Domain-Specific Runtime Variability In Product Line Architectures

Michael Goedicke, Klaus Pohl, and Uwe Zdun

Institute for Computer Science, University of Essen, Germany
{goedicke|pohl|uzdun}@cs.uni-essen.de

Abstract A software product line primarily structures the software architecture around the commonalities of a set of products within a specific organization. Commonalities can be implemented in prefabricated components, and product differences are typically treated by well-defined variation points that are actualized later on. Dynamic, domain-specific aspects, such as ad hoc customization by domian experts, are hard to model with static extension techniques. In this paper, we will discuss open issues for dynamic and domain-specific customizations of product line architectures. We will also present an indirection architecture based on Component Wrapper objects and message redirection for dynamically composing and customizing generic components for the use in concrete products. As a case study, we will discuss two designs from a Multimedia Home Platform product line: end-user personalization across different new media platforms and customization of interactive applications by content editors.

1 Introduction

Software product lines or system families structure a set of products of a specific organization into a product line architecture that contains a set of more or less generic components. Each individual product architecture is derived from the product line architecture. Each product uses a set of generic components and introduces product-specific code as well. Generic components provide well-defined interfaces. In the products-specific code those generic components are configured for the use in the product's architecture.

The typical process of adopting and/or using the product line approach concentrates on reducing structural complexity by finding and extracting commonalities. The system is primarily built from a common set of assets [1]. Often these common assets are desinged as black-box components. In the industrial practice such building blocks may have a substantially different form than in an ideal academic view [2]. Industrial components are usually very large and have a complex internal structure with no enforced encapsulation boundaries. Often there is no explicit difference between interface entities and non-interface entities.

In software product line approaches, component configuration is usually handled by well-defined variation points, implemented with appropriate variability mechanisms. Common (traditional) variability mechanisms are parameterization, specialization, or replacement of entities in the reusable component. The designer decides for variability mechanisms and architectural styles. Once a design and implementation is based on a certain variability mechanism, in traditional approaches, it is often quite hard to exchange them with other mechanisms.

One of the main contributions of the product-line approach is its focus on domain-specific architectures. A product-line should provide a systematic derivation of a tailored approach suited to an organization's capabilities and objectives [4]. Therefore, the approach pays special attention to the traceability between architectural decisions and functional and non-functional requirements. However, many domains impose requirements for domain-specific customizations that can hardly be implemented with variation points that are bound before runtime. In this paper, we will concentrate on techniques that can be applied to introduce domain-specific ad hoc customizability, e.g. for domain experts. To reach that goal in a large product-line, it is important for domain experts to be able to understand a product's variabilities without requiring to learn about other parts of the architecture.

We have studied these issues theoretically and practically in three larger industry projects: TPMHP (focusing on a generic product line architecture for development of digital business television applications on top of the MHP (Multimedia Home Platform) [5]), ESAPS and CAFÉ [6,11] (aiming at engineering software architectures, processes, and platforms for system-families), and a document archive system [10]. In these projects, late-bound flexibility is not only a useful feature, but a requirement for using a product line approach at all. For instance, domains like web engineering are characterized by constant *domain changes*. For rapid incorporation of these changes it is impractical to hand-code the changes in long development and deployment cycles. In some product lines customer-specific *customization requirements* are foreseeable, and then rapid customizability, ideally even by non-programmers, is required. Sometimes variations are dependent on the *runtime context*, and thus, such product lines require runtime variability. Many *24×7 server applications*, such as custom web servers and application servers, usually cannot simply be stopped for deploying new components. Thus in such domains a dynamic component exchange mechanism is required.

In this paper, at first, we will outline open issues in designing variation points in Section 2 that are a prerequisite to practically implement domain-specific variation points. For resolving these open issues, we will present an object-oriented runtime indirection architecture for encapsulating variation points in Section 3. For implementing the architecture we require runtime variability mechanisms, and different runtime variability mechanisms have different consequences. As a case study, in Section 4, we will introduce practical runtime customization requirements from the TPMHP project, and discuss solutions based on the architecture presented beforehand.

2 Open Issues in Designing Variation Points

In this section we will discuss a set of open issues during design of runtime variation points for domain-specific concerns in product line architectures. Variation points are implemented using a variability mechanism. Typical traditional examples for variability mechanisms are association of an object in Decorator style [7], delegating to a Strategy [7], using inheritance for specialization, exchanging a runtime entity (such as an object), parameterization, and preprocessor directives. Obviously all these variability mechanisms have quite different properties. For instance, the binding time differs: some mechanisms have to be bound at design time, some at compile time, some at startup of the program, some at runtime. In general, the later we bind a variation point, the more flexible the solution is. However, we have to deal with certain drawbacks as well, for instance, by binding at runtime, the performance, memory consumption, and runtime complexity may be influenced negatively.

When we build complex design dependencies or architectural artifacts, such as variation points, on top of traditional variability mechanisms we face a set of potential problems if domain-specific customizations have to be rapidly incorporated (some of these issues are also identified in [3]):

- *First-class representation:* An architectural artifact does not have a first-class representation in design and programming languages. Thus the recurring use of the artifact has to be built with certain syntactic conventions. The resulting designs and programs are harder to read. Moreover, the constructs used to implement the architectural artifact cannot be easily reused. For a domain expert, who is not a technical expert for the product line design and implementation, this means the design and implementation are impenetrable because her/his task-specific idioms do not map well to the design and programming language elements.
- *Traceability:* Without a first-class representation in the program code or design, the architectural artifact is not recognizable ad hoc as an entity. Therefore, the artifact as a whole is hard to trace; that is, (complex) variation points are hard to locate for a client of the component. At runtime, traceability can be provided using reflection or introspection mechanisms. Without such functionality, it is hard to extend a given product without intimate knowledge of the product line architecture's internals. A domain expert usually does not have an intimate knowledge of the internal implementation; thus, the system can only be customized by qualified programmers.
- *Implicit dependencies:* Dependencies between architectural elements and features are often only implicit. As a result it is not clear what parts of the product line architecture are needed for a specific product. For a domain expert that means that all dependent components have to be understood to understand a given component.
- *Scattering variation points:* Variability at the requirements level often does not map nicely onto programming language code. Thus, in naive implementations, features are simply scattered across system parts, and multiple features are tangled within system part. Scattered variation points heavily reduce understandability of the design and code, especially for non-technical people.

If the issues discussed above are not resolved, it may become hard to use, extended, and customize products that require rapid changeability. For each customization a larger, non-trivial programming effort is required. Often, initially found abstractions for variation points do not match the reality or the implementation well. Thus it is important that we are able to rapidly exchange variation point implementations and variability mechanisms during software maintenance.

3 Encapsulating Variation Points

In this section, we will present an architecture for indirecting component interactions, so that we can encapsulate variation points. Then we will discuss some choices for runtime variability mechanisms in the indirection architecture as well as its consequences.

3.1 Indirecting Component Interactions With Component Wrapper Objects

In this section we discuss an indirection architecture that enables us to treat different variability fields in a product line separately, but yet let the product line offer an integrating extension architecture. In general, we propose that each component is divided into three parts: a component implementation, an explicit export interface, and an explicit import interface. Export and import relationships are first-class entities of the programming (and design) language. For instance, export and import can be modeled as first-class objects, but they can also be modeled with runtime constructs specifically designed for component wrapping and interface adaptation, such as interception techniques for component composition [8]. Usually we provide explicit relationship introspection options for these export and import entities, so that we can trace at runtime which components are connected to which other components. Thus, we can trace the architectural construction of a product at runtime.

Note that there is only a single design and implementation step necessary to transform a given (industrial) component into a component fulfilling these conventions. That means, the industrial reality of large-scale components with no enforced boundaries [2] is not ignored. Moreover, the former interfaces do not have to be changed; that is, backwards compatibility can be ensured. Both issues are crucial to acceptance of the proposed architecture in industrial practice.

In Figure 1 we can see that the generic component parts, implementing the component's tasks, do not directly access the used components but use a central Message Redirector [9]. The Message Redirector is usually not implemented for each component but derived from the product line's extension architecture. It is used as a simple indirection mechanism: no external component is accessed directly but only with the redirector. A symbolic call is mapped to the correct interface in the Component Wrapper object [14] that defines the expected import from a component. The used component, in turn, provides the same structure. Thus it provides an explicit export interface as well. This provided interface is configured in the Component Wrapper to fulfill the expected interface. Thus, we can deal with slight interface changes and other necessary adaptations on the Component Wrapper. Concerns cross-cutting multiple components can be handled by the Message Redirector.

There are different ways to implement the architecture for component-oriented runtime variability sketched above. For different products and product lines different vari-

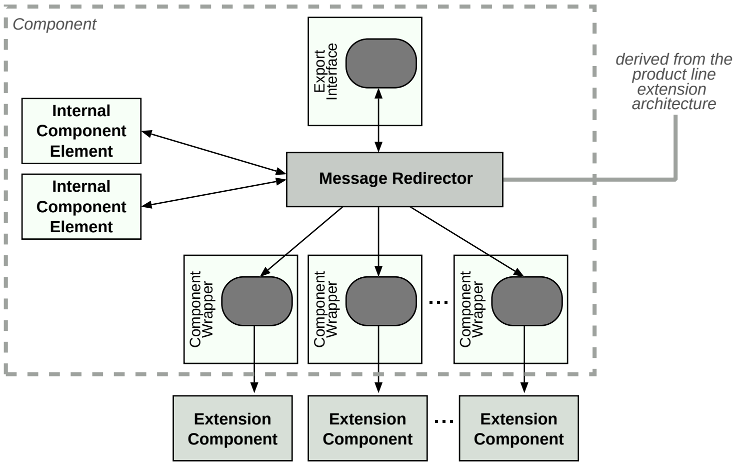

Figure 1. Variant encapsulation in a component-oriented indirection architecture (illustrated for an individual component)

ability mechanisms are useful. To implement the architecture we require runtime variability mechanisms, and in most cases more than one variability mechanism is used, including traditional mechanisms such as inheritance and parameterization.

In contrast to solely using traditional variability mechanisms, in an architecture, as described above, it is quite easy to exchange the used variability mechanisms or adapt their inherent properties to new requirements. This is mainly due to the central Message Redirector that allows for adaptations of the extension process of one or more components. This way the variability mechanism hidden by a symbolic Message Redirector call can be adapted without interfering with the component's internals.

As we can see, there are no impositions on how the component is implemented internally or which variability mechanisms are used. As a requirement, each component has to implement one or more component export interfaces as Facades to the component, and one or more component import interfaces as Component Wrappers to wrap the used components. Each arrow in Figure 1 indicates an explicit component connection that is traceable and dynamic at runtime. Component connections are handled via the Message Redirector; therefore, they expose runtime variability. The Message Redirector is an abstraction for the process of component composition; thus, variations in this process can be applied here. The extensional components, included in the component import, can be derived from the product line architecture or they can be part of the specific product. An interesting aspect is that product-line specific implementations of variability mechanisms are usually integrated as components as well.

3.2 Runtime Variability Techniques

Using the product line extension architecture presented, we can individually make the choice for a runtime variability mechanism for each element in the product line. We are able to use the most appropriate technique for introducing variability in each individual part of the product line. If necessary, it is also possible to change the choice in a component-oriented fashion during product or product line evolution. There are different techniques that we use for introducing runtime variability in our product lines, with different benefits and liabilities. There are different runtime variability approaches that operate at different abstraction levels and have different properties. In different customization situations, different domain expert participation requirements have to be mapped to these properties to find the best alternative. A few examples are:

- The most simple technique is to simply *design and program the variation point by hand* using traditional variability mechanisms. However, this strategy can lead to significant problems, including higher complexity, more maintenance efforts, etc.
- *Scripting* languages, such as Tcl, XOTcl, Python, or Perl, are widely used and accepted in industry for configuration, customization, test automation, and component glueing. In many projects scripting languages are used as languages for domain-specific customization. The scripting language commands can be used as symbolic indirections in the Message Redirector. However, if system languages are used as well, two or more languages have to learned by developers and the language models have to be integrated.
- Our *pattern language for flexible component architectures* [14] can be used for implementing the indirection architecture, sketched in Section 3.1. A benefit of the pattern approach is that the required expertise is rather low: a design with patterns can rather easily be understood by non-technical stakeholders. Thus patterns are a good means to convey and discuss design decisions in early phases without (a) having to specify the concrete variability mechanism used for implementation and (b) being too vague in the technical realm. As a drawback, the patterns have to be implemented before they can be used. This may be too large an effort for very small projects. For larger product lines often this is not problematic. Here, the patterns are used as a conceptual guidance for designing variation points by hand.
- Using *markup languages for customization* is a simple technique for implementing limited dynamic customizations of applications, as discussed in [13]. A markup languages such as XML is used to describe the customizable elements of an application, and application parts are dynamically generated from these XML files. XML tags are used as symbols in the Message Redirector, and the information architecture follows the hierarchical structure of the XML texts in form of a Composite pattern implementation. This technique, is simple and reliable, but of course, it has its limitations as it is only a powerful form of parameterization.

3.3 Consequences of Runtime Variability in the Indirection Architecture

In this section we discuss the contribution of the architecture to resolve the open issues in designing variation points, identified in Section 2, as benefits and potential liabilities of the architecture. There are the following benefits:

- In conventional implementations of our indirection architecture, for each architectural fragment there is a conceptual entity in design and implementation that is also

an entity in the Message Redirector. As the target language itself is not extended, a *first-class representations* is only partially archived. Some languages, as for instance Tcl, XOTcl, Smalltalk, and Lisp, do allow for language extension with components so that these components are actually *first-class representations* of architectural fragments in the language. Nonetheless, in all approaches domain-specific indirections can be provided to represent the idioms and structures of the domain.
- On the Message Redirector we provide a symbolic identifier for each component and for all imported and exported functionality. If we provide introspection options as well, we can ensure runtime *traceability* of architectural fragments in the product line.
- Avoiding *implicit dependencies* is a primary goal of the architecture. The components do not directly refer to each other but use the central Message Redirector. Features are usually encapsulated in architectural fragments. Therefore, dependencies among fragments and to the design-/requirements-level can be made explicit at runtime.
- Since variation points are encapsulated and only accessed through the Message Redirector, *scattering variation points* across the code can be avoided. However, in conventional implementations this is only a convention, and developers are able to violate the convention, when they use direct component accesses that bypass the Message Redirector.

As our indirection architecture relies on runtime variability, we also have to consider the following potential liabilities:

- *Runtime resources:* Performance, memory, and other runtime resources, may be negatively influenced by mechanisms such as reflection, dynamic linking, dynamic invocation, and interpretation.
- *Runtime complexity:* Additional runtime structures do also add more complexity to the runtime environment; thus, understandability and maintainability may be negatively affected. Moreover, if the variation points are hard to trace, it is hard to understand the (complex) interfaces of the product line.
- *Variation point management:* Adding and binding variants at a late moment in time implies extra work for managing, implementing, and documenting variation points.
- *Predictability:* A large amount of variability may make it virtually impossible to test all combinations during development. Mechanisms may be needed that allow testing at a late moment in time and that ensure the consistency of the products. Thus a certain amount of discipline among developers is required to create an extensive regression test suite.
- *Adjusting to product specifics:* The idea of one explicit interface used for multiple implementations implies the problem that products are reduced to their common denominator. In extreme cases, when a workaround is virtually impossible, it may be necessary to re-implement parts of the common product line implementation in concrete products. Usually, adaptations and extensions on the export and import interface objects let us avoid such problems.

A product line designer has to care for these issues in a very early stage of development. Often it cannot even be predicted which concrete requirements a product has, say, because they are not yet known or the impact of a change can only be found out by runtime testing. Usually the issues named above are highly context-specific and domain-specific, thus they can only be adjusted per product line, or sometimes only per product. The techniques, discussed in Section 3.2, entail the consequences named above to a different degree. Therefore, in each architectural fragment of the product line we should use the most appropriate approach for implementing runtime variability.

4 Domain-Specific Customization in an MHP Product Line

In this section we will discuss the problem of domain-specific customization in the context of the EU project "Technological Perspectives of the Multimedia Home Platform (TPMHP)" aiming at a product line architecture for the Multimedia Home Platform (MHP) [5]. The MHP specification is a generic set of APIs for a client-side software layer for digital content broadcast applications, interaction via a return channel, and internet access on an MHP terminal, such as digital set-top boxes, integrated digital TV sets, and multimedia PCs. The MHP standard defines a client-side technical specification for MHP terminal implementations, including the platform architecture, an embedded Java virtual machine implementation running DVB-J applications, broadcast channel protocols, interaction channel protocols, content formats, application management, security aspects, a graphics model, and a GUI framework.

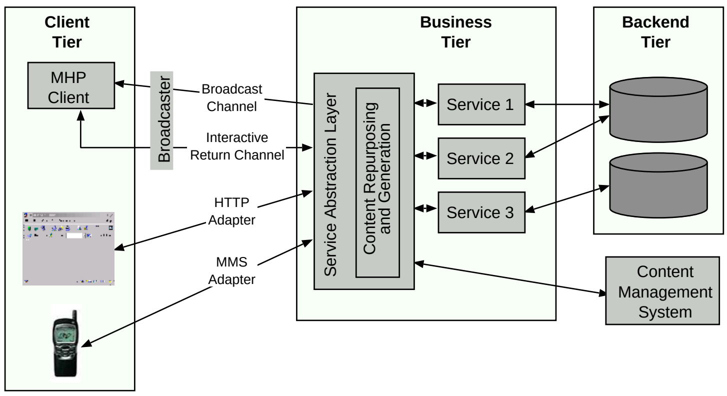

Figure 2. Service abstraction for MHP terminals, mobiles, and web browsers

In the project, we considered the situation of a large German warehousing company with multiple different stores as a content provider for the MHP. Different new media shop types, including web shops, MHP-based interactive television shops, and m-commerce shops (e.g. based on MMS) should be supported. The basic architecture is a Service Abstraction Layer [12], as depicted in Figure 2. Each of the customizable shopping applications is realized by one or more services. In this context we face multiple different requirements for runtime variability:

- *User Personalization:* On client-side, users can personalize their user experience. Customization happens on client-side with a simple (e.g. form-based or grapical) interface. The personalization information are stored on the server. The server generates personalized pages for different content formats, such as DVB-J Java classes, HTML pages, MMS pages, etc.
- *Shop Branding:* For the customer relationship it is important that each individual store has its own unique brand identity, including logos, layouts, banners, colors, etc. Such properties should to be visible for each of the platforms supported. Shop branding is handled by content editors, and it requires rather complex behavior that is not solely expressible by providing a CSS-like style sheet a priori. The customer should not experience the common product line realization during shopping, but should have the impression that each shop has an individual appearance. Thus, shop branding requires writing custom code for layout of the shop sites.
- *Customizing Interactive Applications:* Interactive applications are customized by domain experts and content providers. That is, simple interfaces for customization by these non-programmers are required. Other representations of user interaction such as web forms or applets are also generated from these information.

In the indirection architecture presented, each individual variability requirement can be implemented in a separate component. The locality gained by abstracting details of other components also reduces the knowledge that a domain expert has to have to perform a customization step. In the remainder of this section, we will present two examples of customizing MHP-based applications: first, we present an integration architecture for tracking client-side consumer customizations on server-side, and, secondly, we will discuss behavioral customization by domain experts.

4.1 Personalization on Client- and Server-Side

As usual in web portals, consumers have to be able to personalize their shopping environment. Usually, form-based pages or tools are provided for this task. However, as different platforms are supported, personalizations on the settop-box should also appear on the other platforms, such as the web and mobile devices, and vice versa.

As a solution for the MHP client, the personalization page implementation sends commands to a lightweight client-side Message Redirector (written in Java). The Message Redirector indirects the consumer's customization commands to DVB-J Java implementations for the MHP platform. Moreover, if required, it sends them with the interaction channel to the server as well. Another Message Redirector on server-side understands the same symbolic instruction set (and also other instructions) but maps the instructions to different components implementing server-side building, publishing, and caching of pages for the platforms on which personalization is handled on server side such as the web. This way, the same client-side customization do also appear on other platforms such as web browsers.

For integrated client-side and server-side customizations the following steps are performed, if the consumer performs a customization on the MHP client:

1. The consumer uses a form-based customization page or tool to customize or personalize the application logic. The tool creates little scripts entirely composed out of commands that can be understood by the client-side Message Redirector.
2. The client-side Message Redirector maps the customization commands to Java implementations and applies the customizations directly on the MHP client platform. The personalization components are used by different shop products, and so cross-cutting personalizations are applied to all shops. As the personalization component only exhibits the abstract shop product interface, consumers are not confronted with complex product and component interdependencies.
3. The personalization script is send to the server via the return channel.
4. The next server-side request, say, by a web browser, causes cached customized pages to be invalidated and recalculated with the new customization script. As the server-side Message Redirector understands a super-set of the client-side Message Redirector, it can map all provided commands to implementations for HTML content creation. Note that these implementation do not have to be DVB Java classes, but can be implemented in any programming language.
5. The same customization is visible on all supported platforms.

In this example we have seen that the Message Redirector's personalization commands abstract the different components and products. The commands only have to conform to the interfaces offered by the Message Redirector. Different products running on one platform (as for instance different shops) and also components for other platforms can be customized in an integrated way. Qualified programmers are only necessary to define the "personalization language," and implement it for each product and each platform.

4.2 Shop Branding by Content Editors

Different shops of the warehousing company should have an unique appearance, but should share the same information architecture. Content editors should be able to perform simple interactive behavior customizations such as shop branding across different platforms and channels. Here, we use XML based customization files for shop branding and simple domain-specific customizations on top of the indirection architecture. For each page there are one or more associated classes, called page templates [13]. A page template is a class defining the customization of a page with program behavior, but the class is largely constructed from an XML file stored in the content cache. The XML file contains the customizations. As program code is dynamically generated, we can specify behavioral extensions in the XML file. Each XML tags maps to an instruction in the Message Redirector. Only for new customization styles programming efforts are required: all other customizations can be done in the XML file. For extending the application with a new customization, we simply have to register a new command with the Messages Redirector.

In Figure 3 this architecture is visualized for one product that is able to display and handle different interactive warehousing pages for different platforms. Here, separated concerns for domain-specific customization, such as shop layout in the figure's example, are implemented independently of the warehousing product, but yet incorporated in the product-line architecture. Domain experts and content editors can simply edit the XML files, then new shop layout classes are generated, and the Message Redirector switches dynamically to these new implementations. The product's business logic is not affected by the changes at all. Similarly, other aspects such as format styles, channel specifics, etc. can also be implemented.

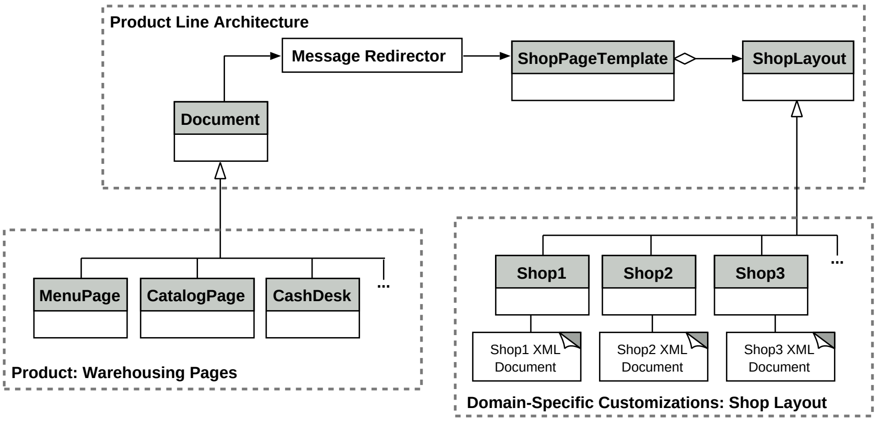

Figure 3. Customizing shop layout with XML files independently from the generic product implementation

5 Conclusion

In this paper we have discussed a set of open issues in designing variation points that we have identified in three larger product line architecture projects. An object-oriented indirection architecture was presented with the aim to reduce the dependency of concrete product implementations and domain-specific variability requirements. This way we can treat domain-specific customizations of products separately from the product-line implementation. We can choose the appropriate variability mechanism for each functionality and encapsulate variation points. As a consequence, we are able to cope with the identified open issues, such as traceability, avoiding implicit dependencies, and scattering variation points.

These issues are especially important for introducing domain-specific customizability, e.g. for domain experts. Customizations should be possible without an intimate knowledge of the whole product-line. The domain experts should only see the task-specific elements of the product and the interfaces of used components. The symbolic calls in the Message Redirector enable us to define a "little" customization language that hides issues that are not relevant for a task-specific view. The indirection architecture presented requires runtime variability resources. In many cases it is beneficial that we can bind every variation point at runtime, but there are also some problematic consequences, such as a negative influence on runtime efficiency and memory consumption. Moreover, for different stakeholders different techniques are appropriate to enable customizations. Therefore, we have to choose the right variability mechanism for each design situation.

6 Acknowledgements

The work described in this paper has partially be founded by the "Technological Perspectives of the Multimedia Home Platform (TPMHP)" EU project, an industry co-operation with the company BetaBusiness TV, and the BMBF, Project CAFÉ, "From Concept to Application in System Family Engineering"; Förderkennzeichen 01 IS 002 C; Eureka $\sum$! 2023 Programme, ITEA Projekt ip00004.

References

1. L. Bass, P. Clement, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, USA, 1998.
2. J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.
3. J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl. Variability issues in software product lines. In *Fourth International Workshop on Product Family Engineering (PFE-4)*, Bilbao, Spain, 2001.
4. G. Campbell. The role of object-oriented techniques in a product line approach. In *Proceedings of OOPSLA 98 Object Technology and Product Lines Workshop*, Vancouver, BC, Canada, 1998.
5. ETSI. MHP specification 1.0.1. ETSI standard TS101-812, October 2001.
6. European Software Institute. Engineering software architectures, processes and platforms for system-families. www.esi.es/esaps, 2001.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
8. M. Goedicke, G. Neumann, and U. Zdun. Design and implementation constructs for the development of flexible, component-oriented software architectures. In *Proceedings of 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00)*, Erfurt, Germany, Oct 2000.
9. M. Goedicke, G. Neumann, and U. Zdun. Message redirector. In *Proceedings of EuroPlop 2001*, Irsee, Germany, July 2001.
10. M. Goedicke and U. Zdun. Piecemeal legacy migrating with an architectural pattern language: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(1):1–30, 2002.
11. K. Pohl, M. Brandenburg, and A. Gülich. Scenario-based change integration in product family development. In *2nd ICSE Workshop on Software Product Lines: Economics, Archtecture and Implications*, Toronto, Canada, May 2001.
12. O. Vogel. Service abstraction layer. In *Proceeding of EuroPlop 2001*, Irsee, Germany, July 2001.
13. U. Zdun. Dynamically generating web application fragments from page templates. In *Proceedings of Symposium of Applied Computing (SAC 2002)*, Madrid, Spain, March 2002.
14. U. Zdun. *Language Support for Dynamic and Evolving Software Architectures*. PhD thesis, University of Essen, Germany, January 2002.