View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CWI's Institutio

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration

Robert Hirschfeld, Ralf Lämmel, Matthias Wagner

REPORT SEN-E0322 DECEMBER 23, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2003, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Design Patterns and Aspects --- Modular Designs with Seamless Run-Time Integration

ABSTRACT
Some solutions proposed in the original design pattern literature were shaped by techniques as well as language deficiencies from object-oriented software development. However, new modularity constructs, composition and transformation mechanisms offered by aspect-oriented programming address deficiencies of object-oriented modeling. This suggests classical design pattern solutions to be revisited. In our paper we point out that aspect-oriented programming not only allows for alternative representations of proposed solutions, but also for better solutions in the first place. We advocate a native aspect-oriented approach to design patterns that emphasizes on improving design pattern solutions both during development and at run-time. We use a simple yet effective method to analyze and describe different solutions on the basis of variation points, fixed parts, variable parts, and optional glue, employing dynamic run-time weaving.

1998 ACM Computing Classification System: D.3.3; D.2.2
Keywords and Phrases: Aspect-oriented programming; Design patterns; Run-time weaving

Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration

Robert Hirschfeld[*], Ralf Lämmel[#], Matthias Wagner[*]

[*]Future Networking Lab
DoCoMo Communications Laboratories Europe
Landsberger Strasse 312, D-80687 Munich, Germany

[#]Department of Software Engineering
CWI Centrum voor Wiskunde en Informatica
Kruislaan 413, NL-1098 SJ Amsterdam, NL

Abstract. Some solutions proposed in the original design pattern literature were shaped by techniques as well as language deficiencies from object-oriented software development. However, new modularity constructs, composition and transformation mechanisms offered by aspect-oriented programming address deficiencies of object-oriented modeling. This suggests classical design pattern solutions to be revisited. In our paper we point out that aspect-oriented programming not only allows for alternative representations of proposed solutions, but also for better solutions in the first place. We advocate a native aspect-oriented approach to design patterns that emphasizes on improving design pattern solutions both during development and at run-time. We use a simple yet effective method to analyze and describe different solutions based on variation points, fixed parts, variable parts, and optional glue, employing dynamic run-time weaving.

1 Introduction

Patterns and pattern languages are ways to express best practices and good designs. They capture experience in a way that makes it possible for others to reuse it [7]. Design patterns, as introduced in [5] mainly deal with static and dynamic relationships among objects found in object-oriented programming (OOP) and software development. According to [4], each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.

The solutions proposed by the original design pattern literature were shaped through techniques as well as language deficiencies from OOP. With the advent of aspect-oriented programming (AOP) [11] new techniques and mechanisms for software composition and adaptation have been presented that suggest these solutions to be revisited and reconsidered. First attempts to reshape design pattern solutions based on AOP have been initiated in [13] and [6]. In our paper, we carry on this challenge and clarify that AOP not only allows for alternative representations of existing solutions but also for better solutions of respective problems. In particular, we exemplify how to avoid indirection layer support and to prevent the loss of object identities. Pattern-based software design has already been studied in the field of dynamic object oriented programming languages and environments [14]. Our approach is centered on a simple analysis of fixed and variable parts of a system's design. Compared to previous work, we add dynamic run-time weaving to our set of development practices. This turns out to be crucial to meet adaptability requirements of certain design patterns. All examples described are implemented in Squeak [15] using AspectS [2][8] which is based on method-call interception [12].

The rest of this paper is organized as follows: In section 2 we emphasize the differentiation of AOP representations of design pattern solutions and native AOP solutions addressing the same problems. Section 3 prepares the solution space by explaining design problems in terms of variation points and their absence. Sections 4, 5, and 6 discuss different solutions for corresponding design problems. In section 4 we reiterate the realization of variation points via object-oriented techniques. In section 5 we reorganize these solutions in an aspect-oriented fashion if feasible, and in section 6 we show how native AOP solutions can improve object-oriented design practices. Finally, in section 7 we will summarize and conclude.

2 Common Problems – Different Solutions

Let us consider the opportunities of AOP with respect to design patterns. Design patterns offer solutions to problems encountered in object-oriented software development. Of course, for almost all problems there are several solutions. Some of them meet certain requirements better than others, depending on the context in which they are to be introduced. While AOP in general aims to address crosscutting and separation of concerns, particular implementations like AspectJ [7][10] and AspectS [2][8] extend object-oriented programming languages and environments in a way that new modularity constructs and code composition mechanisms are made available. These extensions enable system transformation and instrumentation carried out during development, at run-time, or somewhere in-

between. Given that, the representation of design pattern solutions or even the solutions as such need to be reconsidered.

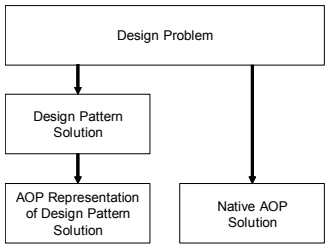

Figure 1: Different solutions through different approaches

Figure 1 illustrates how a common design problem can lead to different solutions. In the following we differentiate between the original solution proposed by a design pattern, its aspect-oriented representation, as well as a native AOP solution. We exemplify this differentiation for two design patterns: the Visitor pattern to discuss issues of extra levels of redirection, and the Decorator pattern to discuss problems of additional layers of behavior.

3 Missing Variation Points

Our focus is on design problems and solutions which are concerned with variation points. Variation points are design practices essential to systems that have to allow their parts to evolve at different pace and independently of each other. They let previously separated fixed and variable parts to be (re-) joined together to form the desired system [1]. In practice, system parts that are conceptually variable of nature are often entangled within the overall system design, and often are not separated explicitly from more stable parts through variation points. The lack of explicit variation points and of modular crosscutting compromises both system adaptability and evolution.

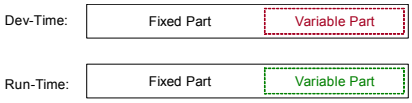

Figure 2: Missing variation points during both development- and run-time

Figure 2 sketches such lack of variation points, missing at both development- and run-time. In general, the lack of explicit variable parts already during development indicates a non-modular design. While the absence of variable parts and variation points might be beneficial at run-time with respect to system performance, it makes it hard for developers to reason about and to evolve such systems. It is desirable to allow for both comprehensibility form a developer's point of view as well as for run-time efficiency at the same instance. In the following we will point out two design problems for which we provide solutions in the subsequent sections.

3.1 Same Set of Objects – Multiple Responsibilities

Often, elements of an object structure have to perform several, but unrelated operations (tangling). Also, defining these operations on all of the relevant classes implies spread functionality over many locations (scattering). In addition, interfaces supported by these classes get broadened, and therefore changes to a particular operation – be it its modification, removal, or the addition of a new one – turn out to be non-localized, cumbersome, and error prone.

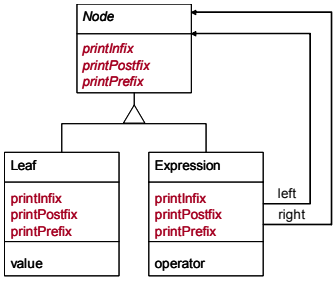

Figure 3: Different operations on expression trees

The class diagram in Figure 3 illustrates how different operations for generating a textual representation of an expression tree – namely a representation in infix-, postfix-, and prefix-notation –

are spread over several classes (here Node, Expression, and Leaf). Adding a fourth operation, for instance the calculation of the value represented by an expression tree, requires changing all three classes. In this example, the non-identified fixed part of the system is the Node class hierarchy. The non-identified variable parts are the operations to be performed on an expression tree.

3.2 Same Instances – Modified Behavior

Extending behavior of the base system by additional instance-specific responsibilities via static inheritance or feature-laden conditional code (usually located in base classes) often leads to inflexibility. Such inflexibility is observable by the fact that these additional responsibilities and the circumstance under which they apply have to be known upfront at development time (limited extensibility). Also, additional instance-specific responsibilities cannot be added or removed that easily, if at all, at run-time (limited adaptability).

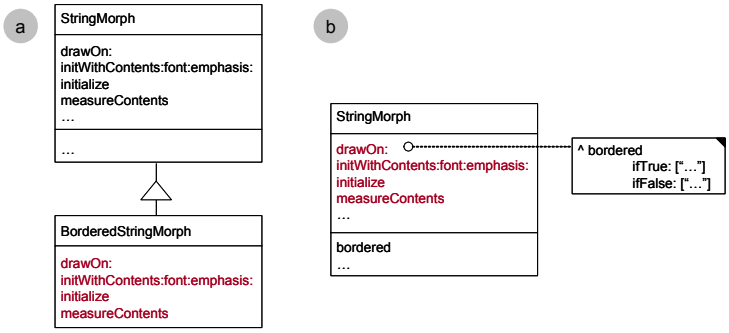

Figure 4: Two ways to extend StringMorph for bordered strings

Figure 4 shows two ways to extend StringMorph from Squeak's Morphic system to allow for bordered strings rendering. In Figure 4 a, StringMorph's functionality was extended by its subclass BorderedStringMorph (the actual approach taken in Squeak's Morphic system). Here, changing instance specific behavior would require instance migration from one class to another, probably involving expensive low-level operations (like become: in Squeak). In Figure 4 b, the extension was done via conditional code. The flag bordered (a Boolean value) will be consulted by all affected operations, and depending on its value the rendering of the string happens to be bordered or regular. Here, conditional code has to be present, and has to be inserted upfront during development. The non-identified fixed part of the system is resembled by StringMorph, the non-identified variable part by the additional behavior allowing for bordered strings.

4 Design Pattern Solutions

Design patterns can help to identify variable parts of a system design. They allow to factor out these variable parts from the base system, and to introduce glue to join fixed and variable parts at designated variation points. Such explicit separation of concerns makes it easier to reason about the system, and to evolve these concerns, independently of each other. In addition, some design patterns specifically aim at run-time behavior adaptation. Separation of variable and fixed parts and their composition is often realized via indirection layers that support variation points. However, besides increasing flexibility, indirection layers require additional computation to coordinate message exchange between the parts decoupled by them. Unfortunately, without special runtime analysis and optimization, this normally implies a decrease in system performance, and often a decrease of comprehensibility, too.

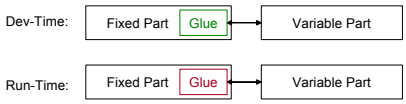

Figure 5: Variation points at both development and run-time

Figure 5 sketches the situation where variation points, and with them the separation of fixed and variable parts, are present at both development- and run-time. However, note that the fixed part is not completely independent of the variable part since glue code needs to be made available in the fixed part supporting integration and interaction. In fact, standard solutions offered by design patterns suggest this glue code to be supplied by the developer explicitly. This requires the developer of the fixed part of a design to anticipate and prepare for variations to be expected.

In more complex designs, a portion of system behavior may be distributed along multiple variation points. This requires developers to coordinate changes and adjustments at several system locations at the same time. Such crosscutting of collective behavior is not localized within one module but divided amongst several. In the following we will revisit the solutions of the Visitor and the Decorator patterns as shaped by the basic object-oriented programming model. We will point out the deficiencies related to the use of indirection layers and object composition to realize variation points.

4.1 Visitor

The intent of the Visitor design pattern is to localize the definition of an operation to be performed on the elements of an object structure, to allow the definition of new operations without changing the classes of the elements on which to operate [5].

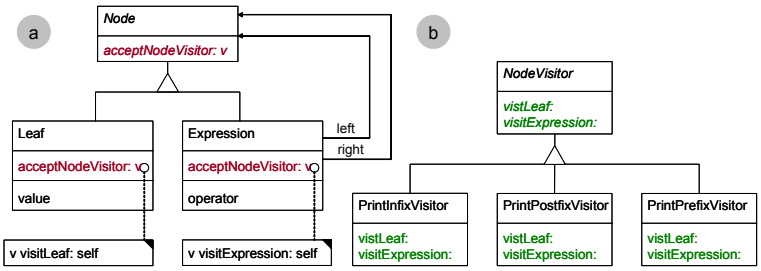

Figure 6: Visitor design pattern applied to expression trees

The application of the Visitor design pattern to generate textual representations of expression trees is illustrated in Figure 6. Visitors heavily rely on double-dispatch to determine the concrete class of elements visited. While classes in the NodeVisitor hierarchy (Figure 6 b) provide behavior for expressing tree traversal, extensions to the Node class hierarchy offer double-dispatch support (Figure 6 a). Both, traversal and double-dispatch behavior have to be provided explicitly by the developer. Double-dispatch is present at run-time and can become problematic in performance-critical settings. The Node class hierarchy clearly resembles the fixed part in this design while subclasses of NodeVisitor amount to the variable part. The method acceptNodeVisitor: in the Node class hierarchy and the abstract class NodeVisitor constitute the glue code that is needed to enable an unlimited set of operations on expression trees. The Visitor glue relies on double-dispatch to determine the concrete class of elements visited. In addition, the glue typically describes the traversal behavior for operations. The double-dispatching scheme is not just difficult to understand – the glue code and the fixed part are also entangled as opposed to the definition of these two concerns in dedicated localizing modules.

4.2 Decorator

The intent of the Decorator design pattern is to allow the attachment of additional responsibilities to objects dynamically, to provide a flexible alternative to subclassing for extending functionality.

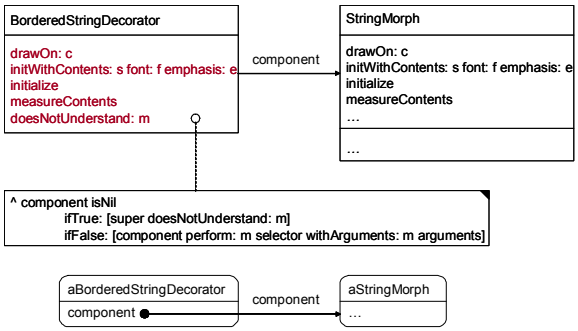

Figure 7: Decorator design pattern utilized for bordered string morph extension

As depicted in Figure 7, the Decorator design pattern can be utilized for instance-specific behavior supplementing borders during StringMorph display rendering. A BorderedStringDecorator can be added to and removed from an individual instance of StringMorph. String morphs constitute the fixed part, bordered string decorators the variable part, and the code wrapping string morphs with bordered string decorators the glue as shown in Figure 5. The code wrapping string morph components with bordered string decorators constitutes the glue. While Decorator gives us the advantage of being able to

change instance behavior dynamically and on a per-instance base, it also has the disadvantage of creating different identities for the original object (a StringMorph) and the decorated object (a StringMorph wrapped by a BorderedStringDecorator) [5]. Being unaware of an object being decorated or not can lead to unexpected run-time behavior resulting in system failure. Furthermore, decorating requires the forwarding of all non-decorated operations which might be problematic with respect to run-time performance and system evolution.

5 AOP Representations of Design Pattern Solutions

For some design pattern solutions that were originally motivated by pure object-oriented best practices it can be beneficial to attempt an aspect-oriented representation, taking advantage of new language extensions and code composition mechanisms introduced by AOP. The aspect-oriented modularity constructs can yield to improvements in terms of better code locality, reusability, composability, implementation modularity, and comprehensibility [6].

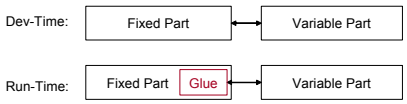

Figure 8: Transparent glue code placement at run-time

Figure 8 depicts how an AOP representation of design pattern solutions can separate fixed and variable parts of a system without requiring developers to explicitly provide glue code up-front. We rather assume that glue code is described implicitly together with the variable part during development, and inserted into the system via the aspect weaver during compile-, load-, or even run-time. Glue code implementing indirection layers, double-dispatch etc. from the original design pattern solutions might still lead to run-time performance degradation.

5.1 Visitor

An implementation of the Visitor design pattern can be realized straightforwardly as demonstrated in Figure 9. The fixed part of the system – represented by the Node class hierarchy – does not need to be prepared at all (Figure 9 a). The NodeVisitor class hierarchy stays the same as already proposed in the original Visitor pattern solution (Figure 9 b). The NodeVisitor aspect is responsible for introducing the double-dispatch code into the base system (Figure 9 c). The NodeVisitor class hierarchy is, similar to the one suggested by the original solution, essentially the same as presented in section 4.1 (Figure 9 a).

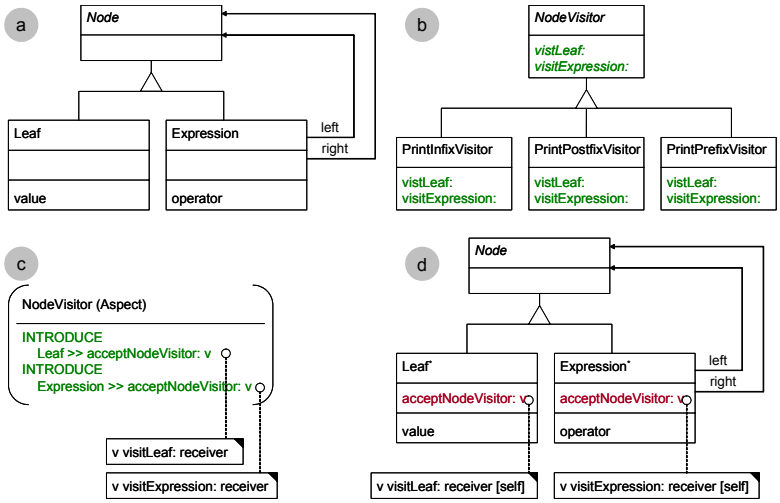

Figure 9: System before and after the application of the NodeVisitor aspect

After weaving this aspect into the system, the double-dispatch behavior gets distributed over the Node class hierarchy and after that the class hierarchy behaves exactly like the one discussed in section 4.1 (Figure 9 d). Here, the unaffected Node class hierarchy represents the fixed part, die traversal behavior introduced by the NodeVisitor class hierarchy the variable part, and the NodeVisitor aspect the glue code found in Figure 8. The plain Node class hierarchy represents the fixed part in this design, the traversals introduced in the NodeVisitor class hierarchy represent alternative variable parts, and the NodeVisitor aspect localizes the glue code for double-dispatch. Note that, even though this solution is

more modular than the initial approach it is nothing more than an AOP representation of the original design in that the woven class hierarchy behaves exactly like the one in the original pattern solution. The system still contains the controversial double-dispatch mechanism after the system transformation performed by the aspect weaver.

5.2 Decorator

There is no meaningful AOP representation of the Decorator design pattern structure. This is because the variation point is already properly manifested by the mere component's interface, and the variable parts including their dynamic plug-unplug properties are realized via plain object composition. Hence, there is simply no need to localize any glue code to be weaved into the base system. As pointed out in [6], one can attempt an advice-based implementation of Decorator by intercepting method calls of the decorated object to perform extra functionality such as for bordered string morphs. However, this design is not faithful to the dynamic manipulation properties of Decorator, and it is thus strictly less flexible than suggested by the Decorator pattern's intent. In section 6.2 we will show how the Decorator can be realized as a native AOP solution employing AspectS.

6 Native AOP Solutions

While it is possible to apply AOP to represent design pattern solutions motivated and constrained by object-oriented programming, it is more desirable to make use of AOP's newly introduced constructs and mechanisms to improve these solutions. We will show how AOP can be utilized to avoid certain drawbacks like model bloat and messaging-overhead caused by indirection levels, and context-dependent change of identity as a consequence of placing intermediate objects mediating between two instances.

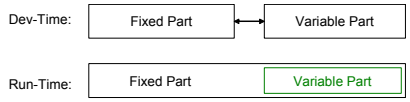

Figure 10: Transparent inclusion of variable system parts

The main property of a native AOP solution is that – while it allows developers to separate concerns clearly and deal with them separately and one at a time during development – the fixed and variable parts of a system are combined seamlessly and are indistinguishable from each other at run-time (Figure 10). Besides, there is no glue code to be provided by developers. The integration of fixed and variable system parts is done implicitly during weaving. As a consequence, developers improve the implementation's modularity by representing selected concerns locally, and run-time performance is not affected by problems such as messaging overhead or loss of identity (one of the symptoms of 'object schizophrenia') introduced by the approaches discussed above. The elimination of glue code simplifies design and implementation. In the following subsections, we will illustrate these new opportunities of native AOP solutions.

6.1 Visitor

We propose the following native AOP solution to the Visitor problem: Instead of utilizing double dispatch to isolate traversal behavior and to reconnect such behavior with the elements to traverse (as done in [6]), we use aspects to introduce such traversal behavior directly into elements classes. With this approach, all the complexity added through the Visitor is eliminated.

Figure 11 depicts the aspects responsible for introducing element traversal code (Figure 11 a), and the class tree transformation performed by the aspect weaver (Figure 11 b and c). The outcome of this transformation is a Node class hierarchy similar to the one explained in section 3.1 – maintaining its advantages, but without carrying on its problems. While developers can cleanly separate element traversal code from the actual elements, this separation is not present at run-time anymore. The design by itself is modular because a particular traversal operation is localized in a dedicated aspect for all the relevant classes. In fact, the absence of the double-dispatch protocol implies a simpler and more modular design. At run-time, the separation of the base system and the traversal concern is not present anymore which also implies the elimination of the associated messaging overhead. The Node hierarchy represents the fixed part of the system, and the various aspects correspond to alternative variable parts. The result of the class tree transformation mirrors the tangled Node class hierarchy as presented in section 3.1, making additional glue code obsolete.

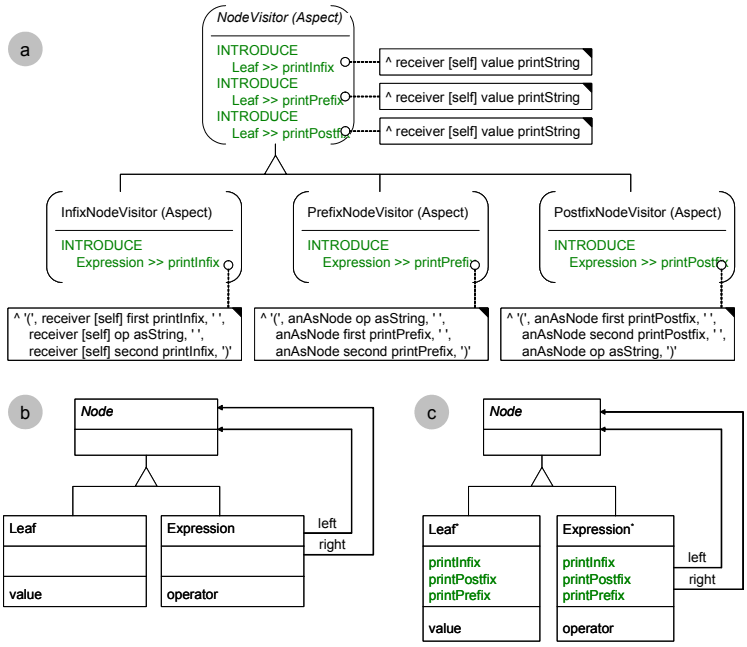

Figure 11: Aspect injecting visitor behavior

6.2 Decorator

Also the intent of the decorator can be realized by a native AOP solution. Here, we take advantage of AspectS' support for the dynamic introduction of instance-specific behavior. Instead of using object composition to wrap the original object and its behavior, we modify the behavior of the actual instance directly.

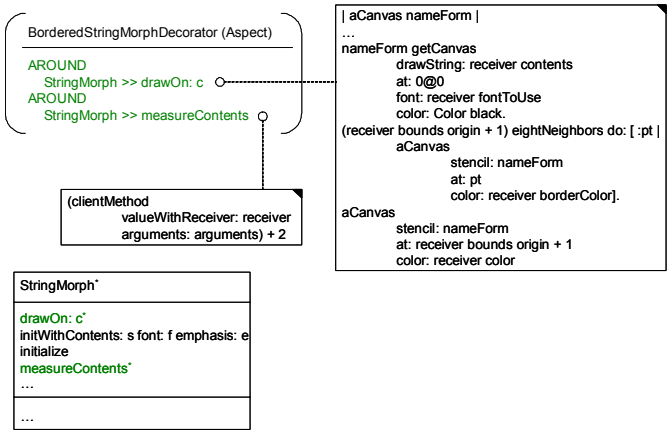

Figure 12: Decorator aspect adding instance-specific behavior

As shown in Figure 12, two pieces of around advice that are provided by the BorderedStringMorphDecorator aspect are applied to the two methods of StringMorph involved with display rendering. This design is completely straightforward and modular: the StringMorph class represents the fixed part of the system, and the BorderedStringMorphDecorator aspect provides the variable part of Figure 8. Note that with an around advice causing in-place modifications, no additional code is necessary to perform forwarding of non-decorated messages. For this solution, it is indispensable to employ run-time weaving to install and uninstall aspects on-demand. Otherwise we would lose the Decorator's dynamic manipulation properties and thus its essential flexibility. The major achievement of our native AOP solution is that object identity problems are ruled out. By operating on instances – so that they change their behavior instead of wrapping them using other objects – the identity of such instances does not change from the point of view of the sender of a message. Therefore, no extra infrastructure is needed to compensate for that.

7 Summary and Final Remarks

In this paper we present an aspect-oriented approach to design patterns. For some design pattern solutions that were originally motivated by pure object-oriented best practices it can be beneficial to attempt an aspect-oriented representation, taking advantage of new language extensions and code composition mechanisms introduced by AOP. However, an aspect-oriented representation of a design pattern solution is just that – an alternative implementation of the same pattern. Besides better code locality, there is no real improvement involved here because the design doesn't change fundamentally.

We advocate a native AOP approach to design patterns. In contrast to [6], [9] and [13] we put an emphasis on improving design pattern solutions both during development and at run-time. While during development the aspect-oriented modularity constructs can yield to improvements in terms of better code locality, reusability, composability, implementation modularity, and comprehensibility, the aspect composition mechanisms mainly carried out through aspect weaving can be utilized to avoid certain drawbacks like messaging-overhead caused by indirection levels and context-dependent change of identity as a consequence of placing intermediate objects mediating between two instances. From a different point of view, our native AOP approach to design patterns [5] can be perceived as aspect-oriented patterns since they change one important implicit force behind these patters from the use of object-oriented programming languages of a certain kind to the use of aspect-oriented programming languages or language extensions. This change in forces allows for the design of new solutions to the same problems.

Our approach is exemplified on two common design patterns, the Visitor and the Decorator. In the context of the Visitor we applied a native AOP solution to eliminate double-dispatch by seamlessly integrating element traversal code back into elements. Our native AOP solution for the Decorator lets us adjust instance-specific behavior without the loss of identity of the object originally wrapped in the Decorator pattern solution. All the examples presented in this paper have been implemented in Squeak using AspectS [2][8]. The work presented is based on the idea that system designs can typically be separated into fixed and variable parts, as well as glue. We have focused on the introduction of variation points into the design which allows us to separate fixed and variable parts that are eventually used to synthesize the desired system. Besides good modularization, this approach allows us to evolve conceptually separate parts independently. Dynamic on-demand run-time weaving as introduced in AspectS is an essential part of our work to resolve issues related to instance-specific object composition.

Acknowledgements

We would like to thank Mira Mezini, Stephan Herrmann, Jorrit Herder, and Dirk Riehle for their comments and contributions.

References

[1] AspectJ Web Site. http://www.aspectj.org.
[2] AspectS Web Site. http://www-ia.tuilmenau.de/hirsch/Projects/ Squeak/AspectS.
[3] K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, TU Ilmenau, 1998.
[4] R.P. Gabriel. *A Pattern Definition*. http://www.hillside.net/patterns/definition.html
[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[6] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proc. of the 17th OOPSLA Conference*, pages 161–173, Seattle, Washington, 2002.
[7] Hillside Group. http://www.hillside.net/patterns/
[8] R. Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, LNCS 2591, Springer, 2003.
[9] E. A. Kendall. Role Model Designs and Implementations with Aspect Oriented Programming. In *Proc. of the 14th OOPSLA Conference*, pages 353–369, Denver, Colorado, 1999.
[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of the 15th ECOOP Conference, LNCS 2072*, pages 327–355, Budapest, 2001. Springer.
[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the 11th ECOOP Conference*, LNCS 1241, pages 220–242, Jyväskylä, 1997.
[12] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD),* pages 41–55, Twente, The Netherlands, 2002.
[13] M. E. Nordberg III. Aspect-oriented dependency inversion. In *Proc. of the OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, Florida, 2001.
[14] P. Norvig. *Design patterns in dynamic programming*. Tutorial at Object World 1995, http://norvig.com/.
[15] Squeak Web Site. http://www.squeak.org.