Feature based Structuring and Composing of SDLC Artifacts

Nishigandha Hirve
TRDDC
Pune 411 013, India
nishigandha.hirve@tcs.com

Tukaram Muske
TRDDC
Pune 411 013, India
t.muske@tcs.com

Ulka Shrotri
TRDDC
Pune 411 013, India
ulka.s@tcs.com

R. Venkatesh
TRDDC
Pune 411 013, India
r.venky@tcs.com

Abstract— **Product organizations often need to develop variants of the same basic product. All the product development life-cycle artifacts from requirements documents to testing artifacts have to be developed for each variant. Storing these artifacts as core assets and structuring them as units that can be composed to get the final artifacts can greatly reduce the cost and improve the quality of the resultant product. This paper proposes a structural alignment of core assets corresponding to the various phases of software development to features of the product family and aspect weaving as a core assets composition operator. Aspect Oriented Programming (AOP) is used for code and an aspect oriented extension to XML is used for other artifacts. These ideas are validated through a prototype toolset that captures the complete Software Development Life Cycle (SDLC) process and allows the user to model the commonalities and variations as features, associate them to core assets and build the product automatically. This paper presents our approach and the details of the toolset along with a case study of a Library System.**

I. INTRODUCTION

.Planned software reuse as recommended by Software Product Line Engineering (SPLE) [1] can considerably improve product development cost and time. SPLE and Feature Modeling [2] identify variations and commonalities between products in a product line and model them as features [3]. This paper proposes the structuring of the different software development artifacts so as to align them with the structure of the feature model. Once this is achieved, composition similar to weaving in aspect oriented programming can be used to derive artifacts at the product level from artifacts at the feature level.

Development of each product from the feature model follows the entire SDLC where the resultant requirements document has to be reviewed and approved by relevant stakeholders and the final product has to be tested for this specific combination of features. Therefore, just producing the final product by composition will not suffice and there is a need to generate all artifacts required by SDLC using similar composition. This paper adapts the idea of Aspect Oriented Programming (AOP) applied to product development, to other non-graphical SDLC artifacts like documents and test harness. Documents are composed using an aspect oriented extension to XML and test harness is composed by writing the test harness in a programming language and using AOP.

The prototype toolset presented here captures complete SDLC process from product requirements gathering to its release. This toolset mainly consists of two parts, core assets repository and product generator. Creation of core assets repository involves identifying features, modeling them using ASADAL [4] and identifying, developing, and maintaining feature-based core assets. ASADAL allows selection of features required for product development. Product generator identifies and composes the required core assets together into a final product.

We demonstrate the toolset and its benefits using library domain case study. This case study illustrates the identification of variations and commonalities in a library product family and their modeling as a feature model. It also illustrates the structuring of other core assets consisting of various development artifacts so that they are aligned with the feature model. AspectJ – AOP implementation for Java [5] is used as the core assets composition operator to handle these variations between product variants for the artifacts implemented in Java. AOP has been chosen as it provides a very general and flexible set of composition operators. A XML weaving tool is used to implement a composition operator for the document artifacts.

Related work: Various other toolsets are available for generating products from a product family, customizing products [7], and configuring product line features [8]. Kyo C. Kang has combined feature oriented analysis with AOP [9]. Studies have shown that feature models can be used for product derivation [6]. Also, feature modeling is supported by several tools [10],[11].

Compared to the tools listed above, our prototype toolset provides support for different artifacts like documents and code, required in a typical SDLC. It mainly investigates AOP based techniques to compose both code and documents.

II. TOOLSET

Figure 1 shows the detailed toolset architecture. The toolset organization is mainly divided into Core assets repository and Product generator.

A. *Core Asset Repository*

The core asset repository stores all the assets including feature models and associated documents and code.

1) Feature Modeling in ASADAL.

We use ASADAL tool for feature modeling that supports FODA notations and allows selection of features for product development. We associate the feature-specific core assets information to the description field in ASADAL using a set of keywords. The keyword set includes keywords such as SOURCE, CONFIG, XML, and TESTCASES.

2) Structuring and Implementation of Core Assets

The required set of core assets is identified from feature model and mandatory, optional and alternative features are implemented separately. The core assets for optional features are implemented as aspects in order to manage variations found in the product family. Whenever an optional or alternative feature is selected, its corresponding aspect gets included in the source list. These aspects modify the functionality of the mandatory feature to provide the product behavior with respect to the selected feature. This type of implementation also provides traceability from features to their implementations.

Code Composition: The composition of code related core assets is similar to the other AOP based applications.

Test Harness Composition: As the test harness is written in a programming language using AOP, its composition is exactly similar to the code composition.

Documents Composition: AOP extensions to XML help to manage documentation at the feature level. This requires structuring of the feature based documents using XML and additional AOP tags. The organizational alignment between feature model and document states that document is either a base document (associated with mandatory feature) or an aspect document (associated with optional, alternate and or feature).

The base document has a standard XML structure and consists of tags, attributes with values and content. Each path from the root to any tag in the document specifies a join point. Aspect documents also have standard XML structure specifying advices to modify the contents at a join point in the base document. An advice is also an XML path from the root to an advice tag and optionally includes the child. The advice tags can be either “after”, “before”, “replace” or “delete”.

The document composition algorithm is:

Let $p = r, t_1 \dots t_n, a, c$ be a path in the aspect document where

- *r* is the root node,
- *a* is the advice node having tag “after”, “before”, “replace” or “delete”
- $t_1 \dots t_n$ are nodes in the path from *r* to *a* in the aspect document
- *c* is the child tag of *a* and is of the form <*ctag, id=val*>

For each such path p in the aspect document, it matches a corresponding path p1 = r1, t1 … tn, in the base document such that the tags of t1 … tn are identical in both p and p1 and if any of t1 … tn has an attribute id, then the value of the attribute also should be the same. For each such match if the tag of a is ‘Before/After’ then c is added as the first/last child of tn respectively in the base document. In the case of Replace/Delete tag, the child of tn with the same tag and value of attribute id as c is replaced/deleted in the base document

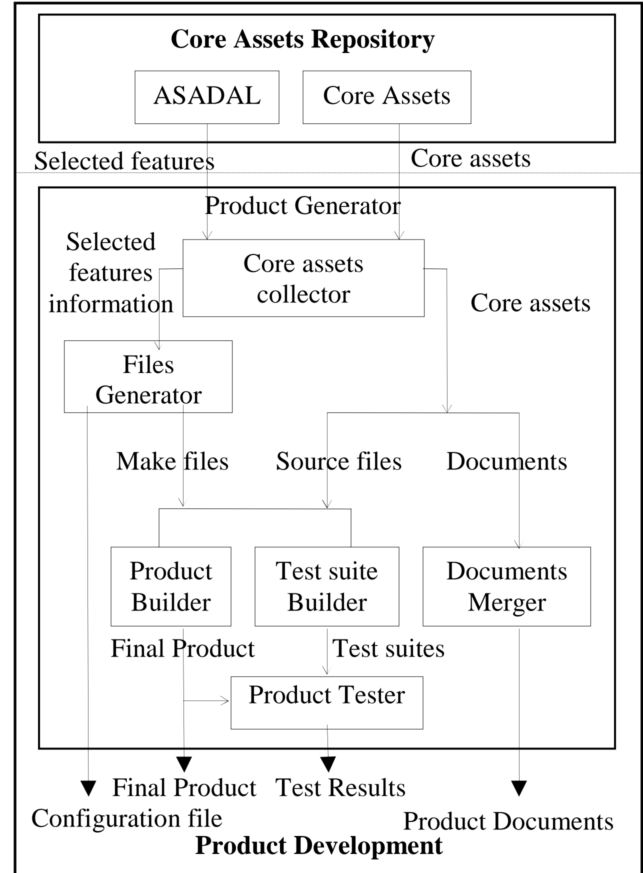

Figure 1. Toolset architecture

The values of attributes other than id are not considered for the match.

B. *Product Generator*

When a new product is to be built, required features are selected from feature model and exported as an ascii file. The exported file also includes associations between the selected features and corresponding requirements, test case, implementation and test harness aspect file names. The *core assets collecter* extracts the required files from the source code and document repository. Once all the files have been extracted the *File Generator* generates a makefile that invokes aspectJava and the java compiler to build the application as well as the test harness. The *product* and *test suite* builder then executes these makefiles. The test harness is then run on the application and the results logged.

Orthogonally the *Document merger* invokes the XML weaver on all the test case and other base and aspect documents extracted by the *core assets collector* to generate requirements and test case documents for the final product.

III. LIBRARY SYSTEM CASE STUDY

This section illustrates the features of the toolset using a Library System product family case study, where a particular set of requirements is considered.

A. Library System Domain

Library systems maintain books and members and offer services to members to reserve, borrow and return books. There are two types of members (Ordinary and Privileged) and three types of books (Journal, Technical book and Magazine). The number of books to borrow and days allowed are determined by the member and book types. Library system may allow members to put claims on the books and the number of claims varies as per member type. There may be a late return fine. Although we have implemented several features as part of our case study, only few of them are described in detail here in the interest of brevity.

B. Modeling and Implementation of Library Systems

Using the concepts of SPLE and feature modeling, we study the Library system domain and systematically elicit the variations – Member type, book type, book claim, number of books and days allowed, Late return fine etc.

1) Feature Modeling

The features identified from variations and commonalities are modeled in the feature model, using FODA notations, as shown in Figure 2. We have shown only a small set of selectable features in Capability and Implementation Layer for clear representation. Mandatory features such as Member and Book Maintenance are omitted from the feature model.

2) Library System Core assets

The various core assets for Library system can be categorized into – primary assets (feature model, requirements, class diagrams), components (code implementations such as core service component, claim processing component), test harness assets (test cases' implementations, test data, test results), various tools used (Tomcat, ASADAL, product generation tools), configuration files, etc.

3) Library System Implementation

We use client-server architecture to implement the library system with Java (JDK1.5.0), JSP, AspectJ (release 1.5) [12] and Tomcat [13]. These implementation details will affect the feature representations. A feature will have its corresponding functionality component represented in implementation layer. For example, the core services are implemented by core service component which is further linked to Java implementation to specify its source files. The Java implementation feature is further divided into client and server components.

C. Core Asset Composition Technique

The source implementation of library functionality has been done in AspectJ with variants of each feature implemented as aspects. AspectJ has also been used to implement the test harness. Since the implementation of functionality is similar to other aspect oriented applications, in this paper we only describe the test harness implementation in detail.

1) Test Harness Composition

The core assets corresponding to test harness are implemented similar to other basic functionality components. The variable part in the test cases is put into the aspects which modifies the behavior of the mandatory test cases.

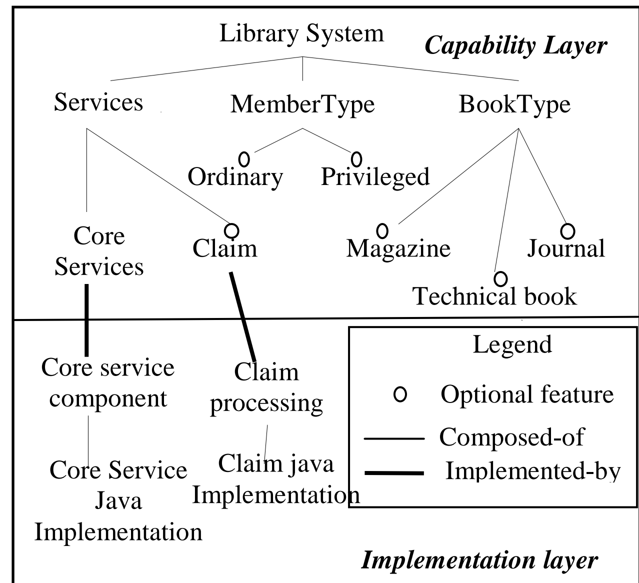

Figure 2. Library system feature model

Example: The test cases for the borrow operation are implemented in BorrowTests class. We consider following test case, in pseudo code, for book unavailable scenario.

```
class BorrowTests{
...........
Function chkPostCondition () {
       Check_for_msg "The requested book is not
       available in the Library";
       return true if successful, else false;
} }
```

In the presence of claim, if the requested book to borrow is unavailable, it gets reserved for the member. Thus, when claim is selected, the borrow operation behaves differently and above test case needs modification to check for claim object. The required modification to borrow test case, due to claim, is achieved using below piece of aspect where method chkPostCondition is overridden by the below aspect.

```
around():BorrowTest.chkPostCondition();{
   check_for_msg "The requested book is not
   available in the Library. The same book is
   reserved for you."
   check_for claim_object;
   return true if successful, else false;
}
```

2) Documents Composition

Various documents are structured with respect to features and formatted as per AOP concepts. For example, for better reuse, test case documents are maintained at feature level for Core services and optional Claim feature separately. These documents will be meaningful only if the Claim feature is selected. This necessitates their formatting in a particular manner to manage variability which includes specifying the required and related join points, pointcuts and advices in an AOP way. A join point from SimpleBorrow.xml, the test case document for Core services feature is shown below.

```
<TestCase id=2> <CheckResult>
<Check id="a"> Check for return message: "The
requested book is not available in the
Library".</Check>
</CheckResult> </TestCase>
```

The elements TestCase and CheckResult specify the join point. In presence of Claim, the information in check results needs to be modified as the test case behaviour is changed. Additional checks for claim message and claim object are added using following advice from BorrowClaim.xml.

```
<TestCase id="2"> <CheckResult> <After>
<Check id="b">Check for return message: "The
same book is reserved for you"</Check>
<Check id="c">Check For loan object for Title1
with member 13 and bookcopy of Title1</Check>
</After> <CheckResult> </Case>
```

Here, the 'After' element specifies the After advice to add the extra checks in the target test case. The test case mapping is achieved using common value for "id" attribute. Thus the information with elements and attribute values specifies the pointcut. When Claim feature is selected, the XML documents composition occurs and the resulting test document for the above mentioned scenario appears as shown below.

```
<TestCase id=2> <CheckResult>
<Check id="a">Check for return message: "The
requested book is not    available in the
Library."</Check>
<Check id="b">Check for return message: "The
same book is reserved for you"</Check>
<Check id="c">Check For loan object for Title1
with member 13 and bookcopy of Title1</Check>
</CheckResult> </TestCase>

Requirements and design documents are
similarly structured and XML weaving is used
for composition.
```

D. *Product Development*

As the entire required core assets base is in-place in repository, product development simply becomes the process of picking-up and combining the related components. We can easily develop Library systems with the required features and release the tested library system to the market.

IV. CONCLUSION

We have demonstrated the feasibility of aligning the structure of all software development assets with feature models and the use of an aspect composition operator to derive a specific artifact corresponding to a specific product. Successful application of the proposed approach requires the documents and code to be structured in a way that is amenable to aspect composition. In particular requirements and design documents should be structured as a tree with the leaves being simple elements.

We believe the ideas presented can be extended to larger projects.

ACKNOWLEDGEMENT

We sincerely thank Suparna Soman for contributing through multiple reviews to make this paper better.

REFERENCES

[1] Clements, P., Northorp, L., "Software Product Lines: Practices and Patterns", 2002

[2] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee., "Concepts and Guidelines for Feature Modelling for Product Line Software Engineering, Software Reuse: Method, Techniques and Tools", ICSR-7, April 2002

[3] Jaejoon Lee, Dirk Muthig, "Feature-oriented variability management in product line engineering", Communications of the ACM December 2006/Vol. 49, No. 12.

[4] Kyungseok Kim, Hyejung Kim, Kyo C. Kang, "ASADAL - a tool system for co-development of software and test environment based on product line engineering" ICSE '06

[5] Ramnivas Laddad, "AspectJ in Action: Practical aspect-oriented programming", 2003

[6] Spinczyk, O., Papajewski, H., "Using Feature Models for Product Derivation", SPLC 2006

[7] Mazen Saleh, Hassan Gomaa, "Separation of Concerns in Software Product Line Engineering", ICSE 2005

[8] Andreas Hein, John MacGregor, Steffen Thiel, "Configuring Software Product Line Features" ECOOP 2001

[9] Kwanwoo Lee, Kyo C. Kang, Minseong Kim, "Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development", SPLC 2006

[10] Alain Forget, Dave Arnold, Sonia Chiasson " CASE-FX: feature modeling support in an OO Case tool", OOPSLA 2007

[11] Thomas Leich, Sven Apel, Laura Marnitz, Gunter Saake, "Tool support for feature-oriented software development: featureIDE: an Eclipse-based approach", OOPSLA 2005

[12] AspectJ (release 1.5), available at as on March 09, 2010, http://www.eclipse.org/aspectj/index.php

[13] Apache Tomcat URL as on March 09, 2010, http://tomcat.apache.org/