Designing Runtime Variation Points in Product Line Architectures: Three Cases

Michael Goedicke[1], Carsten Köllmann[1], Uwe Zdun[2]

[1] Institute for Computer Science, University of Essen, Germany

{goedicke|koellmann}@cs.uni-essen.de

[2] Department of Information Systems, Vienna University of Economics, Austria

zdun@acm.org

Abstract. Software product lines provide a common architecture, reusable code, and other common assets for a set of related software products. Variation is a central requirement in this context, as the product line components have to be instantiated, composed, and configured in the context of the products. In many approaches either static composition techniques or dynamic composition techniques based on loose relationships, such as association, aggregation, or replacement of entities, are proposed to design the variation points. If the domain of the product requires runtime variation, however, these approaches do not provide any central management facility for the runtime variation points. As a solution to this problem, we propose a pattern language that provides a domain-specific variation language and runtime variation point management facilities as part of the product line. We present three case studies from the areas of interactive digital television and document archiving in which we have applied this pattern language.

1 Introduction

Software product lines or system families provide a structure for a set of products of a specific organization in form of a so-called product line architecture. Such a product line architecture contains a set of generic components in addition to other common assets. The individual architecture of a particular product is derived from the product line architecture. Each particular product uses the set of generic components and introduces product-specific code as

well. Generic components provide well-defined interfaces. In the product-specific code the generic components of the product line architecture are configured to build the particular 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 between the various product variants. A particular product is primarily built from a common set of assets [2]. Often these common assets are designed as (black-box) components. In software product line approaches, a 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. At design time the variability mechanisms and architectural styles are decided. Once a design and implementation is based on a certain variability mechanism, it is often in traditional approaches quite hard to exchange them with other mechanisms. Also there is no central facility for managing these variation points at runtime, as they are usually based on ordinary (e.g. object-oriented) relationships.

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 [6]. The approach pays special attention to the traceability between architectural decisions and functional/non-functional requirements. However, many domains impose requirements for domain-specific customizations that can hardly be implemented with variation points bound before runtime. In such cases, without a central runtime variation management concept that can be derived from the product line architecture and that goes beyond (object-oriented) relationships, traceability and similar features can hardly include the traceability links that are established at runtime. In this paper, we will concentrate on techniques that can be applied to introduce domain-specific ad hoc customizability and treat these runtime variation points as first-class entities.

We have studied these issues theoretically and practically in two larger industry projects from which we will present three case studies in this paper:

- TPMHP is an EU project that focuses on a generic product line architecture for development of digital television applications on top of the MHP (Multimedia Home Platform) [9]. We present two cases in this context:

- An MHP application has to determine the hardware configuration of the client terminal (for instance: a digital set-top box) at runtime onto which it is broadcasted and executed in turn. The application has to adapt and configure itself according to these parameters. Then it loads the appropriate MHP product line components, as well as its application domain components, from the broadcast channel.
- In the context of a retail chain a number of different interactive television (e-commerce) shops are offered. These e-commerce applications share a common architecture and common components, and contain additional special components for a particular shop. In this e-commerce area, usually additional channels to interactive television have to be supported at the server side as well. For instance, web shops and integration with mobile phones are also supported. The second case study investigates how to integrate content for all these platforms and dynamically add orthogonal aspects, such as channel specifics, content formats, or content styles.

- We conducted a reengineering project for a large-scale document archive system [14]. In this paper, we present a (runtime) configuration and customization framework with the goal of product derivation, deployment, and installation at the customer as a third case study.

In these projects, late-bound flexibility is not only a useful feature, but also a requirement for using a product line approach at all. These domains 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 rapid customizability is required. Sometimes variations are dependent on the *runtime context*, and such product lines require runtime variability. *24x7 server applications*, such as custom web servers and application servers, have to be integrated. These usually cannot simply be stopped for deploying or configuring components. Thus in such domains a dynamic component exchange mechanism is required. The same problem arises in the context of *broadcast* (MHP-)clients, because the server has no control of the client at all.

Our general approach is to provide a "little" domain-specific language and a central architecture for managing the runtime variation points in this language. This architecture is provided in the generic product line architecture implementation and can be reused for all products derived from it.

At first, we outline some open issues in the design of runtime variation points. Next we discuss a pattern language for designing a runtime variation point management language within a given object-oriented software architecture. Note that these patterns can be used in any object-oriented software framework, not only in product lines. Then we present three cases from the projects mentioned above to illustrate the use of a domain-specific runtime variation point management language in practical examples. Finally, we discuss related work and conclude with a summary of our main findings.

2 Open Issues in Designing Runtime Variation Points

Variation points are implemented using a variability mechanism. Typical traditional examples for variability mechanisms are associating objects in DECORATOR style [12], delegating to a STRATEGY [12], using inheritance for specialization, exchanging a runtime entity such as an object, parameterization, inlining, or 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, and some at runtime. In general, the later we bind a variation point, the more flexible the solution is. However, with runtime variability we have to deal with certain drawbacks as well, for instance, degraded performance, increased memory consumption, and higher runtime complexity.

In this paper, we consider the question how to manage variation points efficiently and in a structured way that have to be bound at runtime. Obviously, only a few of the techniques, discussed above, provide a solution here. As these techniques primarily rely on associating different objects, they are mainly implemented by hand and thus there is no central management of variation points at runtime.

Also there is not first-class representation [3] of variation points. Thus the recurring use of variation points has to be built with certain syntactic conventions and is scattered across the code. The constructs used to implement an architectural artifact cannot easily be reused. As a consequence the traceability of variation points is limited: without a first-class representation in the program code or design, a variation point is not recognizable ad hoc as an entity. Therefore, complex variation points are hard to locate for a client of the component.

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 parts. As a consequence these “scattered” variation points heavily reduce understandability of the design and code.

Variation points that are hard-coded into the code cannot easily be reused at runtime. Once such a variation point is configured, it is not easy to pass its configuration to other applications.

Dependencies between variation points 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.

What is needed, is a way to manage the runtime variation points as first-class entities in the product line architecture. Otherwise it may be hard to use, extend, and customize products that require rapid changeability. A general, generic architecture for managing runtime variation points, resolving all these issues mentioned above, can be hard to build. However, in the product line context, we can benefit from the domain knowledge. Thus our solution is to provide a domain-specific runtime variation point management solution for the product line. Of course, such techniques can also be used in other contexts, such as object-oriented frameworks or component frameworks, for instance.

3 A Pattern Language for Building a Domain-specific Runtime Variation Point Management Language

In this section, we present a pattern language for building, manipulating, and managing domain-specific runtime variation points. We use this pattern language in the case studies, presented in the following sections, as a conceptual foundation.

A pattern is a proven solution to a problem in a context, resolving a set of forces. Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution [1]. A pattern language is a collection of patterns that solve the prevalent problems in a particular domain and context, and, as a language of patterns, it especially focuses on the pattern relationships in this domain and context. As an element of a language, a pattern is an instruction, which can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant [1].

Our pattern language uses some patterns that have been documented in the literature before, either as "isolated" patterns or in other pattern languages. In this pattern language we use these patterns specifically in the context of building, manipulating, and managing domain-specific runtime variation points. Thus we describe the patterns here in this particular context. The main contribution of this section is to describe these patterns in the context of variation point management and their integration is this context.

Figure 1 shows an overview of our pattern language. In particular it consists of the following patterns:

- A COMMAND [12] encapsulates an invocation to an object and provides a generic, abstract invocation interface. In the pattern language, COMMANDS are used as the basic mechanism to implement variation points.
- A COMMAND PROCESSOR [5] allows for processing COMMANDS via an API-based interface and can provide add-ons to processing, such as a COMMAND history.
- A COMMAND LANGUAGE provides a symbolic (usually string-based) language that is mapped to COMMANDS. It can be seen as a sophisticated COMMAND PROCESSOR that embeds a COMMAND INTERPRETER.
- An INTERPRETER [12] defines a representation for a grammar along with an interpretation mechanism to interpret sentences in the language. In the pattern language, we use a COMMAND INTERPRETER that uses the symbolic instruction of the COMMAND LANGUAGE and interprets them as COMMANDS.
- A MESSAGE REDIRECTOR [13] provides a generic invocation mechanism that allows one to integrate an (existing) object system in a COMMAND LANGUAGE.
- A MESSAGE INTERCEPTOR provides an adaptation mechanism that can intercept message invocations and decorate or modify them. A MESSAGE REDIRECTOR can trigger the interceptors. Various kinds of MESSAGE INTERCEPTORS have been documented as patterns (see [14], [25]).
- A SERVICE ABSTRACTION LAYER [26] provides service abstractions as variations orthogonal to the COMMAND LANGUAGE. It can be used, for example, when clients operating in varying contexts should use COMMAND LANGUAGE elements.

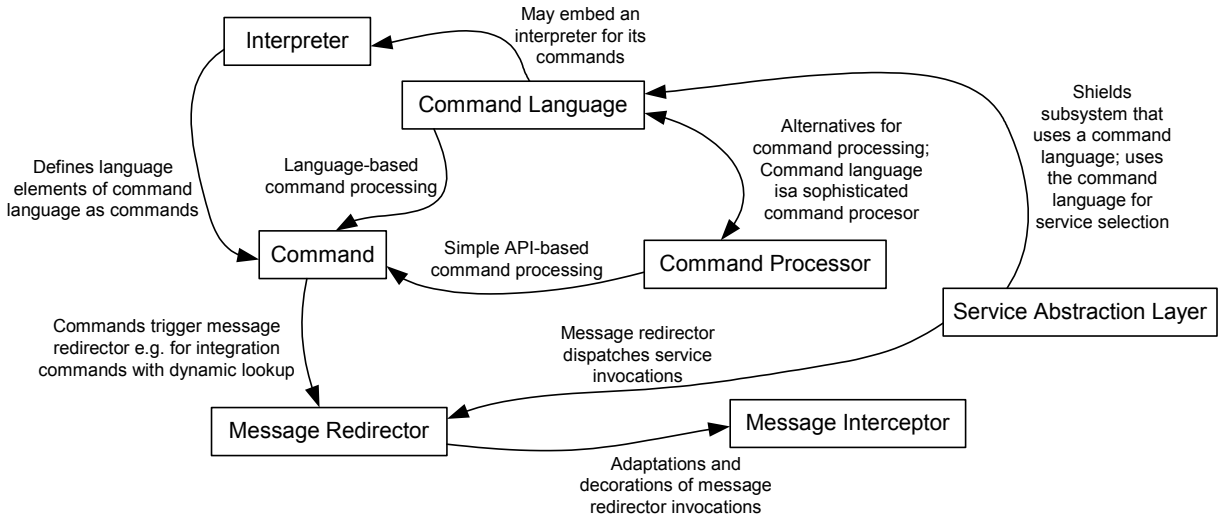

Figure 1. Pattern language overview

3.1 Command

Context. A runtime variation needs to be implemented.

Problem. A runtime variation point has to be dynamic. In an object-oriented system, runtime variation can be expressed by message invocations. But ordinary invocations are embedded in or scattered across the program text. How to provide dynamic invocations that can be managed centrally?

Solution. Encapsulate an invocation to an object in another object, called a COMMAND [12]. Provide an abstract COMMAND class that offers an operation to execute the encapsulated invocation. Special COMMAND classes provide the different COMMAND variations. A variation point associates with the abstract COMMAND class, and clients invoke an "`execute`" operation. At runtime the executing COMMAND can be exchanged; this way runtime variation is provided.

Note that COMMANDS often require parameters to fulfill sensible actions. These are typically given as an argument list of variable length, such as a string array. The COMMAND implementation is responsible for converting the parameters to the correct types. Thus not each COMMAND can be exchanged with any other COMMAND, but they have to be compatible in their parameters. If a COMMAND is called with the wrong parameters, a runtime error has to be raised.

Discussion. The COMMAND pattern is a simple and easy-to-build form of a pre-defined runtime variation mechanism. Particular COMMANDS can possibly be offered by a product line architecture. All classes that are derived from the abstract COMMAND class can be used in variation points; thus a management of the variants is possible. Note that there is no variation management at runtime and no management of the variation points (i.e. invocations of the COMMANDS), when only using COMMANDS. The invocations are still scattered across the code. The COMMAND classes can be seen as a very primitive language for runtime variations, with only an execution model for single COMMANDS. COMMANDS are used in the other patterns, presented below, to encapsulate single instructions.

3.2 Command Processor

Context. COMMANDS are used for runtime variation and management of variants.

Problem. A COMMAND only provides an execution model for itself, but does not allow for more complex relationships between COMMANDS, such as an order of execution or nested execution. These relationships have to be hard-coded in the variation points.

Solution. Let the client of a variation point associate with a COMMAND PROCESSOR [5]. It has the primary responsibility to simply invoke its COMMANDS. The COMMAND PROCESSOR provides a simple form of runtime management of variations: it simply aggregates all accessible COMMANDS. It also maintains a stack of COMMANDS to be able to access the COMMAND history.

Discussion. A COMMAND PROCESSORS does not raise the expressiveness of the "language" offered by the COMMANDS. But it adds some management functionality; especially it maintains a list of all COMMAND objects at runtime. Also the stack of COMMANDS can be used to support slightly more complex variations, such as undo, redo, or macro recording functionalities. However, still there is no variation management at runtime and no management of the variation points supported.

3.3 Command Language

Context. COMMANDS are used for runtime variation and management of variants.

Problem. The COMMANDS provide a very simple kind of a language, but it is cumbersome and hard to read this language, as every instruction requires an API call to the COMMAND objects. For instance in the following example, a condition is constructed, a COMMAND is configured with the condition and an action implemented in another COMMAND, and then the COMMAND is executed:

```
Condition c = constructCondition(“x > 1”);
ifCommand.setCondition(c);
ifCommand.setThenAction(doSomethingCommand);
ifCommand.execute();
```

In addition to the code complexity of this example, there is another central problem: as these instructions are hard-coded in the program code, in a compiled programming language such as Java or C++, it is not possible to exchange the whole COMMAND execution code presented above; only variations by exchanging the referenced COMMAND objects can be performed.

Solution. Provide a (little) COMMAND LANGUAGE and a representation for its grammar. Also provide an INTERPRETER [12] for this language. It uses the language representation to interpret sentences in the language. For each language element, provide a class that performs the interpretation. The interpretation class invokes the respective COMMAND, and the COMMAND provides the implementation for the language element. Then the following script, for example, can be used instead of the cumbersome command instructions above:

```
if {x > 1} {doSomething}
```

Here, a COMMAND “`if`” is used that accepts a condition and a body as string arguments. The condition is evaluated on the fly. Only if the condition is true the body script is evaluated dynamically. In it, another COMMAND `doSomething` is invoked.

Such a script can be handed over to the INTERPRETER as a string; thus it is possible to dynamically evaluate it. It can also be changed and constructed at runtime.

Discussion. A COMMAND LANGUAGE combines COMMANDS with an interpretation mechanism for a (little) language, evaluated by an embedded INTERPRETER. The (little) language scripts can be maintained independently of the program code containing the variation point; thus variation points can be managed separately. The scripts can also be changed and constructed at runtime allowing for dynamic variant point management at runtime. As each script has to be processed by the COMMAND LANGUAGE INTERPRETER, a (limited) runtime control of the variation points and variations is possible.

3.4 Interpreter

Context. A COMMAND LANGUAGE is used.

Problem. For implementing a COMMAND LANGUAGE we need some mechanism for mapping a language syntax and grammar to the respective COMMANDS. Even though a COMMAND LANGUAGE can potentially have a very simple syntax and grammar, there are some additional language constructs required, such as grouping of instructions in blocks and substitutions. These additional language constructs have to be executed before COMMAND execution can be performed.

Solution. Provide an INTERPRETER [12] for the COMMAND LANGUAGE. For instance, the INTERPRETER can define a class per grammar rule of the language. The INTERPRETER parses one instruction after another, according to the line end rules and the grouping (in instruction blocks) rules of the language. Then substitutions are performed. Note that there may be COMMAND substitutions. That is, COMMANDS that nest in an other COMMAND. Substitutions have to be interpreted recursively. After all substitutions are performed, the instruction is interpreted. Interpretation in a COMMAND LANGUAGE means to lookup the responsible COMMAND in the command table and invoke it with the given arguments. Consider the following example, in which the square brackets ("`[…]`") are used for COMMAND substitution:

```
set value [doSomething]
doSomethingElse
```

Here, two instructions are interpreted one after another with the COMMANDS `set` and `doSomethingElse`. But before we can interpret the `set` COMMAND, we have to evaluate `doSomething`. The result of this evaluation is given as a second argument to the `set` COMMAND. In this example two interpretation rule classes of the INTERPRETER are used: `EvalCommand` and `EvalBracketSubstitution`.

Discussion. An INTERPRETER requires a parser for the language. For COMMAND LANGUAGES this parser can be pretty simple, for instance, simply searching for line ends, groupings, and substitutions. Often, when implementing a COMMAND LANGUAGE, an existing interpreter (for example of a scripting language) can be reused with only slight modifications. Thus, writing a full-fledged INTERPRETER and parser is not always necessary. In an INTERPRETER implementation nesting structures in the language are typically modeled as COMPOSITES [12].

3.5 Message Redirector

Context. A COMMAND LANGUAGE with an INTERPRETER is used for interpreting a (little) variation point management language.

Problem. COMMANDS are a good means for implementing variations in a limited or well-defined scope. But consider writing a (little) language that should be able to access all classes in a product line, instantiate them, and invoke methods of the objects. Using COMMANDS as an invocation mechanism here is tedious, as one COMMAND per method of a class has to be written. Another problem of COMMANDS in this context is that – despite that they all perform the same task of invoking encapsulated code – they do not provide a central control mechanism for the variation points.

Solution. Build a MESSAGE REDIRECTOR [13] as a central invocation mechanism. It is called with the symbolic invocations that can be extracted by the INTERPRETER. The message itself is given as an argument to the MESSAGE REDIRECTOR'S dispatch operation. The MESSAGE REDIRECTOR maps the symbolic calls to actual implementation objects and methods. These can for instance be found via reflection or be registered by the application. After this mapping is performed, the redirector invokes the message implementation and returns the result.

Discussion. A MESSAGE REDIRECTOR is a generic invocation mechanism that allows one to integrate an (existing) object system in a COMMAND LANGUAGE. In contrast, COMMANDS are specific invocation mechanisms. COMMANDS are more suitable when the INTERPRETER should be extended with new language elements. For automated wrapping

and generating new variation points at runtime, MESSAGE REDIRECTOR is the better solution. This is because – in contrast to COMMANDS – it can easily wrap a complex subsystem, provide adaptation mechanisms, such as MESSAGE INTERCEPTORS, and control the variation points under its control.

In the context of a COMMAND LANGUAGE, MESSAGE REDIRECTORS are usually triggered by COMMANDS of the language. That is, the elements of the language are built as COMMANDS. Those COMMANDS that represent variation points and/or should access classes of the product line are bound to the MESSAGE REDIRECTOR. These MESSAGE REDIRECTOR COMMANDS handle the class lookup, for instance via reflection, and method invocation. For each object or class that represents a variation point, a new COMMAND, bound to the MESSAGE REDIRECTOR, is created.

3.6 Message Interceptor

Context. A MESSAGE REDIRECTOR is used.

Problem. A MESSAGE REDIRECTOR controls the message flow to all variant points, relevant for a specific problem. To implement rapid variations it is necessary in many situations to support tracing, modifying, or adapting of message invocations. With COMMANDS and COMMAND bindings in a MESSAGE REDIRECTOR as the only variation mechanisms, these tasks have to be hard-coded into the MESSAGE REDIRECTOR or all affected COMMANDS. This solution does not allow for dynamically composing message traces, modifications, or adaptations at run-time. Also the message traces, modifications, or adaptations cannot be reused or refined. How to express message traces, modifications, or adaptations as first-class entities dynamically?

Solution. Build a callback mechanism into the MESSAGE REDIRECTOR working for all messages that have to pass it. These callbacks invoke MESSAGE INTERCEPTORS for standardized events observable by the MESSAGE REDIRECTOR, such as “before” a method invocation, “after” a method invocation, or “instead-of” invoking a method. MESSAGE INTERCEPTORS can be dynamically composed with the invoked objects and operations.

Discussion. The MESSAGE INTERCEPTORS are usually defined in the COMMAND LANGUAGE as a special kind of COMMAND, so that COMMAND LANGUAGE scripts can access them. The MESSAGE INTERCEPTOR usually requires some way to obtain the invocation context, such as the calling object, method, class, and current object ID, from the MESSAGE REDIRECTOR.

3.7 Service Abstraction Layer

Context. Basic variations of the "services" offered by the software product line and its products are handled via COMMAND LANGUAGES and MESSAGE REDIRECTORS.

Problem. The typical variation points when building the services of a product line can typically be well managed with COMMAND LANGUAGES and MESSAGE REDIRECTORS. Usually, in the context of a product line we concentrate on the inter-product variation, when designing variation points. Many product lines have to handle some orthogonal variations as well, especially when they are to be presented in the context of "new media" platforms. That is, the same services have to be present to more than one channel, in more than one format, and/or with more than one presentation style. These are just examples; there are many other possible client-specific or request-context-dependent variations. These issues should not interfere with the implementation of the services of the products or the product line.

Solution. A SERVICE ABSTRACTION LAYER [26] is an extra layer to the business tier containing the logic to receive and delegate requests. The SERVICE ABSTRACTION LAYER abstracts over different service providers by implementing different channel adapters to support invocations using different protocols.

Discussion. The SERVICE ABSTRACTION LAYER provides additional dimensions or aspects of variation to the same services, such as channel, presentation style, and formats. Each of these individual aspects can contain variations that are usually expressed by a COMMAND LANGUAGE as well. For instance a presentation style script contains COMMANDS to format a given content in a specific presentation style.

A SERVICE ABSTRACTION LAYER may be implemented with a MESSAGE REDIRECTOR for indirecting symbolic calls to actual implementations of appropriate services, as well as channel, presentation style, and format implementations. The later aspects (channel, presentation style, and formats) are then typically handled by MESSAGE INTERCEPTORS that decorate and adapt the results of the service.

4 Case Study 1: Adapting to the MHP Hardware Environment

In this section we discuss the problem of domain-specific variation point handling in the context of adaptation to hardware environments for the Multimedia Home Platform (MHP). This case study was conducted within the EU

project “Technological Perspectives of the Multimedia Home Platform” (TPMHP), aiming at a product line architecture for the MHP.

The MHP specification [9] is a generic set of APIs for digital content broadcast applications, interaction via a return channel, and Internet access on an MHP terminal. Typical MHP terminals are for instance digital set-top boxes, integrated digital TV sets, and multimedia PCs. The MHP standard defines a client-side software layer 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. In this section, we focus on the basic variability concerns that MHP applications have to deal with on client side, resulting from the diversity of digital TV hardware environments.

4.1 Variety of MHP Set-Top Boxes, Supporting Systems, and the TV Environment

Although there are just a few MHP set-top boxes on the market right now, a great diversity of products can be foreseen in this field. In addition, there will be a great number of supporting systems, in particular input devices. In addition to these components the whole variety of different TV systems, as traditionally to be found on the TV market, has to be addressed, when building MHP applications. Let us consider this diversity in detail:

- *MHP set-top boxes:* Set-top boxes typically have more limited hardware resources than PC platforms, especially boxes at the entry level of the pricing range. In the field of television hardware there is typically also a market for more sophisticated boxes; that is, high-end MHP boxes with a PC-like hardware. These will support the broad range of MHP functionalities very early, whereas smaller boxes, such as cheap “zapping boxes” with low CPU speed, small graphical chips, little RAM memory, and no persistent storage, only implement (mandatory) parts of the specification. Since existing boxes in the TV market have to be supported for a long period of time, there has to be a broad variety of hardware platforms to be considered when writing interactive TV applications. The configuration of a box has to be taken into account as well, for instance availability of an optional flash

ROM or hard drives, or the particular type of modem. If a return channel can be accessed, we have to differentiate between line modems with varying bandwidth support, cable modems, and satellite modems.

- *Input devices:* The most important supporting device of a set-top box is the remote control. The minimum remote control of an MHP terminal includes a standard set of four color keys, the number pad, and four arrow keys with an OK button. Keyboards are typical extensional input devices, either external or included inside of the remote control. Other input devices might be PDAs or mobile phones using the infrared interface. Also voice control, mouse input, joysticks, and laser pointers are discussed as potential input devices. Alternatives in this context are program-based text inputs with the remote control, as supported by cheap boxes as well. Thus, a variety of different hardware and possibly also software components have to be taken into account as possible input devices.
- *TV environment:* The output device of an MHP application can be any Video and TV display. Different sizes, screen formats (4/3, 16/9), resolutions, and brightness degrees are just some parameters that have to be considered by an MHP application developer.

The variability parameters in the hardware environment, characterised above, can be divided into two groups:

- *Basic Hardware Parameters:* Some parameters have to be addressed by almost any MHP application, such as CPU speed and RAM size. Here, we propose the basic COMMAND pattern and exchange of the COMMAND sets of a COMMAND LANGUAGE as a solution.
- *Advanced Hardware Features:* Other issues can optionally be addressed to support more advanced hardware environments. Variation in parameters of the second group can be handled using small scripts that dynamically download and lookup the required hardware support classes. The scripts then access these Java classes with a MESSAGE REDIRECTOR. This has a slight performance impact, but allows for more flexibility and a higher complexity of parameterization. Many parameterizations can be expected to be recurring; thus these scripts are also implemented via the COMMAND LANGUAGE used for basic hardware parameters.

4.2 Availability of Environment Information

To determine a proper application configuration for the given hardware, as described above, it is necessary for the MHP application to obtain the runtime information about the environment and its parameters into which it is broadcasted. The MHP API only includes a few ways for retrieving environment information, and only if supported by the set-top box manufacturer. For example, the method `System.getManufacturer()` returns the hardware manufacturer of the set-top box and its version. The class `HrcCapabilities` describes basic remote control capabilities of the box, but does not check whether the actually used remote control supports them. Some more data might be obtained without user interference. For instance, we can check for the memory size using garbage collection or use runtime performance measurements. But many relevant parameters cannot be checked automatically, and thus these have to be specified by the user at least once. It depends on the individual hardware components of a set-top box how such information can be retrieved by an application:

- *Flash ROM:* If supported, it is possible to store some environment information in the flash ROM, even so the MHP specification intends to use this storage only for user-specific data, called "user preferences." Concerning the small size of typical flash ROMs it is not possible to store much information here.
- *Persistent storage:* Extended information can be stored on a hard drive. Saving standardized configuration files would enable different applications to get environment information without having to ask the user over and over again.
- *Background application:* A small application running in the background can be used to dynamically offer environment information. This is useful if none of the devices mentioned above is integrated in a box and/or the information offered by this application have to be centrally interpreted. The major problem of this solution is the permanent usage of memory and performance resources.
- *Return channel:* The information can be retrieved from a remote server using the return channel of a set-top box. As this is a rather slow way of access, it should only be used if the methods mentioned before cannot be used.

The techniques mentioned above can be used for storing information of any kind, such as technical parameters, user preferences, or customization information. In general, the user inputs of hardware information should be avoided, if

possible, and system resources should be saved. Note that how system information is accessed and stored by a broadcasted application needs runtime variation itself.

4.3 Runtime Variation Approaches for Hardware Selection in an MHP Application

In this section we describe the use of the patterns for runtime variation, discussed in Section 3, with regard to the problems of interactive TV hardware environments, as introduced in the previous section. Here, we face the following forces:

- The decision for the proper hardware configuration cannot be made before the application is broadcasted to the set-top box and started on the box. Thus runtime variability is required.
- Variation points that are handling issues like performance, storage size, or other basic parameters of a set-top box should be as small and performant as possible because MHP applications always have to use them, whatever the performance of the box is they are running on.
- Runtime flexibility, after the hardware configuration selection is made once, may be required, but this will occur rather seldom, for instance, if the used input devices changes.
- The hardware environment information is required for different products (i.e. MHP applications) derived from the MHP product line. Once collected, the hardware environment information should be reusable for all products downloaded onto a set-top box. Especially repetitious user inputs should be avoided.
- The solution should not require the aspect "hardware configuration" to be scattered across the code: for instance, conditionals, such as `if` statements for each selection in the code, are not a reusable and changeable solution and should thus be avoided. The hardware configuration should be cleanly encapsulated, for instance in a separate component of the product line.

4.3.1 Building a Command Language for Accessing the Basic Hardware Structures

COMMANDS are a good basic solution for accessing the basic hardware structures, providing fast access without the need of further performance-consuming interpretation or dispatch steps. The basic idea of our solution is to provide a small language based on COMMANDS for accessing the basic MHP components that are dependent on the MHP hard-

hardware environment. When the application is downloaded and started on the set-top box, it performs the hardware configuration selection, downloads the proper COMMAND set, and binds these commands to the application. We use the INTERPRETER and COMMAND LANGUAGE patterns to select a proper COMMAND set. Obviously, this decision cannot be made before runtime – that is, not before the broadcast. Thus runtime variability is required.

The basic MHP components that are dependent on the MHP hardware environment are:

- *Input device:* Different input devices offer various input events. How to react on these events depends on the hardware used. Thus generic `getInputDeviceEvent` and a `setInputDeviceEvent` COMMANDS are created with specializations for specific devices, such as `getKeyEvent`, `getDirectionEvent` (for example raised by joystick movement or a pressed arrow button), or a `getCharEvent` created by either a hardware keyboard or a virtual keyboard shown on the TV.
- *Graphical Objects:* Graphical elements in the MHP API include a few basic Java functions and elements of the HAVI standard [15]. HAVI (home audio video interoperability) is a standard specification for digital audio/video appliances, mostly providing lightweight GUI objects like buttons, lists, etc. Unfortunately, at the moment different set-top boxes interpret HAVI elements slightly different, so it depends on the type of set-top box if HAVI should be used or graphical elements have to be painted using the fundamental graphic functions of Java. Thus COMMANDS for each graphical object are needed: A `button` for example is a COMMAND. It has various (aggregated) sub-COMMANDS, e.g. for `paintButton`, `setBorder`, `setColor`, `setText`, `setImage` etc. The sub-COMMANDS can be invoked on every individual button COMMAND. Just replacing these COMMANDS at runtime can change also the style of an application. For instance, we can use simple graphical objects on small set-top boxes, and extended ones on medium-sized and high-end platforms.
- *Styles Design:* To configure a design fitting to the actually used TV screen and the capabilities of the set-top box, COMMANDS for combinations of size, outer appearance, and color of graphical elements are needed. Furthermore the style of texts, fonts, line breaks, and alignment have to be constructed on runtime.
- *Communication:* Different return channels can be identified at runtime and may in some situations be handled differently. Thus COMMANDS for `sendData` and `receiveData` are introduced. These provide, for instance, blocking communication on faster return channels and unblocking communication for slower ones. This way

users of fast channels can be sure that the communication has been successful before going on. Users of slow connections get the acknowledgement later but can go on using the application instantly.

Let us consider the classes for a few buttons as an example:

```
Class GraphicsButton {
  ...
  setVisible() {...}
  paint() {...} // paint button with Java graphics instructions
}
Class HAVIButtonCmd implements Command {
  HtextButton button;
  static final private String validOptions[] = {
    “setVisible”,
    ...
  }
  static final private int SET_VISIBLE = 0;
  ...
  public void execute(String[] arguments) {
    ...
    int subcmd = getIndex(arguments[1], validOptions);
    switch (subcmd) {
      ...
      case SET_VISIBLE: {
        button.setVisible(); //invoke the HAVI HTextButton
      }
    }
  ...
```

```
}
Class GraphicsButtonCmd implements Command {
  GraphicsButton button;
  ...
  execute() {
    // map graphics button operations to sub-commands in the same way
    // as for the HAVI button
  }
}
```

In this code we have introduced a new self-made button, as well as two button COMMANDS, both offering the same COMMAND interface. One COMMAND is using the HAVI button; one is using the self-made graphics button. To enable a developer to use these COMMANDS, they have to be instantiated. For the GUI COMMANDS we create a FACTORY [12] COMMAND that allows developers to create new COMMANDS of the respective GUI types with the FACTORY'S `create` sub-COMMAND:

```
Class HAVIGUIFactoryCmd implements Command {
  ...
  public void execute(String[] arguments) {
    ...
    int subcmd = getIndex(arguments[1], validOptions);
    switch (subcmd) {
      case CREATE: {
        ...
        int type = getIndex(arguments[2], validOptionsCreate);
        switch (type) {
          case BUTTON: {
            ...
```

```
            String cmdID = getArgument(arguments[3]);
            HAVIButtonCmd button = new HAVIButtonCmd(cmdID);
            ...
          }
        ...
      }
  ...
}
Class GraphicsGUIFactoryCmd {
  // create Graphics GUI elements in the same way as in the
  // HAVI GUI factory
  ...
}
```

Now we have different choices to select for the button type in an application. The most simple one is to use an `if` statement at every point where a GUI COMMAND is executed. Because of the danger of code scattering, this technique can only be used for a small number of variation points. The handling of fundamental environmental parameters should be done at a higher level. An additional advantage of a higher-level variation point handling and deployment is the lower amount of class files that have to be downloaded onto a set-top box, if dynamic class loading is implemented. Of course, we can still address COMMANDS using simple conditions. For instance, a complex COMMAND can be built using the GUI FACTORIES. Note again that this style of selection should only be used to build complex COMMANDS, for instance, to avoid performance bottlenecks. The typical selection of hardware functionalities should be performed with one of the approaches discussed in the next sections.

4.3.2 Integration of MHP-specific Commands in a Command Language

Using COMMANDS for encapsulating variation points solely, as in the previous section, still implies the problem that there is no central management of the runtime variation points. With an INTERPRETER that is configured dynamically, it is possible that different COMMANDS can be bound to language elements of a COMMAND LANGUAGE in a simple, yet

flexible way. The variation points, encapsulated in COMMANDS, can be centrally managed with the COMMAND table of the language. Dynamic lookup with a MESSAGE REDIRECTOR is not used for basic hardware selection to avoid its performance overhead. Thus all COMMANDS depending on basic hardware environment parameters are directly bound to language elements.

This approach offers a solution for a central reaction to varying environmental parameters of MHP boxes, directly after the application is started on the set-top box. Environmental changes at runtime, such as changes of input device or getting more performance by stopping a parallel application, can be handled this way as well, simply by providing a dynamic re-configuration of the COMMAND mapping. As the interpretation step is a simple mapping of string to COMMAND, this solution is quite efficient, and can even be used on smaller boxes. However, for performance bottlenecks, it is always possible to define COMMANDS that perform the selection directly using `if` statements, as mentioned above.

In the following example, COMMANDS are dynamically bound to an INTERPRETER of a COMMAND LANGUAGE. The selection of these COMMANDS is based on the detected selection conditions described before, as for instance performance, RAM size, TV size, box type, input device, kind of back channel, etc.:

```
if (tvsize() > 15 && preferences.getPerformance() < 80) {
  interpreter.createCommand(“GUIFactory”, GraphicsGUIFactoryCmd);
  ...
} else {
  interpreter.createCommand(“GUIFactory”, HAVIGUIFactoryCmd);
  ...
}
if (inputDeviceSupported(JOYSTICK)) {
  interpreter.createCommand(“getDirectionEvent”, JoystickEventHandlerCmd);
} else {
  interpreter.createCommand(“getDirectionEvent”, KeyboardEventHandlerCmd);
}
...
```

At first the INTERPRETER has to detect what kind of TV screen is used. This can be specified in a configuration file, for instance. By combining this result with performance parameters of the box a decision is made how the GUI widgets are presented, meaning which respective FACTORY COMMAND class is bound to the COMMAND LANGUAGE instruction `GUIFactory`. Then, as a second example, the direction input event of the user is bound to the detected input device. These COMMANDS implement the respective sub-COMMANDS and handle the argument parsing. For instance, a script for creating and showing two buttons, named "OK" and "CANCEL" may look as follows:

```
GUIFactory create Button OK
GUIFactory create Button CANCEL
CANCEL setVisible
OK setVisible
```

Note that we use a Tcl-style syntax in our COMMAND LANGUAGE, as its INTERPRETER reuses some classes of the TclJava [8] implementation for evaluation, substitution, blocking, and other basic INTERPRETER tasks. COMMAND LANGUAGE scripts, such as the example above, can be passed to the INTERPRETER of the COMMAND LANGUAGE. In this example, it invokes the previously bound FACTORY COMMAND two times. This creates two new COMMANDS both bound to the respective button COMMAND class. The method `setVisible` is then invoked for these two COMMANDS.

The code size of these scripts is much smaller than coding everything by hand. Thus we can provide little scripts through the broadcast channel or return channel. Thus, initially only the COMMANDS and COMMAND LANGUAGE classes are loaded. Thereafter environment parameters are detected and script(s) fitting to these are loaded and executed. This way it is possible to insert a lot of little scripts into the limited space of the broadcast channel (the so-called DSM-CC object carousel [9]) producing different kinds of application logic without the need of hard-coding these variations. This avoids the effect of overloading applications with scattered decision structures (see Figure 2).

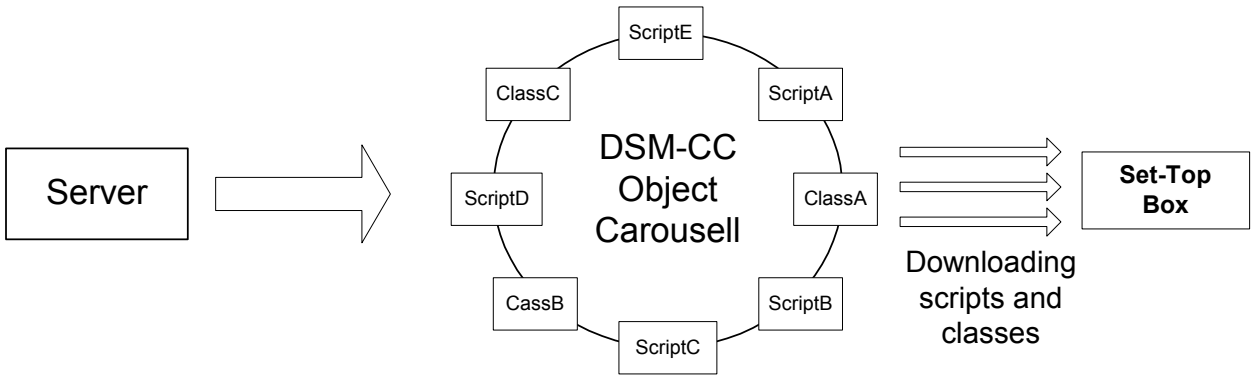

Figure 2. Download of scripts for runtime interpretation

4.3.3 Using a Message Redirector in the Broadcasting Environment

In the context of our pattern language, the MESSAGE REDIRECTOR pattern can be seen as an extension of the program designs described above, adding new or changing functionality of an application at runtime. It can be used to integrate scripts and existing Java classes. It does so without having us to write a new COMMAND wrapper class for each Java class that should be integrated. In the context of hardware selection, it is only desirable to write COMMAND sets for the most commonly used elements, as the ones in the examples above. For less commonly used elements of an MHP platform we need more flexibility, however, a slight performance impact is not as problematic, as less commonly used hardware elements can typically be found on more advanced set-top boxes.

Consider, we have dynamically loaded a Java class for keyboard support onto a set-top box and want to access it from a script. As part of our product line, we register a MESSAGE REDIRECTOR as a COMMAND of the INTERPRETER. It is able to access Java classes by reflection using a method `new`. As first argument this method accepts the name of a Java class. This Java class is looked up and, if found, registered in the MESSAGE REDIRECTOR. As a second argument `new` expects the name of a COMMAND which is dynamically created. This name is also used as an identifier for the newly created object, and it is stored in the MESSAGE REDIRECTOR'S object id/object mapping table. Thereafter, methods can be called for this COMMAND, which are redirected to the Java class by the MESSAGE REDIRECTOR. This design is depicted in Figure 3.

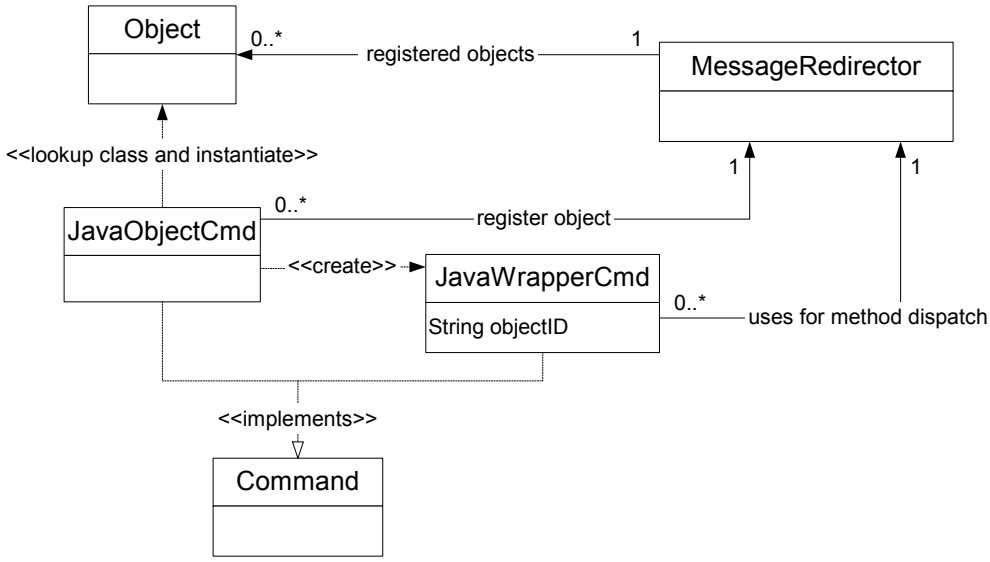

Figure 3. A message redirector for accessing Java objects via reflection

Consider the following simple example script:

```
JavaObject new KeyboardHandler keyboardHandler1
while {[keyboardHandler1 getNextEvent]} {
  ...
}
```

First the COMMAND `JavaObject` is called to create a new Java wrapper object. The method `new` looks up the class given as first argument (`KeyboardHandler`) and instantiates it. An instance of `JavaObject` is created with the object ID and COMMAND name given as second argument (`keyboardHandler1`). The mapping of object ID to the instantiated object is stored in the MESSAGE REDIRECTOR'S "registered objects" table. When the new COMMAND `keyboardHandler1` is invoked, it forwards the invocation to the MESSAGE REDIRECTOR. The MESSAGE REDIRECTOR looks up the method given as first argument for the respective Java class. If it is found, it is invoked on the wrapped Java object. This way we are able to flexibly add Java classes at runtime in MHP applications. Note that the applications need not to be specially prepared for this addition. In other words, we can deal with unexpected hardware devices simply by downloading the classes for device handler and the respective scripts.

5 Case Study 2: Branding and Customization for the MHP

In this section, we consider a second case study in the context of our MHP project, introduced already in the previous section. Within the project, we considered the situation of a large retail chain (the "company") with multiple different stores serving as a content provider for the MHP. The company wants to offer multiple slightly different shops realized on top of new media platforms, including the MHP. Each shop application can be seen as a product derived from a common "new media shop" product line. Apart from the variation in shops there are other, orthogonal variation requirements, as all shops should be presented as web shops, MHP-based interactive television shops, and possibly also as m-commerce shops, for instance, based on the Multimedia Messaging Service (MMS). Developing and maintaining a new interface for each of these portals of each shop is undesirable. Here different runtime variability requirements have to be handled:

- *Branding:* It is important that each individual shop has its own unique brand identity, including logos, layouts, banners, colors, etc. This uniqueness cannot simply be reached by providing always the same document layout for all shops. It is more about a complex combination of all components giving the customer an experience of "recognition." The styling mechanisms of the MHP product line should not be too obvious, giving each service the freedom to develop its own style to a certain degree. So branding requires a unique coding of layout, design, and even behavior of control elements and navigation means.
- *User Personalization:* Users should be able to personalize "their" application to a certain degree using simple interfaces. It should be possible to transfer the personalized behavior to all other services of a product derived from the MHP product line. In MHP user information can be sent from the client to a server using the return channel. This server could act like a central point where all customization information is stored. Another approach is to store personalization information in the persistent storage of the client, so that every application can use these parameters to configure itself at runtime.

Once in place, the customization solutions for branding and user personalization should be largely performed at a level of end-user customizability. Content editors should perform changes without Java programming, and users

should be able to customize their application from the user interface. Customizability requirements often impose ad hoc changeability as well; that is, a change, for instance performed by a content editor or user, should be directly applicable at runtime. In the web context, it is important that the server has not to be stopped for introducing simplistic changes: usually applications should run 24x7 hours a week. Different platforms, such as the web, mobile devices, digital TV sets, etc., may impose orthogonal stylistic customization requirements for the content. For instance, on mobile devices and on the television screen the information is usually presented in a much more condensed way than on the Internet. The user interaction schemes vary as well. That is, on the TV set the user interfaces should be simple, say, to be usable with a remote control. Moreover, specifics of the channel used to transport the content matter as well: the web usually presents a generated page within a few seconds, whereas providing information for TV broadcast channel takes a substantially longer period of time, as it requires building the DSM-CC object carousel and multiplexing on the broadcast channel.

The basic server architecture of our solution is a SERVICE ABSTRACTION LAYER, as shown in Figure 4. The architecture abstracts different channels including MHP terminals, mobile phones, and web browsers. Each of the customizable applications is realized by one or more services of this layer. The SERVICE ABSTRACTION LAYER is responsible for selecting which service has to be mapped to which channel (DSM-CC, HTTP, MMS) in which format (DVB-J classes and scripts, HTML, MMS) and with which shop layout (according to branding and personalization).

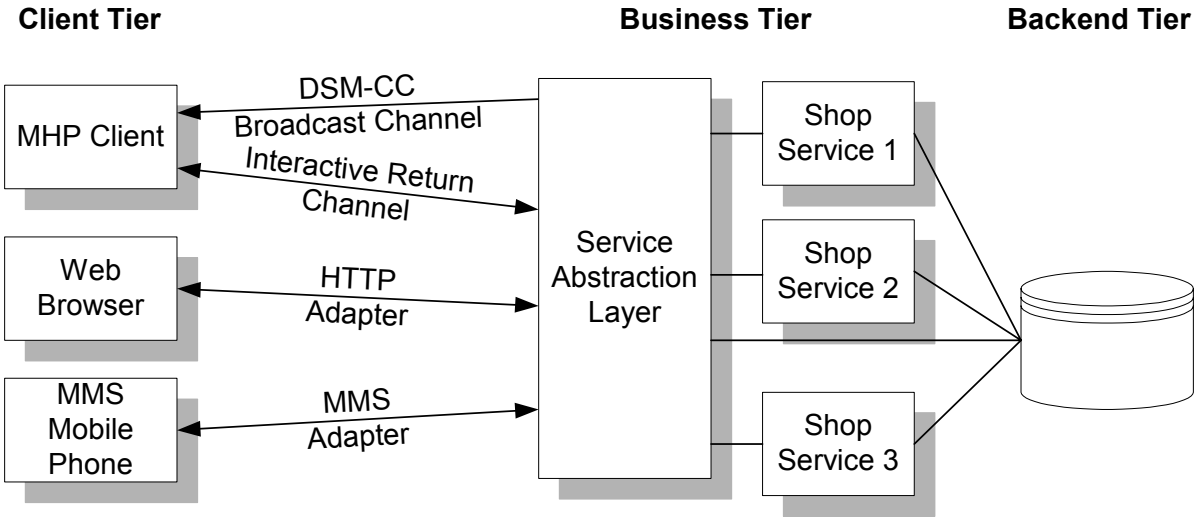

Figure 4. A service abstraction layer for the server side of the MHP product line

In this architecture, the recurring fragments of shop business logic services, as well as styles, formatting instructions, and channel specifics, should be cleanly separated as aspects and composable as required by the current request reaching the SERVICE ABSTRACTION LAYER.

Pages to be displayed on various devices are represented by classes that build up COMMAND scripts for these pages. In Figure 5 there are some `Document` class examples for shop interaction. Thus the product line provides a domain-specific COMMAND language for building the business logic of the shop services. Each of these `Document` classes provides Java methods and emits COMMAND scripts for the business logic of shop interaction services.

Format, channel, and layout aspects are separated from the `Document` classes. These aspects are handled in separate classes that implement a MESSAGE INTERCEPTOR interface.

When the COMMAND LANGUAGE'S INTERPRETER evaluates COMMAND scripts, a MESSAGE REDIRECTOR maps the COMMANDS to Java classes, in the same way as in Case Study 1. Thus the MESSAGE REDIRECTOR is used to compose the elements of the server side application.

The orthogonal aspects channel, layout, and format are introduced as MESSAGE INTERCEPTORS. These are registered with the MESSAGE REDIRECTOR. Before or after the invocation of a `Document` object, the MESSAGE REDIRECTOR invokes the MESSAGE INTERCEPTORS registered for that object. These adapt the computation performed for a COMMAND so that format, channel, and layout specifics are added dynamically.

On server side for web pages and MMS pages this adaptation is necessary, as the concrete combination of aspects and the page have to be dynamically build and composed depending on the actual request. On the MHP platform, the layout aspects have to be added and the page creation business logic rules should be the same. Thus the same business logic scripts and classes are delivered to the MHP client side, as used on server side. Note that these COMMAND LANGUAGE script and `Document` classes can stay the same, regardless whether it is executed to build some business logic on an MHP client or for a dynamically generated web page in a web server. By exchanging the binding of COMMANDS in the MESSAGE REDIRECTOR, as well as the MESSAGE INTERCEPTOR registration, on the MHP client a GUI representation of the business logic is created.

The MESSAGE INTERCEPTORS compose the aspects, cutting across the glue code of different components, to one computational entity, in particular: shop layout, content format, and channel-specifics (see Figure 5).

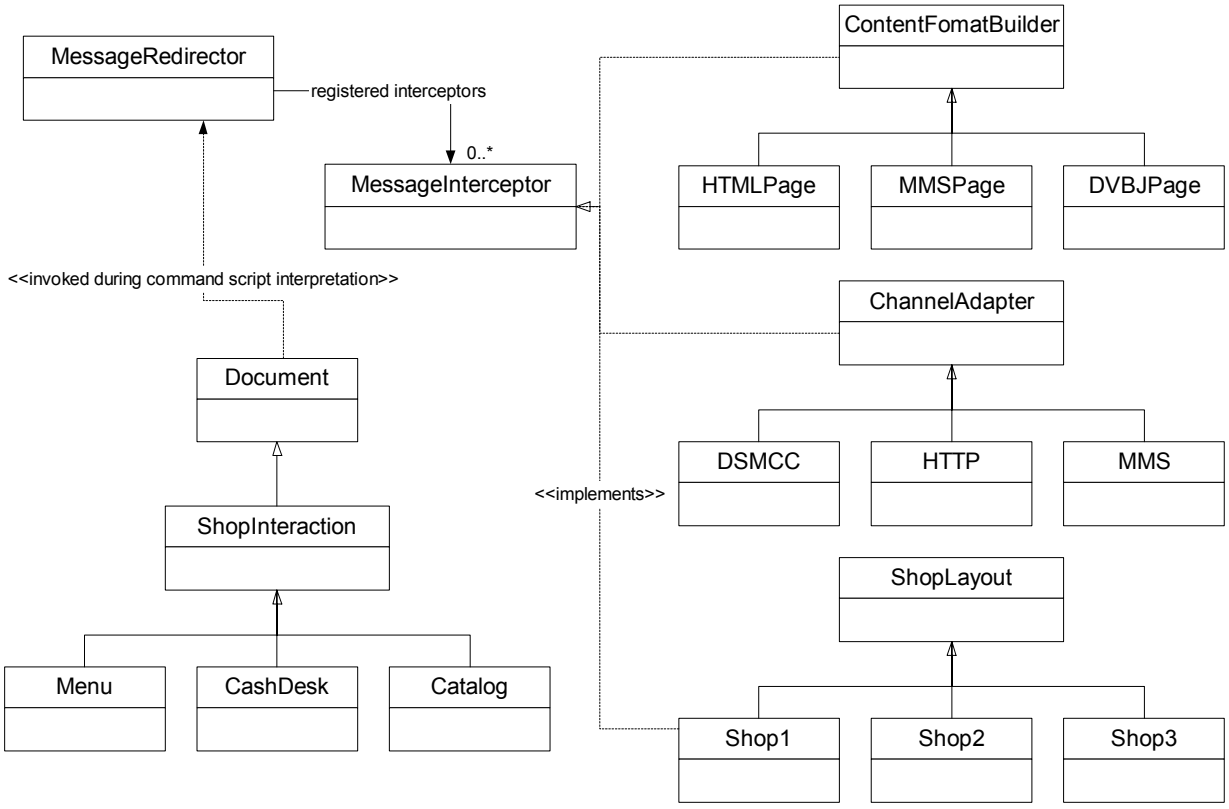

Figure 5. Document classes are composed by message interceptors with the orthogonal aspects format, channel, and layout

Note that very often such an architecture on server side also requires advanced fragments and caching mechanisms to support highly efficient dynamic page creation. Refer to [27] for a pattern language dealing with these issues.

Customizability is provided in the COMMANDS LANGUAGE. Thus content editors only have to learn and use the COMMANDS LANGUAGE primitives, instead of a full-fledged programming language. COMMAND LANGUAGE scripts

can be used on client side (e.g. an MHP terminal) and on the server side (e.g. as part of a web application server's mechanisms to build dynamic pages). Thus it is possible to use the same user personalization on client and server side. Moreover, COMMANDS LANGUAGE scripts also define a standard format how to deliver client side user personalization to the server via the return channel. On server side, the pages' application logic and these components are composed before a content page is generated. Thus a thin client suite is supported, which is important, as weaker set-top boxes should be supported and broadcast bandwidth is limited.

6 Case Study 3: Configuration and Customization of a Document Archiving System

In [14] we present a larger reengineering case study of a document archive system. A document archive system allows users to archive a large number of documents on several sorts of storage devices. The primary devices used in the application area are optical storage devices, but other media types are also used, such as hard drives or networked storage devices. Users can retrieve archived documents with different text-, web-, and GUI-based clients. At the time of the reengineering effort, the system was implemented in C. These clients need to be customizable for different customer requirements. The client through several search criteria delimits searches. Such information is stored in an associated database for each record archived by the system.

The document archive system has a generic product line architecture that can be directly instantiated. But in the area of document archiving it is usual to derive specialized products for the customer, what includes new developments as well as sophisticated configuration and customization tasks for the server side of the document archive system. For instance, often some recurring archive tasks of the customer should be automated, such as archiving some directories every 24 hours or providing a customized scanning routine for physical documents. The system also has to be adapted to specifics of the customer's infrastructure and the archived documents.

In the original system implementation, for such tasks, a small language extension of the korn shell was introduced. It was able to embed applications as plug-ins into the system. These shell scripts essentially have called a binary that executes in a forked process. To raise expressiveness of these plug-ins, a set of new key words was added to the korn shell together with a small parser/compiler. In particular, plug-in initialization, execution, and termination was supported by the following primitives:

- head - Comments, version labels, actions before plug-in execution.
- var - Global Variables.
- func - Callable Functions.
- begin - Plug-in initialization.
- loop - Looping body of the plug-in execution.
- end - Termination of the plug-in.

Here, a little language was implemented for plug-in execution. But there were some problems with this solution leading to the reengineering effort:

- *Language maintenance:* What seemed in first place a small effort of implementation and maintenance has evolved over the time into a complex language.
- *Language extension:* Since the underlying korn shell language is not designed for language extension, it was rather difficult to add quite simple additions, requiring writing new C and korn shell code.
- *Understanding:* There is no integration of the scripts and there is a rather obfuscated syntax, both limiting understandability.
- *Language integration:* The language was integrated with the current system release, but there was no concept to integrate with other systems or planned future system releases of the system (written in Java).
- *No runtime variation:* The solution was neither dynamic nor introspective at runtime. Thus it was hard to add ad hoc changes and "play" with the system, when deploying it to a new customer.

In summary, all these issues have made it hard to quickly derive a custom product and/or deploy and customize the system to a new customer.

Our proposed solution resides on the pattern language presented. Instead of using the korn shell, we use a COMMAND LANGUAGE implementing the same primitives, as discussed above, as COMMANDS. These COMMANDS are themselves implemented as part of the document archive system. They are aggregated by a customization and con-

figuration component, provided by the document archive system's generic architecture, which can be loaded into the script INTERPRETER of the COMMAND LANGUAGE as a standard language extension. The elements of the document archive system to be customized are also provided as COMMANDS. These are accessed with a MESSAGE REDIRECTOR that handles dynamic lookup of these elements.

When deriving an individual product from the generic architecture, the customization and configuration can be done in little scripts written in the COMMAND LANGUAGE. These scripts load the customization and configuration component and use its COMMANDS to customize and configure the elements of the document archive system at run-time.

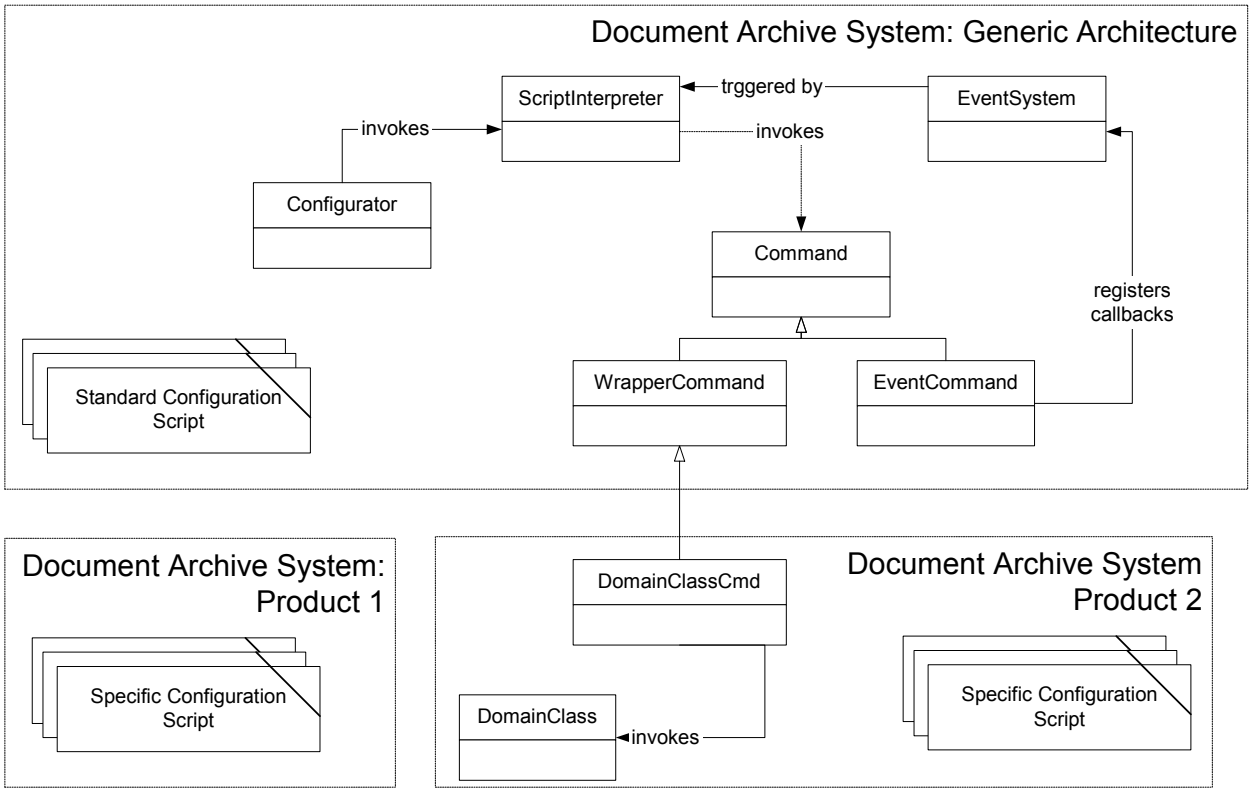

Figure 6. Configuration and customization in document archive products

Figure 6 shows that the document archive system's generic architecture defines the framework for configuration and customization. Especially, an INTERPRETER is defined with a COMMAND LANGUAGE. The COMMAND LANGUAGE consists of COMMANDS that wrap functions of the document archive system. Other COMMANDS register events with

the event system, such as timer events. There is a pre-defined set of standard configuration scripts. Concrete products can simply refine or replace these scripts. They can also define new user-specific code. If this code should be configured as well, a wrapper COMMAND for the domain class has to be implemented. The products then can be configured using instances of these domain classes.

In summary, the customization and configuration component is derived from the product line architecture and can easily be extended with new primitives using new COMMANDS and object-oriented abstractions, such as inheritance. Language maintenance of the COMMAND LANGUAGE is part-of the ordinary maintenance activities of the document archive system's implementation. Thus integration with it is not an issue anymore. Direct access to the elements of the document archive system by dynamic lookup and script composition at runtime allow for more rapidly deriving new products, instead of changing the C code for every installation. Only when new elements of the document archive system should be configured we have to add some code for building COMMAND wrappers for these elements. As COMMAND wrappers only have to convert string arguments and forward an invocation, they can be built quite quickly, what can even be automated using a wrapper generator.

7 Related Work

Different approaches tackle the design of software architectures in the context of product lines (see for instance [4], [7], [16], [2]) focusing on design and implementation. Variability as a basic issue in this context became more and more an apparent need, in parallel with an increasing interest for software product lines in general.

Variability concerns in software architectures can be seen from different points of view, resulting from different phases of the software development process. A taxonomy of variability realization techniques is given in [24], describing determining factors of what kind of variability should be introduced at which time and in which context, giving various case studies. Here configuration management and design patterns are identified as important techniques for realizing variability. Several patterns are introduced and compared like "runtime component specialization," showing how several implementations can use the same interface is described, or "condition on variable," deciding what variant to use depending on a value determined by a condition during runtime.

There are various approaches [22], [17], [10] describing how variability can be managed in product lines during all these lifecycle activities. The approaches are mainly introducing notation techniques based on features that define abstractions from the concrete requirements. In contrast, our approach focuses on variability mechanisms of systems that should be configured (either automatically or by user interaction) to be fitting to a given environment or supporting evolutionary tasks. In [11] the features, realized by Mozilla, and the underlying variation techniques are discussed as a case study, including the used runtime variation techniques. Mozilla uses some techniques that are similarly used in our approach, namely dynamic binding, scripting, and domain-specific languages. Our work proposes an integration of these and other runtime variation techniques by the patterns in our pattern language, and to instantiate this variation concept from the product line architecture.

The COMMANDS in our work were often used as wrappers for existing classes. Wrappers are mechanisms for introducing new behavior to be executed before, after, in, and/or around an existing method or component. Wrapping is especially used as a technique for encapsulating legacy components [23]. In the document archive system case study, our approach was primarily used as a high level wrapping technique to access the underlying system from the configuration scripts.

In [21] adaptive plug & play components are proposed. These define cross-cutting behavioral fragments for different components that can be parameterized. Although the technical realization is different to our approach, a similar goal of flexible composition of given components is pursued.

Domain-specific languages (DSL) are an approach to resolve the problems related to domain-specific aspects. They are small, usually declarative languages. In general they are particularly expressible in a certain problem domain. A typical example of a domain-specific language is MHEG-5 [20]. MHEG-5 is a language for declaring multimedia and hypermedia objects for building scenes within an application. Domain-specific languages are often created in the context of domain engineering projects focused on achieving reuse and reliability in particular problem domains. In the MHEG-5 example, the DSL is also used for the domain of broadcasting application content for set-top boxes. Domain-specific languages have been used in various domains, and aim at higher productivity, reliability, and flexibility. Although domain-specific languages are an attractive alternative, often mainstream languages are preferred due to the wealth of available libraries. Also some domain-specific languages have problems with portability and long-term support, since they tend to be research projects. Several domain-specific languages have problems

regarding integration with other domain-specific languages. In our work, we have also provided a kind of domain-specific language for a specific product line architecture, but by tight integration with the base language we have tried to avoid these typical problems of traditional domain-specific languages.

Aspect-oriented programming (AOP) [18] copes with tangled code fragments which logically belong to one single module (a "concern") but cannot be modularized because of limited composition mechanisms of the underlying programming language. Modularization of crosscutting concerns or tangled code is achieved in most aspect languages, such as AspectJ [19], by statically weaving aspects to a program. In the context of the second case study we have faced a similar concern with the aspects channel, layout, and format of a page. However, here we needed runtime composition of these aspects. We have provided MESSAGE INTERCEPTORS within a COMMAND LANGUAGE as an alternative solution to static AOP. This can be seen as a hand-crafted form of dynamic aspect composition.

To a certain extent, the use of COMMANDS and COMMAND LANGUAGES in our work can also be seen as an aspect-oriented technique, resembling so-called "introductions" in AspectJ. Introductions are a means to add structures, such as methods, to existing classes. COMMAND LANGUAGE script elements can be seen as a similar structural behavior extension of the classes they wrap. The realization technique, however, is completely different: AspectJ introductions are statically added by the aspect weaver (a kind of compiler), whereas COMMAND LANGUAGE scripts are dynamically composed and interpreted.

8 Conclusion

The contributions of this paper are a pattern language for building runtime variation points with COMMANDS and a COMMAND LANGUAGE, as well as three industrial case studies that we have conducted using the concepts of this pattern language.

Our goal was to show a runtime variation approach based on the combination of these pattern. Most of these patterns are explained in more detailed elsewhere, including [12] (COMMAND and INTERPRETER), [5] (COMMAND PROCESSOR), [13] (MESSAGE REDIRECTOR), and [26] (SERVICE ABSTRACTION LAYER). Our pattern language's contributes is a pattern description that focuses primarily on the context of managing variation points. As a pattern language, our description primarily highlights the pattern relationships in this particular context.

For applying the pattern language some central place is required, where the variation point handling architecture is provided. For this reason, the pattern language can well be applied in the product line context, where the common product line code servers as such as central place. Also for building a COMMAND LANGUAGE for variation point handling, it is important to operate in a limited domain, as it is hard to build a generic COMMAND LANGUAGE, working well for all possible variation requirements. This domain-focus is also provided by the product line approach. However, of course, the pattern language can also be used within an object-oriented framework or a component framework.

The three cases we have presented are operating in the product line context. Here, we have explored in how far the pattern language can be applied to some quite different contexts, where runtime variability is required. It turns out that the different patterns are well capable to resolve the forces in these three cases. We believe this is due to the use of patterns as a conceptual foundation. In particular, the presented patterns are successful solution in the runtime variability context. The inherent variability in the pattern format also contributes as a way to convey successful software solution to new contexts. Finally, the contextual integration of the patterns by the pattern language helps to conceptually integrate a variety of solutions in a context, despite all variations in the problem fields and application areas.

There are, however, also some potential drawbacks. Implementing a substantial part of the pattern language is a considerable work. In our work, we are reusing parts of existing Tcl-Java implementations [8] to minimize this implementation effort. If this is not possible for some reason, the approach might be too much an implementation effort for small projects. Also, the proposed techniques should carefully be used in cases where runtime variability is not required, as these techniques imply a performance overhead. In cases where runtime decisions are necessary, the techniques proposed can be implemented very efficiently with only a slight or no overhead compared to other dynamic solution, such as delegation. In cases where a completely static technique can be applied as well, usually dynamic techniques perform not as well, especially if they involve a dynamic lookup or reflection. In such cases, it has to be decided whether the additional flexibility provided by a dynamic solution is really needed. Note that our pattern language has a solution for problems that are implemented as dynamic variation points, but then turn out to be better handled statically: as we rely on a two-level language concept of base language and COMMAND LANGUAGE,

any variation point written in the COMMAND LANGUAGE can simply be moved into a base language implementation. This implementation is then bound to a COMMAND and thus it is still accessible from the COMMAND LANGUAGE.

9 Acknowledgement

The work described in this paper has partially be founded by the "Technological Perspectives of the Multimedia Home Platform (TPMHP)" EU project, an industry cooperation between University of Duisburg-Essen and Materna GmbH; IST project 2000-30114.

References

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jakobson, I. Fiksdahl-King, and S. Angel. A Pattern Language – Towns, Buildings, Construction. Oxford Univ. Press, 1977.

[2] L. Bass, P. Clement, and R. Kazman. Software Architecture in Practice. Addison-Wesley, Reading, USA, 1998.

[3] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl. Variability issues in software product lines. In Proceedings of Fourth International Workshop on Product Family Engineering (PFE-4), Lecture Notes in Computer Science, Springer Verlag, Berlin, 2001, 13-21

[4] J. Bosch. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, 2000.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-oriented Software Architecture. A System of Patterns. J. Wiley and Sons Ltd., 1996.

[6] 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.

[7] P. Clements and L. Northrop. Software Product Lines – Practices and Patterns. Addison-Wesley, 2002

[8] M. DeJong and S. Redman. Tcl Java Integration, http://www.tcl.tk/software/java/, 2003.

[9] European Telecommunications Standards Institute (ETSI). MHP specification 1.0.2. ETSI standard TS101-812, February 2002.

[10] J. van Gurp and J. Bosch, Managing Variability in Software Product Lines, Proceedings of 2nd Landelijk Archtectuur Congress, 2000.

[11] J. van Gurp, J. Bosch, and M. Svahnberg, On the Notion of Variability in Software Product Lines, Proceedings of 2nd Working IEEE/IFIP Conference on Software Architectures, 2001, 45-54.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[13] M. Goedicke, G. Neumann, and U. Zdun. Message redirector. In Proceedings of EuroPlop 2001, Irsee, Germany, July 2001, 317-330.

[14] 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.

[15] HAVI specification 1.1, May 2001.

[16] M. Jazayeri, A. Ran, F. Van Der Linden, and P. Van Der Linden. Software Architecture for Product Families: Principles and Practice. Addison-Wesley, 2000.

[17] B. Keepemce and M. Mannion, Using Patterns to Model Variability in Product Families, in IEEE Software, July/August 1999, 102-108.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In Proceedings of ECOOP 97, Finnland, June 1997. LCNS 1241, Springer-Verlag, 220-242.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. Communications of the ACM, October 2001, 59-65.

[20] MHEG Working Group. MHEG. http://www.mheg.org, 2000.

[21] M. Menzini and K. Lieberherr. Adaptive plug & play components for evolutionary software development. In Proceedings of Conference on Object-Oriented Programming Systems, Languages (OOPSLA), 1998, Vancover, Sigplan Notices, Vol. 33, No. 10, 97-116.

[22] T. Myllymäki. Variability management in software product-lines. Technical report 30, Institute of Software Systems, Tampere University of Technology, January 2002.

[23] H. M. Sneed. Encapsulation of legacy software: A technique for reusinglegacy software components. Annals of Software Engineering, 9, 2000, 293-313.

[24] M. Svahnberg, J. van Gurp, and J. Bosch, A Taxonomy of Variability Realization Techniques, Technical report, ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, February 2002.

[25] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.

[26] O. Vogel. Service abstraction layer. In Proceeding of EuroPlop 2001, Irsee, Germany, July 2001, 113-128.

[27] U. Zdun and O.Vogel: Content Conversion and Generation on the Web: A Pattern Language, In: Proceedings of EuroPloP 2002, Irsee, Germany, July, 2002, 195-232.