Journal of Web Engineering, Vol. 5, No. 1 (2006) 025–042

SUPPORTING WEB APPLICATIONS DEVELOPMENT WITH A PRODUCT LINE ARCHITECTURE

LUCA BALZERANI, DAVIDE DI RUSCIO, ALFONSO PIERANTONIO
Dipartimento di Informatica, Università degli studi di L'Aquila
Via Vetoio, I–67010 L'Aquila, Italy
{*balzerani|diruscio|alfonso*}*@di.univaq.it*

GUGLIELMO DE ANGELIS
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Via G. Moruzzi 1, I–56124 Pisa, Italy
guglielmo.deangelis@isti.cnr.it

Received April 19, 2005
Revised September 9, 2005

Web applications have become crucial elements of the global information infrastructure, evolving from simple collections of static pages to distributed applications. Since 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 pursue synergy effects in software development.

The paper illustrates Koriandol, a product line architecture designed to develop, deploy and maintain families of Web applications. Specific family members are assembled from reusable components which support variability determination through built–in reflective mechanisms. These provide the ability to bind variation points to specific variants even post deployment, making applications widely reconfigurable.

Keywords: Koriandol
Communicated by: S Comai

1 Introduction

Over the last years, Web applications have become crucial components of the global information infrastructure, evolving from simple collections of static pages to distributed applications intended as hybrid between hypermedia and information systems [1]. The economic relevance and the increasing intricacy of such applications have often shown the limitations of many ad-hoc and spontaneous development processes and have demonstrated the need for techniques and models that can offer a greater return on development time and quality factors.

Shifting the focus from the design of single applications to that of system families is an effective way to attain synergy effects in software development. Web applications often share similar behaviors whose commonalities and systematic variabilities can be exploited to effectively pursue planned software reuse. Indeed, software family engineering aims at developing a reuse infrastructure by anticipating the different product contexts and delaying design decisions to a later moment in the software development process [2]. These delayed design decisions are often referred to as variation points. Profiting from this potential demands

adequate planning and management of the reuse approach as otherwise relevant economic benefits will be missed due to an incongruous alignment of the reuse infrastructure. 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 [3], a product line architecture [4] (PLA) to design, deploy, and maintain families of applications. Domain–specific abstractions are captured by generic components which are designed in a prescribed way such that can be easily accommodated in the system which takes full advantage of the built–in variability management mechanism. 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 background notions about product line engineering are given. Section 3 presents the ideas behind Koriandol as a Web–specific PLA and provides insight into the development process, the organizational model and the use of reflection to handle variability. Section 4 describes tool implementing the architecture exposed in the previous section. Next section discusses an example case study. Section 6 relates the work presented in this paper with other approaches. Finally, a section devoted to the conclusions and future work 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 [5]. 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 [4]. A software product line captures commonalities between software products for a product family, aiming to a systematic reuse of core assets for building related products. The core idea of software product line engineering is to develop a reuse infrastructure that supports the software development for a family of products.

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 [6]. Software product line approaches accrue benefits at multiple levels, resulting in a competitive advantage for organizations that adopt them [7]. Once the product line core asset repository is established, there is a direct savings each time a product is built due to reuse of the core assets in a strategic and prescribed way. Furthermore, overall software quality is enhanced since each new system take advantage of all of the defect elimination in its forebears. Among the benefits of product line approches it's worthwhile to mention large–scale productivity gains, decreased time–to–market, increased product quality, increased customer satisfaction, more efficient use of human resources, ability to effect mass customization, to maintain market presence, and to sustain unprecedented growth. By using a software prod-

uct line, developers are able to focus on product specific issues rather than on questions that are common to all products [2]. 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 [8]. Variability is realized through *variation points* (originally introduced in [9]), i.e. places in the design or implementation that are necessary to achieve some functionality variance. During software development, many design decisions are taken while others are postponed. Variation points correspond to such delayed design decisions, i.e. decisions that are deliberately left open in order to support variability. Once variation points are introduced, they need to be resolved later on during the software lifecycle. Each variation point has an associated set of *variants* which express its variability; variants can be defined as the different ways a variation point can be resolved to. In essence, a variation point can be considered a formal parameter which is eventually actualized by a variant. 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 run time [10].

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 [4]. 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 for capturing commonalities and variabilities in system families and product lines. Several methods have been proposed to model features; among these, Feature–Oriented Domain Analysis (FODA) [11] is often referred to as one of the most emerging. FODA is based on feature models which represent the common and variable features of concept instances, and their interdependencies. A feature model consists of a feature diagram and 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 [12].

- 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 [7] among others) have been documented, along with successful product line practice patterns. An updated *hall of fame* is maintained by the Software Engineering Institute [13].

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 [14] is a PLA designed to develop, deploy and maintain families of Web applications. 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. 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.

3.1 Design and development process

The process of designing and developing a Koriandol–based Web application can be described in terms of domain and application engineering, where the former is devoted to the production of shared assets which the latter uses to instantiate concrete products. It worths to remark that Koriandol architecture captures commonalities among Web applications, while domain–specific commonalities and variabilities are addressed by components, which are specialized bodies of knowledge, areas of expertise, or collections of related functionalities.

According to the literature (see for instance [15]) our approach to product line development can be considered as *proactive*, since the development starts with the core assets. Organizations which take this kind of approach define their product line scope, which provides a mission statement for designing the architecture, components, and other core assets with the right built–in variation points to cover such scope. Producing any system within that scope becomes a matter of exercising the variation points of the components and architecture that is, configuring and then assembling and testing the system [13]. The fundamental activities can be then described according to the following:

1. **Domain analysis.** During this activity the application domain is analyzed to define its scope and to find out common and variable features of the systems in that domain. The main goal of this phase is to identify the places where variability can occur and to explicitly represent it with a domain model, so that we can reason about system families rather than single applications. Work products of domain analysis are domain definitions, dictionaries, and concept models. Among these, feature models are used to represent variability, distinguishing mandatory, optional and alternative features, as prescribed by FODA.

2. **Domain design.** The primary objective of the domain design phase is to develop an architecture for the system family. This task has already been accomplished in Koriandol through the development of a common architecture for all Web applications, independently from the considered specific domain. Even when a new application domain is approached, such architecture does not need to be modified since domain–specific issues are completely addressed in components.

3. **Domain implementation.** Once designed the common architecture, components and other assets have to be implemented. This is done during the domain implementation phase. Since Koriandol already provides a ready–to–use architecture, development is restricted to component realization. More explicitly, after a feature has been identified and modeled, a reusable component which implements it is developed. This activity corresponds to designing and implementing an independent subsystem with its own model, business logic and eventually presentational units. When completed, the component is registered into the repository and becomes available to all applications. In some cases it is not necessary to create a component from scratch; this happens when the required component can be obtained by adaptation of an existing one. Developers have to decide whether to use the existing component as a basis to create a new one, or extend it with new behaviors, increasing its variability and thus widening the system family scope.

4. **Requirements analysis.** During the application requirements analysis, functionalities (e.g. dynamic content generation) that cannot be obtained with static elements are identified. Our experience in Web applications development has shown that such services are often recurrent and can be reconducted to common *abstractions* whose behavior exhibits a certain degree of variability. When a new abstraction is discovered, it's submitted to domain analysis in order to investigate its variability. An abstraction groups a set of related features and is itself a feature. Thus there is a strong feedback between the requirements elicitation and domain analysis steps.

5. **Product configuration.** During this phase, features are mapped to components which are selected and instantiated into pages. If one or more features have no components implementing them, the abstraction identification and component realization steps are iteratively repeated until all needed components are available.

6. **Custom development.** While a system family can be arbitrarily wide, a certain amount of custom development is often needed to complete a particular application. For instance, an important aspect of every Web application is presentation, which involves

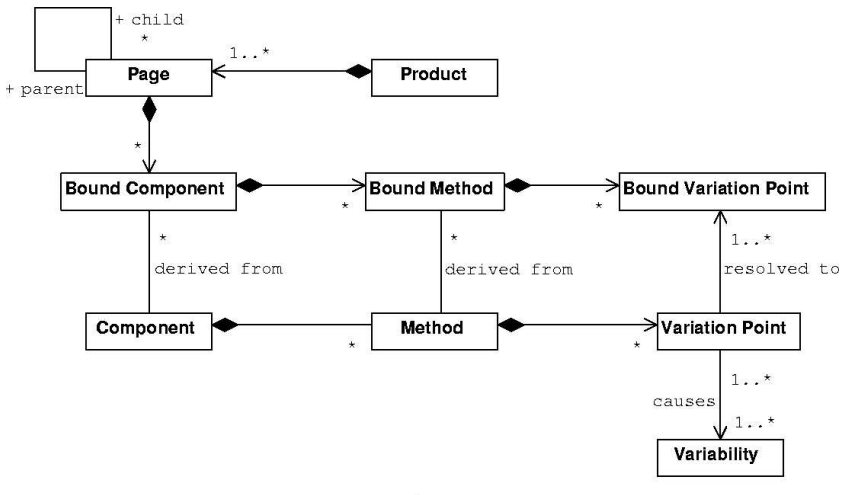


Fig. 1. Associations among the assets in a product

the development of graphical templates and other static elements such as images, style sheets and so on. This type of work products is almost always application–specific and is not reused among different systems.

3.2 Organizational model

According to the organizational model provided by Koriandol, a Web application can be viewed as an association of pages and components as depicted in the UML class diagram of Fig. 1. In particular, a Web application is a Product consisting of a number of Pages that are arranged hierarchically. Each page consists of contents dynamically provided by the selected Components embedded in a skeleton layout which contains others static informations and grafical contents. 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.

3.3 Variability handling through reflection

In Koriandol we use reflection to introspect components and discover the functionalities they provide, which include dynamic content generation, data entry facilities and many other. We have introduced a kind of introspection specialization in order to distinguish among these different types of functionalities.

As said above, components also embody a built–in variability handling mechanism which allows application engineers to dynamically bind each variation point to the appropriate vari-

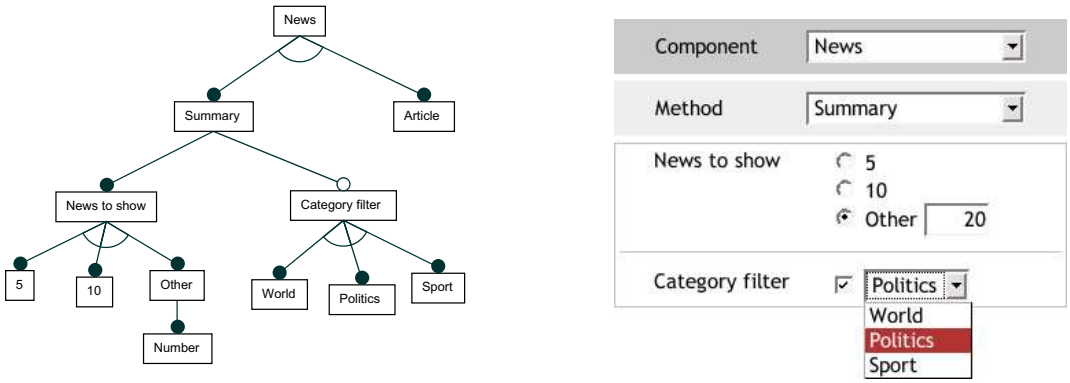


Fig. 2. Feature diagram for a News component

ant. 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 FODA 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 summarized, and an optional one which applies a category filter. The corresponding variability handling mechanism, as rendered in a Web browser, 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 [16].

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.

3.4 Component Structure

Components in Koriandol are pluggable structures which require to be recognized by the system, thus they must be endowed with some machinery in order to make the system aware

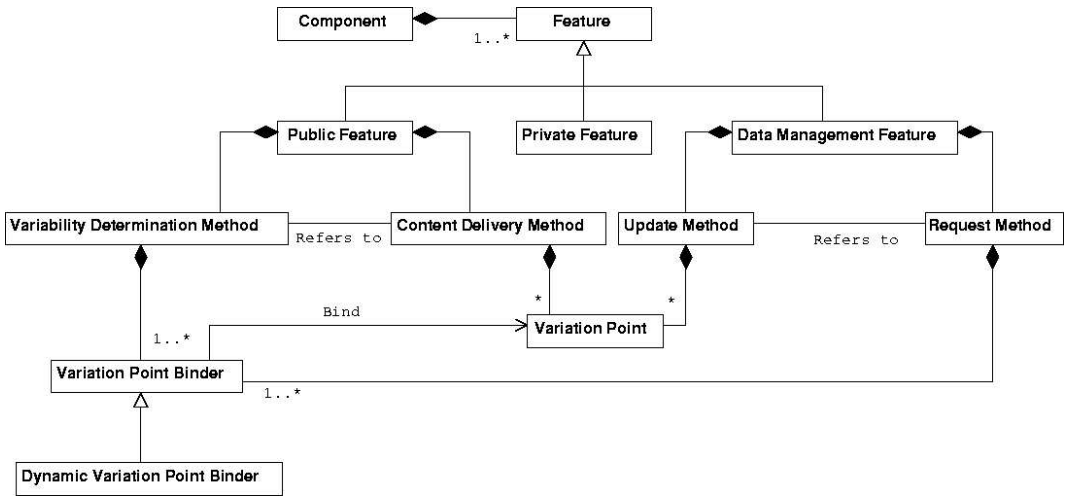


Fig. 3. Component Metamodel

of their functionalities.

The metamodel in Fig. 3 specifies the structure of a Component whose features are distinguished among private, public and of data management, respectively. In particular, the Private Features are methods whose scope is local and are intended as auxiliary functions to be used from within the component. The Public Features serve for two different purposes: on one hand they are used to perform the variability determination during the instantiation of a product; on the other hand they deliver semi–structured contents in response to a client request. Thus a public feature consists of the composition of a Variability Determination Method which Refers to a Content Delivery Method. The latter usually presents some genericity through one or more Variation Points, i.e. formal parameters which have to be provided when the feature is put to use and which are the same as the diagram in Fig. 1. The instantiation of such parameters is realized by the variability determination method that for each variation point uses the corresponding Variation Point Binder. In particular, a binder defines the functionality for determining the specific variant among a collection of admissible values, consequently the referred content delivery method is parameterized with such values in order to return the result. In essence, each feature can be customized according to a wizard which is obtained by aggregating the functionalities of the binders associated with each feature's variation point.

With reference to the example proposed in Sec. 3.3, the News component has the Summary content delivery method which implements the feature for publishing the news summary. The correspondent variability determination method is also provided by the component for determining a number of parameters, such as the number of items which have to be listed in the summary, eventual category the news have to refer to, and so on. Thus, the designer can customize the way a feature is put to use. The right hand side of Fig. 2 illustrates how, for instance, the variation point Category filter is instantiated by means of the correspondent form fragment provided by the referred binder.

Each component manages also the data aspects. In fact, the contents provided by a com-

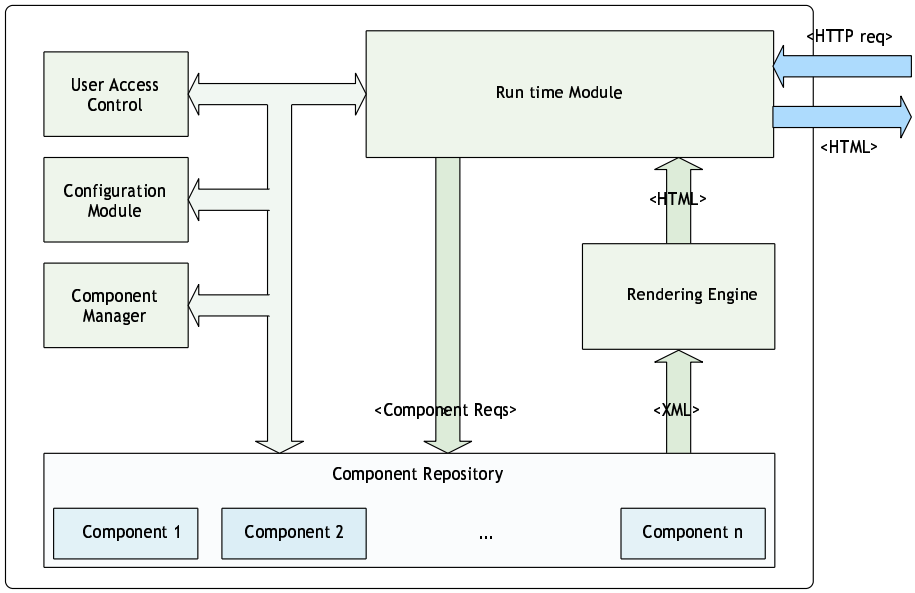


Fig. 4. Koriandol system

ponent are usually stored in a database fragment handled by the component itself through the Data Management Features. Analogously to the public ones, such features are the composition of the Request Methods with the Update Method, where the former relies on the variation point binders for building another wizard which serves for the data management which is, in turn, performed by the latter.

In some cases, not all the variation points can be bound at static time, such as during the product instantiation or a configuration stage. This is due to the fact that a great deal of information is available only at run time and specifically is related to data–driven navigation, i.e. *contextual links* [17] able to propagate data keys eventually used to retrieve contents from a database. In order to have such information available, the Dinamic Variation Point Binder denotes meta–information used to instruct the run time module how to extract data from the HTTP query string.

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. Additional modules are also present and mainly devoted to the management and configuration handling as illustrated in the simplified schema of Fig. 4. The modules are compositionally arranged in order to conform to the Model–View–Controller [18] (MVC) architectural pattern, which essentially aims at minimizing the degree of coupling between the user interface and the underlying data models proposing a three-way factoring paradigm as follows:

– the model holds all data relevant to domain entity or process, and performs behavioral processing on that data;

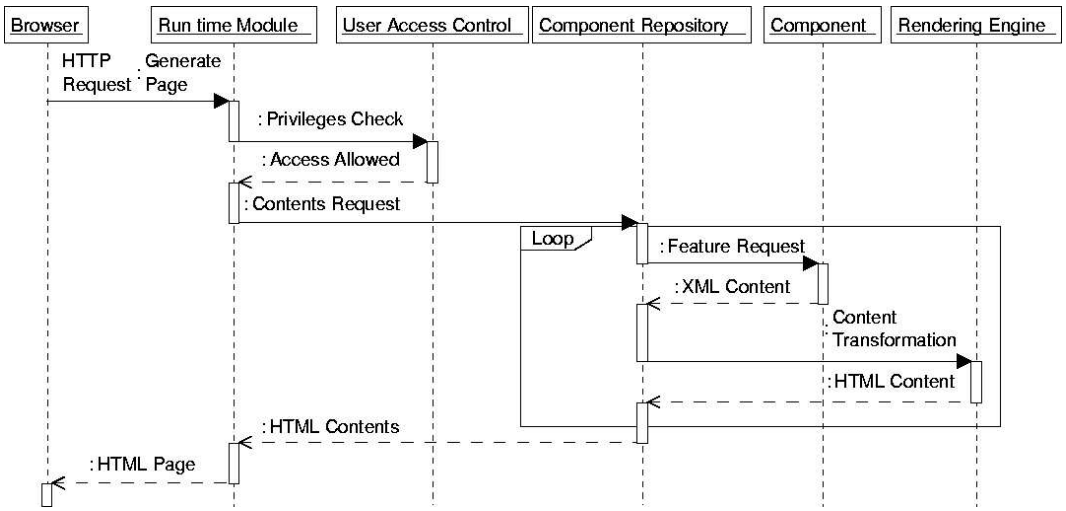


Fig. 5. HTTP request resolution

– the view displays data contained in the model and maintains consistency in the presentation when the model changes; and

– the controller is the glue between view and model reacting to significant events in the view, which may result in manipulation of the model.

According to this architecture, the run time module and the presentation engine play the role of the front controller [19] and the view, respectively. However, the model has not a unique correspondence because the data layer is scattered in different components, each of which representing an independent subsystem. In fact, models are those components of an application that actually do all the work. They are kept quite distinct from views, which display aspects of the models. Controllers are used to send messages to the model, and provide the interface between the model with its associated views and the interactive user interface devices.

Each incoming HTTP request triggers a number of interactions between the different modules as in the sequence diagram of Fig. 5 where the dynamics of a client request is illustrated. In particular, the incoming client request, denoted by the HTTP Request message, is sent from the Browser to the Run time Module, which interprets it according to the correspondent page specification. In other words, it validates the user privileges through the Privileges Check action and identifies which components have to be invoked to serve that particular request (Contents Request action). Each page functionality is obtained through component requests, i.e. a Feature Request action is sent to the component that was selected and configured during the variability determination for the instantiation of the requested page. The method returns XML data which is passed to the Rendering Engine (through the Content Transformation action) which generates HTML fragment by means of transformation stylesheets. Once all component requests are realized (by implying the exit from the Loop) the obtained HTML segments (HTML Contents) are put together and returned to the user (HTML Page).

A prototypical implementation of the Koriandol is available for download on SourceForge [14]. In the sequel of the section, the core units of the system are described referring to Fig. 4.

4.1 Management/Configuration module

The management module provides for support to manage users, components, and products 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 in Koriandol as an interface specification as described in Sec. 3.4. The variability determination mechanism is also included in the components in order to provide 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 operations 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 module

The role of the run time environment is on one hand to coordinate all the activities among the modules once a request arrives from a client; 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 Rendering 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.[a] 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]The template language and engine which has been adopted is *patTemplate* [20].

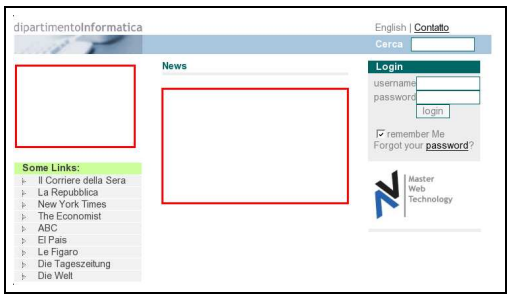


Fig. 6. Home pages layout

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.

This section illustrates a product configuration which aims at creating a simple news portal. In this example only functional requirements, in the sense of [21], are considered. The application is supposed to be logically organized as a tree: the home page of the news portal represents the root and the first level nodes represent the home pages for each news category. As navigational requirements the application home page has to provide access to each category ones. In order to satisfy these needs, the home page interface provides a menu to home pages of the different categories and to the most recent article. According to this arrangement, each category home page contains a summary of the corresponding news and a menu similar to the one provided by the application home page. As interface requirements, each summary entry shows the date, the abstract of the news and a link for accessing the whole article. When the user select a news item, the application computes the request and redirect the user to the target page where the selected body news is shown.

The product instantiation begins with the elicitation of the required assets, more specifically a layout for the portal and category home pages, a layout for the remaining pages, and the news and navigational components which are supposed to be already available. Each layout contains static graphical contents and a number of specific place–holders to denote where dynamic contents can be filled as illustrated in Fig. 6. Such place–holders are instantiated with the News and Navigator components whose diagrammatic structures are reported in Fig. 8 and Fig. 9, respectively. The outcome consists of the pages in Fig. 7.

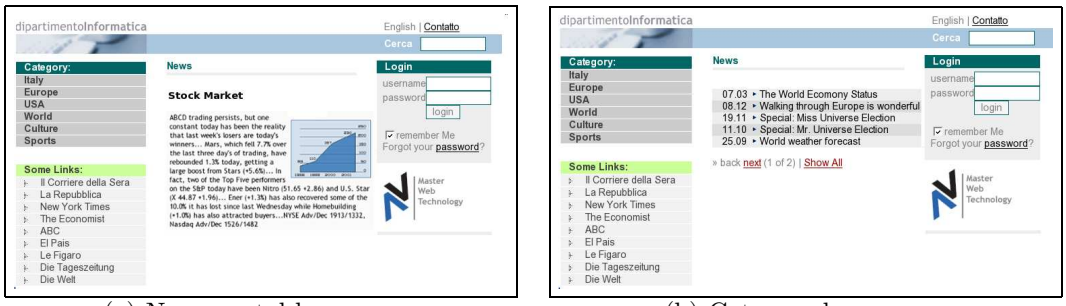
The News component provides two content delivery methods: Summary yields the list of the available news, while Article provides the body of a news. As already mentioned, the use of features requires the determination of the component variability. In the example, Summary is parameterized with the number of newsToShow in accord to a categoryFilter; additional parameters concern the date, the abstract, the associated image and the reference to a page where the complete article is visualized. The Article feature needs the specification of the news identifier whose value will be given during news browsing as explained at the end of the section. The other component, Navigator, defines some navigation functionalities, such as

Summary variation point	Value
newsToShow	1
abstract	true
category	false
img	true
date	false
target	64

Table 1. Summary parameter configuration

TreeMenu which starting from the rootPage returns the links to all reachable pages within a certain depth and a menu PathMenu which returns the path from an ancestor page (ancPage) to the current one.

As in any other product line, the application development corresponds to a product instantiation, where the designer chooses the desired feature. In this case, by means of the configuration module each place–holder is bound to a specific feature providing the desired content. In particular, in order to instantiate the central place–holder in the news portal home page, the configuration modules retrieves the list of the available components (see Fig. 10). As the designer picks the News, the system introspects the component in order to obtain the features which can be used, i.e. the News components returns Summary and Article. Once Summary is chosen, the configuration module invokes the variability determination method associated with Summary in order to allow the designer to enter the configuration parameters. In Tab. 1 are reported the values for the variation points of the Summary feature which allow to obtain a page containing the last issued news with an associated image and the abstract as illustrated in Fig. 7.a. The parameters for the variation points are in turn stored by means of the run time module. The configuration for the other place–holders is accomplished by following analogous steps. A special case is represented by the page which shows a complete article, since such content depends on data which are retrievable only dynamically by means of the dynamic variation point binders, as described in the previous sections. In fact, the Article feature requires a unique identifier which has to be encoded in the HTTP query string; the dynamic variation point binder inferArticleToShow instructs the Run time Module to find such



(a) News portal home page (b) Category home page

Fig. 7. Home pages

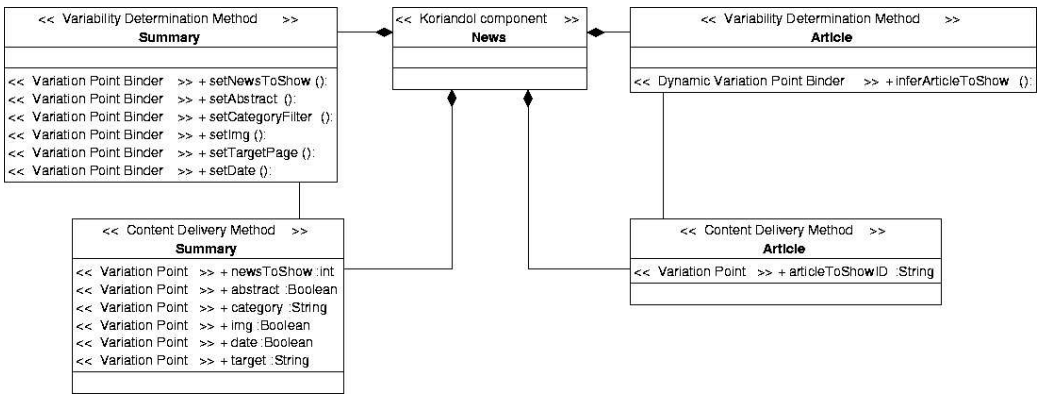


Fig. 8. Logical News component structure

an information and to set the variation point articleToShow.

The interactions depicted in the sequence diagram in Fig. 10 explain how it is possible to have variability determination both during product instantiation and in any post–deployment phase. Especially, the configuration module can be executed without requiring to stop the run time module.

6 Related Work

Most of the literature on software product lines ([13, 8, 22] 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 [23] 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 MVC architecture as extensions. The main difference with Koriandol is in the lack

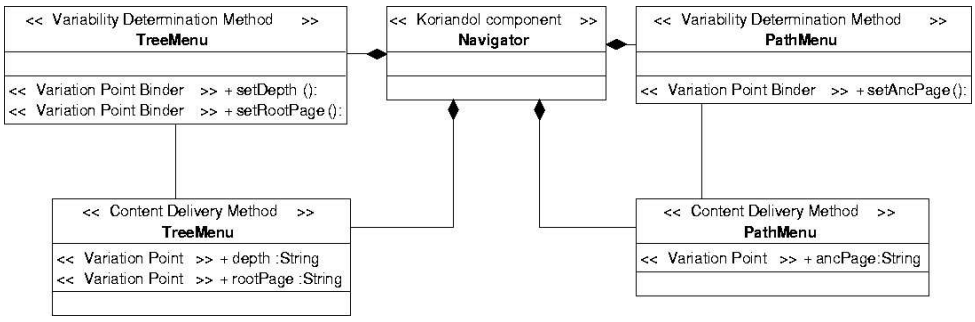


Fig. 9. Logical Navigator component structure

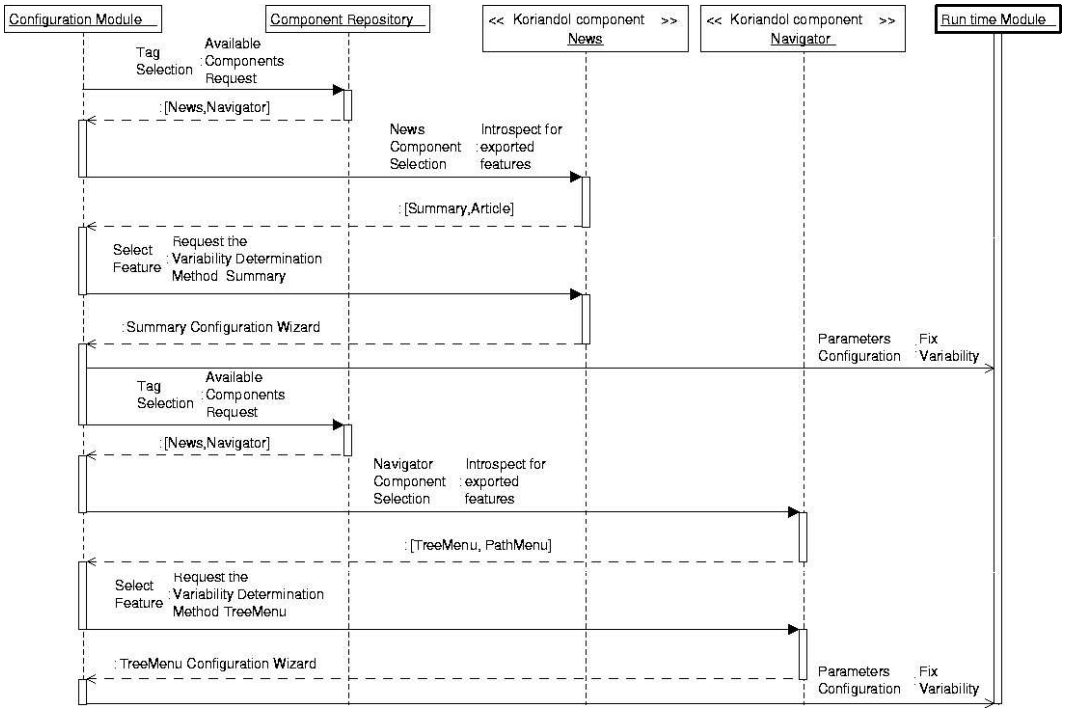


Fig. 10. Home Page Configuration Example

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

The last decade witnessed the development of several Web modeling languages such as OOHDM [24], OO-Method [25], UWE [26], W2000 [27], Webile [28] and WebML [17]. None of them encompasses any support to variability handling or to any other product family concept, nevertheless they have been recognized as being among the most prominent modeling methods for Web applications. In [29] variability in Web applications development is supported at a limited extend for documentation and communication purposes only. In particular, architectural and design patterns are used to describe the variability which can be introduced in a design process. Another approach [30] proposes product lines to re-engineer existing websites to let them migrate to system families. The representative assets are Web files whose code is inspected to extract the common and the variable points in order to identify core assets such as JavaScript, ASP, PHP or HTML code. Such approach differs from Koriandol which uses components, rather than code, as primary core assets that can be assembled in a prescribed way in order to provide required behaviors.

More efforts have been made in applying component–based techniques to Web applications development. For instance the WebComposition approach introduced in [31] allows modeling Web applications from components. WebComposition components are defined and represented using the Web Composition Markup Language, an XML application.

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) [11] 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 [16] which is a data–intensive extension of a textual version of the feature diagrams.

7 Conclusions and Future Work

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.

We are currently investigating a convergence scenario among our proposal and model–centric ones to provide with an approach where applications are conceptually modeled by explicitly dealing with variability and in turn generated through automatic transformations by imposing Koriandol as target platform. In particular combining product lines and model–driven architectures may address some of the shortcomings of the former and makes the benefits of the latter available in the context of product families (see [32]). Admittedly, Koriandol is more implementation–oriented and lacks an high–level design model, so extending it with conceptual modeling capabilities would result in a significant improvement. On the other hand, model–centric methodologies are often focused on the design of single applications and cannot properly deal with variability, so managing system families is a problematic task. Such approaches can therefore take advantage both from product lines concepts (e.g. the distinction among mandatory and optional features) and Koriandol variability support.

Several convergence strategies are possible. The first one is having model–to–code transformations to produce Koriandol–compliant code. This way, it would be possible to introduce variability in applications generated from abstract models at the component level, and to widen their scope. Furthermore, generated applications would become dynamically reconfigurable even after deployment, like any other Koriandol–based product. Another possibility is to enrich the application modeling language, adding support for variability representation. The advantages would be twofold: (1) the modeling language wouldn't be dramatically changed but just extended with new constructs, and (2) this would give the opportunity to use Koriandol components in the models (e.g. in the form of packages), allowing to represent

even complex code–intensive behavior.

References

1. P. Fraternali. Tools and approaches for developing data-intensive Web applications: a survey. *ACM Computing Surveys*, 31(3):227–263, September 1999.
2. 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)*, pages 45–54, Amsterdam, 2001. IEEE Computer Society.
3. L. Balzerani, G. De Angelis, D. Di Ruscio, and A. Pierantonio. A product line architecture for web applications. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC)*, pages 1689–1693, Santa Fe, New Mexico, USA, March 13-17, 2005.
4. J. Bosch. *Design and Use of Software Architectures – Adopting and evolving a Product-Line Approach*. Addison-Wesley, 2000.
5. C.W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.
6. D.S. Batory, R. Cardone, and Y. Smaragdakis. Object-oriented frameworks and product lines. In P. Donohoe, editor, *Procs. 1st Software Product Line Conference*, pages 227–247, 2000.
7. P. Clements and L.M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
8. M. Jaring and J. Bosch. Representing variability in software product lines: A case study. In *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, pages 15–36, San Diego, CA, August 2002. Springer.
9. I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.
10. L. Geyer and M. Becker. On the influence of variabilities on the application-engineering process of a product family. In *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, pages 1–14, San Diego, CA, August 2002. Springer.
11. K. Kang, S. Cohen, J. Hess, W. Novak, and P. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI Carnegie Mellon University, 1990.
12. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
13. 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.
14. G. De Angelis, P. De Medio, D. Di Ruscio, and A. Pierantonio. Koriandol project site, 2004. http://sourceforge.net/projects/koriandol/.
15. C.W. Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product Family Engineering*, pages 282–293, 2002.
16. L. Balzerani. Problemi di generazione e configurazione dei sistemi a componenti, 2004. Tesi di Laurea in Informatica, Università degli Studi di L'Aquila.
17. S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a Modeling Language for Designing Web sites. *Computer Networks*, 33(1–6):137–157, 2000.
18. G.E. Krasner and S.T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object-Oriented Programming*, 1(3):26–49, 1988.
19. D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns*. Sun Microsystems Press (Prentice Hall), 2nd edition, 2003.
20. PHP Application Tools. patTemplate, 2004. http://www.php-tools.de.
21. M. J. Escalona and N. Koch. Requirements engineering for web applications - a comparative study. *Journal of Web Engineering*, 2(3):193–212, 2004.
22. D.M. Weiss and C.T.R. Lai. *Software Product-Line Engineering: A Family Based Software Development Process*. Addison-Wesley, 1999.
23. M.D. Jacyntho, D. Schwabe, and G. Rossi. A software architecture for structuring complex web

applications. *Journal of Web Engineering*, 1(1):37–36, October 2002.
24. D. Schwabe, G. Rossi, and S.D.J. Barbosa. Systematic hypermedia application design with OOHDM. In *Proceedings of the Seventh ACM Conference on Hypertext*, Models of Hypermedia Design and Evaluation, pages 116–128, 1996.
25. J. Gòmez and C. Cachero. Oo-h method: extending uml to model web interfaces, 2003.
26. N. Koch and A. Kraus. The expressive power of uml-based web engineering. In *IWWOST*, volume 2548 of *LNCS*, pages 105–119. Springer, 2002.
27. F. Garzotto, L. Baresi, and M. Maritati. W2000 as a MOF metamodel. In *The 6th World Multiconference on Systemics, Cybernetics and Informatics - Web Engineering track*, July 2002.
28. D. Di Ruscio, H. Muccini, and A. Pierantonio. A Data Modeling Approach to Web Application Synthesis. *Int. J. Web Engineering and Technology*, 1(3):320–337, 2004.
29. R. Capilla and N.Y. Topaloglu. Representing Variability Issues in Web Applications: A Pattern Approach. In *Computer and Information Sciences - ISCIS 2003*, volume 2869 of *LNCS*, pages 1035–1042, January 2003.
30. R. Capilla and J. C. Dueas. Light-Weight Product-Lines for Evolution and Maintenance of Web Sites. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 53, Washington, DC, USA, 2003. IEEE Computer Society.
31. H.W. Gellersen, R. Wicke, and M. Gaedke. Webcomposition: An object-oriented support system for the web engineering lifecycle. *Computer Networks*, 29(8-13):1429–1437, 1997.
32. D. Muthig and C. Atkinson. Model-driven product line architectures. In *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, pages 110–129, San Diego, CA, August 2002. Springer.