Implementing Design Patterns in CaesarJ: an Exploratory Study

Edgar Sousa
Departamento de Informática
Universidade do Minho
Campus de Gualtar
4710-057 Braga
PORTUGAL
edgar@di.uminho.pt

Miguel P. Monteiro
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica
PORTUGAL
mmonteiro@di.fct.unl.pt

ABSTRACT
In the past, repositories of examples of the well-known Gang-of-Four design patterns brought insights on the potential contributions of aspect-oriented programming, as well as providing a suitable case study for subsequent research. In this paper, we present the first results of an ongoing effort to bring these advantages to a broader range of aspect-oriented languages. We present several implementations in CaesarJ of seven Gang-of-Four patterns. A short analysis follows, in which a comparison is made with AspectJ.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – patterns, information hiding, and languages; D.3.3 [**Programming Languages**]: Language Constructs and Features—patterns, classes and objects

General Terms
Design, Languages

Keywords
Design patterns, CaesarJ.

1. INTRODUCTION
Design patterns are a distillation of many real systems for the purpose of cataloguing and categorizing common programming and design practice. The most well-known design patterns are the 23 Gang-of-four (GoF) patterns, which propose flexible solutions for many design and structural issues [9]. The presentation of each pattern is structured in a number of parts, including purpose or intent, applicability, structure of the solution and code examples.

As a collection, the GoF patterns provide a rich catalogue of problems and corresponding solutions that can be found in large and complex systems. A repository of implementations is a good case study for some kinds of research. It also has the advantage that the patterns can be tackled one at a time and each individual example can be kept simple. This eases the task of someone approaching the collection, specific code examples and/or the language in which the example is written.

In the GoF book, the examples are coded in languages that were mainstream (mostly C++) at the time in which the book was published. It was noticed that the patterns provide many insights on both the strengths and limitations of the languages in which the patterns are coded, as well as providing hints as to what language features could overcome the limitations [3]. In this paper, we are mainly concerned with languages for aspect-oriented programming (AOP) [5].

In recent years, the GoF patterns were used for various purposes, namely as a showcase for some AOP languages [12], to illustrate the advantages of a given AOP language over some other language used as benchmark [17][21], and as a basis for tutorials [18]. In some cases, the code examples were made publicly available [12], which opened the way for its use in further research [10][20].

Though the GoF patterns proved useful for a number of purposes and there are now many AOP languages available [5], currently most AOP languages lack its own repository of GoF implementations. To our knowledge, just two repositories were developed [12][21], the one in AspectJ being publicly available. We believe that it would be beneficial if a wider range of aspect-oriented languages was used to develop implementations of patterns such as the GoF. If such repositories were available, they would provide case studies for various kinds of research as well as facilitating comparisons and assessments.

In this paper, we present the early results of an ongoing effort to develop implementations of the GoF patterns in the CaesarJ language [2]. Examples for seven patterns were developed[1]. In the long term, we plan to develop repositories of GoF implementations in various aspect-oriented languages, to be used as a basis for assessing the relative strengths and weaknesses of the languages involved, both relative to each other and to more traditional languages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPLAT 2008, March 31, 2008, Brussels, Belgium.
Copyright © 2008 ACM 978-1-60558-144-6/08/0003…$5.00.

[1] The examples are available for download at: http://ctp.di.fct.unl.pt/~mpm/CaesarJGoF4SPLAT.rar

The rest of this paper is structured as follows. Section 2 provides a short introduction to CaesarJ. Section 3 outlines our approach to the task. Section 4 provides an initial analysis of the implementations. Section 5 suggests several opportunities for future work and section 6 concludes the paper.

2. INTRODUCTION TO CAESARJ

CaesarJ shares several features with AspectJ [1], presently the most popular and widely used AOP language. However, CaesarJ also has some important differences from AspectJ. This paper assumes familiarity with AspectJ and outlines the main features of CaesarJ by underlining the differences from AspectJ. For more information on CaesarJ, we refer to Aracic et al [2].

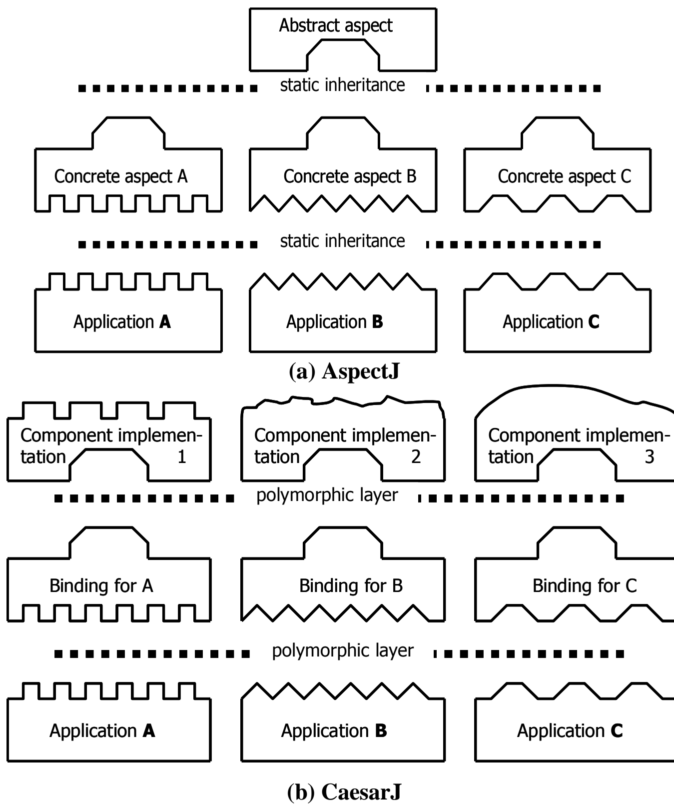

Figure 1. Mechanisms for reuse for AspectJ and CaesarJ.

CaesarJ uses a joinpoint model that is akin to that of AspectJ. Figure 1 provides an overview of the similarities and differences between AspectJ and CaesarJ as regards support for reusability. Part **a** of Figure 1 outlines AspectJ's division between a reusable abstract aspect, a case-specific concrete aspect and the specific application to which aspect functionality is to be composed. To bind the abstract aspect to each case-specific system, a different concrete sub-aspect is created for each case. However, there is a single abstract aspect for all concrete aspects. It is one of possibly many alternative implementations, but in AspectJ it is frozen and cannot be replaced by an alternative implementation without invasive changes on the source code. By contrast, in CaesarJ it is possible to polymorphically replace one implementation of the reusable part of a component with another, without impact on the remaining modules of the component (part **b** of Figure 1).

In addition to plain-Java classes, CaesarJ supports a second kind of class (**cclass**) that supports *family polymorphism* [7], i.e. the capability to treat sets of inner, nested classes polymorphically, the same way as methods. These nested classes are termed *virtual classes* [15]. The name of any class nested within a **cclass** can be dynamically bound to different concrete classes in different circumstances, the same way method calls are late bound to the most specific implementation in plain Java. Figure 2 outlines the structure of a CaesarJ component.

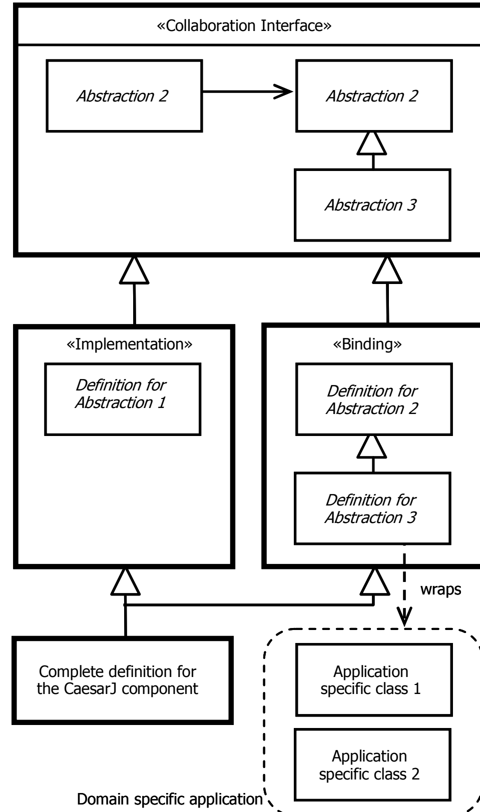

Figure 2. General structure of CaesarJ component.

An key notion in the CaesarJ approach to design is that of *collaboration interface* (CI) [16]. A CI is a top-level abstract **cclass** that declares a collaboration between objects, thus representing design-level relationships between abstractions such as those often represented in the abstract types of a class diagram. Figure 2 shows one example of a CI at the top, with an arbitrary internal structure involving three classes.

CaesarJ recognizes that component integration comprises several parts which should be kept separate and independent. The higher-level part that provides the general framework is the CI, with the remaining parts extending the CI through inheritance. In addition, CaesarJ takes into account that there are two sides to component integration: (1) the CaesarJ *implementation*, i.e., the implementation of the component itself and (2) the *binding*, i.e., the glue that integrates the component to a specific application. According to these concepts, implementation code does not make references to case-specific classes and is therefore reusable. A CaesarJ implementation corresponds to an abstract aspect in AspectJ and comprises a top-level **cclass** module that inherits

from the CI. Likewise, CaesarJ bindings are top-level **cclass** modules that also inherit from the CI and enclose the logic that glues the component to a specific application. These correspond to the concrete sub-aspects in AspectJ. To integrate the component the application, nested classes of CaesarJ bindings can *wrap* one or several objects of arbitrary types and extend them with additional state and behaviour. CaesarJ wrappers perform the same role as the inter-type declarations and declare parents clauses of AspectJ.

CaesarJ implementations and bindings must between them provide definitions for all declarations placed in the CI. Finally, CaesarJ uses mixin composition [4] to compose the two different, parallel hierarchies of implementations and bindings that extend a CI so as to create a single, unified component. In all cases discussed in this paper, it is simply a matter of creating an empty **cclass** that extends the implementation and the binding.

3. APPROACH TAKEN

Both AspectJ and CaesarJ enable aspects to be divided into a general part that is applicable to multiple cases and a part that encloses the details specific to the case at hand. However, CaesarJ goes significantly further than AspectJ in structuring those parts. The parts of a CaesarJ component are clearly separated into different modules whose implementations can be polymorphically replaced, without impact on the other modules. Relative to AspectJ, this provides more opportunities for reuse. Presenting a *single* implementation of each pattern wouldn't illustrate the full extent of CaesarJ's capabilities. We therefore set the aim of developing multiple implementations of the selected patterns.

All scenarios used to develop the CaesarJ examples are based on existing repositories rather than invented anew. Using independent scenarios yields more credible results and is more likely to provide interesting problems and situations. For this reason, we are basing our work on existing repositories of implementations freely available on the Net[2] (see Table 1).

Table 1. Repositories of GoF implementations publicly available

Repository name	Author	Lang-uage	URL
Thinking in patterns	Bruce Eckel	Java 2	http://www.mindview.net/Books/TIPatterns/
Design pattern Java companion	James Cooper	Java 2	http://www.patterndepot.com/put/8/JavaPatterns.htm
Fluffycat	Larry Truett	Java 2	http://www.fluffycat.com/Java-Design-Patterns/
Hannemann et al	Hannemann/ Kiczales	Java 2/ AspectJ	http://hannemann.pbwiki.com/Design+Patterns
Huston	Vince Huston	Java 2	http://www.vincehuston.org/dp/

There is a wide range in scenarios, style and implementations decisions used for the patterns. For instance, some authors heavily rely on graphics objects from the Java standard APIs while other repositories are mostly send messages to the console. In some cases, the participants in the pattern are instances of classes from the Java standard API, though more often these are represented by specific classes for a simple scenario. Naturally, data structures used also vary and in some cases functionality from the standard Java API is used instead. For instance, Eckel resorts to Java's Observer/Observable API to implement Observer, while Hannemann and Kiczales use weak hash maps and array lists.

The examples developed are those shown in Table 2. The patterns were selected for various reasons. Observer is most often used to illustrate CaesarJ's strengths and there its implementation is already published [17] and thus represents an ideal entry point to someone approaching CaesarJ. Chain of Responsibility is structurally similar to Observer, though simpler (just one participant instead of two) and thus looked a direct follow up. Singleton is simple and many people's entry point to the GoF, being selected for these reason. Decorator was chosen to test the wrapper mechanism and possibly mixin composition. Abstract Factory seemed the ideal candidate to test family polymorphism and virtual classes. Visitor and Bridge were selected because the patterns represent interesting structural problems suitable for testing CaesarJ's composition capabilities. See also the analysis (section 4).

Table 2. CaesarJ implementations developed for the GoF

	Thinking in patterns	DP Java companion	Fluffycat	Hanneman n et al	Huston
Abstract factory		X	X		
Bridge			X		X
Chain of responsibility			X	X	
Decorator				X X	
Observer[3]	X X				
Singleton				X	
Visitor	X		X		

The component structure supported by CaesarJ provides a clear guide as to where each piece of code is to be placed and suggests the following process:

- First, analyze of the original example, with the help of a class diagram. In some cases, the abstract part of the structure maps directly into a CI.
- Next, place all implementation code not dependent of case-specific types in the implementation module.
- Place of all code depending on case-specific types in the binding module.
- Create the complete component through an empty **cclass** that extends the implementation and the binding.

[2] The list of repositories presented in Table 1 is not an exhaustive one. Covering more repositories is a possibility, left for future work.

[3] Two different scenarios of Observer by Eckel were implemented, with three different implementations.

4. PRELIMINARY ANALYSIS

Structurally, all CaesarJ examples are more or less complex variants of the framework outlined in section 2. One instance is Observer, whose case is well-known. Thanks to the enhanced polymorphism, we managed to hold multiple implementations and bindings in the same system [17].

4.1 Mechanisms used

Figure 3 outlines the AspectJ and CaesarJ approaches to integrating an aspect to an application. In AspectJ, the mapping between abstractions from the component world to abstractions in the application world is performed in two phases. First, inner marker interfaces represent the concepts and declare parents compose extra members to them. Second, declare parents clauses bind the interfaces, and thus the extra members, to the classes from the application. One consequence of this approach is that the extra members lack a proper "home" to encapsulate them. Instead, those inter-type members are top-level members within the aspect. This explains way AspectJ aspects tend to have a flat internal structure [17]. By contrast, the use of CaesarJ bindings with wrappers means that extra members are enclosed in their own modules, which results in a richer internal structure. This also explains why CaesarJ is less asymmetrical than AspectJ. In CaesarJ, the programmer reasons less in terms of "aspects" and "base code" than in terms of components and their constituent modules.

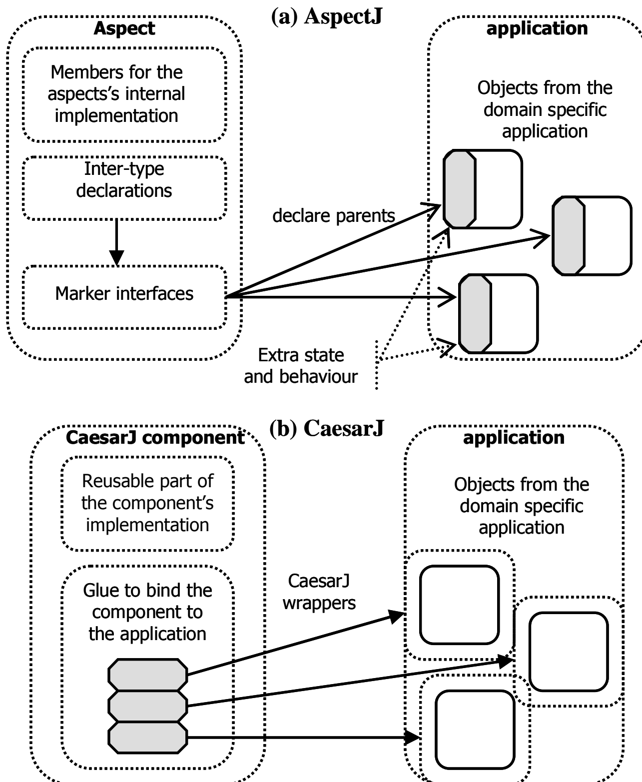

Figure 3. Mechanisms for integrating with an application

The fact that CaesarJ shares with AspectJ the mechanism of pointcuts and advice invites some comparisons. One interesting question is whether pointcuts are used the same way as in AspectJ, or the CaesarJ-specific mechanisms have an influence on the use patterns for pointcuts and advice. It turns out that in CaesarJ, pointcuts and advice feature less prominently than with AspectJ. Thanks to CaesarJ's more sophisticated mechanisms to deal with structure, it is possible to separate the various parts of a component in different modules to a greater extent than with AspectJ. One is thus more sparing in the use of pointcuts and advice, whose use is essentially reduced to that of glue code, i.e., in the bindings. The experience gained so far suggests that in CaesarJ, pointcuts and advice should be left to those situations in which the behaviour to be composed does not follow an identifiable pattern in the static structure of the system, e.g., scattered calls to a method or constructor. That is the case in all uses of pointcuts and advice in the examples presented in Table 3.

The Decorator pattern is a good example to illustrate the differences between the two approaches to pointcuts. We developed two different implementations of Decorator, one that uses a pointcut and around advice similar to that proposed by Hannemann and Kiczales [12], and one that resorts to the wrapper mechanism. The use of wrappers conforms more closely to the original intent of the pattern, namely in the dynamic nature of the compositions [14] and in the possibility of varying the order with which decorators are composed [19]. In the context of CaesarJ, the approach based on pointcuts and advice is a case of overuse of the mechanisms. However, there remains the issue of application specific objects loosing their original identity, whose thorough analysis is left for future work.

Table 3 presents a summary of the use of CaesarJ pointcuts, CIs, implementations and bindings, in the implementations referred in Table 2. As it would be expected, not all examples include all features. The exception are the bindings, which is also what we expected. Note that some of the constructs available in CaesarJ are not covered in this paper (e.g., deploy on object) as the implementations from Table 2 do not comprise sufficient material to perform an assessment.

Table 3. Use of mechanisms in the CaesarJ examples.

Use of the mechanism:	pointcut /advice	CI	Imple-mentation	binding
Abstract Factory	No	No	No	Yes
Bridge	No	Yes	Yes	Yes
Chain of Responsibility	Yes	Yes	Yes	Yes
Decorator	No(*)	No	No	Yes
Observer	Yes	Yes	Yes	Yes
Singleton	Yes	No	Yes	Yes
Visitor	No	Yes	No	Yes

(*) *One scenario does use pointcuts/ advice but in this case we do not consider it good practice and do not count it for this reason.*

The CaesarJ Singleton resembles that in AspectJ, since the CaesarJ example is based on pointcut and advice, just like the AspectJ example. However, even in this simple example CaesarJ provides an opportunity to separate a reusable advice into an implementation module. Since the example does not include a CI,

the implementation is placed at the top of the inheritance hierarchy. Similar design decisions were made in some of the other examples.

Since the inter-type declarations of AspectJ can be regarded as a manifestation of mixin composition, we initially hypothesised that mixins could be used as a more structured alternative to the inter-type declarations, as well as providing direct language support to decorators. However, mixin composition in CaesarJ cannot be used “on the fly”: a specific declaration of a cclass extending the pertinent modules is required for each different combination. This defeats the aim by Decorator of preventing a combinatorial explosion of class declarations. In addition, mixin composition is restricted to cclass modules. For these reasons, the primary of mixins is to compose modules that extend a CI.

Abstract Factory is really about how to enforce family polymorphism in a language that is not endowed with the feature. Since CaesarJ is, it makes a perfect fit for the pattern. However, due to the nature of the problem, no reusable code was obtained. Each scenario maps to different, case-specific code (see Figure 4).

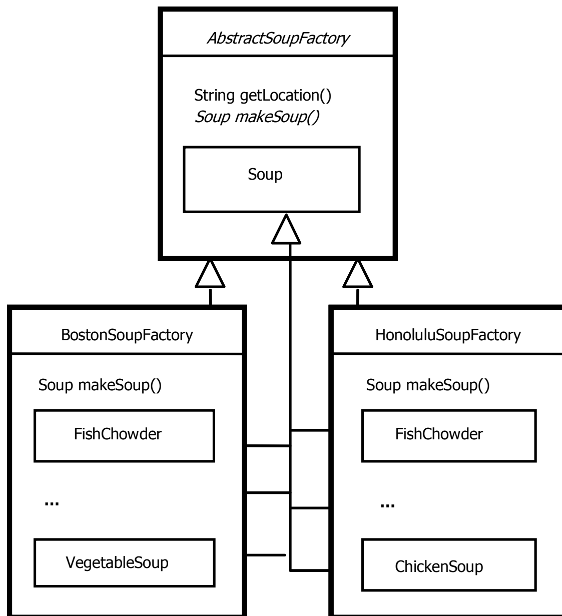

Figure 4. Class diagram of the CaesarJ example of the fluffycat scenario for the Abstract Factory pattern

Visitor is about an inheritance tree to whose classes one may want to add various different additional operations. It quickly becomes cumbersome to place the logic for many different operations in the classes of the tree. The solution proposed by Visitor is to provide an accept operation to each of the classes, through which a visitor object is passed, containing the logic for the additional operation. In other words, Visitor embodies the problem of double dispatch, i.e., the ability to select a given block of code based on two different types that can evolve independently. This is an instance of the more general case of *multiple dispatch* [6]. Ideally, CaesarJ would support this feature as regards a family of types, but it supports only single dispatch based on the type of the family object associated with the concrete family at hand. This limitation motivated the proposal by Gasiunas et al [11] of multiple dispatch on virtual types.

4.2 Reuse

Besides features shared with plain Java, AspectJ provides one kind of module to enclose reusable code: abstract aspects. In CaesarJ, these correspond to two kinds of module: CIs and implementations. Table 4 shows what CaesarJ patterns result in such reusable modules and how the results compare with the AspectJ examples for the same patterns [12].

Depending on the nature of the pattern, different levels of reuse are obtained. In some cases, only the high level design (i.e., the CI) is reusable and in others only the implementation is reusable.

Table 4. Reusable modules in AspectJ and CaesarJ

Reusable parts:	**AspectJ** (abstract aspects)	**CaesarJ** (CIs)	**CaesarJ** (implementations)
Abstract Factory	No	No	No
Bridge	No	Yes	Yes
Chain of Responsibility	Yes	Yes	Yes
Decorator	No	No	No
Observer	Yes	Yes	Yes
Singleton	Yes	No	Yes
Visitor	Yes	Yes	No

4.3 Reasoning with Collaboration Interfaces

One advantages of CaesarJ clearly felt relative to both Java and AspectJ was when reasoning about the examples through class diagrams. CIs provide a design-level view of a component or subsystem found in class diagrams but generally absent in mainstream languages. For this reason we initially expected that CIs would be of help to reason with the overall structure of the component when looking at the code. It turned out that class diagrams of CaesarJ designs provided significant benefits as regards comprehensibility. The diagrams representing the CaesarJ designs are conceptually closer to the original design intentions than traditional class diagrams, exposing the relationships between classes and individual operations more faithfully than traditional class diagrams (e.g., those representing the Java implementations from Table 1). The distinction between top-level classes and nested classes facilitated the reasoning with the overall design, as well facilitating the task of mapping the original Java designs to CaesarJ designs. The enhanced clarity also applies to top-level methods. For instance, the factory method from the CaesarJ design for Abstract Factory (see Figure 4) is a top level method placed on the same level as the virtual classes. This exposes a design decision that is absent from traditional designs because nested classes are absent from traditional class diagrams and top-level methods do not carry the same meaning.

5. FUTURE WORK

5.1 Complete the CaesarJ Repository

The results presented in this paper are the preliminary results of an ongoing effort. Our immediate goal is to complete the GoF repository for CaesarJ. It is to be expected that the CaesarJ implementation of some patterns will be identical to Java (e.g., Iterator). CaesarJ features not covered in this paper will be explored (e.g., dynamic deployment) and a more thorough analysis will be carried out. One interesting issue is to assess the comparative advantages of the wrapper mechanism w.r.t. AspectJ-style inter-type declarations. Using inter-type declarations, target objects (instances of the original, target classes) and additional members share the same identity. Though inter-type declarations are structurally poorer than wrappers, we conjecture that this characteristic may be advantageous in some cases.

5.2 Derive Refactorings for CaesarJ

In the past, availability of a GoF repository in a given language was used as a basis for pinpointing refactorings [20] for that language. This work provides similar opportunities for CaesarJ.

5.3 Larger Case Studies

After developing a repository of GoF implementations, a logical next step is to apply the insights gained to object-oriented frameworks whose structure is based on the patterns. This can be carried out by developing a given system anew, or through refactoring experiments on existing systems.

5.4 Extend Work to Other AOP Languages

We aim to cover more aspect-oriented languages. For instance, the Object Teams language [13] has some features in common with CaesarJ, namely family polymorphism. One interesting proposition is to compare Object Teams' support for family polymorphism with CaesarJ's.

6. CONCLUSION

This paper presents the first results of an ongoing effort to develop a repository of implementations of the GoF patterns in CaesarJ. Examples for seven patterns are presented and a short comparative analysis is made with an existing AspectJ repository. The experience gained suggests that CaesarJ' features to deal with static structure, namely family polymorphism and a clear separation of implementations and bindings, lead to a more flexible management of the constituent parts of a component and avoids overusing pointcuts and advice, which are used primarily for glue code.

7. ACKNOWLEDGMENTS

This work was partially supported by projects SOFTAS (POSI/EIA/60189/2004) and AMADEUS (POCTI, PTDC/EIA/70271/2006) funded by *Fundação para a Ciência e Tecnologia*, and project AspectGrid (GRID/GRI/81880/2006). Three anonymous reviewers gave feedback that helped to improve the paper and provided useful ideas to be explored in future work.

8. REFERENCES

[1] Aspectj project home page, http://www.eclipse.org/aspectj/.

[2] Aracic I., Gasiunas V., Mezini M., Ostermann K. Overview of CaesarJ. Transactions on Aspect-Oriented Software Development I. Springer LNCS vol. 3880, 2006.

[3] Baumgartner G., Läufer K., Russo V. F. On the interaction of Object-oriented Design Patterns and Programming Languages. Technical report CSD-TR-96-020, Purdue University, Feburary 1996.

[4] Bracha G., Cook W. Mixin-Based Inheritance. Proceedings of ECOOP/OOPSLA 1990.

[5] Brichau J., Haupt M. Report describing survey of aspect languages and models. AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01, May 2005.

[6] Chambers C. Object-oriented multi-methods in Cecil. ECOOP '92, Utrecht, The Netherlands, 1992.

[7] Ernst E. Family polymorphism. ECOOP 2001, Heidelberg, Germany, 2001.

[8] Filman R. E., Elrad T., Clarke S., Aksit M. Aspect-Oriented Software Development. Addison Wesley 2005.

[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[10] Garcia A., Sant'Anna C., Figueiredo E., Kulesza U., Lucena C., Staa A. Modularizing Design Patterns with Aspects: A Quantitative Study. LNCS TAOSD I, Springer vol. 3880, 2006.

[11] Gasiunas V., Mezini M., Ostermann K. Dependent types. ooPSLA 2007, Montréal, Canada 2007.

[12] Hannemann, J., Kiczales, G., Design Pattern implementation in Java and in AspectJ, OOPSLA 2002, Seattle, USA, 2002.

[13] Herrmann S. Object teams: Improving modularity for crosscutting collaborations. Net. Object Days, Erfurt, Germany, 2002.

[14] Hirschfeld R., Lämmel R., Wagner M. Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration. 3rd German GI Workshop on AOSD, 2003.

[15] Madsen O. L., Moller-Pedersen B., Virtual classes: a powerful mechanism in object-oriented programming. OOPSLA'89, New Orleans, Louisiana, USA, 1989.

[16] Mezini M., Ostermann M. Integrating independent components with on-demand remodularization, OOPSLA 2002, Seattle, USA, 2002.

[17] Mezini M., Ostermann K. Untangling Crosscutting Concerns with Caesar. Cap. 8 of [8].

[18] Miles R. AspectJ Cookbook. O'Reilly 2004.

[19] Monteiro M. P., Fernandes J. M., Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns. DSOA'2004 workshop, Málaga, Spain, 2004.

[20] Monteiro M.P., Fernandes J.M. Towards a Catalogue of Refactorings and Code Smells for AspectJ. LNCS TAOSD I, Springer vol. 3880, 2006.

[21] Rajan H., Design Patterns in Eos, PLoP '07, Monticello, Illinois USA, September 2007.