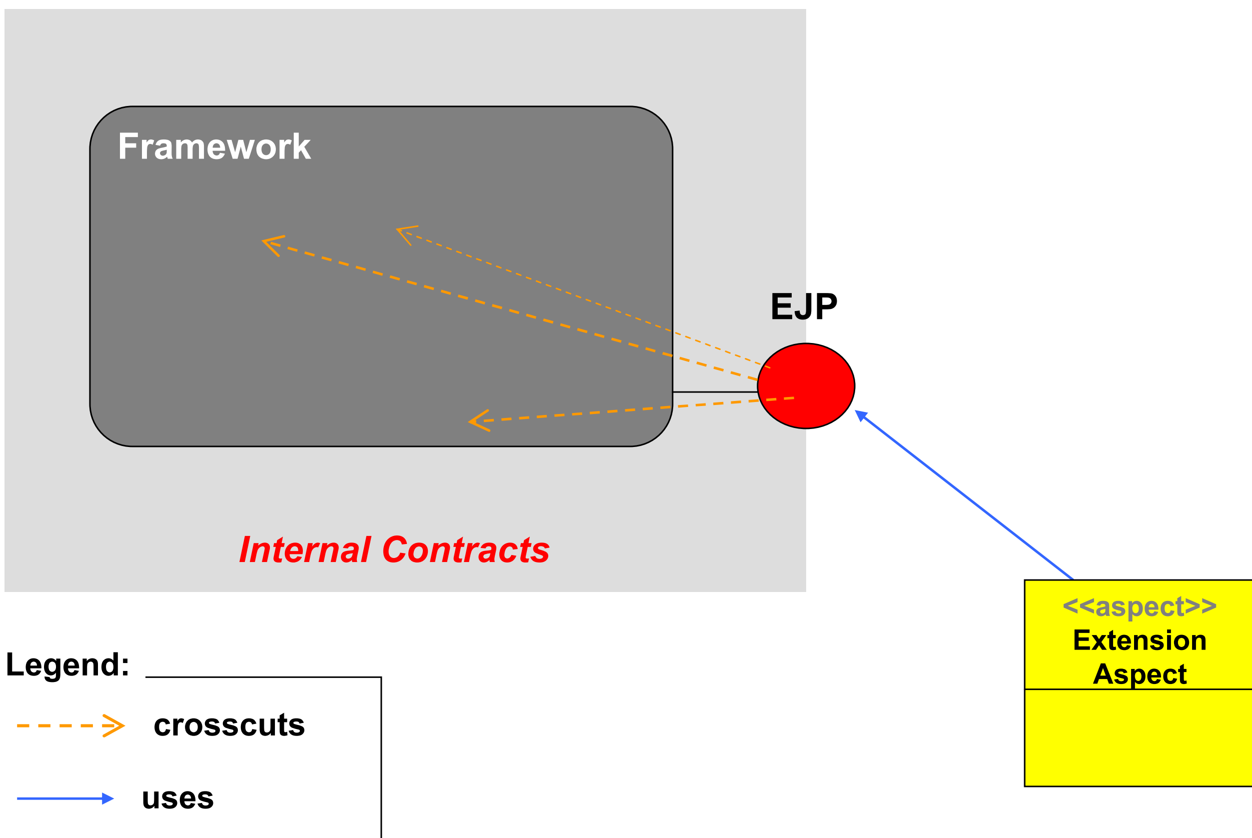
© LES/PUC-Rio

Back Ground: Extension Join Points



Laboratório de Engenharia de Software

EJP Structure – Framework Internal Contracts



© LES/PUC-Rio

EJP Structure – Framework Internal Contracts

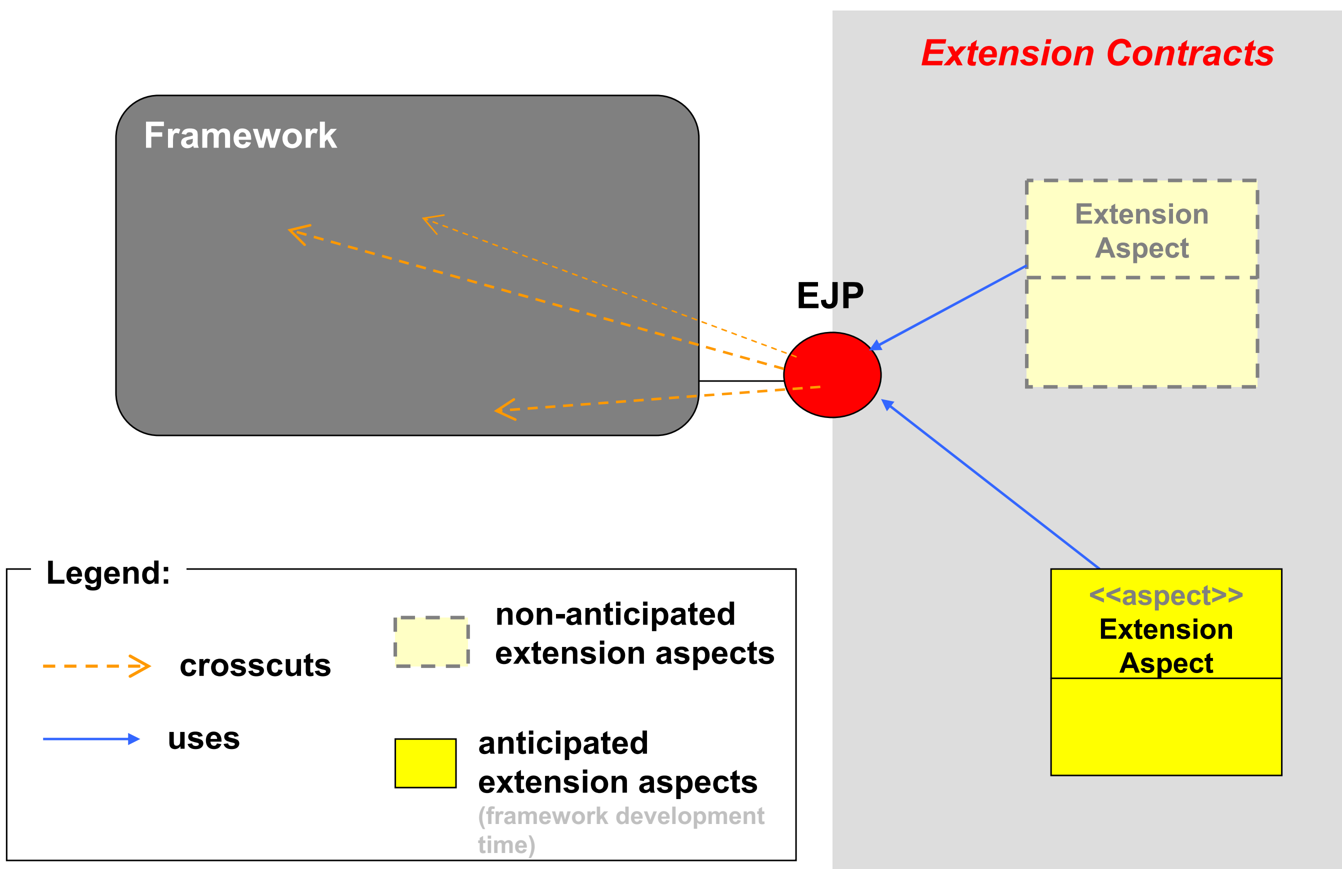
- They are classified in the following categories:

- **Structural Contract:** the framework should implement specific interfaces.

```
public abstract aspect EJP{
    declare parents: ClassA implements EventReporter;
    declare parents: ClassB implements EventReporter;
 ...
}
```

- **Behavioral Contract:** the framework EJPs comprises all and only the framework events that the EJP should expose.

© LES/PUC-Rio



EJP Structure – Framework Extension Contracts

- The following categories were defined:
- **Structural Contract:** Aspects has restricted access to framework elements.
- **Behavioral Contract:** framework invariants, pre/pos-conditions

Contracts expressed in AspectJ will not just be stated but they will be enforced during compilation time or runtime.

Case Study: Game SLP

- The overall structure: *Game Engine* framework;
- The *Game Engine* consists in a state machine;
- State changes according to the elapsed time and user input;
- State changes → various images should be re-drawn.

Laboratório de Engenharia de Software

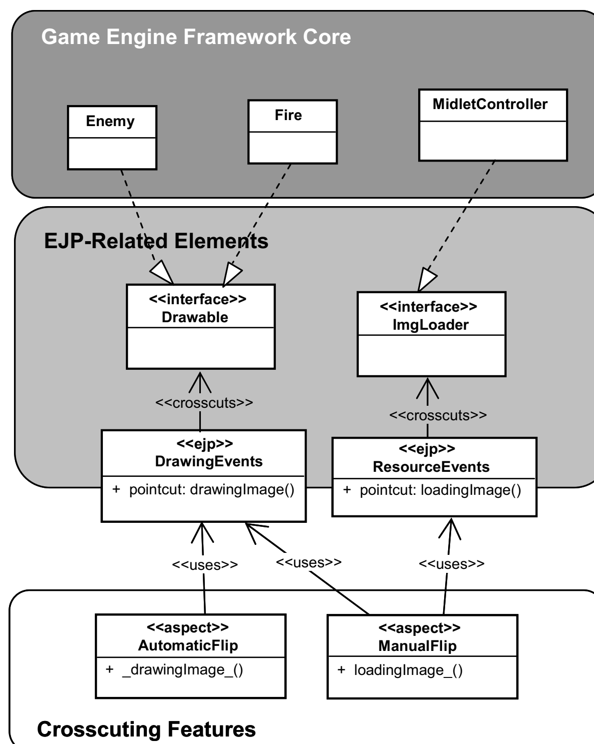
Case Study: Game SLP

- How an image can be flipped:
  - Some devices have built-in flip API and
  - To others, flipping algorithms have to be defined.

- The flipping feature is crosscutting:
  - it depends on image drawing events spread over a set of framework modules.

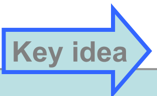
- We implemented this crosscutting feature using aspects and EJPs.

Game Engine



The Approach

- A verification approach for crosscutting features based on EJPs;
- According to the EJP-based development approach:
  - A set of EJPs and contracts are defined.
- EJPs are the basis for the verification approach.



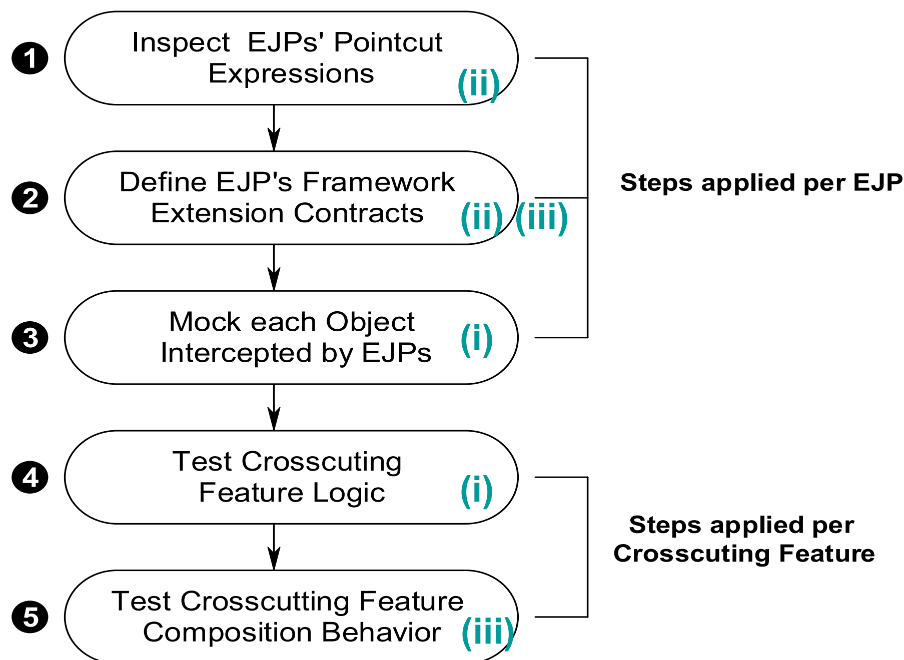
Laboratório de Engenharia de Software

Sources of Faults in AO Programs

i. crosscutting feature logic;
ii. inaccurate pointcuts;
iii. interaction between a crosscutting feature and the base code;
iv. faults in the base code;
v. interaction between the crosscutting features.

© LES/PUC-Rio

Approach's Steps



Legend:

- ● Step number
- (i..) Kind of fault detected at the step

Step 1: Inspect EJPs' Pointcut Expressions

LES

Laboratório de Engenharia de Software

- Manually check the inaccuracy of EJP pointcuts;
- Time consuming;
- But only the EJPs have to be inspected;
- The crosscutting features are not inspected;
- They reuse the EJPs pointcuts.

© LES/PUC-Rio

Step 2: Define EJPs' Framework Contracts

- EJP's Extension Contracts assures that:
  - crosscutting features will respect invariants, pre and post conditions.
- EJPs' Contracts can be checked:
  - during manual inspections,
  - verified at compilation time,
  - or runtime (test oracles).

© LES/PUC-Rio

Step 3: Mock each object intercepted by the EJPs

Laboratório de Engenharia de Software

- Developer should define mock objects for the code intercepted by the EJPs.
- Real and Mock objects should implement the same interface.
  - The EJP should refer to an object by it's interface.
- During tests:
  - EJP remain ignorant to whether it is intercepting the real object or the mock object.

© LES/PUC-Rio

Step 4: Test the Crosscutting Feature Logic

- The Mock objects used to enable the test of crosscutting features logic.
- At this step:
  - the crosscutting feature is weaved with the mock object;
  - and each method of the resultant component is unit tested.

© LES/PUC-Rio

Step 5: Testing Crosscutting Feature Composition Behavior

Laboratório de Engenharia de Software

- Crosscutting feature in the base code;
- The base control structure and behavior is possibly modified;
  - Existing test suite may be insufficient;
- A second test suite should be defined;
- New coverage criteria may be defined to verify the effectiveness of the test suite (ex. EJP-specific).

© LES/PUC-Rio

Conclusions and Future Work

LES

Laboratório de Engenharia de Software

- A systematic approach for detecting faults in crosscutting features.
  - Implemented by means of aspects and EJPs.

- We intend to refine it through:
  1. More case studies
  2. Analysis Aspects Interactions
  3. Metrics
  4. Tool support

© LES/PUC-Rio

ODC Crystallizes Test Effectiveness

Ram Chillarege
www.chillarege.com

10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA

© 2006

chillarege

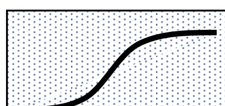
ODC: The sweet spot between the extremes of quantitative and qualitative analysis

QUANTITATIVE . . . . QUALITATIVE

ODC

- Counting
- Defect rates
- Specific
- Most defects, superficially
- Fast
- Low cost

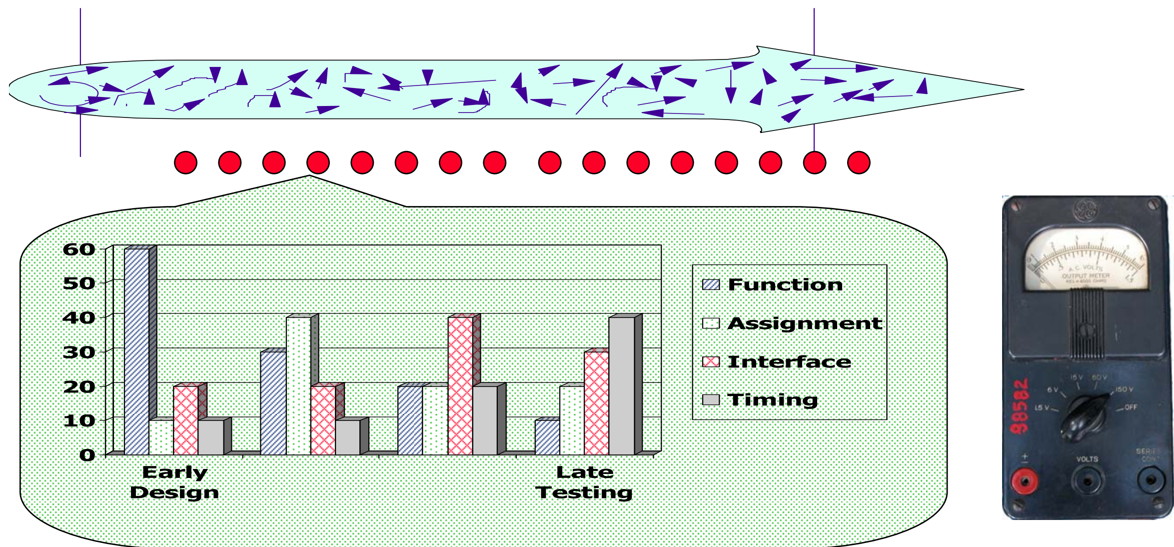
- Quality Circle
- Root Cause
- Open
- Few defects, thoroughly
- Slow
- High cost



© 2005

chillarege

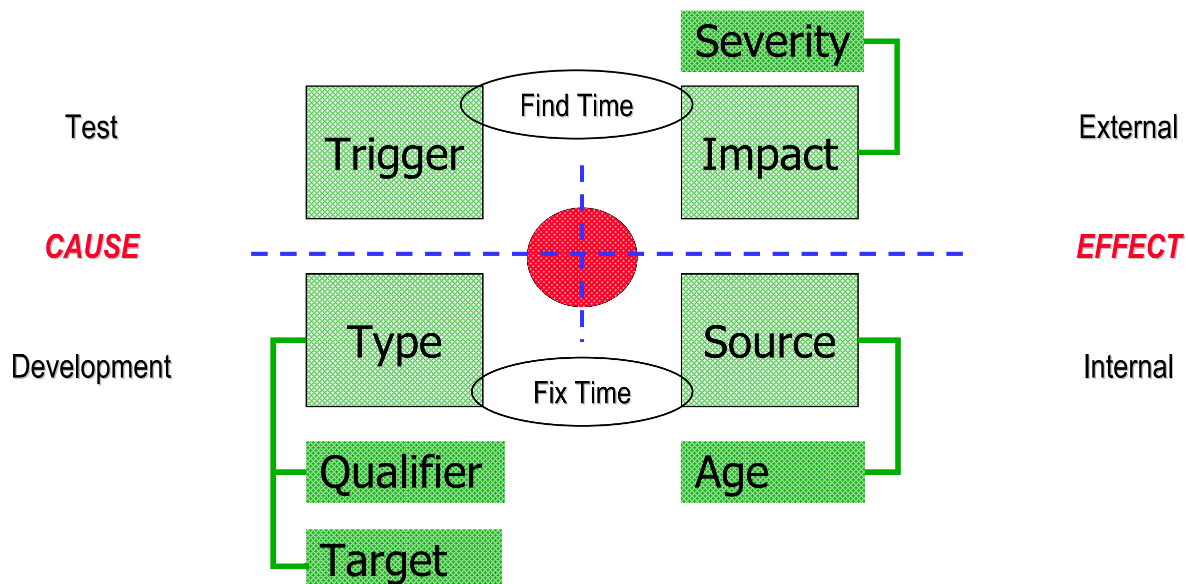
Change in the distribution measures the progress of the product through the process



© 2005

chillarege

ODC is captured across the defect life cycle- yields cause / effect understanding



© 2005

chillarege

ODC Attributes tell a story

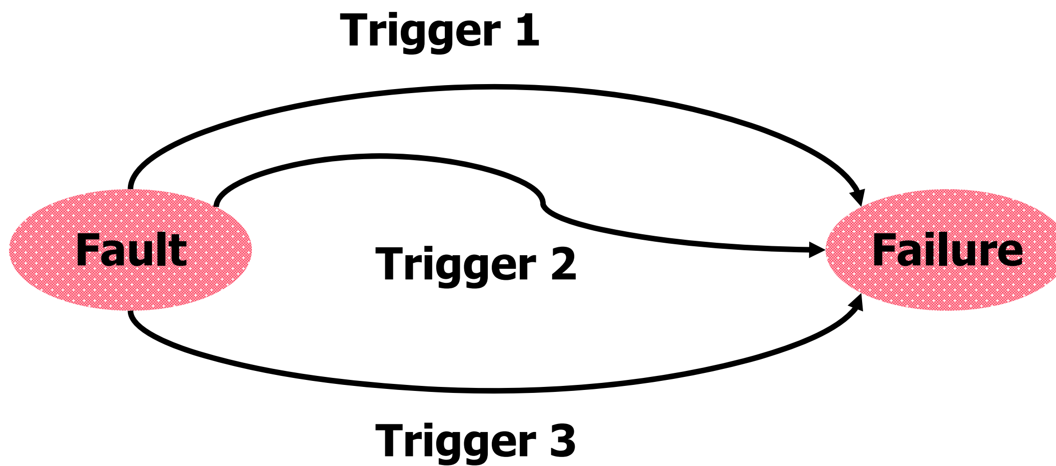
Attribute	***Meaning***	***Application***
Activity	Doing what?	Maps to work and skill (cause)
Trigger	How detected ?	Maps to Verification (cause)
Impact	What happened ?	Maps to Customer (effect)
Target	Which artifact ?	Maps to Development (cause)
Type	Fixed what ?	Maps to Development (cause)
Qualifier	Omission? co-mission?	Maps to Development (cause)
Source	Where ?	Subsets Codebase (cause)
Age	How old ?	Subsets Codebase (cause)

Understanding Triggers

Terms, definitions, concepts:

- **Fault**
- **Error**
- **Failure**
- **Defect**
- **Trigger**

ODC Trigger .. Catalyst that activates Faults into Failures



Contact Information

Chillarege Inc.
Ram Chillarege
+1 (917) 790 9390
1505 Consett Ct.
Raleigh, NC 27613

ram@chillarege.com
www.chillarege.com

SPLiT 2006
Breakout Session

SPLiT 2006

Results and Open Issues from SPLiT '04 and '05

- Manage the complexity of the test space
  - → be conservative on the scope of variability in the core assets
- Match techniques and methods for development and testing
  - → extend PL infrastructure towards testing
- Design for testability
  - → develop a score for which components are testable
  - → use aspects/code instrumentation for test interfaces etc. (research)
  - → use the potential from automated tests (industry)
- Align test strategy with architecture
  - → whichever comes first determines/limits the other
  - → how to check confirmance (during evolution)?
- Is it better not to reuse tests (because of cross checking)?

SPLiT 2006

We want each Group to develop an...

Elevator Statement (1 Slide)

- **Short, clear, compelling description of the topic:**
  - **How would you describe the topic in a few sentences?**
  - **What is the problem?**
  - **Why is it important? Why should we spend effort on it?**
- **Technology gap:**
  - **Where are we in industry and where are we in research?**
  - **Where would we like to be?**
  - **What are concrete problems/challenges?**
    - **Can you prioritize the problems/challenges?**
    - **Do you have ideas of possible solutions?**
- **Related work:**
  Is there other work/areas/projects that might be relevant to look at/investigate/consult?

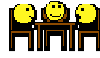
Organization of the Breakout Groups

- **Each discussion group should have one person 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**
  → including names of all participants
  → ppt preferred

And now …

Let’s start working!



SPLiT Workshop Aug 22nd

Design for Testability

Jamie Williams, Cummins
Kathrin Scheidemann, BMW Car IT
Christian Tischer, Bosch
Peter Knauber, Univ. Mannheim

Testability

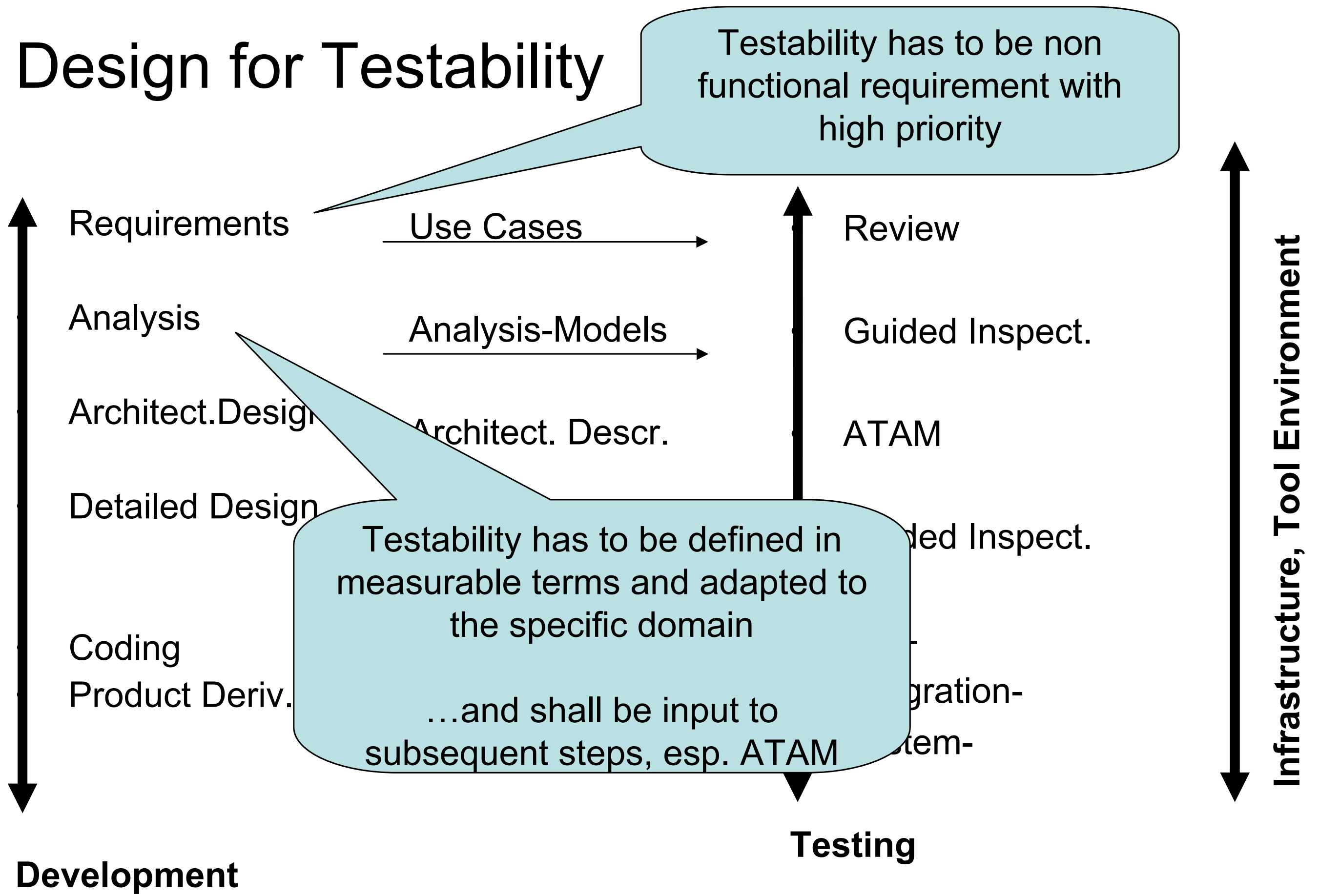
"The ease with which software gives up its faults" (John McGregor)

Some thoughts in the beginning…

- What Test coverage is required?
  Exhaustive testing is not possible within Product Lines (?)
  Exhaustive testing is not necessary within Product Lines (?)
- Portability, reusability of tests: what are required variation points in the test, e.g. parameterizing data

Examples for possible measures/criteria

- Encapsulation: Modularization with separation of concerns, encapsulation of variability
- Measurement of Coupling and Cohesion as criteria for testability
- Abstraction: Independency e.g. of underlying Layers


Design for Testability
Testability has to be non functional requirement with high priority
Requirements
Use Cases
Review
Analysis
Analysis-Models
Guided Inspect.
Architect.Design
Architect. Descr.
ATAM
Detailed Design
Testability has to be defined in measurable terms and adapted to the specific domain
…and shall be input to subsequent steps, esp. ATAM
Coding
Product Deriv.
Testing
Development
Infrastructure, Tool Environment
Based on: John McGregors in Tutorial4

Results Breakout Discussion Group on Test Strategy

Participants

- Ram Chillarege
- Georg Grütter (Bosch)
- Ronny Kolb (Fraunhofer IESE)
- Tim Trew (Philips Research)
- Kentaro Yoshimura (Hitachi)

Problem

- Define a competitive test strategy
  - when
  - what
  - how
  - to what extent
  - by whom
- a particular product line is tested at least cost to gain the most

Gaps

- Lack of information about applied test strategies
- Experience base with defect data
- Lack of understanding of business impact
- Composability in testing space

Elements of Test Strategy

- Assessment
- Action

Test Strategy Assessment

- Results are
  - Effectiveness of test and risk
  - Criticality of customer impact
  - Cycle time and cost

Test Strategy Action

- Balance between testing core assets over their variation space and testing applications
- Balance the granularity of the artifact for observability and controllability
- Optimize testing for cost and/or time

Adapting Techniques and Methods for Testing Software Product Lines

Erika Olimpiew
Hotae Kim
Rob Reckzine
Vander Alves

Problem Statement

- Variability in SPLs
  - Is hard to manage
  - It is difficult to predict and manage the many feature combinations
- There are many methods and variability mechanism for testing single systems
  - How can these be extended for testing SPLs?

Where we are

- Methods
  - RUP
  - Agile
  - IEEE standards
  - Model-based testing
  - Formal methods (model checking)
- Variability mechanisms
  - Compiler directives
  - Inheritance
  - Parameters
  - Templates
  - Separation of concerns

Where do we want to be

We want to efficiently manage the following testing problems in an SPL

- Test configuration management problems
- Feature interaction problem
- We want concrete guidelines for choosing
  - methods for testing
  - variability mechanisms for test artifacts

Related work

- Crosscutting Interfaces (Sullivan et at, 2005)
- AOP (Kiczales et al, 97)
- XVCL (Jarzebek,90s)
- Alloy (Jackson et al,2006)

Third Workshop on Software Product Line Testing

August 22, 2006
Baltimore, Maryland, USA

Agenda

- Keynote: Georg Grütter, Robert Bosch GmbH: Challenges for Testing in Software Product Lines
- Short Paper Presentations
  - A Reuse Technique for Performance Testing of Software Product Lines
  - Considerations in Developing a Controls Product Line Test Architecture Implementation
  - Customizable Requirements-based Test Models for Software Product Lines
  - On Testing Crosscutting Features using Extension Join Points
- Invited speaker: Ram Chillarege, Chillarege Inc.: ODC crystallizes Test Effectiveness
- Three Breakout Sessions, Result Presentation and Discussion, Wrap-Up

SPLiT 2006

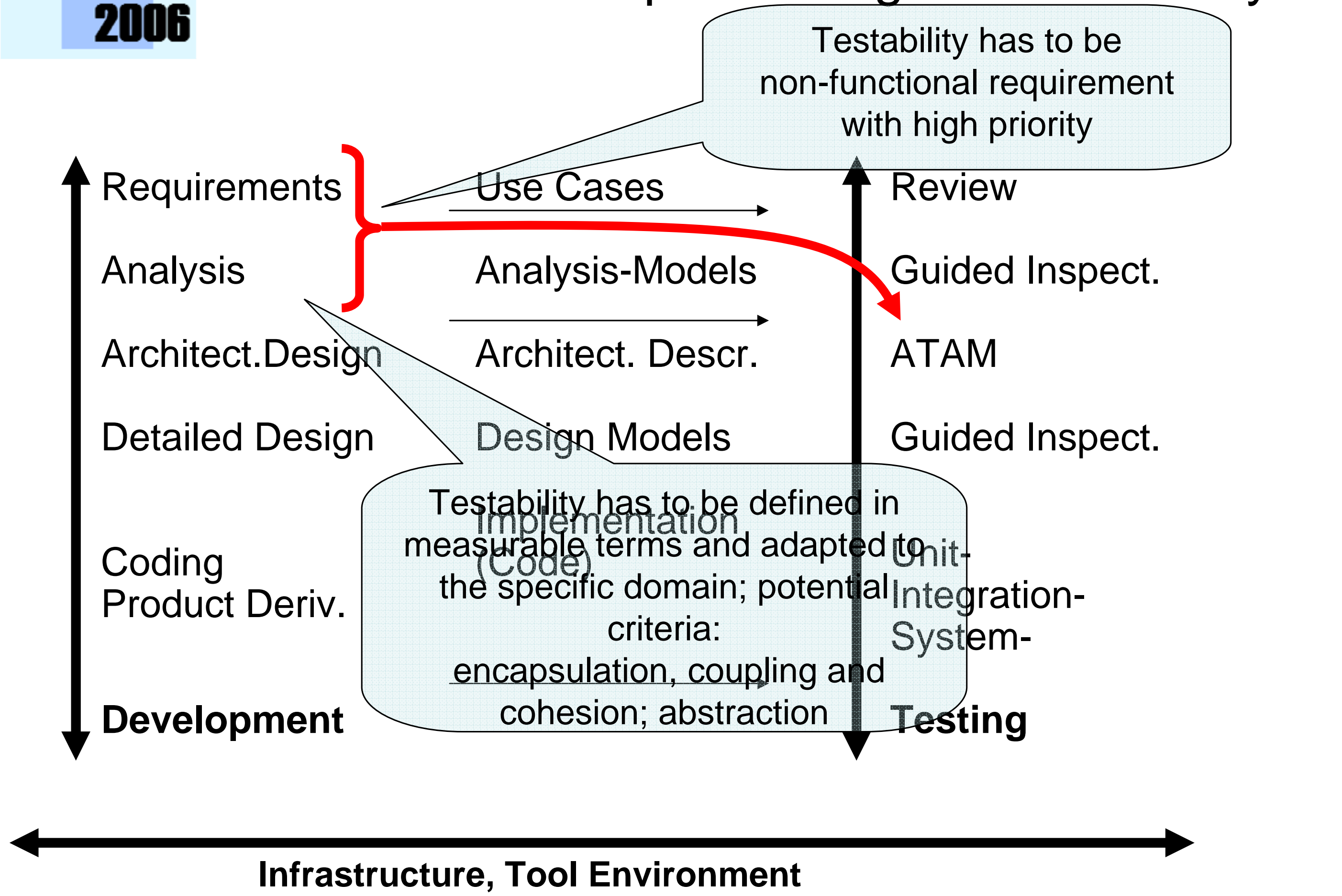
Discussion Group on Test Strategy

- Problem:
  Define a competitive test strategy
  when, what, how, to what extent, and by whom
  a particular product line is tested at least cost to gain the most
- Challenges
  - Lack of understanding of business impact of testing
- Assessment of Effectiveness of Test and Risk
  - Composability in testing space
- Criteria for determining the Test Strategy
  - Balance between testing core assets over their variation space and testing applications
  - Balance the granularity of the artifact for observability and controllability
  - Optimize testing for cost and/or time

SPLiT 2006

Discussion Group on Adapting Techniques and Methods for Testing Software Product Lines

- Problem: Variability in SPLs
  - Test configuration management problems
  - Feature interaction problem
- There are …
  - Methods:
    RUP, Agile, IEEE standards, Model-based testing, Formal methods (model checking), etc.
  - Variability mechanisms:
    Compiler directives, Inheritance, Parameters, Templates, Separation of concerns, etc.
  - Many methods for testing single systems
- We need concrete guidelines for choosing
  - methods for testing
  - variability mechanisms for test artifacts


SPLiT 2006
Discussion Group on Design for Testability
Testability has to be non-functional requirement with high priority
Requirements
Analysis
Architect.Design
Detailed Design
Coding
Product Deriv.
Development
Use Cases
Analysis-Models
Architect. Descr.
Design Models
Implementation (Code)
Review
Guided Inspect.
ATAM
Guided Inspect.
Unit-
Integration-
System-
Testing
Testability has to be defined in measurable terms and adapted to the specific domain; potential criteria: encapsulation, coupling and cohesion; abstraction
Infrastructure, Tool Environment
[Based on: John McGregor, Tutorial 4]