Product Line Variability with Elastic Components and Test-Driven Development

George Kakarontzas
Dept. of Informatics
Aristotle University
Thessaloniki, Greece, and
Dept. of Comp. Science and Telecom.
T.E.I. of Larissa
Larissa, Greece.
gkakaron@teilar.gr

Ioannis Stamelos
Dept. of Informatics
Aristotle University
Thessaloniki, Greece.
stamelos@csd.auth.gr

Panagiotis Katsaros
Dept. of Informatics
Aristotle University
Thessaloniki, Greece.
katsaros@csd.auth.gr

Abstract

In this work we present a systematic approach for the creation of new variant software components via customization of existing core assets of a software product line. We consider both functional and quality variants and address the issue of a controlled creation of variants which considers the reference architecture and its co-evolution with a number of other artifacts including components and functional and quality test suites. Furthermore we discuss the relationship between the popular agile practice of Test-Driven Development (TDD) and how it can be used to assist the evolution of software components of a software product line.

1 Introduction

A product line approach to software development [1] exploits the commonalities among several products to introduce a systematic reuse approach in which several important artifacts are considered core assets from which several different products are built. Core assets include software components, the product line architecture, test suites, documentation manuals, specifications etc. During product development the core assets are specialized to specific product assets by varying them to fit the specifications of the new product [2]. In general there are two methods to introduce variability in product line components: configuration and customization. Configuration assumes that all possible component variants have been considered and that source code and other appropriate configuration mechanisms have been built in the component to allow its "switching" to new product variants. This is the preferred way for novice users since it requires zero software development for new product assets however it has several disadvantages including (possibly harming) "dead" code and also very expensive upfront development efforts to consider all possible variants and introduce configuration mechanisms to realize them in the core asset base. The other alternative is customization: Customization is the development of new product assets from existing core assets to create new variants for the new products. Although the creation of new product assets (which may be included in the core assets base) requires some effort this effort, if guided by basic software engineering principles of modularity such as high cohesion and low coupling, is not necessarily large and it avoids the previously mentioned pitfalls of configuration.

The product line's features and variabilities are captured in a feature model [3] which provides the features of the product line including mandatory and optional features of products. These features are implemented by the components of the reference architecture of the product line. If we decide to evolve the product line to include a new feature or new variants of features, then new components must be built or existing components must be customized to introduce the required variation. The reference architecture will also vary accordingly.

Challenges to this evolution are how we proceed in a way that guarantees the minimum effort for the customization and how we ensure that the quality and functionality of the whole product is not undermined during this evolution step. Test-Driven Development on the other hand is an agile practice that essentially reverses the development cycle for an application from design-code-test to test-code-design [4] . The developer first writes a test which will be the oracle of success for the to-be developed code. Initially the test does not pass since the code under test has not been developed yet. The developer then writes the simplest possible code to make the test pass. After the test has passed the devel-

978-0-7695-3514-2/08 $25.00 © 2008 IEEE
DOI 10.1109/CIMCA.2008.84

oper refactors the code to eliminate "bad smells" (e.g. code duplication).

In the rest of this paper in Sec. 2 we describe the concept of elastic components and suggest its usage for the evolution of the software components of a product line. Next in Sec. 3 we describe how Test-Driven Development (TDD) can be used constructively to support the implementation of elastic components. In Sec. 4 we present some related work and finally in Sec. 5 we give future research directions and conclude.

2 Elastic components

To address the aforementioned challenges to component evolution the concept of *elastic components* has been proposed in [5]. Elastic components are not just a single component but a hierarchy of components with a common root which is a component of the reference architecture. The children of this hierarchy are variants of the root component which is called *pure*. The characteristic of the variants is that they have some functional or quality additions as compared to the pure component. For example a variant might provide some new functionality or an extra fault tolerance feature. Components are considered as composites of objects with one object playing the role of the façade to the internals of the components which are not accessible from other components. The component provides its services through provided interfaces and might require additional services through its required interfaces. The component is also accompanied with metadata describing essential information for its successful deployment such as the version number and its requirements.

The initial construction of a software component is no different than the construction of a simple cohesive object. We start by defining scenarios that we want to achieve using this component. These scenarios serve as the specification and verification mechanism under which a component is specified initially and verified after its implementation. The component methods form a provided interface which is used by the scenarios to verify the component's operation. Internally the component contains in the first phase just a cohesive object providing the implementation of the provided interface. The component may use for its implementation the services of other external components which are the required interfaces of the component. Having this first cohesive component, which is called *pure component*, we can then extend it by adding more scenarios which are new functional features or quality enhancements. To implement these additional features we introduce new objects inside the component which collaborate with the initial cohesive object to achieve the new functionality. During this evolution we look for disharmonies in the component's internal classes which will undermine the component's future evolution such as low cohesion and high coupling and use metrics to highlight possible anomalies (e.g. God classes with excessive number of responsibilities and Data classes with no or few responsibilities) [6]. The result is a new variant of the original pure component as long as the new component still succeeds in verifying all the previously defined scenarios. The new variant component is less cohesive than the original component but more useful since it provides more features or quality enhancements. It is crucial however to notice that usefulness is fitness for a particular use and therefore the variant component is more useful to a client that requires the additional features or quality enhancements as those are specified with the additional scenarios. For another client the new features may be completely inappropriate. If we were keeping only the new variant of the original component in our hierarchy then a new client requiring some other type of features would have to cancel some or all of the newly introduced features of the variant component. This is known as negative variability [7]. To avoid this we keep the old components in a repository and the designer can always pick the best place in the hierarchy from which to start the creation of a new component. This place would be the component in the hierarchy that provides the maximum number of scenarios required also by the new component without requiring the cancelation of any of the scenarios of the old component. The evolution for the creation of the new component in our approach is supported by three factors:

1. The objects inside each component are highly cohesive and low coupled enabling the evolution with ease

2. The user selects the component that satisfies already the maximum features that he requires without requiring the cancelation of any features, and

3. The approach is supported by a repository that enables the discovery and selection of the most appropriate component for extension

The process described for the creation of variants is depicted in Fig. 1. Notice that although Fig. 1 introduces just one additional class for the variants this is not necessary nor essential. More classes or no additional classes can be introduced and any variability paradigm can be employed such as templates, inheritance and interception. Also we depict only first level variants, but more variants can be created by extending existing variants if all the scenarios of a variant are still required. During the process of component evolution in the elastic component hierarchy it is possible that scenarios of the pure component will be violated. This is an indication of a different concept altogether: a new component that although conceptually similar to the elastic component that we tried to extend is not sufficiently similar to

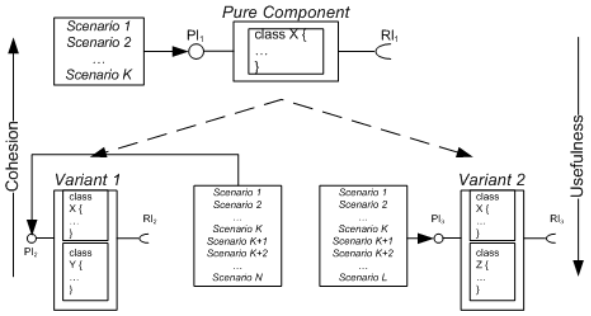

Figure 1. The creation of variants

be considered part of the hierarchy since it violates the basic assumptions for the functionality and quality of the pure component. In these cases a new pure component will be created with a new elastic component hierarchy. During the evolution of the elastic components other core assets evolve as well: (1) The reference architecture is updated to reflect the new component variants, (2) The newly developed test suites and other specifications are all saved in the elastic component repository for future evolution efforts as well as for searching and retrieval to support component reuse, (3) The feature diagram is updated to include the new features provided by the new variants, (4) Documentation explaining the rationale for the introduced variants as well as details for the implementation decisions for them is developed.

3 Constructing Elastic components with Test-Driven Development (TDD)

Our approach to constructing elastic components uses tests and TDD to explicitly scope a component's functionality. The developer defines a set of tests T as the set of tests that define this component's role in this and the future systems that it will be used as-is. The developer does not make any attempt to extend this set of functions to include functionality beyond this basic functionality that the developer considers essential. This is consistent with the YAGNI (You Aren't Gonna Need It) practice of agile development. After passing this set of tests and refactoring the code the component is thought as a complete new version of a newly created component hierarchy. This component is called pure in the sense that no attempt was made to fit it in the context of any particular system. It forms the base of a component hierarchy that has this component at its root. If the component needs transformation and evolution both for the current and future projects then there are two possible ways to evolve:

1. The first one is an evolution that violates the current set of tests. This violation is considered from our approach as violation of the basic contract for this component and therefore it spins-off the new component to its own new hierarchy in the repository. The component that violates one or more of the tests of T is therefore the base for a new elastic component hierarchy.

2. The second kind of evolution is one that respects the set of tests T and therefore for all purposes is considered the same component. However the developer may add new tests in this set. This addition of tests indicates new features that this component has. These new tests are required to be categorized by the developer to one or more categories according to the type of change (e.g. new functionality, better performance etc.) This categorization is done using the categories of the ISO quality model [8] which includes the following general quality categories: functionality, reliability, usability, efficiency, maintainability and portability.

To put it a little bit more formally assume that we have a pure component x with a test suite T_x. Then an evolution of this pure component will result in a new component y with a test suite T_y.

- The component y is called a variant of the component x if $T_x \subseteq T_y$.
- For the pure component x the set H_x contains all the components that are part of the hierarchy of x.
- For each component $y \in H_x$, Q_y is the ordered sequence of all the quality properties according to ISO-9621 that this component improves. For example if $Q_y = [q_1, q_2]$ this means that the component y has improved two quality properties over its base component x, q_1 (e.g. performance) and q_2 (e.g. modifiability).
- The difference $T_y - T_x \geq 0$ in the size of test suites of the component y from its base component x, is called *IHC (Internal Hierarchy Difference)*.
- If after the evolution of x to y, $T_x \nsubseteq T_y$ then the component y will be considered a new pure component in its own elastic component hierarchy.
- For two pure components x and y the size of the set of all tests belonging in both test suits of x and y, is the similarity $S(x, y)$ of the two components. For example if $T_x = \{t_1, t_2, t_3\}$ and $T_y = \{t_1, t_4, t_5\}$ then the similarity of the two components is 1, since they have one test in common.

In Fig. 2 (a) we see schematically an elastic component hierarchy and in (b) three different elastic component hierarchies. A critical question regarding software product lines is how to evolve them in order to accommodate new emerging requirements. In the context of this work we assume

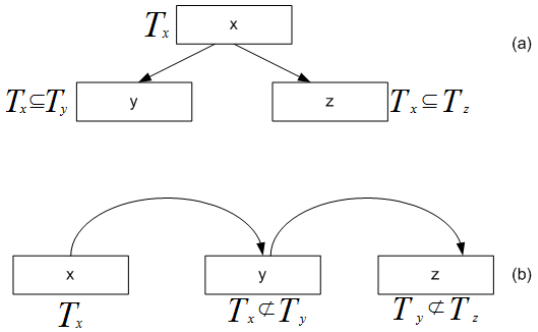

Figure 2. Elastic component hierarchies based on tests

that the application engineering team given a certain new feature for a product instantiates and modifies the software architecture of the product line in order to accommodate the new feature. The modifications of the software architecture will result in consequent required modifications for the existing software components. The components are then modified using TDD as we explained earlier, with modifications of existing core assets. This is the application engineering workflow of the product line engineering process. An important issue is wether the modified components should be incorporated in the core asset base and how. This is usually depicted as a feedback loop from the product line engineering workflow to the domain engineering workflow. To make this decision one should consider if the new core assets are potentially reusable to a number of products or if they are just unique for the newly developed product. Given the potential usefulness of the components in more products than the newly developed product, the newly developed components should be incorporated to the core asset base for future reuse. With our approach the component source code and the additional tests will be stored in the component repository which will also record the relationship to the existing components at a logical level: i.e. the fact that the new component is a variant of an existing component, which customizes its base component to accommodate a new functional or quality feature. The repository will also include the quality categories improved (under the ISO-9126 categorization). Furthermore the additional tests will be the proof that the component does indeed achieve the required goal. The compatibility of the component to the existing software architecture is judged with the previously existing test cases. If the component does not violate the existing test cases of its base component (i.e. it is a variant of an existing component) then the component can be used in all the configurations that the previously existing component could be used, in addition to the new configuration resulting from the application engineering phase for the new product. The modifications of the software architecture for the new product will result in a number of components being varied and all these components should be included in the core asset base along with their tests and they accompanying metadata. The feedback of the application engineering process to the domain engineering process as described, is depicted in Fig. 3. The important observation with the proposed ap-

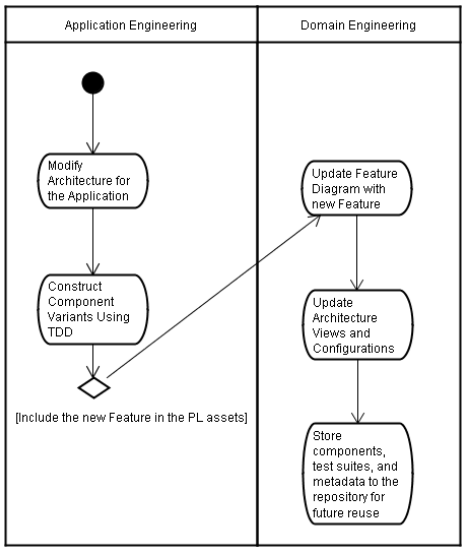

Figure 3. Application engineering feedback to the domain engineering process

proach is that Test-Driven Development which is an agile practice, does not interfere with the flexible upfront design required by product-line engineering. TDD is used instead in the context of application engineering to evolve the components of the product line in order to customize them for the newly introduced requirements. The produced artifacts are then evaluated for their significance in the product line context by the domain engineering team and are included in the core assets only if this is deemed useful. Therefore the two approaches of product line engineering and test-driven development are no longer in tension, but rather complement each other harmoniously.

To give a short illustrative example consider that we have a compute engine product line which is capable of executing jobs on several processors simultaneously. Each job is embarrassingly parallel which means that comprises a number of tasks that can be executed in parallel without the need to communicate to each other. In the initial product line architecture the compute engine was designed so that it runs on a cluster server that is highly reliable since all its processors are attached to the same local cluster. Imagine now that we wish our compute engine to make opportunistic use of idle

cycles in our local area network or the Internet. The idea is that users who are willing to help us with a demanding computation can visit a URL and download a Java applet. The applet then connects back to our compute engine which assigns tasks to the users' computers, that are executed and then the results are returned back to the compute engine. Since now the environment is much less reliable we want to extend our initial reference architecture so that some form of fault tolerance is achieved. There are several extensions that need to be made in the overall architecture:

- The hosts accepting tasks from the compute engine (i.e. the Host component) should now be extended to include a Java applet version, since we do not wish our users to install any specific software in their machines, but merely to download an applet and provide a specific security policy.
- The hosts should now also include the capability to periodically checkpoint the Tasks of the application and send these checkpoints back to the compute engine. This is necessary since given the opportunistic environment we want to be able to restart the tasks in a different machine if for example the connection to the execution host is lost.
- The compute engine should now be able to send ping messages to the hosts to discover their health status and get back responses or timeout. If the host do not respond within a specific timeout threshold the compute engine will assume that the host connection is lost.
- The compute engine should be capable of migrating the task to another host from the checkpoint last acquired if one available when the connection to a remote host executing a task is deemed as lost.

In this scenario the components that will evolve as a result of the new feature are the Host and the ComputeEngine components. The development of the new components however will not start from scratch. The developers of the new components will reuse the existing Host and ComputeEngine components and modify them using TDD. The existing test cases for the two components will be reused as well as the internal architectures of the components and they will be expanded as needed for the new feature. In this particular case both components will pass the existing tests since the existing requirements do not contradict the newly-introduced requirements. The test suites of both components will be expanded:

- We need to test that the host component periodically sends the tasks' checkpoints back to the compute engine and that it responds to ping messages from the compute engine in a timely manner.
- We also need to test that the ComputeEngine component accepts the checkpoints send by the Host component, it pings the Host component to discover its health status, and migrates the Tasks if and when needed.

Notice that the newly introduced components do not violate the existing tests and are considered variants of the existing components, because the existing components test suites do not include any performance tests, but only functional tests. If the existing components had specific performance requirements and the respective tests then these existing tests would probably be violated. In that case the new components would fail to pass the old tests. This is correct since we were interested in the performance of our compute engine in the first version and now we are more interested in the fault tolerance aspect willing to entail a performance penalty for this. The important thing to notice however is that even in this case the two different versions of the components will not be entirely unrelated since their similarity (i.e. the number of tests in common) will be positive, but the new components will not be backward compatible with the old versions.

After the construction of the new variants of the components the decision to include them in the core asset will be taken to the product line engineering team which will consider their possible reuse in future products. The team for example may decide that a good idea would be to open the possibility of some reward system for the owners of the host machines and introduce auctioning for tasks in the Host components. Auctioning will then be a new feature which also requires the checkpointing and migration of tasks. The best components to adapt in this case are the variants which already include this functionality instead of the original base components.

4 Related Work

A few approaches have considered reuse in general and product line approaches to reuse in particular, in relation to agile methods.

In [9], the authors describe Extreme Harvesting a method for discovering reusable components on the web using test matching. Tests are developed as usual in the development of a new product using extreme programming. However before the development of the code that passes the test, the developer uses an eclipse plugin that attempts to discover components on the web using code search engines such as Koders and Merobase, that pass the test. If such a component exists, the developer can then reuse it. Two approaches are described: definitive harvesting which attempts to match all of the test cases at once and is more suitable for well-understood domains and speculative harvesting which proceeds in more conservative steps, for situations where the

developer is willing to adapt his or her system to discovered components. Our approach similarly to Extreme Harvesting tries to reuse software components in the context of a product line approach. However we are not attempting to find a component that exactly matches the test cases of the component, but rather reuse the component to produce a new variant.

In [10] the authors consider the relation between agile methods and software product line engineering. Similar to us the authors believe that the two approaches should be combined and that agile practices can assist during the application engineering phase: "the two approaches must be combined. PLE as a proactive, strategic reuse approach forms the basis to develop new products...Agile methods are then used in application engineering to perform the customization or calibration of a product for a specific customer". The authors consider how agile practices (e.g. the planning game) can be customized and used in the context of the application engineering workflow of a product line process. Our approach concentrates instead in the most important agile practice of Test-Driven Development and the way that this practice can assist to the evolution of a product line.

In [11] the author considers the strong characteristics of both agile methods and plan-driven methods, and suggests that both have a "home ground of project characteristics within which perform very well". The author also suggests that "a combined approach is feasible and preferable". In this work we have presented such a combined approach, adapting however the agile practice of TDD and not considering the mixing of other agile practices with product line engineering approaches.

5 Conclusions and future research directions

In this work we proposed a new method for component variability and reuse of product lines which is driven by test suites. The method addresses the issue of the variability introduction in a product line using customization of existing core assets. Some variation mechanisms for software product lines are listed in [2] including inheritance, component substitution, plug-ins, templates, parameters, generators, aspects, runtime conditionals and configurators. Technically our approach is a component substitution approach in which new components are created reusing existing components and test cases. However the substitute component may be created internally using many of the other mentioned variation mechanisms. Test suites assist the correctness of the placement of the new variant component in the core asset base which is organized for the components in a hierarchical fashion to enable their effective reuse, searching and retrieval. Tests also assist in the verification that the product is correct after its evolution since test suites are available to repeat the quality assurance process in the new product. In the future we plan to concentrate in the development of the elastic component repository and use specific metrics such as the Similarity between components and the Internal Hierarchy Difference to speed the searching process in repositories with large numbers of components. We also plan on applying our approach to a number of large case studies to refine it further.

References

[1] Paul Clements and Linda Northrop: "Software Product Lines: Practices and Patterns", Addison-Wesley, 2002

[2] Felix Bachman and Paul Clements: "Variability in Software Product Lines", Technical Report CMU/SEI-2005-TR-012, Software Engineering Institute, September 2005

[3] Kyo C. Kang et al.: "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990

[4] Lasse Koskela: "Test Driven: Practical TDD and Acceptance TDD for Java Developers", Manning, 2008

[5] George Kakarontzas et al., "Elastic Components: Addressing Variance of Quality Properties in Components", Euromicro 2007 - CBSE Track, Lübeck, Germany, pp. 31-38, IEEE, 2007

[6] Michele Lanza and Radu Marinescu: "Object-Oriented Metrics in Practice", Springer, 2006

[7] James O. Coplien: "Multi-Paradigm Design for C++", Addison-Wesley, 1998

[8] ISO, "Software Engineering - Product Quality - Part 1: Quality Model", ISO, ISO/IEC Standard 2001.

[9] Oliver Hummel and Colin Atkinson: "Supporting Agile Reuse Through Extreme Harvesting", 8th International Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2007), LNCS 4536/2007, pp. 28-37, Springer, 2007

[10] Ralf Carbon et al.: "Integrating Product Line Engineering and Agile Methods: Flexible Design Up-front vs. Incremental Design", in Proceedings of the 1st International Workshop on Agile Product Line Engineering, 2006

[11] Barry Boehm: "Get Ready for Agile Methods, with Care", IEEE Computer, vol. 35, no. 1, pp. 64-69, IEEE, Jan. 2002