A product line architecture for web applications

Conference Paper · January 2005

DOI: 10.1145/1066677.1067059 · Source: DBLP

4 authors, including:

Davide Di Ruscio
Università degli Studi dell'Aquila
135 PUBLICATIONS **1,548** CITATIONS

SEE PROFILE

Alfonso Pierantonio
Università degli Studi dell'Aquila
143 PUBLICATIONS **1,814** CITATIONS

SEE PROFILE

A Product Line Architecture for Web Applications

L. Balzerani, D. Di Ruscio, A. Pierantonio
Dipartimento di Informatica
Università
degli Studi di L'Aquila
I–67100 L'Aquila, Italy
{balzerani, diruscio, alfonso}@di.univaq.it

G. De Angelis
Istituto di Scienza e Tecnologie dell'Informazione
"Alessandro Faedo" CNR
I–56124 Pisa, Italy
guglielmo.deangelis@isti.cnr.it

ABSTRACT

Increasingly, Web applications are used in similar environments to fulfill similar tasks. Sharing a common infrastructure and reusing assets to deploy recurrent services may be considered an advantage in terms of economic significance and overall quality. Thus, it may be appropriate to design web applications as members of a product family.

The paper illustrates Koriandol, a product–line architecture designed to develop, deploy and maintain web application families. In contrast with usual component–based systems, Koriandol prescribes that variability handling mechanisms are reflective and built–in into the components.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design; D.2.11 [**Software Engineering**]: Software Architecture; D.2.13 [**Software Engineering**]: Reusable Software

Keywords

Web applications, Product line architectures, Variability determination, Feature oriented domain analysis, Reflection, Component-based systems

1. INTRODUCTION

During the last years, Web–based systems rapidly evolved from simple collections of static pages to complex applications. The economic relevance and the growing intricacy of such applications require techniques and models that can offer a greater return on development time and quality factors. On the contrary, traditional methodologies are often based on the mere skills of individual programmers and does not always apply the principles of software engineering.

As Web applications often share similar behaviors, shifting the focus from the design of single applications to that of system families is an effective way to pursuing software reuse. In particular, Web–based systems can be considered as software products derived from a common infrastructure and assets which capture specific abstraction in the domain, e.g. shopping cart, checkout, and user registration in an online retailing system.

This paper discusses Koriandol, a product–line architecture [3] (PLA) to design, deploy, and maintain families of applications. It is a component–based system with explicit variability management built–in into the components in a prescribed way. In fact, any variation in Koriandol is accomplished by means of reflective mechanisms able to bind a variation point to specific variants, rather than writing code and keeping the variants distinguished.

The structure of the paper is as follows. In section 2 some preliminary notation on product–line engineering is introduced. Next section presents the ideas behind Koriandol as a Web–specific PLA. Sect. 4 illustrates the tool implementing the architecture exposed in the previous section. An example is proposed in Sect. 5. Finally, a section devoted to related work and the conclusions are closing the paper.

2. PRODUCT LINE ARCHITECTURES

Software reuse is a simple yet powerful vision that aims at creating software systems from existing software artifacts rather than building systems from scratch [14]. Leveraging commonalities between members of a product family as well as across similar product families emerged as an effective way of pursuing software reuse.

A software product line typically consists of a product–line architecture (PLA), a set of reusable components and a set of products derived from the shared assets. Each product inherits its architecture from the PLA, instantiates and configures a subset of the product line components and usually contains some product specific code [3]. The major motivation for PLAs is to simplify the design and maintenance of program families and to address the needs of highly customizable applications in a cost-effective manner [2]. By using a software product line, developers are able to focus on product specific issues rather than on questions that are common to all products [9]. Summarizing, a PLA is a blueprint for a family of related applications.

The key concept to the development of system families is variability, intended as the ability to derive various products from the product family [12]. Variability is realized through variation points (originally introduced in [10]), i.e.

SAC '05, March 13-17, 2005, Santa Fe, New Mexico, USA.

places in the design or implementation that are necessary to achieve some functionality variance. Each variation point has an associated set of variants which express its variability; a variation point is unbound until a particular variant is selected, then it's said to be bound to that variant. Associated with each variation point is a binding time, at which the resolution of the variation point takes place. Typical binding times are architecture configuration (component selection), component configuration (component parameterization), startup time, and runtime [8].

Handling variability is a difficult task. The differences among the products in a product family can be described in terms of features. A feature is a logical unit of behavior that is specified by a set of functional and quality requirements [3]. Accordingly, features realize a mean to abstract from requirements, which are tied to features by a *n-to-n* relation. Feature modeling is an important approach to capturing commonalities and variabilities in system families and product lines. Several methods have been proposed to identify and model features; among these, Feature–Oriented Domain Analysis (FODA) [13] is often referred to as one of the most emerging. FODA provides both a process to determine common and variable features of concept instances, including their interdependencies, and a notation to represent them in feature models consisting of feature diagrams with some additional information such as short semantic descriptions of each feature, constraints, default dependency rules, etc.

Products within a product family are typically developed in stages which tend to be asynchronous, i.e. a *domain engineering* and a concurrently running *application engineering*, respectively.

- Domain engineering involves, amongst others, identifying commonalities and differences between product family members and implementing a set of shared software artifacts (e.g. components) in such a way that commonalities can be exploited economically, while at the same time the ability to vary the products is preserved. During this phase variation points are designed and a set of variants is associated to each of them. Work products of the domain engineering process are software components, reusable and configurable requirements, analysis and design models, and so on. In general, any reusable work product is referred to as a reusable asset [6].
- During application engineering individual products are derived from the product family, constructed using a subset of the shared software artifacts. If necessary, additional or replacement product-specific assets may be created. In this phase each variation point, as defined in the previous stage, is bound to a specific variant, selected from the set of variants associated with it.

The above-mentioned stages constitute two relatively independent development cycles, i.e. development for reuse, meant as development of the product line itself, and development with reuse, also called product instantiation. Many case studies (see [4] among others) have been documented, along with successful product line practice patterns. An updated *hall of fame* is maintained by the Software Engineering Institute [5].

3. KORIANDOL, A WEB–SPECIFIC PLA

Increasingly, Web applications are used in similar environments to fulfill similar tasks, i.e. systems may often be part of a product line. Sharing a common infrastructure (which builds the core of every product) and reusing assets which can be delivered to deploy recurrent services is always more becoming commonplace. With this basis it is only necessary to configure and adapt the infrastructure to the requirements of the specific application. If product family engineering is done right, this results in a considerable decrease in effort needed for the construction of a single Web application.

Koriandol [7] is a PLA designed to develop, deploy and maintain Web application families. The definition of an application involves the selection of components, which are assembled in a prescribed way, from an in–house library or the marketplace. Components include built–in reflective mechanisms for variability determination in order to put them to use in specific products. The identification and representation of common and variable features are performed using the FODA methodology. In contrast with traditional component–based development, any variation in Koriandol is accomplished by means of such mechanisms, rather than writing code and keeping the variants distinguished. These are pivotal aspects of Koriandol as the handling of the differences between family members is a key factor for the success of a product family.

According to the organizational model provided by Koriandol, a Web application can be viewed as an association of pages and components as depicted in Fig. 1. In particular, a Web application is a product consisting of a number of pages that are arranged hierarchically. Each of them collects contents dynamically provided by the selected components. As already mentioned, components are generic, i.e. they are designed to capture both the commonalities and variabilities of a class of behavior in an application domains, for instance as the typical functionalities of an online retailing system (e.g. cart, catalog, checkout, etc.) which are realized at different degree of complexity. Each method provides a different functionality and the admitted values for its formal parameters are representative for the functionality variants whose selection resolves a variation point to a bound one (bound variation point in Fig. 1). Analogously, the methods selected in a page are bound method once all their variation points have been resolved. A component is a bound component as soon as some of its method are selected in the product.

As said above, in Koriandol components also embody a built–in variability handling mechanism which allows application engineers to dynamically bind each variation point to the appropriate variant. Depending on previous user selections, the mechanism reveals through introspection applicable variation points and their associated variants. This provides a convenient mean to navigate and operate on the component feature model. To better illustrate such a variability handling mechanism, let us consider the feature diagram for a simple News component presented in the left–hand side of Fig. 2, where variability is recognizable at different abstraction levels. The component encompasses two methods: Summary and Article which display a list of news and an article content, respectively. They are represented as alternative subfeatures of the component, i.e. exactly one of them has to be selected at once. Summary in turn has a mandatory subfeature which specifies the number of news to be sum-

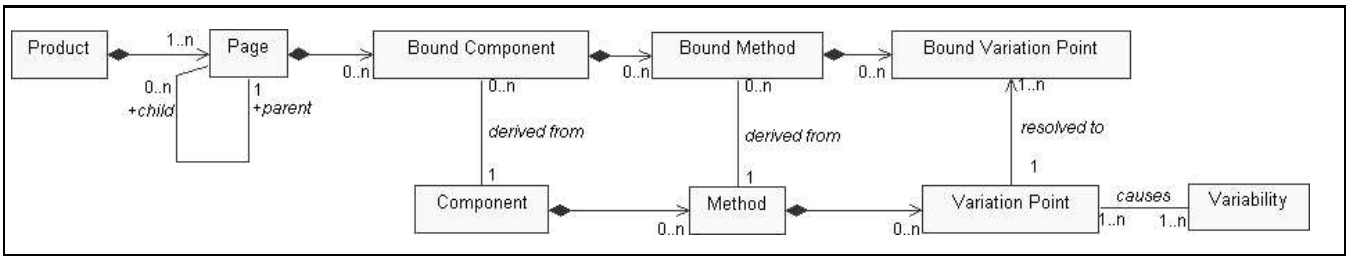

Figure 1: Associations among the assets in a products

marized, and an optional one which applies a category filter. The corresponding variability handling mechanism is shown on the right–hand side of Fig. 2.

A mapping between the notation of feature diagrams and HTML widgets (e.g. check boxes, radio button and so on) has been defined. Exploiting this mapping, variability handling mechanisms can be automatically generated from declarative specifications. In order to provide a convenient textual representation for feature diagrams a domain–specific language has been given in [1].

Typical binding time for component (and method) selection is the architecture configuration. In our approach the variability determination can be accomplished both during the product instantiation phase and post-deployment, i.e. at run-time, making applications widely reconfigurable. This is one of the major contributions of Koriandol since this approach can improve the variability management capabilities of the application engineering phase. Furthermore, during product instantiation, previously unrevealed variation points can came up, offering additional variability to application engineers. Conceptually this means that they have the opportunity to decide how much variability to exploit, a design task which is usually restricted to the domain engineering stage.

The proposed architecture can be used to develop a wide range of applications depending on the domains whose abstractions and their relationships are captured by registered assets. Of course, the nature of the products depends necessarily on the available components and in case some behavior is missing the designer should care about its analysis, design, and realization.

4. TOOL SUPPORT

The Koriandol system consists of specialized software modules which realize the common infrastructure of the product–line architecture described in the previous sections. Additionally, some auxiliary modules mainly for the management and the configuration handling are provided too. In particular, the following core units are among the most important:

- the management and configuration module,
- the run–time environment, and
- the presentation engine.

A simplified schema of the system is illustrated in Fig. 3, where the control flow triggered by a client request is denoted by numbered labels. When a user request arrives (see spot (1) on figure), the runtime module interprets w.r.t. the correspondent page specification (2), in other words it validates the user privileges in order to access the page and,

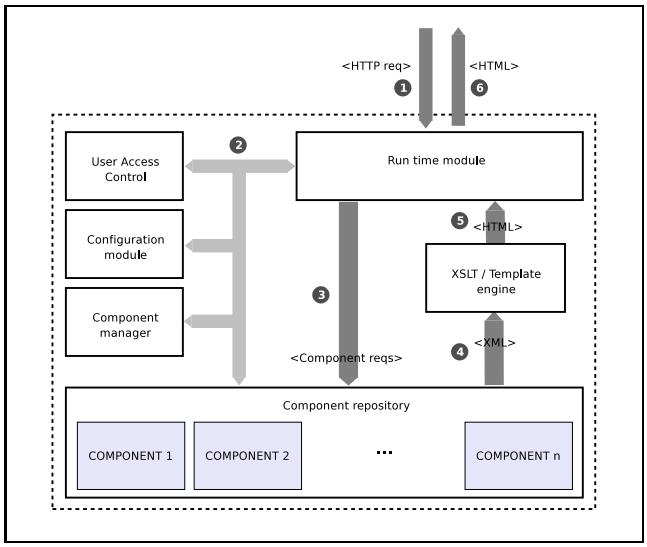

Figure 3: Koriandol system

in case the user is authorized (or the page is freely accessible), identifies which components have to be invoked to serve that particular request. Each page functionality is retrieved through a component request (3), i.e. a method invocation to the component selected and configured during the variability determination for the instantiation of the requested page. The method returns XML code which is passed to the XSLT/Template engine (4) which generates HTML code (5) by means of transformation stylesheets. Once all component requests are realized the obtained HTML segments are put together and returned to the user (6).

In the sequel of the section, the core units of the Koriandol system are described referring to Fig. 3.

4.1 Management/Configuration module

The management module provides for support to administrative tasks such as user, component, and product accounting among others. These are fairly usual functionalities, but the component management and configuration. In fact, each component must be registered and validated prior to use, according to established prescriptions given by Koriandol as an interface specification. The variability determination mechanism is included in such a specification in order to grant configuration capabilities to the system.

In particular, the system assists the application designer in recording decisions and entering directives, such as preparing and putting a feature to use within a page. This op-

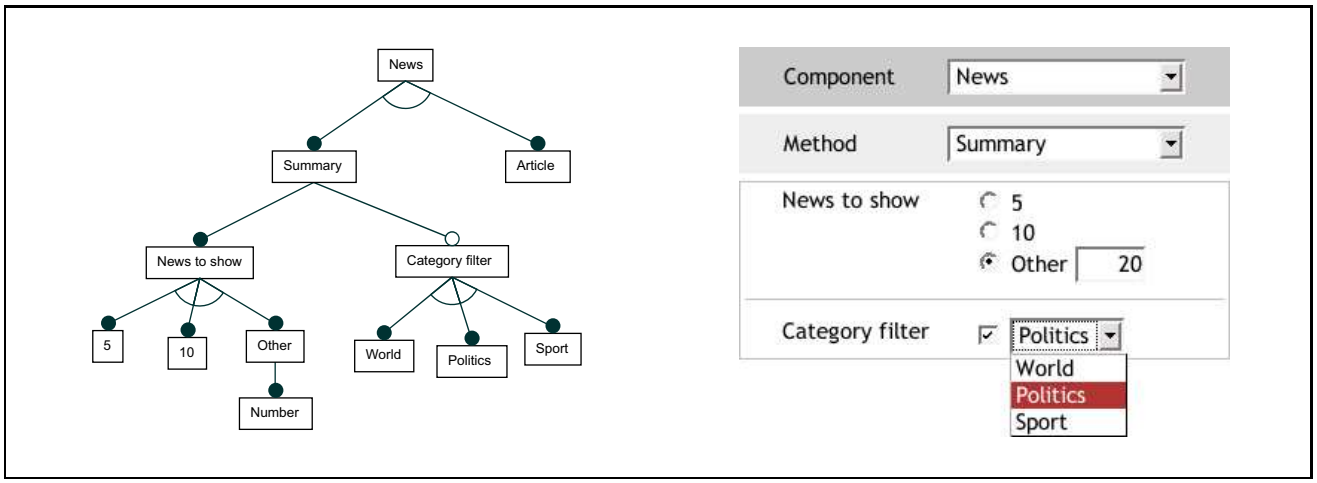

Figure 2: Feature diagram for a News component.

eration (and similar ones which are not described due to space limitations) can be accomplished by means of interactive wizards as the one illustrated on the right–hand side of Fig. 2. Once a component is selected, the system is able to retrieve the available features through reflection. Not all the methods are visible during this stage, only those which actually implement a feature; most of the other are either used locally to the component or directly invoked by the system. Once a method is selected, the system makes available to the application designer the lower part of the form in Fig. 2 (returned by another method of the component) that presents all the parameters the design has to provide in order to determine the coordinates denoting the desired variant.

4.2 Run-time environment

The scope of the runtime environment is on one hand to coordinate all the activities among the modules once a request arrives from outside; on the other hand, to retrieve functionalities (dynamic contents and services) which are demanded by the pages being served. Such functionalities are dynamically retrieved by identifying the component methods and passing them relevant information which are evaluated according to the directives given by the designer during the variability determination for that specific instantiation.

4.3 Presentation Engine

As said, each component is able to yield contents which are structured and given in XML. Contents are delivered independently from any presentational aspects which are mainly dealing with the appearance of an application. The system provides a presentation engine which allows the designer to associate to each feature/method in a component a different XSL transformation stylesheet or alternatively a HTML template[1]. XSL transformation stylesheets are hierarchically arranged according to the parental relation among the assets, e.g. a stylesheet associated with a product can be overridden by another one associated to lower assets in the hierarchy, such as the one associated with a component or a feature. Thus, the component designer can take important decisions regardless of eventual constraints which may be imposed by the presentational aspects.

A prototypical implementation of the Koriandol system is available for download on SourceForge [7].

5. EXAMPLE

The Web applications developed using the Koriandol system consist of services and contents delivered in pages which are hierarchically arranged. The tool discussed in the previous section provides the facilities to create and organize them consistently with the requirements of the application being developed.

The definition of the homepage depicted in Fig. 4 involves three components providing product search functions, a showcase, and an authentication form, respectively. Each functionality is instantiated with parameters which are specified exploiting the built–in reflective variability determination mechanism. For instance, the component providing the showcase facility can have a number of variation points, such as product categories and frequencies, which are useful in order to tune the vendor strategies. In this case, the Koriandol system introspects the component and retrieves a form similar to the one in Fig. 2 which is used to bind such variation point with one of the available variants.

6. RELATED WORK

Most of the literature on software product–lines ([5, 12, 16] just to mention a few) focus on the technology and the processes that are related to the development of product–line based software. Although, the cost effective development of web applications is perhaps one of the most challenging areas of software engineering today not too much work has been carried out in viewing Web applications as software product–line to our knowledge. In [11] a specialized architecture, called OOHDM–Java2, is given to develop Web applications. At a certain extant, the system family defined by OOHDM–Java2 is represented by the whole class of possible Web applications, since the commonalities are very general and deriving from the model–view–controller architecture as extensions. The main difference with Koriandol

[1]The template language and engine which has been adopted is *patTemplate* [15].

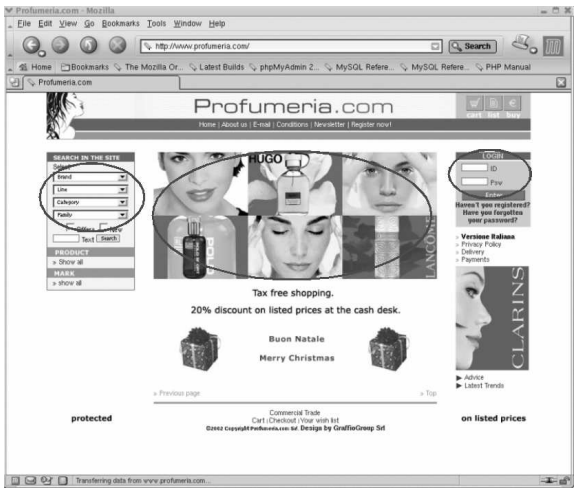

Figure 4: Abstractions featured by an e-commerce

is in the lack of variability determination mechanisms in the architecture, since they are systematically treated during the application design.

Performing correct domain analysis is crucial for correct development of software product–line. Among the different approaches, it worths mentioning the Feature–Oriented Domain Analysis (FODA) [13] which based its foundations (and popularity) on an in–depth study of other domain analysis techniques. The feature-oriented concept introduced by FODA is based on the emphasis placed by the methods on identifying prominent or distinctive features within a class of related software systems. These features lead to the creation of a set of products that defines the domain. Koriandol relates to FODA since the variability determination mechanisms which are given within the components are generated by means of a domain–specific language [1] which is a data–intensive extension of a textual version of the feature diagrams.

7. CONCLUSIONS

The paper described a product–line architecture for Web applications, which are obtained as compositions of reusable components. By means of suitable mechanisms, which are assembled directly into the components, products can be instantiated and managed with a higher degree of flexibility. In fact, each component can be put to use by interactively binding its variation points to specific variants. Most important, this configuration activity can be performed not only during product instantiation but also after the application has been deployed. This is due to the variability determination mechanisms part of the components and directly invocable by the system by means of reflective techniques. Consequently, the stages of feature analysis and variation determinations, that usually are performed during domain analysis, can be postponed during application engineering (and even later after the application deployment).

The architecture has been totally implemented together with management tools which allow the user to administrate both the system and the derived products.

8. ACKNOWLEDGMENTS

We thank Pasquale De Medio for the invaluable work in implementing the Koriandol system and coordinating the student programmers. Also, we are grateful for the insightful comments we received from the anonymous reviewers.

9. REFERENCES

[1] L. Balzerani. Problemi di generazione e configurazione dei sistemi a componenti. Tesi di Laurea in Informatica, Università degli Studi di L'Aquila.

[2] Don S. Batory, Rich Cardone, and Yannis Smaragdakis. Object-oriented frameworks and product lines. In P. Donohoe, editor, *Procs. 1st Software Product Line Conference*, pages 227–247, 2000.

[3] J. Bosch. *Design and Use of Software Architectures – Adopting and evolving a Product-Line Approach*. Addison-Wesley, 2000.

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

[5] P. Clements, L. M. Northrop, and et al. A framework for software product line practice, version 4.2. Technical report, SEI Carnegie Mellon University, Pittsburgh, 2004.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[7] D. Di Ruscio A. Pierantonio G. De Angelis, P. De Medio. Koriandol project site, 2004. http://koriandol.sourceforge.net.

[8] Lars Geyer and Martin Becker. On the influence of variabilities on the application-engineering process of a product family. In *Procs. 2nd Int. Conf. on Software Product Lines*, pages 1–14. Springer-Verlag, 2002.

[9] J. Van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Procs. Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, 2001.

[10] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[11] M.D. Jacyntho, D. Schwabe, and G. Rossi. A software architecture for structuring complex web applications. *Journal of Web Engineering*, 1(1):37–60, October 2002.

[12] M. Jaring and J. Bosch. Representing variability in software product lines: A case study. In *Procs. 2nd Int. Conf. on Software Product Lines*, pages 15–36. Springer-Verlag, 2002.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and P. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, SEI Carnegie Mellon University, 1990.

[14] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[15] PHP Application Tools. patTemplate, 2004. http://www.php-tools.de.

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