Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software

March 22, 2004

Held in conjunction with the Third International Conference on Aspect-Oriented Software Development (AOSD 2004)

Lancaster, UK

College of Computer and Information Science
Northeastern University
Boston, Massachusetts 02115
360 Huntington Avenue, 161CN

NU-CCIS-04-04

Yvonne Coady and David H. Lorenz (Eds.)

The Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)

Held as a one day Workshop at AOSD'04, The International Conference on Aspect-Oriented Software Development, March 22-26, 2004, Lancaster UK

Aspect-oriented programming, component models, and design patterns are modern and actively evolving techniques for improving the modularization of complex software. In particular, these techniques hold great promise for the development of "systems infrastructure" software, e.g., application servers, middleware, virtual machines, compilers, operating systems, and other software that provides general services for higher-level applications. The developers of infrastructure software are faced with increasing demands from application programmers needing higher-level support for application development. Meeting these demands requires careful use of software modularization techniques, since infrastructural concerns are notoriously hard to modularize.

Building on the ACP4IS meetings at AOSD 2002/2003, this workshop aims to provide a highly interactive forum for researchers and developers to discuss the application of and relationships between aspects, components, and patterns within modern infrastructure software. The goal is to put aspects, components, and patterns into a common reference frame and to build connections between the software engineering and systems communities.

Program Committee

Elisa Baniassad, Trinity College
Don Batory, University of Texas at Austin
Shigeru Chiba, Tokyo Institute of Technology
Yvonne Coady, University of Victoria (co-organizer)
Pascal Costanza, University of Bonn
Jeff Gray, University of Alabama at Birmingham
Stephan Herrmann, Berlin Technical University
Wilson Hsieh, University of Utah
Julia Lawall, University of Copenhagen
Cristina Videira Lopes, University of California Irvine
David Lorenz, Northeastern University (co-organizer)
Klaus Ostermann, University of Technology Darmstadt
Renaud Pawlak, University of Lille
Calton Pu, Georgia Tech
John Regehr, University of Utah
Mario Südholt, École des Mines de Nantes
Jan Vitek, Purdue University
Rob Walker, University of Calgary

Papers

Personalization as a Cross-cutting Concern in Web Servers: A Case Study on Java Servlet Technology
Jordi Alvarez, Ignacio Gutierrez, Miguel-Angel Sicilia

Encapsulating Crosscutting Concerns in System Software
Christa Schwanninger, Michael Kircher, Egon Wuchner

Supporting Product Line Evolution with Framed Aspects
Neil Loughran, Awais Rashid, Weishan Zhang, Stan Jarzabek

Transaction Management in EJBs: Better Separation of Concerns With AOP
Johan Fabry

Software Plans for Separation of Concerns
David Coppit, Benjamin Cox

Towards the development of Ambient Intelligence Environments using Aspect-Oriented Techniques
Lidia Fuentes, Daniel Jimenez, Monica Pinto

Towards an Aspect Weaving BPEL engine
Carine Courbis, Anthony Finkelstein

Separation of Concerns in Compiler Construction using JastAdd II
Torbjorn Ekman

The Proxy Inter-Type Declaration
Michael Eichberg

Applying Aspect Orientation to J2EE Business Tier Patterns
Therthala Murali, Renaud Pawlak, Houman Younessi

Aspect-Oriented Design and Implementation in Java Bytecode Analyzer Framework
Susumu Yamazaki, Michihiro Matsumoto, Tsuneo Nakanishi, Teruaki Kitasuka, Akira Fukuda

Posters

Software Connectors in COSA Approach
Adel Smeda, Tahar Khammaci, Mourad Oussalah

Addressing Ubiquitous Software Complexity with Mobile Containers
Vasian Cepa

Towards unifying aspects and components
Houssam Fakith, Noury Bouraqadi, Laurence Duchien

JAsCoAP: Adaptive Programming for Component-Based Software Engineering
Wim Vanderperren, Davy Suvee

Infrastructural support for data dependencies in data-centered software systems
Lieven Desmet, Frank Piessens, Wouter Joosen

Personalization as a Cross–cutting Concern in Web Servers: A Case Study on Java Servlet Technology

Jordi Álvarez
IT and Multimedia Department
Open University of Catalonia
Avda Tibidabo 39–43, Barcelona (Spain)
jalvarezc@uoc.edu

Ignacio Gutiérrez
Carlos III University
Madrid (Spain)
nacho.gutierrez@worldonline.es

Miguel–Ángel Sicilia
Computer Science Department. Polytechnic School.
University of Alcal. Ctra. Barcelona km. 33.6
28871 Alcal de Henares, Madrid (Spain)
msicilia@uah.es

Abstract

Aspect–oriented design allows for a better modularization of cross–cutting concerns in software systems. The design of personalized (adaptive) Web applications — which can be considered as concrete realizations of hypermedia systems — essentially adds user modeling *actions (UMs) and* adaptive behaviors *(ABs) to the associative hypermedia structure comprised by nodes, contents and links. In consequence, UMs and ABs are typically spread across Web pages, becoming candidates for modularization in separate aspects, which would eventually result in improved development and maintenance. In this paper, we describe an approach for the inclusion of aspect–based user and adaptation modeling actions, based on the conceptual* `Labyrinth` *hypermedia model, along with a concrete case study of their implementation in* `Servlet` *technology.*

1. Introduction

Web systems are nowadays one the most important subclass of hypermedia systems, due to its widespread adoption as a means for a variety of activities like electronic commerce, publishing, communication and advertising. Personalized Web systems (also called adaptive Web systems [4]) are in turn a subclass of Web systems that build and elaborate some kind of model about their users, using it subsequently to tailor the hypermedia structure to the knowledge, preferences or objectives of each individual or group.

Building user models is a matter of spreading user modeling actions across the Web pages, so that certain navigation events are recorded or are used to conjecture about the characteristics of the user [15]. In a similar fashion, adaptive behaviors are spread across the generation of dynamic Web pages, since the process of tailoring requires ultimately querying the user model and generating the appropriate markup inside the span of a single `HTTP` request. For example, the reordering of links in a page according to the preferences of a concrete user (a technique often referred to as adaptive link sorting [3]) requires a query to her preference representation that can be subsequently used to sort the links. This cross–cutting structure of the code is a consequence of the fact that personalization is a separate concern in Web systems.

Existing Web personalization systems [7] provide specialized application programmer interfaces (APIs) that can be invoked from server–side markup languages like `Jsp` [16]. But even when such API calls are encapsulated in tag libraries, their inclusion obscures presentation markup and often produce tangled code. This nuisance becomes more problematic in system that provide extensive, fine–grained personalization, since the generation of personalized links an contents involves many scattered calls for each single page. In addition, it may be desirable in some cases to be able to extend an existing Web system for personalization without changing the code of their pages, and also to separate personalization from the rest of business logic to avoid interferences among the tasks of Web designers, Web programmers and personalization experts.

The techniques of aspect–oriented design (AOD) provide

improved modularity to software systems by focusing on separation of concerns [18]. In consequence, AOD can be considered a promising technique to obtain a better modularization for personalization–oriented actions, as sketched in [1].

AOD has already been applied to provide better modularity to the structure of the code in specific Web systems, like `Atlas` [9], and also to the internal architecture of Web servers [11], but it has not been studied how it could be applied to the Web implementation of systems based on generic hypermedia models. These models provide support for the specific characteristics of Web applications as concrete realizations of hypermedia systems, and therefore, they provide a good point of departure for the identification of the principal concerns of the architecture of Web applications. In addition, some preliminary work on early aspects of personalization exists [12, 8], but design and implementation issues regarding the topic have not been still studied.

In this paper, we describe a novel approach that applies AOD to the design of adaptive Web systems [4]. To do so, we describe the extension of a concrete and widely used Web technological infrastructure in terms of an existing comprehensive hypermedia model. More specifically, we have focused on the extension of `Servlet`–based technology with aspects for the purpose of implementing user modeling and adaptive behaviors based on the `Labyrinth` abstract hypermedia model described in [5].

The rest of this paper is structured as follows. Section 2 provides a basic model of hypermedia as applied to Java Web Servers, and the details of the AOD–based implementation of personalization are sketched in Section 3. Finally, conclusions and future research directions are provided in Section 4.

2. User Modeling and Adaptation as Cross-Cutting Concerns

The `Labyrinth` hypermedia model is centered around the notion of basic hyperdocument, representing a hypermedia application defined by seven sets of conceptual elements and a number of associated functions [5]. Some of these elements represent concerns for designers, e.g., the set U of users and groups represents *user modeling*, sets N (nodes) and C (contents) represent *content modularity*, sets L and A of links and anchors represent navigation, and the set B of attributes is a general–purpose facility for *description* of the rest of the elements of the model by means of $(name, value)$ pairs.

User modeling and content modularity can be characterized — according to the definitions in [2] — as *problem domain* concerns, if we consider its users to be Web designers, and they are also *canonical*, due to the generic nature of the models. The arbitrary definition of attributes is largely domain-independent, and as such, its applicability extends the scope of hypermedia applications. In what follows, we deal with a restricted model of `Labyrinth` hyperdocument that can be described by the expression (1) where lo (localization) describes the structure of contents as constituents of nodes.

$$H = (N, C, U, L, A, B, lo) \tag{1}$$

Prior to describing aspect–oriented implementation details, it becomes necessary to provide a mapping from the elements in (1) to physical elements in Java Web server–based applications, along with an account of the essential mechanism for the implementation of adaptive behaviors inside the standard processing of `HTTP` requests. These two issues are briefly described in the following sub–sections.

2.1 Mapping Users and Contents to Servlets

Java-enabled Web servers are based on the concept of `Servlet`, which is an abstraction of the (arbitrarily complex) resolution of an `HTTP` request. In addition, *Java Server Pages* (`JSP` pages), which are combinations of `HTML` and `Java` code, are internally translated to `Servlets`. In consequence, if we use `JSP` pages to represent both contents and nodes, the whole content structure is managed by the Java runtime, providing the necessary hooks to manage it in the server side. This does not conflict with the structuring of the application according to the *Model-View-Controller* (MVC) architecture, since contents and nodes can be static or dynamic, and their generation can be carried out by `Servlet` calls combined with `Jsp` code.

In abstract terms, we can define two different sets of `JSP` pages : one representing nodes (N) and another one representing contents (C). The distinction is left to the decision of the designer, and contents of low granularity like single image files, sounds or markup fragments can be designed as contents by putting them inside a separate `JSP` file.

Nodes and aggregated contents are formed conceptually by composition of other contents (and physically, for example, through server-side includes), so that the function $lo(n_i, c_j)$ describes the position of content c_j inside the node n_i (we have not considered time for simplicity), and $lo(c_i, c_j)$ can be used to describe aggregated contents. Physically, this compositions can be achieved implicitly by using `<jsp:include>` actions [16], for example.

Links can be embedded inside `Jsp` pages as usual `HTML` markup, or they can be designed as independent entities (by simply putting them in a separate `JSP` file just as contents) if required to implement a concrete adaptive behavior like contextual links, as described, for example, in [14]. For the sake of simplicity, link anchors — i.e. link source and destination points — will be restricted to entire nodes and

contents, not allowing finer grained specification[1].

User modeling can be carried out by associating a `LabyrinthUser` instance to the current session (anonymous users are simply an special case), which is automatically maintained by the `Java` Web runtime. It should be noted that user modeling encompasses every data collection and data elaboration functionality regarding the models the system builds about its users (by setting and/or updating elements in B), so that it includes simple profiling actions (like recording the visits to a given node) but also complex processes, perhaps entailing reasoning procedures (e.g. with the help of rule bases).

2.2. Designing Adaptive Behaviors in Servlet–Based Contexts

The above described straightforward mapping of hypermedia concepts to Servlet technology enables a range of adaptive behaviors to be implemented by manipulation of the composition of dynamic Web pages. Since separation of concerns is our main goal here, such manipulation will be carried out by post–processing the `HttpResponse` generated by a Servlet, so that business logic for each page is seen as a black–box. More elaborated and efficient implementation alternatives could be explored in the future.

Each adaptive technology [3] can be designed that way at content or node–level. Objects attached to the current session can be used to trace the process of composition of the node for its constituents contents.

3. Case Study: Extending Servlet Technology with Aspects for Personalization

As a proof of concept for our approach, we have used the `AspectJ` framework [10] to extend the `Tomcat` Web server[2]. Technically, it required the recompilation of a part of Tomcat source code in order to make its `JSP` compilation engine use the `AspectJ` compiler as the default internal `JSP` compiler in order to generate `.class` servlet files.

This modification allows to integrate the definition of user modelling behaviours into the `Servlet`'s protocol. These behaviours can be attached around the dynamic joint points involving method calls defined in the `Servlet` interface, including instantiation (constructors), initialization (`initialize`), request handling (`service`) and end of service (`destroy`). With the use of aspects containing definitions for the corresponding *joint points* and *advices*, behaviours for each content and node (items in $I = C \cup N$) can be defined. The specification of aspects on concrete items can be realized by linking the aspect with the `JSP` implementation class [16]

For example, behaviors can be attached to the end of the visit to any item in I by writing an aspect with an *advice* like the following:

```
after(HttpServletRequest req,
      HttpServletResponse resp,
      HttpJspBase h) returning :
  execution(* _jspService(ServletRequest,
    ServletResponse,HttpJspBase))
  && args(req,resp) && target(h) {
   ServletContext c =
      h.getServletContext();
  // code (possibly manipulating
  // session's information)...
}
```

Figure 1 shows an example of an adaptive behaviour defined using this technique. The example is further explained in in section 3.2. Figure 1 shows how the use of AOP helps in introducing adaptive behavior while a JSP page request is being processed. The UML notation being used was introduced in [17].

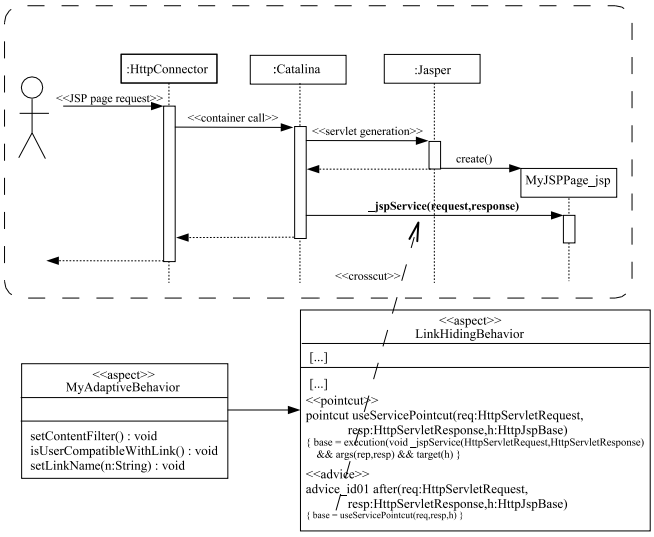


Figure 1. Introducing adaptive behavior into a servlet container

In the rest of this section, the more relevant implementation details of user and content modeling actions and the overall scheme to implement adaptive behaviors are sketched.

3.1. User and Content Modeling

Two approaches to include aspect-based code have been implemented, one for programmers and the other for Web

[1] Recent XML–based languages like XPointer enable arbitrarily complex anchor specifications that should be subject of future work.

[2] `http://jakarta.apache.org/tomcat/`

designers. Both of them can be considered an aspect-based implementation of the concept of *event* in Labyrinth [6], and have the main advantage of the separation of event implementation from content elements (JSP pages).

Programmers can develop aspects by extending two predefined abstract aspects: UserModellingAspect and ContentModellingAspect. Both of them contain

The first one provides two attributes: one refers to the LabyrinthUser instance (user) representing the current user, and the other one corresponds to the type LabyrinthContent (content) and represents the current node or content being visited. The second aspect only provides the latter attribute. In both cases, the attributes are intended to write imperative code that updates the user and content model.

Our current implementation only provides setAttribute and getAttribute generic methods — with a number of variants that ultimately map String names to Object instances — as a minimal implementation of the B conceptual element for single- and multi-valued attributes.

In addition, both aspects provide two methods (setUserFilter and setContentFilter), that can be redefined, and allow to specify whether the aspect is applicable to a given content element or not. In our current implementation, the expressions assigned by both set methods to the corresponding attributes must be valid SQL where clauses refering to user or content attributes. This is illustrated in the following aspect fragment that records "long" node visits:

```
aspect MyUserModellingAspect
     extends UserModellingAspect {
  void setUserFilter()
     { userFilter="age>25 and type='student'";}
  void setContentFilter()
     { contentFilter="name like lesson*";}
  before(Servlet s) : any-node(s) {
    if (!user.getAttribute("last-visit")
                   .equals(content.getName())))
    if (System.currentTimeMillis() -
      user.getIntAttribute("t-visit"))>
                                Const.TIME)
      user.setAttribute("read",content.
 getStringAttribute("last-visit"),true);
    }
  after (Servlet s) returning : any-node(s) {
    user.setAttribute("t-visit",
        System.currentTimeMillis());
    user.setAttribute("last-visit",
content.getName());
   }
}
```

In the example, any-node is a predefined *pointcut* denoting service requests on nodes. The aspect is only applicable to "students above 25 years old" and only affects "lessons". The before advice stores as "read" those lessons that have been visited at least once. The after advice performs a typical simple user profiling behavior.

Web designers require more transparent interfaces, that hide the complexities of AOP. As a first prototype for them, we have designed a graphical interface that allows to perform two different kinds of operations: update content or user attributes to literal string values, and increment numerical attributes. This allows, as showed in Figure 2, to easily define visit counters ("seguimiento visita") for a set of contents by specifying a new "counter" action associated to an after service event (equivalent to the concept of advice).

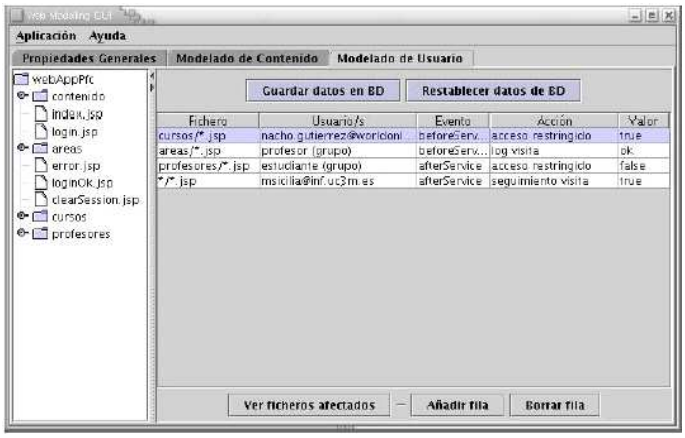


Figure 2. Prototype for the Definition of Actions

3.2. Implementing Adaptive Behaviors

The post–processing of the HTTP request in HttpServlets can be carried out by intercepting successful returning service requests, and manipulating the HttpServletResponse interface. As a proof of concept, we have designed a generic mechanism to support *link hiding* behaviors, so that certain links are removed from the response if a given predicate about the user is not satisfied.

```
aspect MyAdaptiveBehavior extends
       LinkHidingBehavior{
  void setContentFilter(){
     contentFilter="name='Welcome.jsp'";
   }
  void isUserCompatibleWithLink(){
     predicate = "novice=true"
  }
  void setLinkName(String n){
```

```
      linkSpec="name="+n;
    }
}
```

The LinkHidingBehavior aspect (see also figure 1) allows to determine whether the aspect is applicable to given content elements by means of the setContentFilter method (similar to the methods described above), and carries out the removal of the link specified in the setLinkName method depending on the result of the isUserCompatibleWithLink method. Link names are provided by means of the id HTML attribute, but the design could easily be extended to hide links identified by other means (e.g. location, destination).

4. Conclusions and Future Research Directions

User modeling and adaptive functionality can be considered as cross–cutting concerns in Web applications, spread across the server–side processing of dynamic Web pages. Current Java–enabled Web server technology can be enhanced with aspect-programming capabilities. If an explicit approach to content modeling based on JSP is used, these capabilities can be used as a mean to implement user modeling behaviors in a modularized ways; and also to attach adaptive processing to the markup stream generated by application's business logic components.

This enables the introduction of adaptive behaviors, and user and content modeling actions with no need to add JSP tags, neither code attached to Servlets implementing MVC pattern. As a result, a quite clean separation of end–user functionality, and system internal actions required to personalize the application is obtained. For that approach to be feasible, a minimal set of conceptual hypermedia design elements are needed. Those elements can be adapted from existing abstract hypermedia models like Labyrinth.

Future work should address a comprehensive implementation of Labyrinth and its extension to handle fuzziness called Maze [13] on aspect-enhanced Java technology, including well–known adaptive technologies [3]. In addition, attention should be paid to the processing burden of the aspect–oriented approach, due to the stringent requirements that many Web applications have with regards to server response times.

References

[1] Gutiérrez, I., Sicilia, M.A., García, E. (2003): On the Java Implementation of aspect-design enabled web content and user model. In: Proceedings of the 1st Workshop on "Desarrollo de Software Orientado a Aspectos" Alicante, Spain.

[2] Akşit, M., Tekinerdoğan, B. and Bergmans, L: "The Six Concerns for Separation of Concerns". In: Proc. of the ECOOP Workshop on Advanced Separation of Concerns (2001)

[3] Brusilovsky, P.: Adaptive hypermedia. *User Modeling and User Adapted Interaction*, 11 (1/2) (2001): 87–110

[4] Brusilovsky, P. and Maybury, M. T.: From adaptive hypermedia to adaptive Web. In P. Brusilovsky and M. T. Maybury (eds.), *Communications of the ACM* 45 (5), Special Issue on the Adaptive Web (2002): 31–33

[5] Díaz, P., Aedo, I. and Panetsos, F.: "Labyrinth, an abstract model for hypermedia applications. Description of its static components". *Information Systems* **22**(8), (1997) 447–464

[6] Díaz, P., Aedo, I. and Panetsos, F.: "Modeling the dynamic behavior of hypermedia applications". *IEEE Transactions on Software Engineering* **27**(6), (2001) 550-572

[7] Fink, J. and A. Kobsa (2000): A Review and Analysis of Commercial User Modeling Servers for Personalization on the World Wide Web. *User Modeling and User–Adapted Interaction* 10(3–4), 209–249

[8] Robert Hirschfeld, Matthias Wagner, Wolfgang Kellerer, Christian Prehofer (2003). AOSD for System Integration and Personalization. In Proceedings of the Workshop on Commercialization of AOSD Technology, Boston, Massachusetts, USA.

[9] Kersten, M.A. and Murphy, G.C.: "Atlas: A Case Study in Building a Web-based Learning Environment using AOP". In: Proc. ACM Conf. on Object-oriented Programming, Systems, Languages and Applications (2000) 340–352

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: "An Overview of AspectJ". In: Proc. of the European Conference on Object-Oriented Programming (ECOOP) (2001)

[11] Kulesza, U. and Silva, D.M: "Reengineering of the JAWS Web Server Design using AOP". In: Proc. of the ECOOP Workshop on Aspects and Dimensions of Concerns (2000)

[12] C. Mesquita, S. Barbosa, C. de Lucena (2002). Towards the Identification of Concerns in Personalization Mechanisms via Scenarios. In Proceedings of the 1st Aspect-Oriented Requirements Engineering and Architecture Design, 25–31

[13] Sicilia, M. A., Aedo, I., Díaz, P., García, E.: Fuzziness in adaptative hypermedia models. In: *Proc. of the Intl. Conference North American Fuzzy Information Processing Society NAFIPS 2002*, IEEE Press (2002): 268–273

[14] Sicilia, M.A., Garca, E., Daz, P., Aedo, I. (2002): LEARNING LINKS: Reusable Assets with Supports for Vagueness and Ontology-Based Typing. In *Proceedings of the International Conference on Computers in Education*, 1567-1568

[15] Sicilia, M.A. (2003). Observing web users: conjecturing and refutation on partial evidence. In *Proceedings of the 22nd North American Fuzzy Information Processing Society*, NAFIPS 2003, 530 -535.

[16] Sun Microsystems, "Java Server Pages Specification, Version 2.0", August 2002

[17] D. Stein, S. Hanenberg, R. Unland. An UML-based Aspect-Oriented Design Notation for AspectJ. Proceedings of the 1st International Conference on Aspect Oriented Software Design (AOSD) (2002), 106–112

[18] Sutton Jr., S. M. and Tarr, P.: "Aspect-Oriented Design Needs Concern Modeling". In: Proc. of the Aspect Oriented Design Workshop on Identifying, Separating and Verifying Concerns in the Design (2002), Enschede, The Netherlands.

Encapsulating Crosscutting Concerns in System Software

Christa Schwanninger, Egon Wuchner, Michael Kircher
Siemens AG
Otto-Hahn-Ring 6
81739 Munich
Germany
{christa.schwanninger,egon.wuchner,michael.kircher}@siemens.com

ABSTRACT
System software has to encapsulate crosscutting concerns properly. Aspect Orientation (AO) is a paradigm that supports modularization of crosscutting concerns. But as AO is relatively new it still lacks support suited for the industry in many domains, e.g. support for the programming languages C and C++ which are heavily used in the embedded domain exists but not yet in the desired scope and quality. To compensate for missing tools and languages we need architectural solutions for the problems around crosscutting concerns. Different system software layers, starting from simple libraries to full blown component containers can be used to provide support for concerns that cut across whole applications. Patterns can help to establish good architectures for this purpose. This position paper briefly describes how design patterns can be evaluated for their suitability to solve problems caused by crosscutting concerns.

Keywords
System Software, Patterns, Frameworks, Components, Aspect Oriented Programming.

1 INTRODUCTION

This position paper documents experiences in building run-time system software in several domains. The company, for which the authors work, typically does not build commercial off-the-shelf (COTS) system software, but it develops software for its hardware products. Those hardware products stem from several domains, including telecommunication, medical systems or automotive systems. Associated with the hardware product families, are the software product families, needed to operate the hardware. Such software product families need to be supported with frameworks or even custom made component containers, which play the role of system software for the software application developers. As system software they foster reuse and help to develop good software in short development cycles. With our experience in building platforms for product families we want to contribute to the field of system software.

During the last years, the authors saw several attempts to build frameworks for system families fail, because the architects of the frameworks were not aware of the crosscutting concerns in the system. Because the project did not capture and localize the concerns in the architecture, the project faced several problems, such as redundant implementations of the same functionality, wastage of system resources, missing resource consumption traceability, uniform error handling and communication strategies resulting in cumbersome integration.

This paper describes how crosscutting concerns can be captured and localized in system software and how patterns can help to build software that separates concerns properly.

Section 2 will explain our view on system software. Section 3 lists the software artifacts used to localize crosscutting concerns, while section 4 enumerates selected patterns for building architectures considering crosscutting concerns. The paper concludes with a brief discussion of related work and a conclusion in section 5 and 6, respectively.

2 SYSTEM SOFTWARE

According to [FODC00], system software is defined as: "Any software required to support the production or execution of application programs but which is not specific to any particular application."

System software can be aligned in to two categories:

- **Production software** – Production software includes tools that help developers in the process of designing, writing and managing software e.g. compilers, linkers, debuggers, profilers or complete IDEs, but also version control, building tools, tracers, runtime checkers and analyzers,
- **Run-time software** – Software that is needed for execution of applications at run-time or integrated in the application, e.g. OS, supporting libraries, middleware, services like persistency or event services, frameworks or even component containers, that offer their own runtime environment.

Figure 1 shows typical layers in software. The layers range from application software, to middleware, to the operating

system. Besides those layers, also the supporting compilers, configuration management software, etc. is considered as system software.

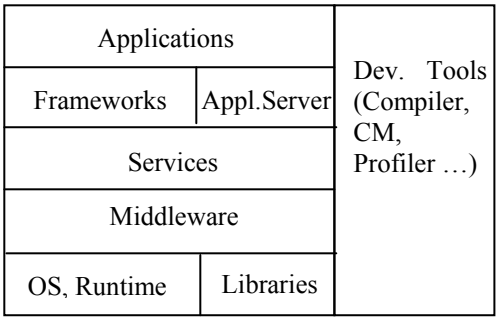


Figure 1: System Software

Software production tools, such as compilers and profilers, are "stand-alone" applications and are usually not part of any delivered system in our organization; therefore they are not of interest for us in the context of this paper.

System software that runs or is part of the application software, such as frameworks and component containers, faces different challenges than stand-alone software. It has to be built for reuse in various projects, or even domains, of which many requirements are not known up front. Additionally, the run-time system software has to be built to integrate into other software.

System software, in our context, mainly deals with resource provisioning and management (OS), communication (communication middleware), event handling (application frameworks), and GUI management (GUI frameworks).

In the next chapter we give an overview on the different kinds of run-time system software and explain how they can be used to support the localization of crosscutting concerns.

3 CAPTURING CROSSCUTING CONCERNS IN SYSTEM SOFTWARE

Depending on the layer, shown in Figure 1, and the domain it is used in, system software has to handle one or several of the following crosscutting concerns:

- Adaptability to application needs, e.g. configuration of middleware, and exchangeability.
- Optimized resource management, e.g. memory management or thread management.
- Transparent, non-invasive inter-process and network communication.
- Initialization and destruction for efficient start up and secure shutdown in resource restricted systems
- Event dispatching and handling

The listed crosscutting concerns (also referred to as aspects) are non-functional. Many domains also have additional functional aspects, for example mobile phones require messages to be passed without copying of the message data, or the sharing of personalization information across all applications in an automotive multimedia system.

Generally, AO tries to achieve the following goals via encapsulation and localization of crosscutting concerns (CCC):

- Modularity – The code for one CCC should be located in one source code file.
- Uniformity – A CCC should be treated uniformly in the whole application.
- Non-invasiveness – It should be possible to change or extend the implementation of the CCC non-invasively.
- Transparency – The CCC should be transparent to the developers.
- Reusability – Reusable software components have to be developed that can not know about the environment and the crosscutting concerns they will be reused for.

The previous two lists show the big overlap between the problems faced in system software and the promised solutions of AO.

Encapsulation of Crosscutting Concerns

Once the crosscutting concerns are identified, there are several ways how to capture them in an architecture:

The simplest way for handling crosscutting concerns is to provide an implementation in form of a *library* together with guidelines how to properly use this functionality. This is something that is usually done for simple crosscutting concerns such as tracing and logging, but also for resource management, where a library is provided that is the only access point for acquiring and releasing a specific resource.

Libraries offer a collection of functions for dealing with crosscutting concerns, *frameworks* do more. They not only provide reusable code, but also influence the architecture, for example by the inversion of the control flow. Also, frameworks often address several related functionalities, e.g. GUI frameworks implement GUI elements and the mechanisms to deal with user events. Frameworks need to be extensible, therefore they typically are built using patterns, like Strategy and Interceptor, which allow framework users to extend and customize the framework functionality.

Component containers are advanced frameworks, separating technical concerns, such as resource and

lifecycle management, from business concerns, containing the actual logic and functionality. They provide a run-time environment for components that relieves the developer from the technical concerns. Commercial component containers are often only suited for business or finance applications because they mostly cover only enterprise-specific technical concerns, but not those of typical embedded software or at least not as configurable or lightweight as required.

Aspect-oriented (AO) programming seems to be the most appropriate way of implementing crosscutting concerns in a modular way. AO brings a number of advantages. Appling AO crosscutting concerns can be modularized in exactly one place, they can be weaved in or out as needed, and their implementation and application is transparent to the developer. On the downside, AO is a rather young paradigm and there are not enough proven languages and tools on the market, yet. Except AspectJ [Kicz97] [Referenz to AJDT] no language can claim to provide industrial strength stability and tool support. AspectJ is an AO extension to Java, especially in embedded systems the dominant languages are C and C++. AspectC++ is a noble attempt to provide the same functionality for C++ as AspectJ does for Java, but the language and the tools (a plug-in for an MS IDE) are not widely used and can't be considered stable enough to implement critical features in reusable system software.

When trying to achieve the goal of reusability for a family of applications, traditional platforms define extension points where the application developer plugs in application logic in a prescribed way usually through base classes, interfaces and templates. System software defines a contract; applications use its functionality by fulfilling their part of the contract. For typical framework approaches the application has to know how to handle the system software, but not vice versa. The programming model of AspectJ like languages is different. Since the connection between the aspect and application code often requires detailed knowledge of the application code, it is a lot harder to pre-implement generic, reusable system software.

Further, how will the quality of the resulting software be ensured after introducing so many variation points? The original assets – the software that should be augmented by an aspect - are typically not designed to be extended; for example join points are defined only later, independent of the software to be extended.

So other alternatives are needed, as long AO, as the most appropriate way to modularize crosscutting concerns in system software, is not mature enough to get 'picked'.

4 PATTERNS FOR BUILDING EXTENSIBLE ARCHITECTURES

Since AO is still in its infancy, but crosscutting concerns have to be handled properly, we evaluate how patterns, as alternative concepts, can be used to build libraries, frameworks, and component containers, which fulfill the requirements like non-invasiveness, exchangeability, reusability, and modularity for crosscutting concerns. This is an 'architectural approach' to solve crosscutting concern related problems. In a first step, we study the rich pattern literature to find design and architectural patterns that help to address the above mentioned requirements.

The table on the last page shows part of our current state of evaluation of design and architectural patterns regarding their usefulness to capture crosscutting concern related problems. All selected patterns touch the area of extensibility and/or integration of concerns, which were our selection criteria.

For AO it is not relevant if the reason for encapsulating a crosscutting concern is to make the implementation easily exchangeable, to make the encapsulation transparent, or to make the encapsulated concern reusable. With AO mechanisms they are all addressed at once. Investigating design patterns shows, that they are focused on specific problems of separating the concern. But this is not really a problem, since often, only one or a small number of the above mentioned requirements have to be fulfilled at the same time. Applying one or two patterns is often sufficient to solve the specific problem related to a crosscutting concern.

For example Decorator [GOF95] helps to add functionality transparently without changing the decorated class and thus can be used to add part of a crosscutting concern implementation without polluting the original class. Additionally, an Abstract Factory [GOF95] helps to hide the decorated functionality from the client.

If the goal is reuse the crosscutting functionality the above combination of patterns can not be used, since the decorator class has to provide the same interface as the decorated class. The concern implementation therefore needs to be encapsulated, for example by a Strategy [GoF95]. Further, if the concern is resource management specific, one or several patterns of [POSA3], such as Pooling or Caching can be used directly.

Because patterns (can only) address specific forces in encapsulating and localizing crosscutting concerns, they have to be categorized accordingly. This is the intend of the attached table. The table contains the following information:

- Patlet: a short description of the pattern.
- Addressed problem: what is the main problem the pattern solves?
- Modularity: does the pattern help modularize a crosscutting concern; how does it help?

- Uniformity: does the pattern help implement a CCC uniformly throughout a system?
- Non-invasive exchangeability and extensibility: does the pattern help to exchange the crosscutting concern implementation without having to change all the places the concern crosscuts?
- Transparency: does the pattern help to keep a concern implementation and application transparent to the application developer?
- Reusability: does the pattern support the reusability of the concern code and/or of the component code that is crosscut by the concern?
- Improvability with AO (AspectJ): could AO improve the implementation of the pattern? Or does AO make the pattern obsolete?
- Possible solution in AspectJ: describes the AspectJ means we would use to implement the patter

For every pattern one to three "+" say how well it is suited to address a specific problem, a "-" indicates that the pattern is not suited at all to solve the problem. The additional text explains the rating. Several patterns fulfill one or several of these criteria, but none of them fulfills all of them the same way AO does. Also, many patterns that are useful in localizing crosscutting concerns can further benefit from an AO implementation, as can be seen in the last but one column of the table.

Let's take the Strategy pattern [GoF95] as an example. Strategy encapsulates application logic and makes it exchangeable transparently. It pretty well modularizes the logic, but still depends on the state of the entity to be extended (Modularity: ++). It is not meant to be used to solve one crosscutting concern uniformly over a whole application, rather targets one specific task (Uniformity: -) but it keeps exchanging of the contained logic perfectly transparent to clients (Non-invasiveness +++). The implementation of Strategy is not completely transparent to the client, since the client has to hold an instance and trigger the Strategy's functionality (Transparency +). A strategy requires state information and can only be reused if the required state is provided (Reusability +).

We want to continue evaluating patterns for a pattern catalogue that is dedicated to problems related to crosscutting concerns only. The next step after that will be the evaluation of successful product family frameworks to find the best practices for encapsulating crosscutting concerns in design and architecture beyond the currently documented patterns.

By recovering how to capture and localize crosscutting concern in system software by 'traditional' means, we hope to also learn more about how to use AOP for capturing and localizing crosscutting concern in system software in the future.

5 RELATED WORK

Jan Hannemann and Gregor Kiczales implemented all 23 GoF design pattern in AspectJ [Han02] and found out that modularity and reusability were improved with AspectJ [Kicz97] remarkably.

Books like Patterns for Concurrent and Distributed Objects, [POSA2], Patterns for Resource Management [POSA3], or Security Patterns [SPC02], are a few examples for pattern collections and languages that offer solutions to problems that partially stem from crosscutting concerns in specific domains.

The work of Eide et al [ER+02] analyzed patterns regarding their static and dynamic structures. As solution the authors suggest to make participants of pattern implementations easier to exchange, based on the understanding that participants in the pattern literature [GoF] can only be objects.

6 CONCLUSION

In this paper we gave a brief overview over the different layers of system software and how they localize crosscutting concerns. We argued that AO is not yet mature enough to be used in the domains of interest to us. Thus we investigated how patterns can help to build crosscutting concern aware architectures for system software. We started to evaluate design and architectural patterns that can help building frameworks and component containers that solve the problems crosscutting concerns bring up. Doing this we raise the awareness of architects and designer for crosscutting concerns. This not only supports contemporary software development, but also paves the way for AO technologies in the future.

REFERENCES

[ERR+02] E. Eide, A. Reid, J. Regehr, and J. Lepreau, Static and Dynamic Structure in Design Patterns, ICSE 2002, 2002

[FODC00] "The Free On-line Dictionary of Computing" http://www.nightflight.com/foldoc/, 2004

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns-Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[Hann02] J. Hannemann, G. Kiczales, Design Pattern Implementation in Java and AspectJ. Proceedings of OOPSLA 2002

[Kicz97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda and C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect Oriented Programming. In Proc. of European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science Vol. 1241, pp. 220-242, 1997.

[PLOPD4] N.B. Harrison, B. Foote, H. Rohnert, “The Role Object Pattern” in Pattern Languages of Program Design 4, p.14-31

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture – A System of Patterns, John Wiley and Sons, 1996

[POSA2] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture – Patterns for Concurrent and Distributed Objects, John Wiley and Sons, 2000

[POSA3] M. Kircher and P. Jain, Pattern-Oriented Software Architecture – Patterns for Resource Management, John Wiley and Sons, 2004

[SPC02] Security Patterns Community: Security Patterns Homepage. http://www.securitypatterns.de, 2004

Pattern Name	Patlet	Addressed problem, domain	Modularity	Uniformity	Non-invasive exchangeability, extensibility	Transparency	Reusability	Comparison with AO (AspectJ)	Possible solution in AspectJ
Decorator	Attach additional responsibilities to an object dynamically. This provides a flexible alternative to subcalssing for extending functionality	extending existing functionality	+;each decorator encapsulates one concern for a single class, it is not suited to encapsulate concerns that span several classes	-; localized to one class	'+++; a decorator class can be directly exchanged within a chain of decorators, no effect on the decorated class	++ transparent for developer of original class but not at instantiation time	-, decorator has to implement the decorated class's interface	+++; with AspectJ no code changes necessary when inserting a new decorator class into a chain of decorators	comparable to before/around advice using the method arguments as pointcut context
Proxy	Provide a surrogate or placeholder for another object	transparent integration	+++, encapsulates additional functionality, but not meant to encapsulate crosscutting functionality	-, one proxy for several classes not applicable, since signature has to match	++, though proxy classes have to be instantiated instead of original class	+++	-, interfaces have to match	+++, adherence to interface not necessary, therefore reusable for several classes	all kinds of advice
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.	encapsulation of operations on tree structures	+++, Visitor encapsulates additional operations on an existing tree structure	++, only for the tree structure possible	++, extending the tree structure with a new node type requires the adaptation of the new class; new visitors can be added non-invasively	++, every class has to implement an accept method	-, visitor is specific to visited classes' functionality	+++, AO allows to implement the Visitor non-invasively even in case of extending the original class hierarchy	introduction of new method
Strategy	Make application logic exchangeable.	extension of base functionality	++, encapsulates specific code, but depends on the state of the entity to be extended.	-, just locally	+++, it is the main purpose to exchange application logic.	+, the original entity must foresee a hook.	+, if the state of the entity is represented similarly.	++, additional (specific to the extended application logic) state might get weaved in.	Introduction with new methods and all kinds of advices.
Interceptor	Allow functionality to be added transparently to a framework and trigger automatically when certain events occur.	extending functionality in call chains	+++, interceptor implementation can target several classes	+++	**+++**	+++	+++	+++ also execution join points possible	before and after advices with calls join points
Resource Lifecycle Manager	Decouples the management of the lifecycle of resources from their use by introducing a separate Resource Lifecycle Manager, whose sole responsibility is to manage and maintain the resources of an application.	encapsulated lifecycle management; domain: resource management	+, localizes the lifecycle management of one or several resources; transparently provides pooling and caching of resources; manages interdependencies transparently	++, resources are managed uniformly throughout the application	+++, allows to exchange the resource management strategies transparently; to support new types of resources, its interface might need to get extended.	+, resource users need to use the resource lifecycle manager instead of existing resource providers; changes to the strategies are transparent	++, the implementation of the resource lifecycle manager can get reused	-, aspects must have knowledge with regards to when and how resources are acquired or released; pointcuts are hard to define	replace existing acquisition and release calls with around advices
Policy Enforcement Point	Isolate policy enforcement to a discrete component of an information system; ensure that policy enforcement activities are performed in the proper sequence.	consistent enforcement of security policies; domain: security	+++; localizes policy related activities to one point	+++, guarantees uniform policy handling	+++, strategy easily exchangeable	- only transparent with additional framework support	+++	++ policy enforcement could be made transparent	introduction and before advices

Supporting Product Line Evolution with Framed Aspects

Neil Loughran[1], Awais Rashid[1], Weishan Zhang[2] and Stan Jarzabek[3]

[1]Computing Department, Lancaster University, Lancaster LA1 4YR, UK
(loughran | awais) @comp.lancs.ac.uk
[2]School of Software Engineering, Tongji University, Shanghai, China 200311
zhangws@mail.tongji.edu.cn
[3]Department of Computer Science, National University of Singapore, Singapore 117543
stan@comp.nus.edu.sg

Abstract. This paper discusses how evolution in software product lines can be supported using framed aspects: a combination of aspect-oriented programming and frame technology. Product line architectures and assets are subject to maintenance and evolution throughout their lifetime due to the emergence of new user requirements, new technologies, business rules and features. However, the evolution process can be compromised by inadequate mechanisms for expressing the required changes. It maybe possible to anticipate future evolutions and, therefore, prepare and design the architecture to accommodate this, but there will eventually come a time when a certain feature or scenario appears which could not have been foreseen in the early stages of development. We argue that frames and aspects when used in isolation cannot overcome these weaknesses effectively. However, they can be addressed by using the respective strengths of both technologies in combination. The amalgamation of framing and aspect-oriented techniques can help in the integration of new features and thus reduce the risk of architectural erosion.

1. Introduction

Software systems evolve over time as new requirements and functionality emerge. It has been estimated that up to 80%[16] of lifetime expenditure on a system will be spent on the activities of maintenance and evolution. However, software product line (SPL) evolution is a much more complex problem than traditional single system evolution due to the differing configuration requirements and possibilities for different systems within the product family. Product lines, particularly those in volatile business domains such as banking, will constantly be subject to maintenance and evolution throughout their lifetime due to the emergence of new requirements, new technologies, business rules and features. Clearly, tools and paradigms which manage this complexity, facilitate modification of the architecture or ease the introduction of new features are needed if we are to reduce the risk of architectural erosion [21]. In this paper we discuss how a combination of two such techniques namely, frame technology [1] and aspect-oriented programming (AOP) [2], can be used to improve evolution of SPLs and their assets. We argue that both techniques offer complementary support for software product line evolution and, hence, improved support can be derived by using them in combination.

The next section provides some background on evolution needs in SPLs. Section 3 introduces frames and AOP and discusses their respective strengths and weaknesses in supporting SPL evolution. Section 4 describes our approach: *framed aspects* and demonstrates its effectiveness in supporting evolution in comparison with the frame-based and AOP implementations discussed in section 3. Section 5 discusses some related work while section 6 concludes the paper.

The discussion in sections 3 and 4 is based on the development of an SPL for electronic city guide systems such as GUIDE [3]. Variation points in this SPL range from the customisation of GUI components and stylings to the capability for the system to run on devices with limited resources (such as PDAs and mobile phones). The feature used as the basis for the discussion in this paper is the implementation of a cache which stores previously visited pages. Variants in this instance are maximum size of cache, deletion strategy (i.e. delete least accessed records, oldest records, etc.), percentage of records to delete and the ability for systems to be configured to be cached or uncached.

2. Evolution Issues in Software Product Lines

Software evolution is difficult to predict and rarely uniform over time. During software development requirements can change by up to 30% [4]. Managing this volatility is difficult because the changes can have major impacts on the design of the architecture. Therefore, effective mechanisms are required which can handle requirement changes through all stages of SPL development as well as evolution of the architecture throughout its life. Traditional generative approaches parameterise components and leave hooks in the architecture for most likely evolutions. The problem with this approach is that complex changes not thought of cannot be effectively handled and often give rise to the need to reorganise existing modules. Some of these issues have been highlighted by [5] in the context of evolution of SPLs for middleware.

Traditional approaches also mainly focus on the classic categories of evolution [6] namely, *corrective* (fixing of bugs), *adaptive* (adding a new feature), *perfective* (improving performance) and *preventive* (preventing problems before they occur). While this categorisation is useful in showing the *type* of evolution to be performed, it does not demonstrate how the change affects the software architecture itself. In order to support this, it is more useful to think of *crosscutting* and *non-crosscutting* evolution.

When a proposed evolution requires changes to more than one module it is said to be crosscutting, while non-crosscutting evolutions can be localised. The need to address crosscutting evolution is crucial in SPLs as a change can affect different variants and branches. Note that an SPL can be subject to a variety of changes over its lifetime ranging from addition, retraction, restructuring and replacement of a feature to

introduction of a new product or an entirely new product line (in instances when variability becomes too large). The example in this paper focuses on evolutions pertaining to a particular feature. Introduction of new products or product lines will form the subject of a future paper.

3. Frames vs AOP

3.1 Frame Technology

Frame technology was conceived during the 1970s as a means to providing a mechanism for creating generalised components that can be easily adapted or modified to different reuse contexts. Frame technology is essentially a language independent textual pre-processor that creates software modules by using code templates and a specification from the developer. Examples of typical commands in frames are <*set*> (sets a variable), <*select*> (selects an option), <*adapt*> (refines a module with new functionality) and <*while*> (creates a loop around repeating code).

To illustrate the use of frames, consider the object-oriented (OO) implementation of the cache feature for the guide SPL. Using OO alone we implemented the cache by creating a Hashtable instance in the Editor class and then wrapped calls to a requestInfo with a check to see if records existed in the cache before proceeding with the requestInfo method call (cf. fig. 1).

```
class Editor extends JEditorPane implements HyperlinkListener
  {
  private Network network;
  private Hashtable cache = new Hashtable();
  // .. methods for adding and retrieving data to/from cache
  //.. constructor and editor initialisation

  public void hyperlinkUpdate(HyperlinkEvent e)
    {
    if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
      {
      String url = e.getURL().toString();
      Document cachedPage = (Document)getFromCache(url);
      if(cachedPage == null)
        {
        network.requestInfo(this, url);
        addToCache(url, this.getDocument);
        }
       else
        {
        // get record from cache and display it
        this.setDocument((Document)cachedPage.getContent());
        }
      }
    }
  }
```

Fig. 1. OO implementation of the Cache feature

The code shown in bold in fig. 1 is the code added by the integration of a cache into a simple editor pane. Using a frame processor such as XVCL [7] we can tag this code to ease its retraction from the codebase (cf. fig. 2).

While the framing solution helps to clearly identify the caching concern, it is not a particularly elegant solution to the problem as the class now becomes cluttered with tags which can make the code difficult to read, understand and therefore evolve.

```
class Editor extends JEditorPane implements HyperlinkListener
  {
  private Network network;
  <option cache>
  private Hashtable cache = new Hashtable();
  // .. methods for adding and retrieving data to/from cache
  </option>
  //.. constructor and editor initialisation
  public void hyperlinkUpdate(HyperlinkEvent e)
    {
    if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
      {
      String url = e.getURL().toString();
      <option cache>
      Document cachedPage = (Document)getFromCache(url);
      if(cachedPage == null)
        {
      </option>
        network.requestInfo(this, url);
        <option cache>
        addToCache(url, this.getDocument);
        }
       else
        {
        // get record from cache and display it
        this.setDocument((Document)cachedPage.getContent());
        }
        </option>
      }
    }
  }
```

Fig. 2. Using frame option tags to identify caching code

Another solution might involve making a copy of the hyperlinkEvent method and having separate frames for the two variants. While this would be a neater solution, it fragments the module and future requirements pertaining to the hyperlinkEvent method would require that the code is updated in both frames, therefore inducing unneeded duplication.

3.2 AOP

AOP mechanisms such as AspectJ [8], Hyper/J [9] and emerging frameworks such as AspectWerkz [10], JBoss AOP [17] and Nanning [18], are now gaining considerable support as a means for managing the separation of concerns and features which would traditionally lead to unmanageable code tangled across multiple classes in OO systems. Examples of concerns in OO systems that exhibit this fragmentation of context are logging, profiling and tracing. AOP languages such as AspectJ allow multiple modules to be refined statically using *introductions* or through injection of additional behaviour in the control flow at runtime via *advices*.

AOP can alleviate the problem of tangled caching code (or tags in case of frames). To illustrate this, consider the AspectJ implementation of the cache in fig. 3, which can simply be plugged into the Editor.

The key part of the aspect is the *around* advice which encapsulates the following sequence of operations:

1 Whenever the requestInfo method within the Editor class is called, grab the argument URL.
2 Search the cache for the URL.

3 If the URL doesn't exist, proceed with the call and add the content of the editor to the cache. If it does exist then simply update the editor pane with the content without proceeding with the call to requestInfo.

Note that PageContent is a data structure used to store the editor content along with other data (i.e. number of accesses) in the cache.

```
aspect CacheAspect
  {
  private Hashtable cache = new Hashtable();
  // ..code
  void around(Editor g, String url): args (g,url) &&                    } 1
      call (public void Network.requestInfo(Editor, String))
      {                                                                 } 2
      PageContent cachedPage=(PageContent) cache.get(url);
      if(cachedPage==null)
          {                                                             } 3
           proceed(g,url);
           PageContent page=new PageContent(g.getDocument());
           addToCache(url,page);
           }
      else
           {
           g.setDocument(cachedPage.getContent());
           }
      }
  // inner class for data structures
  }
```

Fig. 3. AOP implementation of the cache using AspectJ

While the AOP implementation cleanly modularises the caching code, no parameterisation support is available. Consequently, the aspect needs to be modified to vary the caching behaviour. Alternatively, an abstract aspect needs to be provided with concrete aspects specifying the specific caching variants required by a particular product. In deeper inheritance structures this can lead to inheritance anomalies [11] and also require that the developer or maintainer possesses an understanding of the operations encapsulated by the abstract aspect as is the case for hot spots exposed in such a white-box fashion [12].

3.3 Comparing Frames with AspectJ

The strengths and weaknesses of frames and aspects are summarised in table 1.

Table 1. Comparing frames and AOP

Capability	Framing	AOP
Configuration Mechanism	Very comprehensive configuration possible	Not supported natively, dependent on IDE
Separation of Concern	Only non crosscutting concerns supported	Addresses problems of crosscutting concerns and code tangling.
Templates	Allows code to be generalised to aid reuse in different contexts	Not supported
Code Generation	Allows static autogeneration of code and refactoring.	Generates code which (in the case of advice) is bound at run time.
Language Independence	Supports any textual document and therefore any language	Constrained to implementation language although this will change as AOP gains wider acceptance
Use on Legacy Systems	Limited at present	Supports evolution of legacy systems at source and byte code level
Dynamic Runtime Evolution	Not supported	Possible in JAC and JMangler. Future versions of AspectJ will have support.

We can observe that the strengths of one technique are the weaknesses of the other and vice versa. A hybrid of the two approaches can provide essentially all the combined benefits thus increasing configurability, modularity, reusability, evolvability and longevity of product line assets.

4. Framed Aspects

Our approach to framed aspects is based on using aspects to encapsulate otherwise tangled features in the SPL and use frames to provide parameterisation and reconfiguration support for the feature aspects. The approach has been realised in the form of the Lancaster Frame Processor which is a trimmed down implementation of the functionality offered by XVCL. It only takes certain frame constructs and forces the programmer to use AOP techniques for the remainder. This balance of AOP and frames reduce the template code clutter induced by frames alone and at the same time provides effective parameterisation and reconfiguration support through the ability to create meta variables and options which can be bound to a specification from the developer when the frame processor is executed.

Returning to our caching example, in the guide SPL, it should be possible for the cache to be configured to different specifications. Utilising framed aspects we have developed a cache that can be configured with the following parameters: <Scheme, MaxCacheSize, PercentToDel, ContentType> where Scheme = Access or Date or Size, MaxCacheSize = any integer, PercentToDel = any value between 1 and 100, and ContentType = Document, String, etc.

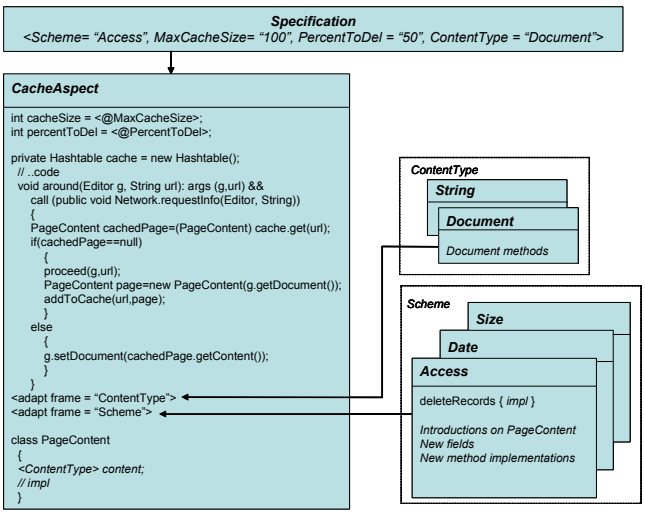


Fig. 4. Using parameterised <adapt> to provide variations in the cache aspect

The choice of different scheme strategies has an impact on the data structure within the cache as well as the deletion method. We can capture this within an aspect very easily by using the introduction mechanism where new fields and methods are inserted on defined objects. We could then use inheritance to inherit these properties when we need them. However, a much cleaner approach is to frame these properties and use a parameterised adapt to incorporate them into our aspect (cf. fig. 4). To make the aspect more reusable across different platforms (i.e. J2SE and J2ME) we could generalise parts of the cache

aspect so that they can store information without being constrained to the J2SE Document. The use of a framed aspect for the cache has effectively created a reusable and simpler to manage component, which would have been difficult to realise in AOP or frames alone without inducing some degree of complexity. We believe that the same technique can be applied to ease the introduction of other features into product lines.

There are numerous ways of utilising the framed aspect approach. In the previous example the aspect code was affected directly with frame tags, however we have found an alternative approach for use in more complex scenarios where there is a need for more control of how different modules (alternative and optional features) can be merged together in terms of constraints and rules for configuration (cf. fig. 5).

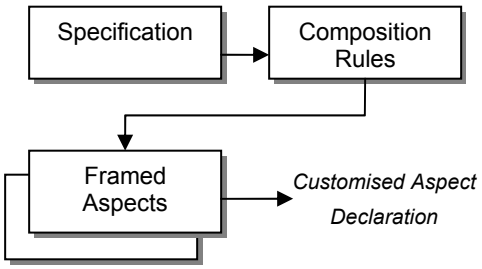


Fig 5. Alternative approach to using framed aspects

We have found that this approach offers a very powerful mechanism for removing even more of the invasive frame code (mainly due to the moving of option and adapt tags from the framed aspect code to the composition rules) and have developed a methodology which allows a feature diagram using FODA [19] for a given reusable aspect component to be created and mapped directly to framed aspects. A future paper [20] will demonstrate this approach in more detail.

5. Related Work

The framed aspect approach displays many similarities with feature oriented programming (FOP, Genvoca et al) [13], where modules are created as a series of layered refinements, SALLY [14], where introductions can be parameterised and Aspectual Collaborations [15] where modular programming and AOP techniques are combined. In FOP, composition is performed by layers stacked upon one another, with upper layers adding refinements to the lower ones via parameterisation, however, the technique is limited at present to static crosscutting feature refinements. With regards to SALLY, only its special style of introductions can be parameterised whereas in framed aspects any AOP construct can be in any AOP language. Aspectual Components have a similarity to framed aspects as they allow for external composition and black box reuse. Emerging AOP frameworks such as AspectWerkz, JBoss AOP and Nanning Apsects allow for aspects to be created as standard classes and configured via XML files which contain advice and other AO details. The main difference with framed aspects over the aforementioned is in the language independence of frames and the flexibility of parameterisation where any programming construct can be a parameter.

6. Conclusion

In this paper we have shown how aspects can benefit from the parameterisation and generalisation support that frame technology brings. We have demonstrated how the integration of new features into a product line can be simplified and believe the same technique can be applied to different concerns. We believe that our approach offers an effective approach to achieve the best of what both technologies have to offer in terms of flexibility, reusability and evolvability. Product line engineering benefits from the configurational power that framed aspects bring and helps to improve the integration of features that would normally crosscut multiple modules in OO and traditional framing technologies. Utilising AO and Frames allows crosscutting concerns to be localised thus improving system comprehensibility and minimising design erosion of architectures.

References

[1] Bassett, P.: Framing Software Reuse - Lessons from the Real World, Prentice Hall (1997).

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J.: Aspect Oriented Programming, Proc. ECOOP '97.

[3] Davies, N et al.: Lancaster GUIDE Project homepage http://www.guide.lancs.ac.uk/

[4] Cusumano, M.A. and Selby R.W.: Microsoft Secrets, Simon & Schuster, New York (1998).

[5] Colyer, A., Blair, G., Rashid, A.: Managing Complexity in Middleware. Workshop on Aspect Components and Patterns, AOSD 2003.

[6] Lientz, B., Swanson, E., and Tompkins, G.: Characteristics of Application Software Maintenance, CACM 21(6) June 1978.

[7] XVCL homepage, http://fxvcl.sourceforge.net

[8] AspectJ Team, "AspectJ Project", http://www.eclipse.org/aspectj/, 2003.

[9] IBM Research, Hyperspaces, http://www.research.ibm.com/hyperspace/

[10] AspectWerkz homepage, http://aspectwerkz.codehaus.org/

[11] Mikhajlov, L. and Sekerinski, E.: A Study of The Fragile Base Class Problem. Proc. ECOOP '98, Lecture Notes in Computer Science, 1445, (Springer-Verlag 1998), pp. 355-382.

[12] Fayad, M. E. and Schmidt, D. C.: Object-Oriented Application Frameworks. CACM 40(10), pp. 32-38, (1997).

[13] Batory, D., Sarvela, J. N., Rauschmayer, A.: Scaling Step-Wise Refinement. ICSE 2003.

[14] Hanenberg, S. and Unland, R.: Parametric Introductions. Proc. of AOSD 2003, pp. 80-89.

[15] Lieberherr, K., Lorenz, D. H., Ovlinger, J.: Aspectual Collaborations: Combining Modules and Aspects. The Computer Journal, 46(5) 2003.

[16] Lehman M. M., Ramil J. F. and Kahen, G. A Paradigm for the Behavioural Modelling of Software Processes using System Dynamics. Technical report Imperial College London 2001.

[17] JBoss AOP homepage, http://www.jboss.org/developers/projects/jboss/aop.jsp

[18] Nanning Aspects homepage, http://nanning.codehaus.org

[19] Kang, K. C. et al. Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21,Software Engineering Institute, Carnegie Mellon University, 1990.

[20] Loughran, N., Rashid, A. Framed Aspects: Supporting Configurability and Variability for AOP, *submitted to ICSR 2004.*

[21] Van Gurp J. and Bosch J., Design Erosion: Problems and Causes, Journal of Systems & Software, vol 61, issue 2, 2002.

Transaction Management in EJBs: Better Separation of Concerns With AOP

Johan Fabry*
Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussel, Belgium
Johan.Fabry@vub.ac.be

March 8, 2004

1 Introduction

The long-term goal of our research is to enhance transaction management in multi-tier distributed systems through the use of higher-level, semantical information and advanced transaction models. As a part of this research, we have performed an evaluation of transaction management in Enterprise JavaBeans, and found it lacking in multiple respects.

Most relevant for this workshop is the bad separation of concerns, as we will show below. In the worst cases, code handling the concern of transaction management can be split in three distinct locations. We feel it would be better to centralize this code into one location, by using a well-defined aspect. Our current work is the implementation of such an aspect, which we introduce here.

This paper first details how Enterprise JavaBeans aims for container-based separation of the transaction concern, and discusses how this concern remains split. Second, we describe our proposed transaction management aspect for EJBs, which aims to integrate this split concern into one. Third, we conclude and project our future work.

2 Container-Based Tx Management in EJBs

Enterprise JavaBeans (EJB) [8] is a well-known and widely used Java component architecture for middleware applications, which, among other services, provides for Container-based Transaction management. The EJB Object, which defines business code, lets its' Container manage transactions by declaring transactional properties for each method in a separate file: the deployment descriptor.

2.1 Declarative Transaction Management

In EJB, transaction management is provided by the container, and this behavior is usually determined by the transaction attributes set in the EJB's deployment

*Author funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) in the context of the CoDAMoS project.

descriptor. This is called *declarative transaction management* [5] and is an instance of Container-based separation of concerns, since the Container will perform transaction management.

In the deployment descriptor, the Bean declaratively states its transaction requirements for every business method. For example, `Required` states that the method must be executed within the scope of a transaction, and if needed, a new transaction will start, while `Mandatory` states that the method invocation will fail if not executed within a transaction scope.

Because all calls to an EJB Object are mediated by the Container, this Container can now transparently start and end transactions. If a transaction is created upon execution of a method, the Container will commit or rollback this transaction when the method ends. The decision to rollback the transaction is primarily based on exceptions thrown. If the method (or a nested method called by this method) throws a *system exception*: a `RuntimeException`, a `RemoteException`, or a subclass of these exceptions, the transaction will be rolled back, and the method will throw a `TransactionRolledbackException`. Also, a transactional method can mark itself for rollback by calling the `setRollbackOnly()` method on the Container, and determine its rollback flag by calling the `getRollbackOnly()` method on the Container. When a transactional method that is marked for rollback ends, the transaction will be rolled back but no exception will be thrown.

Declarative transaction management is said to achieve a greater separation of concerns [5] and is said to allows the transactional behavior of the EJB to be modified without needing to change the implementation of the business logic [8] . Therefore, the same Bean should be able to be reused over different applications with different transactional requirements.

However, while declarative transaction management is a promising evolution in transaction management, this new concept, as implemented in EJB, has a number of significant drawbacks. One of these drawbacks is especially relevant here: the incorrect separation of concerns.

2.2 Separation of Concerns?

As said above, declarative transaction management is said to achieve a greater separation of concerns, resulting in cleaner business method code. However, if we further investigate how declarative transaction management is currently implemented in EJB, we see that this is not really the case.

Consider commits and rollbacks: the decision to commit or rollback a transaction is made primarily based on exceptions thrown during method execution. If a system exception is thrown, the transaction will be rolled back when the method ends, if not, the transaction will be committed. Also, the method may call the `getRollbackOnly()` and `setRollbackOnly()` methods on the Container to obtain and set the rollback flag.

However, to call the above container methods, or to manually throw a system exception breaks separation of concerns. Since the method now contains code whose concern is to handle a section of the transaction, transaction management is not cleanly separated out. In other words: to cleanly use declarative transaction management, the method may never get or set its' rollback flag. Manipulating this flag not only taints the method with the transaction management concern, but also splits this concern in two disjunct parts.

Furthermore, consider what should be done in case of a rollback. Conceptually, handling of the rollbacks of transactions is a part of the concern of transaction management. Therefore, to have a clean separation, such error-handling code should also be defined when stating the methods' transaction requirements. Sadly, in EJB this is not the case. When a transaction is rolled back due to a system exception, the method will throw a `TransactionRolledbackException` to the caller of the method, and when a transaction is rolled back due to the use of `setRollbackOnly()` the caller will not be informed of this in any way. So, when using the `setRollbackOnly()`, the method needs to additionally signal this to the caller by either returning an 'error' value or throwing an exception.

The above implies that handling a rollback can only be done from within the method callers' code. Conceptually, this means not only that the caller is now tainted with the error-handling part of transaction management, but also that transaction management is split up in three disjunct sections: the transaction declaration, manipulation of the rollback flag and a-posteriori handling of rollbacks. Furthermore, since the callers to the transactional method need to specify the error-handling code, this may lead to code duplication if there are multiple callers to the method.

In other words, declarative transaction management, as currently implemented in EJB, only provides a clean separation of concerns in the most trivial cases. This is when a transaction never rollbacks: no transactional system errors may occur, deadlocks may not be broken through a rollback, and the application itself may not decide to perform a rollback. In all non-trivial cases, declarative transaction management, as currently implemented in EJB, provides a worse separation of concerns than traditional transaction demarcation. This is because the transaction concern is forcefully split up in three distinct parts, in different sections of the application, whereas in traditional transaction demarcation these three parts are contained within one location: the implementation of the transactional method.

3 Toward a Comprehensive Aspect for Tx Management of EJBs

We feel it would be better to separate out transaction management of EJBs into one complete section, instead of three disjunct sections. To do this, we propose defining transaction management in one, comprehensive aspect.

There is a body of existing work on defining transaction management as an AspectJ [1] aspect, either stand-alone by Kienzle and Gerraoui [4] or as a by-product of specifying a persistence aspect by Soares et al.[7] and also by Rashid and Chitchyan [6]. However, each approach not only is unrelated to EJBs but also falls short of our proposed comprehensive aspect. Briefly put, the work of Kienzle and Gerraoui [4] is inadequate with regard to exception-handling, as stated by the authors themselves. The work of Soares et al. [7] has been deemed as too application-specific [6], and the work of Rashid and Chitchyan [6] omits the handling of rollbacks.

We have started work on defining a more comprehensive aspect, using the technique of *logic meta-programming* to write our custom aspect weaver [9]. The use of LMP for AOSD is not new: LMP has, among others, been used to define

domain-specific aspects [2], and to argue for more expressive crosscut languages [3].

An integral part of our transaction management solution is a custom-built transaction monitor. Woven code uses traditional transaction demarcation calls to our transaction monitor, instead of using the Container's transaction monitor. This is because, in future work, we want to provide extended transaction capabilities which are not available using the standard transaction monitor, as defined in the EJB standard.

3.1 Declaring transactional methods

Our weaver uses logic programming to reason about the method code, and can easily detect some of the methods' properties, which helps in crosscut definition. For example: if a method M calls getters and setters of an entity bean, it makes sense to make M transactional. Furthermore, if M only calls getters, we can mark the transaction as read-only[1]. Note that, at this time, we do not perform any recursive analysis: methods called by M are not investigated for getters and setters.

This automatic detection of transactional methods is the default behavior of our weaver: all classes within a given package are investigated, and transaction demarcation code is inserted for all methods that should be transactional. This demarcation code also includes exception handling, equal to the standard EJB behavior, i.e. the transaction is rolled-back in case of a system exception. The woven code now behaves as if all transactional methods were declared as `RequiresNew` in the EJB's deployment descriptor.

The weaver can also be used in a more conventional manner, by specifying which methods should be made transactional in a separate `transactions` aspect file. In such a file, for a given bean, the method signatures are listed and postfixed either with `new` or `none`, indicating whether a new transaction should be started or the method is not transactional. For the parameter list of method signature, the `*` wildcard may be used, indicating applicability regardless of parameter types. Also, default behavior for a bean can be set to be either `new`, `none`, or `detect`, this last signifying automatic transaction detection. An example transactions aspect is below:

```
transactions CounterBean
{
   increment(int count) new;
   get(*) none;
   default detect;
}
```

Note that we can also use the weaver to automatically generate these aspect declarations, effectively explicitizing the information implicit in the code.

3.2 Adding Exception Handlers for Rollbacks

However, as we have said above, we want tot go further than this; our transaction aspect also centralizes exception handling for transaction-related excep-

[1] This information can be used by the transaction monitor at run-time to optimize throughput

tions. Indeed, we can define exception handlers for methods, by simply appending a number of `catch` blocks, containing java code, to the transactional declaration of the method. Within this code, the transaction can be rolled back by simply calling a `txRollback()` method. If the transactional method throws an exception that is not caught by these handlers, or the handlers do not call the `txRollback()` method, the transaction will commit when the method ends.

Lastly, we add an extra feature which is not available in EJB transaction management: restarting a transaction from an exception handler. When restarting, the transaction rolls back and the method is restarted (by re-calling the method), which implies the creation of a new transaction. Transaction restart is indicated by using a `catch` block with as body the `restart` keyword.

An example aspect definition containing these kinds of exception handlers is given below:

```
transactions CounterBean
{
   increment(int count) new catch (RuntimeException ex)
         {txRollback(); ex.printStackTrace(); System.exit(1)}
      catch (RemoteException ex) restart;
   get(*) none;
   default detect catch(Exception ex)
         { txRollback(); ex.printStackTrace(); System.exit(1)};
}
```

This effectively centralizes transaction declaration and handling of rollbacks due to exceptions in one location, and avoids unnecessary code duplication (for exception handling) in callers of the Beans' methods. One last element that is missing in this centralized transaction aspect, is the use of `setRollbackOnly()` to set the rollback flag. This occurs when, somewhere within the execution of the method, some heuristic determines that the transaction should be rolled back. At this time, we do not yet support this use of heuristics, we consider it as future work.

While it may seem unnecessary to have rollback handlers in the aspect when not being able to declare heuristics that will trigger a rollback, this is not the case. Significant causes for a rollback can already be found in this setup: transactional system errors may occur and deadlocks may be broken though a rollback. For these cases, our system is arguably better than the EJB implementation: instead of transaction declaration and rollback handling separated in two places, these are now integrated into one.

4 Conclusion and Future Work

This paper started with a discussion of Container-Based Transaction Management, as currently implemented by enterprise JavaBeans. Bean methods declare their transaction properties in the deployment descriptor, and the Bean Container automatically starts and ends transactions if required. Transactions are rolled back if the method ends in a system exception, or if the method set the rollback flag of the transactions.

Sadly, handling of exceptions and rollbacks produces a bad separation of concerns: code concerned with transaction management is now not localized

in one place but in three places. First we have declaration of the transaction properties in the deployment descriptor, second we have determining of rollbacks in the bean itself and third handling of rollbacks in the beans' callers.

After discussing transaction management in EJB's, we proposed a better solution with regard to separation of concerns. We have shown our current work, which can detect the need for transactional methods, and integrates transaction handling and the handling of rollbacks in one aspect. Heuristically determining a rollback within the method code is not yet supported. However, the current incarnation already has its merits because it is useful for handling exceptions due to e.g. network outages, forced rollbacks due to deadlocks, and so on.

Also, as a result of explicitly treating rollback handling, we considered default strategies for handling a rollback, and have already implemented a transaction restart. This leads us into future work: we are further exploring handling of rollbacks in the context of advanced transaction models, such as the use of compensating transactions.

Lastly, as future work, we will investigate how we can integrate the 'missing' concern part: rollback heuristics, into the transaction aspect.

5 Acknowledgments

Thanks to Thomas Cleenewerck and Jessie Dedecker for proof-reading and Theo DHondt for supporting this research.

References

[1] The AspectJ project. http://eclipse.org/aspectj.

[2] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, pages 110–127. Springer Verlag, 2002.

[3] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003.

[4] J. Kienzle and R. Guerraoui. Aop: Does it make sense? - the case of concurrency and failures. In *Proceedings of ECOOP 2002*. Springer Verlag.

[5] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, third edition, 2001.

[6] A. Rashid and R. Chitchyan. Persistence as an aspect. In *2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003.

[7] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA 02*. ACM.

[8] Sun Microsystems. Enterprise JavaBeans specification. http://java.sun.com/products/ejb/docs.html.

[9] R. Wuyts. *A Logic Meta-Programming Approach to Support Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, January 2001.

Software Plans for Separation of Concerns

David Coppit
Department of Computer Science
McGlothlin-Street Hall
The College of William and Mary
Williamsburg, VA 23185 USA
david@coppit.org

Benjamin Cox
Department of Computer Science
McGlothlin-Street Hall
The College of William and Mary
Williamsburg, VA 23185 USA
btcoxx@cs.wm.edu

ABSTRACT
Complex software often has concerns which cut across the modules of the system. Aspect-oriented programming languages such as AspectJ attempt to address this problem by providing a new abstraction for encapsulating such concerns called *aspects*. Aspects are integrated automatically during compilation with the base code at well-defined join points. This approach is difficult to apply when concerns are highly context-dependent and have complex relationships not supported by the language. In this paper, we propose an alternative approach based on *software plans*. In this approach, a specialized editor is first used to annotate code segments as belonging to one or more concerns. The user can then specify a limited view of the code, a *plan*, which consists of some desired subset of the concerns. Using this plan view, the user can directly implement any complex relationship between overlapping, interdependent concerns. We present our approach using a motivating example from the GNU *grep* tool. We also present our prototype editor implementation.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Coding Tools and Techniques–*program editors.*

General Terms
design, languages

Keywords
software plans, separation of concerns, aspects

1. INTRODUCTION
Complex software often has multiple overlapping and interdependent concerns. The traditional approach is to attempt to aggregate related concerns using a functional or object-oriented decomposition of the code. More recently, language designers have provided more powerful language abstractions for representing concerns as cross-cutting *aspects* [6,7]. In all of these approaches, source code is re-modularized in an attempt to improve the cohesion of code serving certain concerns while minimizing the coupling between the modules.

Unfortunately, these approaches are difficult to apply to overlapping and interdependent concerns. In such cases, modularizing a system to improve the coupling and cohesion of one concern may increase the tangling of other concerns. For example, debugging code is often scattered throughout the software. Attempting to restructure the system to improve the cohesion of the "debugging" concern would aversely affect the functional or object-oriented decomposition.

Unfortunately, aspect-oriented programming languages only partially address this problem. An inherent assumption of aspect-oriented programming languages is that it is possible to provide general declarative mechanisms for specifying the location of cross-cutting concerns, and that these mechanisms can be used by the *aspect weaver* to automatically integrate the concern code into the existing modular structure. AspectJ [7], for example, allows the programmer to specify *join points* at calls to methods and constructors, references or uses of fields, executions of exception handlers, object initialization, etc. Each of these join points requires only limited context, and are suitable for automatic integration by the weaver. None allow a concern to be integrated into an arbitrary program location. Indeed, it is not clear that this can be done automatically for concern code which depends heavily on its context in the base code. For example, trace messages used during debugging to record the flow of execution of a program depend heavily on context, and can not be integrated automatically by a weaver.

Carver and Griswold's [2] analysis of concerns in GNU `sort` demonstrates that such complex interdependencies between concerns and base code, which they call "invasive compositions", do arise in practice. Proper integration of such concerns by the weaver would require code modifications to be coordinated at multiple locations, and for concerns to be composed in the correct order. They note that one approach to dealing with such difficulties using existing join point models is to decompose complex expressions into a series of more "atomic" expressions, and extend the model to allow any series of statements to be a join point.

Murphy et. al [10] describe the process of restructuring code so that aspects can be encapsulated using the limited join point mechanisms provided by languages such as HyperJ and AspectJ. For example, they describe a somewhat byzantine approach to dealing with concern-specific code in `if-then-else` branches. For AspectJ, an "around" advice is used to bypass the original implementation of a method, instead executing "concern-optimized" versions of code which contain the appropriate branch of the code depending on the concern. They also describe a similar approach in which concern-specific code is moved to the beginning or end of a method, where the aspect weaver can integrate aspects.

In this paper we propose an alternative approach which avoids the difficulties associated with an automatic weaver, while still allowing concerns to be conceptually separated. The code is

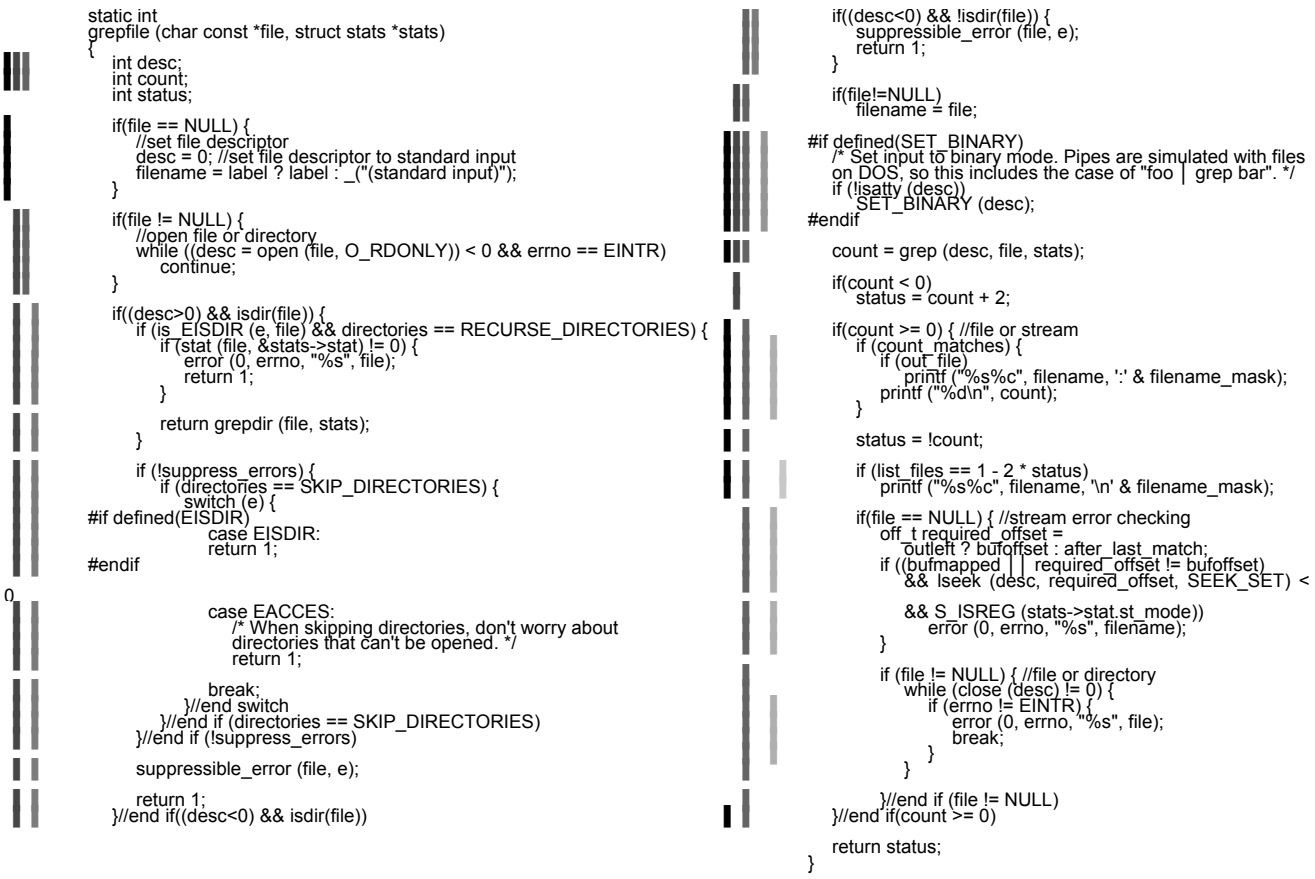


Figure 1: The grepfile function tagged with concerns

treated by the source editor as multiple inter-related layers or *plans*. A plan is a view of the software that contains only the code segments related to those concerns of immediate interest. The developer can edit the code in this view, in which case the editor automatically updates the concern information (e.g. tagging new code as belonging to the same set of concerns as the edited code). Because a particular code segment may be tagged as belonging to multiple concerns, it may also be visible in a different plan. When the source code is finally compiled, the editor renders the tagged code as a traditional monolithic code representation.

Currently, we have finished enhancing an integrated development environment to support editing of plans. Our next step is to test the approach in one or more case studies. Eventually we hope to enhance the editor to provide better automated support for tagging and editing of code related to particular concerns.

In Section 2 we present our approach in more detail, with a motivating example. Section 3 describes the implementation of plans in the Eclipse IDE. Section 4 describes our planned evaluation. Section 5 presents related work. Section 6 describes key challenges, and Section 7 concludes.

2. APPROACH

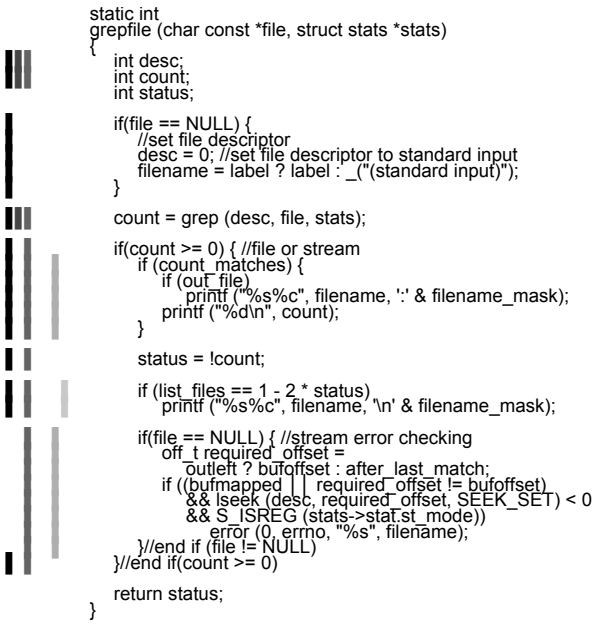
In this section we present our approach in more detail. We will use the GNU `grep` [4] program as a running example, showing how even a simple program can have complex relationships between concerns.

Figure 1 presents the key function in `grep` for searching a file, directory, or input stream for a given pattern. [1] In this example, we have used a line of code as the smallest code segment that can be related to a concern. The bars to the left of the lines indicate the concerns that are related to the line. The bars are colored, and bars of the same color are aligned in the same column. In this case, we have tagged the code with seven concerns:

- Processing of input streams
- Processing of a directory
- Processing of a file
- Error handling
- Binary files
- The `-c` option to output the number of matches
- The `-l` option to output the matching filenames

For example, the first and last few lines are not tagged, indicating that they appear in all plans. The first conditional block is tagged as belonging to the "Processing of input streams" concern, and the next conditional block is tagged as belonging to both the "Processing of a directory" and "Processing of a file" concerns.

[1] The code has been modified slightly to improve clarity.



```
static int
grepfile (char const *file, struct stats *stats)
{
    int desc;
    int count;
    int status;

    if(file == NULL) {
        //set file descriptor
        desc = 0; //set file descriptor to standard input
        filename = label ? label : _("(standard input)");
    }

    count = grep (desc, file, stats);

    if(count >= 0) { //file or stream
        if (count_matches) {
            if (out_file)
                printf ("%s%c", filename, ':' & filename_mask);
            printf ("%d\n", count);
        }

        status = !count;

        if (list_files == 1 - 2 * status)
            printf ("%s%c", filename, '\n' & filename_mask);

        if(file == NULL) { //stream error checking
            off_t required_offset =
                outleft ? bufoffset : after_last_match;
            if ((bufmapped || required_offset != bufoffset)
                && lseek (desc, required_offset, SEEK_SET) < 0
                && S_ISREG (stats->stat.st_mode))
                    error (0, errno, "%s", filename);
        }//end if (file != NULL)
    }//end if(count >= 0)

    return status;
}
```

Figure 2: The stream-only plan for the grepfile function

Note that even in this simple function there are many crosscutting concerns that make the code difficult to understand. For example, the binary filesystem concern is completely independent of the error handling concern. In this case, we could create a plan in which either concern is viewed and edited without the other.

There are also concerns that are dependent on other concerns. For example, the error handling concern is dependent on the directory, file and stream concerns. Viewing the error handling concern code which deals with directories without also viewing the directory concern would result in meaningless code. There is also an implicit ordering dependency between the "Binary files" concern and the file, directory, and stream processing concerns– the file descriptor must be set to binary mode before calling the `grep()` function.

The editor automatically tags new lines of code as belonging to the concerns of the edited text. For example, if the programmer is editing a block of code related to the "binary files" concern, the editor will automatically tag new code as belonging to that concern. While this approach suffices for the majority of editing operations, it is not a complete solution. For less common editing of concern code, the developer can manually tag a code segment as belonging to a concern. In using our prototype implementation, we have identified several situations where program analysis by the editor can provide automated assistance to further reduce the need for manual tagging. We discuss this issue in more detail in Section 6.

Once the code is tagged, the developer can specify a plan consisting of one or more concerns. Plans allow the developer to deliberately ignore concerns which are not apropos to the current activity. For example, consider the plan shown in Figure 2, a view of the system that contains the stream concern but not the file, directory, or error-checking concerns. The code is more than half as short and is easier to understand. In addition, the plan provides a coherent, even compilable, view of the code.

Plans are easy to use and allow the programmer to focus on different aspects of interest. The programmer can use plans to manage complex overlapping concerns, and can easily resolve interactions between two concerns by creating a new plan that shows both. Tags also serve as documentation, helping a developer unfamiliar with the code to easily and quickly determine the concerns associated with a given line of code, as well as interactions between concerns.

3. PROTOTYPE IMPLEMENTATION

Figure 3 shows a screenshot of our prototype implementation. In this view, the code for the `grep` utility is currently being edited. In the left are the colors associated with the various concerns. The programmer has selected some text to be tagged, and one can see the names of the available concerns in the cascaded context menu. As the programmer modifies the code, the IDE will automatically update the concern meta-data.

In our current implementation, the smallest code segment that the editor allows to be tagged is a single line. Currently the source code is stored internally as a single monolithic representation (even though, in general, lines of code for unrelated concerns can have any ordering). When the file is saved, the monolithic representation is saved as the file, and the concern information is saved separately. This provides backwards-compatibility with tools that expect a traditional monolithic format. Currently the tool does not perform any analysis for automatic tagging of code.

In order to implement this functionality, we customized the open source Eclipse IDE [2]. Eclipse provides an API for the IDE which allows developers to extend its functionality. For example, we mark ranges of text for a particular concern using the `Position` class. Similarly, our annotations are implemented using the `Annotation` and `AnnotationRulerColumn` classes. We have also modified the Eclipse IDE to allow the user to specify a plan as a set of visible concerns. Our current policy allows the user to force concerns to be hidden, or to optionally hide concerns. Code related to the latter type of concern will be visible if it is also tagged with some other visible concern.

4. EVALUATION

In order to evaluate our approach we will conduct several case studies in which our editor is used to develop several software systems. While developing the software we will investigate the theoretical as well as practical strengths and weaknesses of our approach:

- Are concerns conceptually separable? It may be the case that there is a poor correspondence between concerns and code.
- Is an editor-based application sufficient to easily separate the concerns? A primarily syntax-based tool may not be powerful enough to allow the user to easily separate concerns.
- Does this approach lower the conceptual complexity? Is it easier to write and understand code with tangled concerns? Is it easier to maintain code using this method?
- Is it possible to effectively filter irrelevant concerns while preserving all the necessary details in a coherent manner? We believe that our proposed approach to filtering lines

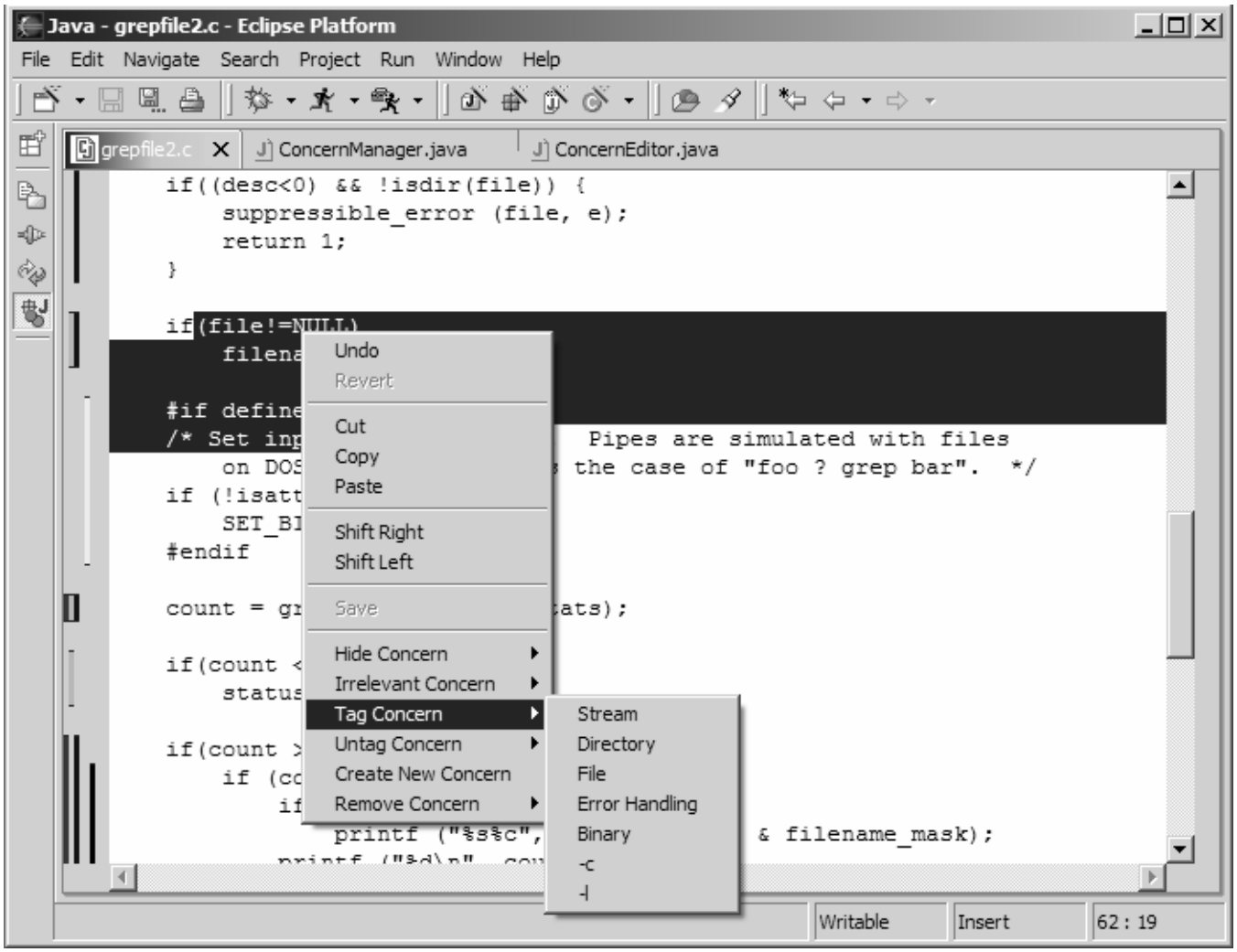


Figure 3: The grepfile function tagged with concerns

will yield coherent plans. However, it may be the case that this approach, more often than not, results in plans that are not understandable.

- What programming languages work well using this approach? Because of the line-oriented nature of this approach, procedural languages seem most suited. However, object-oriented languages may also work well.

5. RELATED WORK

Aspect-oriented programming (AOP) [6] uses "aspects" to encapsulate the concerns. The aspects are then "woven" into the code automatically by the compiler. The original formulation of AOP required custom compiler support for weaving different types of aspects. More recent efforts in the development of AspectJ [7] have attempted to provide a general method for writing aspects and weaving them into the base object-oriented code. Our approach is editor-oriented rather than language- or compiler-oriented, and can therefore be used with a range of languages. In addition, our approach allows (and requires) the programmer to express the complex relationships between overlapping and interdependent concerns. In contrast, languages such as AspectJ limit the integration of aspects and base code to only those program locations (join points), which are supported by the language. In particular, the language does not allow arbitrary aspect code to be inserted into arbitrary locations in the main code. For example, the two lines in Figure 1 which implement the `-l` functionality (near the middle of the right column) are dependent on the context. They are dependent on the execution of the grep function call, as well as the previous line assigning setting the value of `status`. The former is supported by AspectJ's "after returning" advice, but AspectJ's "set()" pointcut designator does not provide enough context to allow the `-l` code will be integrated after the assignment.

Lai and Murphy [8] analyze the relationship between concerns and code structure. Their FEAT tool allows the user to tag lines of code in a manner very similar to ours. However, their tool does not support the notion of software plans—all code related to all concerns is always visible. However, their tool does parse the code to create an abstract syntax representation, which allows them to analyze the relationship of a set of concerns to the existing code structure. In particular, they measure the proportion of files which contain code related to a concern ("spread"),

the proportion of tokens for a concern which are also involved with another concern ("tangle"), and the proportion of tokens in files for a concern which involve that concern ("density").

Program slicing [11] attempts to reduce the complexity of code by extracting only those lines of code that can alter, or are altered by, a particular variable. The extracted subset is a working program that is similar to our "plans". Unlike their automated approach, our approach is manual but more flexible in that any set of lines can be associated with a concern. Also, it is not always the case that a program variable correlates to a single concern. A variable may have multiple uses in different concerns in a program; conversely, a particular concern may require the use of multiple variables.

Information transparency [5] attempts to identify related sections of code that are dispersed throughout the source code, by using inference and searching tools. The basic idea is to identify concerns lexically, based on characteristics such as variable names, or syntactically, based on characteristics such as loop structure. Unlike information transparency, in our approach the tool helps the programmer explicitly define which sections of code are related, and does not involve after-the-fact deduction. More effort is involved to tag lines of code, but our approach can provide coherent views of the code, while information transparency presents disconnected but related lines of code.

Finally, some editors support hiding of `#ifdef`/`#endif` text based on user-specified values for the relevant symbols. Emacs [3], for example, has a hide-ifdef-mode [9]. The basic idea is similar to what we propose, although editor support is limited. In fact, our early experiments to assess the feasibility of a line-based tagging strategy used the C pre-processor in this manner. However, using pre-processor directives is obviously tedious and results in overly difficult to read code.

6. OPEN QUESTIONS

Initial use of our tool has already revealed a number of key open questions. The first question is the extent to which the management and tagging of code with concern information can be automated. Aspect languages relieve the developer of the burden of integrating aspects into base code. Our approach, in contrast, allows the user to integrate highly context-dependent concerns into the base code, but provides editor-based concern management capabilities instead of automated integration. The costs associated with manual integration of concerns are no worse than that of code developed without aspects in mind. However, tagging of concerns is an additional cost, and should therefore be as inexpensive as possible. We are not yet sure of the extent to which our current editing operations help the user to tag code. One method to enhance automatic tagging is to employ program analysis to infer that lines of code belong to the same concern. For example, the use of a variable defined to be in another concern would indicate that the code using the variable belongs to that concern.

The second issue is the view consistency problem. Editing operations in a given plan should modify the hidden code in a consistent manner. For example, there are a number of ways to handle the situation in which the user deletes a block of code containing hidden text belonging to a concern not in the current plan. Our tool's current strategy is to detect this situation and disallow the operation. In effect, this forces the user to make the hidden text visible and resolve the conflict. An alternative is to use an internal representation of the code which better models concern dependencies—if the hidden concern is independent of the current plan, the visible code can be deleted while leaving the hidden concern. Obviously, a complete solution requires program analysis to guarantee that the deletion of the visible code does not change the semantics of the hidden code.

A third open question is the extent to which code can truly be simplified in the manner illustrated in Figure 2. It seems that some rewriting of the visible code in a plan is necessary in order to arrive at a concise, easy-to-understand representation. We took some liberty in Figure 1 by splitting an `if-then-else` statement into the first two `if-then` statements. This allowed us to tag the entire `if-then` statements as belonging to one concern or the other. In the original representation, we would have been forced to tag the contents of the branches and not the `if-then` statements themselves in order to avoid `else` clauses without associated `if` statements. A side effect of this strategy is empty "`{}`" blocks in certain plans. Clearly some sort of "pretty printing" of the code is necessary to remove such noise, as well as careful management of editing operations.

Finally, we must expand our own evaluation of the approach outlined in Section 4 to include evaluation via user studies. Addressing the issues described above can help reduce the costs associated with using this approach. However, it should be possible to evaluate the basic idea using the prototype we have already implemented.

7. CONCLUSION

In this paper we have presented a new, editor-based approach to dealing with tangled concerns. Inspired by the use of plans in other engineering disciplines, our approach attempts to provide the developer with the capability to create complex relationships between concerns, while, at the same time, providing mechanisms for keeping them manageable.

While our approach shows some promise, evaluation is an obvious area of future work. In addition, there is an opportunity to exploit information from analysis of the source code in order to automate much of the manual labor required by our initial prototype. In addition, the filtering can be made "smarter" to address anomalies such as empty "{}" brackets resulting from hiding the body of the block.

ACKNOWLEDGEMENTS

We would like to thank the Eclipse developers, especially Tom Eicher, for their technical assistance in modifying Eclipse to support editing of plans. We would also like to thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] Lee Carver and William G. Griswold. Sorting out concerns. Position paper for Multi-Dimensional Separation of Concerns Workshop, OOPSLA 1999.

[2] Eclipse.org, The Eclipse homepage. URL: http://www.-eclipse.org/

[3] The GNU Project, The Emacs homepage. URL: http://-www.gnu.org/software/emacs/emacs.html

[4] The GNU Project, The grep homepage. URL: http://www.-gnu.org/software/grep/grep.html

[5] W. G. Griswold. Coping with Crosscutting Software Changes Using Information Transparency. In *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, September 2001.

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97: Proceedings of the European Conference on Object-Oriented Programming*, pages 220-42. Springer-Verlag, 9-13 June 1997.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001: Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327-353. Springer-Verlag, Budapest, Hungary, June 2001.

[8] Albert Lai and Gail C. Murphy. The Structure of Features in Java Code: An Exploratory Investigation. Position paper for Multi-Dimensional Separation of Concerns Workshop, OOPSLA 1999.

[9] Brian Marick and Daniel LaLiberte. hide-ifdef-mode.el. URL: http://www.mit.edu/afs/athena/contrib/epoch/lisp/hideif.el

[10] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating Features in Source Code: An Exploratory Study. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 275-85, Toronto, Canada, 12-19 May 2001. IEEE.

[11] Mark Weiser. Program slicing. IEEE Transactions on Software Engineering, SE-10(4):352-7, 1984.

Towards the development of Ambient Intelligence Environments using Aspect-Oriented techniques*

L. Fuentes, D. Jiménez, M. Pinto
Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga
Boulevard Louis Pasteur, 35 29071 Málaga (SPAIN)
Email: {lff, priego, pinto}@lcc.uma.es

Abstract

Nowadays the interest in Ambient Intelligence Environments has grown considerably due to new challenges posed by the evolution of society requirements to more friendly environments. Ambient Intelligence technology is not fully developed and integrated in everyday life, but a lot of organisations are interested in it. On the other hand, Aspect Oriented Software Development is considered a growing technology that improves the modularity and adaptability of complex large-scale systems. Then, our goal is the development of an Ambient Intelligence Aspect-Oriented platform adapting DAOP, a component and aspect platform.

1. INTRODUCTION

Today computational devices are being used in hundreds of human activities, ranging from office work, industrial systems and domotic systems. Hardware and computational technologies have evolved to satisfy the services demanded from these activities, like phone communications or games. This evolution has been in parallel with the development of new kinds of devices. These new devices like the last generation of mobiles and PDAs, pose high CPU and storage capacity requirements demanded to perform the new services. Examining carefully the current state-of-art we can see that the future of all these new technologies is the convergence to the Ambient Intelligence concept or AmI.

The origins of the AmI are first found in 1991, when Mark Weiser wrote an article about Ubiquitous Computing [1]. For Weiser, the ubiquitous computing term is the opposed to virtual reality. Where virtual reality puts people inside a computer-generated world, the ubiquitous computing forces the computer to live in the real world. The Ubiquitous Computing is then referred as the capacity of integrating autonomous computational devices in the real world. The devices are able to extract data from the real world that surrounds them and perform some data processing in order to obtain results that can be used afterwards by others devices. The main characteristic of the ubiquitous computing is that minimises the computer device intrusion in real world. Ideally, the human beings in an AmI environment will not notice the devices.

As an example of this type of interactions we can consider the communication between a humidity sensor device and a garden irrigation device. The former device will inform the later one about humidity variations, and the irrigation device will take actions to preserve the environment humidity according to the user preferences. Another example that we will examine more carefully in this paper is the application of AmI technologies to a car equipped with several devices that communicate among themselves and provide some useful services like user identification, local traffic information services, road obstacle detection sensors or car diagnostic services.

Weiser shows the need of finding new ways to obtain a better integration of the information technology in everyday life activities. He postulates that this integration must include the people social behaviour and the technology accessibility as main concepts. The proposed concepts are very general, but the main idea from his work is that devices will need to be adapted to people and that this will only be possible by developing much more natural interfaces between human beings and machines (voice, hand gesture, etc.). But in spite of the efforts the project was not successful, mainly because in that date, the available technology does not meet the requirements necessary to support the proposed ideas.

In the last few years, Weiser's ideas were retaken when the new technologies like mobile and wireless networks started to evolve. In consequence, the AmI concept has been adopted after several meetings of the European ISTAG [2] (*Information Societies Technology Advisory Group*) and encompasses a broader vision of the ubiquitous computing idea proposed by Weiser. The meetings started in 2001 and the goals were to promote and extend the use of ubiquitous computing technologies in the 6th European Community Programme for Research and Technological Development.

The initial meeting ended with the creation of several documents for the IPTS (*Institute for Prospective Technology Studies*). The documents can be found at http://www.cordis.lu/ist/istag.htm. The ISTAG said that the examples proposed in those documents would be technologically viable for year 2010. In the documents, they have also pointing out the critical development areas of technology and the main fields of application (genomic, biotechnology, information society technology, nano-technology, nano-science, aeronautic and space, food production security, sustainable development and economic and politics sciences).

* This research was funded by the CICYT under grant TIC2002-04309-C02-02.

In addition to the advances in hardware and computational technologies AmI should take advantage of the new software development technologies that have emerged in the last years. The importance of applying advanced software technology have been made manifest after seeing the difficulties of developing concrete AmI oriented projects like Aura [12] or Oxygen [13]. In this sense, we think that AmI applications are good candidates to be modelled using Aspect-oriented Software Development techniques or AOSD [3]. Since AmI environments are dynamic and are characterized by the runtime changes of interactions among users and devices, presenting strong requirements of dynamic adaptability, they are good candidates to benefit from aspect separation techniques.

In AOSD, aspects are defined as properties of an application that cut across some or all the application objects or components [11]. The AOSD tries to identify, isolate and extract these properties from the application core functionality, modelling these properties as aspects. Aspects can evolve independently from component functionality, so applications become more modular and, in addition, we can reuse those aspects in others similar applications. Moreover, we can replace the number and type of aspects that are applied to an application without modifying the application code. We think that identifying and separating aspects we reduce the complexity of the evolution management of AmI applications both at design and at runtime. In this paper we will try to identify the most relevant aspects that are found when decomposing an AmI application in software components and how DAOP, a component and aspect platform can help us in the application development process.

After this introduction we will show in section 2 the most relevant aspects we found in AmI applications. In section 3, we will show our proposal towards the definition of a platform for AmI applications and finally, we will expose our conclusions and future work.

2. ASPECTS IN AMBIENT INTELLIGENCE ENVIRONMENTS

Before starting to explain which aspects we have identified in AmI environments, we will provide a brief summary of the characteristics that any AmI application must show and the problems that arise when we develop these applications. The three main characteristics are:

Ubiquitous Computing, that is the ability of providing computational capabilities to any device everywhere in a non-intrusive way. These devices range in size from a board to a simple tag. There are three problems that we face when developing Ubiquitous Computing applications. First, the limited amount of energy [7] that those devices have available to function. Second, the low computing power that those devices provide and finally, the limited device storage capability known as the "nomadic data" problem. See [5] and [6]. This last problem appears due to the limited storage capacity of devices that make impossible store and retrieve all the information generated by them from everywhere.

Ubiquitous Communication, that is the ability to communicate among them any kind of device. When we try to implement this characteristic, we face several different problems. The first one is the device and communication protocol heterogeneity. This heterogeneity prevents good communication interoperatibility between devices. The second problem is the dynamic nature of these environments. The applications are executed like being part of a large distributed application and the coordination between them is difficult. It requires a solid distributed network system, a homogeneous interchange information format, a communication coordination system, a dynamic aware location mechanism and a homogeneous way to achieve the heterogeneous devices interconnection. Another problem is the scalability in a distributed network system. When the system is crowded of devices, the communications channels become saturated and interferences and errors in communications start to rise. When this happens, the need of scalable adaptation strategies is a must. Finally, the last problem encountered is the communication security and privacy. The special nature of wireless communications makes them vulnerable to intrusions or data interception problems. We must assure the privacy of confidential and sensible data by encryption or others procedures.

Natural Interfaces. The main goal of Weiser was the integration of devices in a human world. We must develop new ways of human-device interaction. To achieve this, we must solve two problems. Non–intrusive hardware, the hardware devices must be easily integrated into everyday object and become "invisible" to people. So, people do not have to care about how to interact with it. Natural interfaces, the devices must provide alternative human interfaces like voice or hand gesture recognition. While non-natural ways of communication like keyboard or mouse interfaces must tend to disappear.

Ambient Intelligence relevant Aspects

After describing the development problems that AmI applications must face, our next goal is to identify the main properties common to most AmI applications. Usually, these properties are independent of the functionality of different devices and therefore, it is a good approach to model them as aspects. Now, we are going to describe the most important properties that we have found:

Access Control. Most part of AmI services define restrictions on which devices can access to them. For example, it will be a really bad idea to allow a child to open his parent door. The access control property applied to the AmI service will handle which actions can be done and which information will be available by defining a control access list for hardware or software components. This is a typical property that should be modelled as an aspect to allow the replacement of the access control mechanism without affecting the application code.

Authentication. All devices and users in the environment must have a unique identifier. The authentication service is provided by most part of non-trivial applications in AmI environments. The device or user authentication can be performed using a username and a password, a digital signature, a user voice recognition mechanism, a digital certificate or any other identification method. We can model this property as an aspect and select the adequate method to achieve the user or device authentication.

Awareness. The AmI environment is constantly changing, new services and applications appear and others disappear without warning. The awareness property will be responsible of notifying the changes in the state of devices. This property comes from the Collaborative Virtual Environments [4], but it is also applicable to AmI environments. We should select carefully the kind and amount of information transmitted because we must not flood the receiver with useless information. The awareness notifications are captured by the environment devices and the data retrieved keeps their environment perception updated. We think that this would be modelled as an aspect because detaching this code from device core functionality allows us to manage different levels of information that is sent or received and even modify this information before it reaches the target. This aspect is crucial in AmI environments due to their dynamic nature and the necessity of most devices in the environment to be aware of changes.

Coordination. Occasionally it is interesting to perform certain specials operations when several circumstances are met in the environment. For example, when an event is sent indicating that a user has opened an AmI car door, we probably need to coordinate several AmI car components to react to this event. A coordination aspect can be modelled to handle the event and send adequate messages to the involved components. It is interesting to model this as an aspect because in AmI environments this kind of interaction is very common.

Communication. AmI environments support heterogeneous devices with different communications protocols and data interchange structure. A communication aspect is useful to act as a bridge to interconnect different devices that normally cannot communicate. If we model this property as an aspect it would be possible to adapt the device communication protocol at runtime to accept communications from others devices.

Encryption. Security in AmI environments is a must, because all the communications are open and thus easy to intercept and alter. We require a sophisticated mechanism to adapt the communication trust necessities at any time. The encryption property decouples the encryption security system from devices. Modelling this property as an aspect permits the replacement of the encryption model without affecting the application. Another possible aspect use is when the device is not capable of providing this encryption mechanism due to processing or storage limitations and the aspect can redirect the task to other specialised devices in the environment.

Error handling and recovery. Applications in this moving and dynamic environment encompasses a high error rate, usually due to interferences, communication transmission errors, communication channel saturation or application unavailability. This aspect helps to achieve a better performance in this field providing specifics solutions considering the services and the devices available in the environment. For example, imagine that we are in a congress registration hall totally automated using AmI automatic registration devices. If one or several of this devices are out of service, an error handling aspect can select other working registration device from the environment and start the login process without reporting the error to the user.

Language Internationalisation. In an AmI environment it is possible that not all users understand the natural language of AmI interfaces or the voice messages of certain devices. For example, an English user probably does not understand a Japanese character display or message. The applications should be able to adapt automatically the interface language to the user preferences. This property can be easily modelled as an aspect since crosscuts several components, and it is used to adapt a component to a user profile.

Persistence and Nomadic Data or Pervasive Data [5]. Due to the fact that the information used by a device can be distributed in different locations, we must provide a mechanism to store it efficiently and the possibility of migrating this information to different devices while the user is moving. Thus, this property can be modelled as an aspect. Another use of this aspect is to remotely store information for devices that cannot locally store some information due to storage restrictions.

To finish this section, we are going to show an example of a simple AmI application modelled with aspects. Suppose that we have a car that is equipped with AmI technology devices. Each device executes one or more specialised programs. The car has a navigation computer, a device connected to a traffic station and some other utility devices like proximity sensor, speech detection devices, car engine diagnosis device, air conditioner, etc.

In the example, we are going to focus in devices that operate the car doors and windows. These devices are controlled by an AmI application modelled as a software component called "*Car Door Component*" as shown in Figure 1. In an AmI world there is no need of keys for cars. The user will be recognised by his voice when he approaches and orders the car to open the door. The authentication aspect will perform the user recognition task before executing the open door command. The authentication aspect is useful to avoid that unauthorised people open the door. If the user is correctly identified, the device will continue the order execution. After the authentication aspect has been applied, the car door device will open the door and finally the

awareness aspect will be evaluated as shown in Figure 1. This aspect will deliver an event that contains information about who has open the door. This event is broadcasted to the environment so that all the components interested in it can catch and evaluate it. For example the navigation computer component after receiving the event executes a personalised greeting or adapt the seat to the user stored preferences, or notifying a traffic station about a new car in this street.

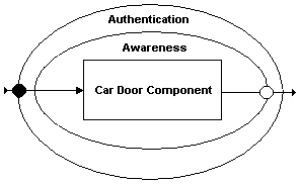


Figure 1 Aspects and Car Door Component.

We justify the use of aspects to be able to change the different authentication or awareness, methods without changing the application code. For instance, we can use digital signature or eye recognition instead of voice recognition for the authentication aspect. We can also (re) use the defined authentication and awareness aspects in others devices, like a car navigation device, a device that controls the driver seat preferences or an air conditioner device.

To finish this section we will highlight several questions. In our model, aspects can be executed in sequential or parallel mode. We can also define if an aspect will be applied before of after the component has processed a message. In the example we have executed the two aspects sequentially, authentication before the device command execution and awareness after completing the command execution. If any of the aspects fail during the evaluation, the execution process stops and an exception is raised notifying the problem. Finally the last issue is that most of the proposed aspects like authentication, access control awareness or encryption can be developed using any aspect platform or aspect oriented language (for example AspectJ [14]).

3. ASPECT-ORIENTED AMBIENT INTELLIGENCE PLATFORM

The CAM/DAOP platform has been designed by our group to support the development of component and aspect based distributed applications. We have successfully developed a Java/RMI implementation of the platform and several collaborative applications [4]. This platform defines components and aspects as the application building blocks and performs the weaving process at runtime [8]. You can find additional information about how DAOP performs the dynamic weaving between components and aspects in [8] and [9]. The language we use to specify a kind of component and aspect composition is performed using an Architecture Description Language called DAOP-ADL [10]. This language uses XML to explicitly describe the architecture of an application that can be modified by DAOP at runtime. This is a powerful feature to reconfigure systems such as AmI environments. Our actual efforts are oriented to adapt the DAOP platform to the AmI requirements.

The DAOPAmI platform

Following the Weiser's vision [1], in a typical AmI environment there are hundreds of devices. This leads us to a heterogeneous environment populated of devices with very different capabilities and requirements. The current DAOP platform kernel implementation exceeds the storage and processing capabilities of many of the AmI typical devices and the communication and security requirements of AmI applications. So, we present a new platform, named DAOPAmI that extends the DAOP platform capabilities and the DAOP-ADL language to support the specific needs of AmI applications. Now we are going to explain the modifications performed upon the DAOP platform to adapt it to the AmI applications requirements.

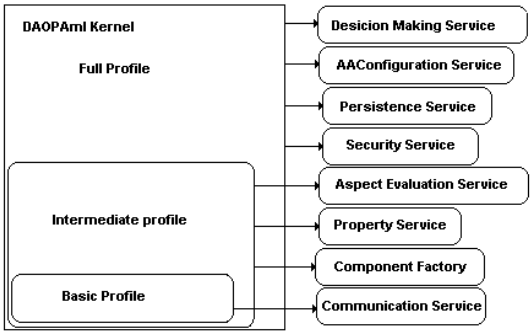


Figure 2 DAOPAmI kernel Architecture.

In [1] Weiser proposed three different devices categories according to its capabilities (taps, tabs and boards). We follow this standard division in our architecture classifying the devices in three profiles (which are similar to J2ME [15] profiles). Figure 2 shows the services that each profile supports. Now, we are going to characterize each one.

```
<component role="temperature">
....
  <delegatedComponent>
    <roleInstance>temperature1</roleInstance>
    <address >    150.214.108.46:1234</address>
  </delegatedComponent>
</component>
```

Figure 3 Component delegation definition.

Basic profile, this profile covers the simplest devices like sensors, identification cards, calendars or calculators, corresponding to Weiser's taps. As is shown in Figure 2, devices must implement at least the communication service that allows them to send asynchronous or synchronous messages to other devices. Currently, we have re-implemented this service using the J2ME communication API due to the low storage and processing capabilities of these devices. Additionally, they delegate the rest of platform functionality to other devices using a new service delegation mechanism provided by the DAOPAmI platform. Notice that devices not supporting the minimum DAOPAmI platform functionality are modelled as components inside other larger devices.

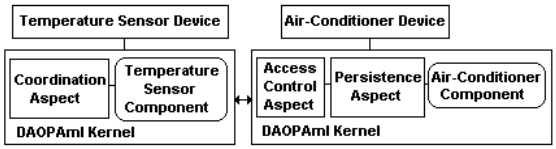


Figure 4 DAOPAmI Air-Conditioner device configuration.

DAOP uses role names to identify and address components and aspects. So we extend the component definition in DAOP-ADL with the physical address of the device modelled by this component. We also add a role instance name to identify individual components playing the same role (see Figure 3). Likewise, in the device side we specify the address of the software component that will receive the device output messages.

Intermediate profile, this profile comprises most of the typical AmI devices that have a medium computational and storage capacity like PDAs, mobile phones or Laptops corresponding to Weiser's tabs. They implement the basic profile functionality and additionally, as shown in Figure 2, the component factory service to manage software components. The property service that resolves some component and aspect data dependencies and finally the aspect evaluation service that manages the component-aspect weaving mechanism.

Full profile, this profile implements all the platform functionality including the intermediate profile and several additional services like the decision-making service, that performs automatic actions based on the environment and a set of logical rules. This service is very valuable to model flexible coordination aspects. The AAConfiguration service, that serves to configure applications and runtime. A persistence service to store temporal and persistent application data and finally a security service that guarantees the data and communications privacy. These devices have large amounts of storage and processing capabilities like advanced desktop computers and enterprise servers. This kind of devices are equivalent to Weiser`s boards.

In the current CAM/DAOP platform implementation, when an application starts, it retrieves the application architecture configuration from local storage or from an application repository. But, what happens if the device does not implement any storage facilities or cannot access directly to an application architecture repository. In this case the DAOPAmI kernel can be configured to retrieve this information from other kernel, using the communication service. After retrieving the application architecture data the application execution starts normally. But a second problem arises. What about if the application demands some services that the device cannot support due to resource or hardware restrictions? In this case, the kernel can be configured to communicate with other kernels delegating the service execution using the delegation mechanism.

Delegation mechanism example

Suppose that we have an air-conditioner device and a temperature sensor located inside a car equipped with AmI technology. The temperature sensor and the air-conditioner devices both support an intermediate kernel, as is shown in Figure 4. The first one is modelled as a temperature sensor component and a coordination aspect that is applied to the output messages of this component. The component takes periodic heat measures and emits an event containing this information. The coordination aspect evaluates this event and distribute it to others components like the air-conditioner. The air-conditioner device is modelled by a component that manages the physical device plus two aspects. The access control aspect ensures that the incoming messages come from valid device sources and the persistence aspect stores a list of received messages.

But, what happens if the sensor device cannot support this configuration due to kernel memory constrains? To solve this, the temperature sensor device delegates the services execution to other kernel. In this new scenario, the temperature sensor application architecture definition has been modified, as explained before, to delegate all kernel services, except communication service, to the kernel executed in the sensor array device shown in Figure 5. The temperature sensor components and aspect are executed within the sensor array kernel, and all events and messages sent to the temperature sensor component are redirected to the sensor array kernel to be evaluated.

An example: an AmI car

Following the previous car example, now we are going to show a more complete car door configuration using several of the previously identified aspects. As shows Figure 6, when a user approaches to the car and order "*open the door*", an authentication aspect will be executed in order to determine the user identity before the voice reaches the speech recognition component. After identifying the user voice as a valid one, the speech recognition component

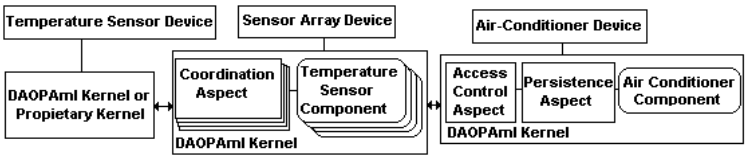


Figure 5 Temperature sensor service delegation.

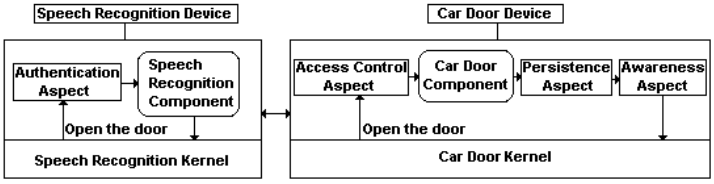


Figure 6 Extended Car Door device kernel configuration.

processes the voice signal and sends a message to the car door component notifying about the command. Afterwards, the car door kernel receives the message and the access control aspect is executed verifying that the user has the right to perform the action. After the verification, the component executes the requested command and then, the persistence aspect is applied to save an activity record. Finally the awareness aspect is executed, which broadcasts information about the performed action to the environment that can be used by other components.

This example shows how the DAOPAmI kernel provides support to AmI application development using an AOSD approach. The use of aspects let us handle the AmI applications dynamic behaviour in a natural way and easily adapt it to unexpected situations. For example we can adapt the previous example to prevent that children can open the car door from inside. To achieve this, we add a new aspect before the access control aspect that retrieves the user location and profile including for example the user age from the AmI environment. With this information the new aspect determines if the command execution can proceed or not. Thus, we have modified completely the application behaviour without changing the code of existing components; we have just added a new aspect to the application architecture definition.

4. CONCLUSIONS AND FUTURE WORK

All ideas proposed in this article are a first approach to determine the architecture requirements to develop AmI applications and define the basic services that are needed. The next step will be extending the CAM/DAOP platform to support the new kernel services and the DAOP-ADL language to express the new requirements. We think that delegation mechanism is an improvement that solves problems related to resource-constrained AmI applications in the DAOP platform and proves that our AOSD approach is feasible.

We think that AmI technology will be very important and will affect not only the way we see the software development but also the society and how people interact with computers and technology in general in the near future. We also think that the AOSD technology can help to produce more configurable and easy to manage AmI environments. AmI technology is still in its first stages and we need to do a lot of work before reaching the Weiser Ideas [1]. Our proposal tries to mix the best of both AmI and AOSD technologies by adapting an existing component-aspect dynamic platform. We think that this eases the AmI application development process and encourage the aspect reuse.

5. REFERENCES

[1] Weiser, M., "The computer for the Twenty-First Century", Scientific American 165, 1991, p. 94-104.

[2] Information Societies Technology Advisory Group. http://www.cordis.lu/ist/istag-reports.htm

[3] Aspect-Oriented Software Development http://www.aosd.net

[4] Pinto, M., Amor, M., Fuentes, L., Troya, J.M., "Collaborative Virtual Environment Development: An Aspect-Oriented Approach", Proceedings of DDMA'01, 2001.

[5] J. Kubiatowicz et al, "OceanStore: An Architecture for Global-Scale Persistent Storage". Proceedings of the ASPLOS November 2000.

[6] The Coda File System. http://www.coda.cs.cmu.edu/

[7] Goldsmith A.J, "Design Challenges For Energy-Constrained Ad Hoc Wireless Networks", IEEE Wireless Communications, August 2002.

[8] Pinto M., Fuentes L., Fayad, M.E., Troya, J.M., "Separation of Coordination in a Dynamic Aspect-Oriented Framework", Proceedings of AOSD'02, April, 2002.

[9] Pinto M., Fuentes L., Fayad, M.E., Troya, J.M., "Towards an aspect-oriented framework in the design of collaborative virtual environments", Proceedings of FTDCS'01 workshop, November, 2001.

[10] Pinto, M., Fuentes, L., Troya, J.M., "DAOP-ADL: an architecture description language for dynamic component and aspect-based development", Proceedings of GPCE 2003. pp 118-137, Erfut, Germany 2003.

[11] Kiczales, et Al., "Aspect Oriented Programming". Proceedings of ECOOP'97.

[12] Toward Distraction-Free Pervasive Computing. Project Aura. IEEE Pervasive Computing 2002. http://www-2.cs.cmu.edu/~aura/

[13] MIT Oxigen Project http://oxygen.lcs.mit.edu/

[14] AspectJ http://eclipse.org/aspectj/

[15] Java 2 Platform Micro Edition. http://java.sun.com/j2me/

Towards an Aspect Weaving BPEL engine *

Carine Courbis
University College London
Department of Computer Science
Adastral Park - Martlesham
IP5 3RE, UK
carine.courbis@bt.com

Anthony Finkelstein
University College London
Department of Computer Science
Gower Street, London
WC1E 6BT, UK
a.finkelstein@cs.ucl.ac.uk

ABSTRACT

This position paper proposes the use of dynamic aspects and the visitor design pattern to obtain a highly configurable and extensible BPEL engine. Using these two techniques, the core of this infrastructural software can be customised to meet new requirements and add features such as debugging, execution monitoring, or changing to another Web Service selection policy. Additionally, it can easily be extended to cope with customer-specific BPEL extensions. We propose the use of dynamic aspects not only on the engine itself but also on the workflow in order to tackle the problems of Web Service hot deployment and hot fixes to long running processes. In this way, composing a Web Service "on-the-fly" means weaving its choreography interface into the workflow.

Keywords

BPEL engine, dynamic aspect, visitor design pattern, Web Service, SOA.

1. INTRODUCTION

Increasingly, applications are built from existing components or services at a coarser-grain level than manipulating classes. The advantage of using Web Services in comparison to components is to enable the development of loosely coupled distributed business applications that are highly interoperable and cross organisational boundaries. This paradigm is called Service-Oriented Computing[1] (SOC). There is a fundamental shift toward a Service-Oriented Architecture (SOA) supported by the use of standards: WSDL (*Web Service Description Language*) to describe the business interfaces of the services (i.e. the contracts), UDDI (*Universal Description, Discovery and Integration*) to publish and discover them, and SOAP (*Simple Object Access Protocol*) to exchange messages between them, independently of the underlying communication protocol.

With Web Services, there is a layer of abstraction above the components that makes it possible to integrate a wide variety of incompatible systems (interoperability) to build an application. This layer of abstraction is often called the orchestration layer. The most well established orchestration technology for Web Services is BPEL (*Business Process Execution Language*) [1], originally created by BEA, IBM, and Microsoft, and currently submitted for standardisation to the OASIS consortium. This XML-based language is rather small [10] but sufficient to handle variables with scopes, loops, conditional branches, synchronous and asynchronous communications, concurrent activities with correlated messages, transactions, and exceptions. With this language, a business process can be described by gluing different Web Services together, creating a new Web Service. This process description is interpreted by a BPEL engine.

In our view, the BPEL engine should be minimal but easy to configure and extend to cope with new requirements and features. But orthogonal functionalities such as execution monitoring do not need to be enabled at each BPEL interpretation as they are themselves performance-inefficient. Selecting Web Services is another example of a possible adaptation. Instead of choosing at design or deployment time which Web Service to use, the engine can choose one at runtime, in accordance with specified criteria, on the first occasion the service is invoked.

As BPEL is an extensible language (that is new instructions can be used in a process description to cope with user-specific needs), its engine also needs to be extensible to integrate new behaviours for user-specific instructions. To build a flexible BPEL engine, we uses two techniques: Aspect-Oriented Programming (AOP) [7] and the visitor design pattern [5, 12]. By using these techniques, the engine can be extended both statically[2] by inheritance and dynamically by aspects.

We intend to apply dynamic aspects not only to the engine itself but also to the BPEL process to tackle the problems of Web Service hot deployment and hot fixes to long running processes. For example, it can be useful to add an unforeseen Web Service at runtime. We have also been investigating a specific case in which services are used to support a large Grid-based computational chemistry application. In this application, there is a need for steering, in

*This research is supported by the Generative Software Development project funded by BT Exact.

[1]The first international conference on SOC has just taken place in Italy (see http://www.unitn.it/convegni/icsoc03.htm).

[2]Dynamically, if the engine supports dynamic class loading.

The Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) March 2004, Lancaster, UK

other words changing the end of the workflow depends upon results identified in earlier stages.

Our BPEL engine will manipulate different types of aspects; it can be seen as an aspect weaver that orchestrates Web Services.

The aim of this paper is to explain how such an adaptable BPEL infrastructure engine can be created taking advantage of the visitor design pattern and dynamic aspects. It is organised as follows. Section 2 presents a brief overview of related work. Section 3 describes the design of our BPEL engine. We conclude the paper in Section 4.

2. RELATED WORK

It is important that "systems infrastructure" software such as application servers, virtual machines, middleware, compilers, and operating systems be open and adaptable. Otherwise no user-specific feature or requirement can be added after implementation time. To integrate new functionalities requires redevelopment of the whole software. For example, Gilad Bracha *et al* have developed their own Java compiler to create a superset of the Java programming language, *GJ*, with generic types and methods [4]. It was not possible to adapt the Java compiler to cope with this language extension. Other examples are the VM-based runtime MOPs (*Meta-Object Protocols*) such as *Guaraná* [11] that have developed their own JVM (*Java Virtual Machine*) to intercept operations at runtime (with a VM-based solution, the passage from the base level to the meta level is invisible to the programmer).

One solution to build more adaptable system infrastructure software is to use dynamic aspects. They are appropriate for run-time adaptations in service architectures [15] and more precisely as hot fixes. By contrast with static aspects such as the ones used in *AspectJ* [6], dynamic aspects can be woven or unwoven into/from a program "on-the-fly". Sato *et al* present [16] a good introduction to dynamic AOP. They also describe their dynamic weaver, *Wool*, that is a hybrid of two aspect implementation approaches. At runtime and on demand, it either embeds hooks into a class for executing the advices and reloads it into the JVM, or inserts hooks as breakpoints into the JVM. At least two dynamic aspect systems, *JAC* [14] and *Handi-Wrap* [3], use static code translation on the byte-code, statically inserting the hooks at all the potential join points. Using aspects on SOAs will make it possible, for example, to check constraints (design by contracts), such as the ones proposed in the Web Service Offerings Language (WSOL) [18], or to monitor the execution with agents.

Using design patterns when implementing systems make them more flexible. In the case of interpretors, the visitor design pattern is very often chosen. Recently, this pattern was used, for example, to implement *Joeq* [19], a virtual machine and compiler infrastructure.

The existing service description languages and Web Service flow languages address business process dynamics and non-functional properties poorly. For example, in the current BPEL version, it is not possible to add on demand or replace a Web Service (a partner) at runtime; the workflow needs to be stopped to be adapted. The idea of using aspects for dynamic workflow adaptations or execution controls has been outlined in [2]. With aspects, new activities can be added or replaced, the control flow modified, the policy resolution to assign resources to activities changed or extended, and resource invocations replaced.

To address the problem of dynamic selection and composition of Web Services, DAML-S [9], an ontology of services, proposes the use of semantic descriptions. These descriptions will then be manipulated by different agents or software such as a semi-automatic composer of Web Services [17]. With the latter, compositions on demand are based on semantic descriptions and are validated by human controllers. Daniel Mandell and Sheila McIlraith describe in [8] how to augment BPEL with Semantic Web technology.

3. AN OPEN, EXTENSIBLE, AND CONFIGURABLE BPEL ENGINE

To have more flexible BPEL processes, we have chosen to design and implement an open, extensible, and configurable BPEL interpreter. Its core logic will be rather small as the language does not contain many instructions, but we plan to enrich it with new features such as:

- To easily extend or modify its behaviour;
- To select or replace Web Services after deployment time;
- To plug or unplug aspects in/from the engine on demand;
- To hot-fix the workflow; for example, to compose on demand new Web Services;

The advantages of these features and how we plan to implement them are now presented. At this end of this section, we also briefly put together the architecture technical details of our language interpreter.

3.1 Engine behaviour extension or modification

BPEL is a language that can be extended with new user-specific instructions such as launching an executable, or replacing a Web Service. This means that its engine needs to be easy to extend. Also it would be useful to have the capability to modify the engine behaviour to take into consideration user-specific requirements. The visitor design pattern meets these requirements as it separates the data structures and the semantics. The behaviour of each BPEL element is represented as a visit method and the set of these methods contained in a class (the visitor). As the engine code will be modular, it will be easy to understand, maintain, extend by inheritance, and modify by visit method overridings.

3.2 Selection and replacement of Web Services

Selecting a Web Service can depend on different criteria and constraints: QoS (*Quality of Service*), price, the result of a request, the trust you have in the provider, etc. In the well-known Web Service example, the travel agency, the selection policy for the airline company can be to take the lowest fare from London to Morocco at Christmas time, or the quickest trip without a stop, or to take British Airways for the frequent flyer points. To find the lowest fare, each airline company needs to be invoked; the selection policy can be a minimal business process. Each partner (Web Service) involved in a business process can have a different selection policy. The selection may be performed at runtime on the first occasion the service is invoked or at replacement request, not at design or deployment time. We plan to accept

one selection policy per partner and a generic one if none is provided. This policy will be used at runtime by the engine to select, from an UDDI registry, a Web Service that is signature and constraint compliant.

There is also a need to be able to replace, at runtime, a Web Service that is slow, unresponsive, or no longer useful for the current iteration. In this way, the workflow can be adapted to improve performance or QoS, to avoid termination because there is no answer from one partner, and to use another similar service in a loop or on user demand. The substitution can only occur if the new Web Service is service-signature compliant (same WSDL description as there is no service adaptor) and if the service to be replaced is in a stable state (not in a transaction, and without an initialisation or one that does not impact on other partners).

To be compliant with the specifications, the core logic of our BPEL engine (the visitor) should contain no Web Service selection or replacement code. The solution is to set a hook before service invocations (the `invoke` visit method) to add these functionalities. In this way, services can be selected and even replaced at post-deployment, as well as selection policies.

3.3 Orthogonal concerns

It can be useful to enrich the core logic of the engine with different concerns at post-implementation. In this way, the engine is more modular, adaptable, and easy to maintain. As some of the concerns can be impact-performant, such as execution profiling or debugging, their corresponding aspects should be enabled to be woven or unwoven on demand during execution. With these dynamic non-functional aspects, the engine can, for example, be controlled by agents that monitor the execution and take actions if one service provider is not responding. Such a concern can be useful especially for long running processes. We can also identify the need for functional aspects between two service invocations to perform local code execution such as converting the data into another format.

We have defined an aspect BPEL-specific language using XPath as a pointcut language to identify the join points (matching the BPEL document) and Java as the advice language[3]. In our first version, we have statically set hooks to execute advices at all the potential join points; that is before and after any BPEL instruction (visit method) such as `invoke` or `receive`, and at any process variable modification. Plugging in an aspect means registering it on the current process and also selecting the different nodes of the process document (AST - *Abstract Syntax Tree*) identified by XPath expressions to annotate them with the aspect name and the name of the advice to execute. Before and after interpreting an instruction, our system checks if there is any annotation and calls the method to execute (advice) if this aspect is still registered. Unweaving an aspect only means removing the aspect from the registry.

3.4 Hot fixes applied to the workflow

For long running processes, adapting a workflow, according to earlier results, by stopping it is not acceptable. There is a need to modify, at runtime, the end of the workflow by adding new computational instructions, and replacing or deleting some instructions. This can also be seen as BPEL aspects, using XPath to identify the join points but BPEL as the advice language (instead of Java for the aspects on the engine). As these aspects act upon the workflow (functional aspects) and have their advice in BPEL, we plan to directly transform the process AST. These transformations can only be applied to the workflow at some precise points and under certain conditions that we need to identify to ensure the stability of the system. For example, deleting a BPEL sequence can only occur if the engine has not started interpreting it.

An important example of such hot-fixes is the composition, on demand, of a new Web Service and thus the addition of its choreography interface. The dynamic aspect technology is our solution to address dynamic composition of Web Services: the choreography interface (BPEL instructions) can be seen as advices and where to weave them as pointcuts. Composing a new Web Service means transforming the AST workflow to integrate the piece of its choreography interface (BPEL advice). With this capability, the workflow can be extended to meet unforeseen post-deployment requirements and user needs.

3.5 Architecture technical details

The core logic of our system (see Figure 1) is the BPEL interpretor, implemented using the visitor design pattern. It contains one visit method for each BPEL instruction and traverses the typed structures, the BPEL trees, from top to bottom. These trees are not only strictly typed to meet the pattern requirements but are also based on the DOM API to enable XPath selections of their nodes, which is useful for the implementation of our BPEL aspect languages.

The code to handle the selection and replacement of Web Services, and the engine aspects is represented as two aspects respectively that we can plug in or unplug from the BPEL interpretor. In this way, the interpretor can be used alone (faster) or extended, at runtime, with these functionalities. Its code is independent from the BPEL aspect and the Web Service selection code, and is compliant with the BPEL specification. This possibility of plugging aspects is due to our visitor design pattern implementation that checks before and after each visit method call to see if some advices need to be executed. More precisely, this check is done in the visit method dispatcher (in our case, a generic visit method instead of the different accept methods implemented in each BPEL element class). More details about the visitor design pattern implementation we are using can be found in [13].

The workflow aspect manager is also code independent from the engine. It just needs to suspend the engine when performing the transformations on the interpreted BPEL document at some stable points and to get access to its data environment to add or remove members (variables, partners, etc.). Additionally, the annotations of the engine aspects already plugged in should be propagated onto any new BPEL instruction added by insertion or replacement.

4. CONCLUSION

In this position paper, we argue that the visitor design pattern and dynamic aspects can be used to implement an extensible and adaptable BPEL engine, thus in SOAs. The benefit of using the visitor design pattern is to write modular code that is easy to extend by inheritance. This characteristic, in the context of an extensible language such as BPEL, is important so as to ease the incorporation of new instruction behaviours into the interpreter. Involving aspects into the

[3]The implementation language of our engine is Java-based.

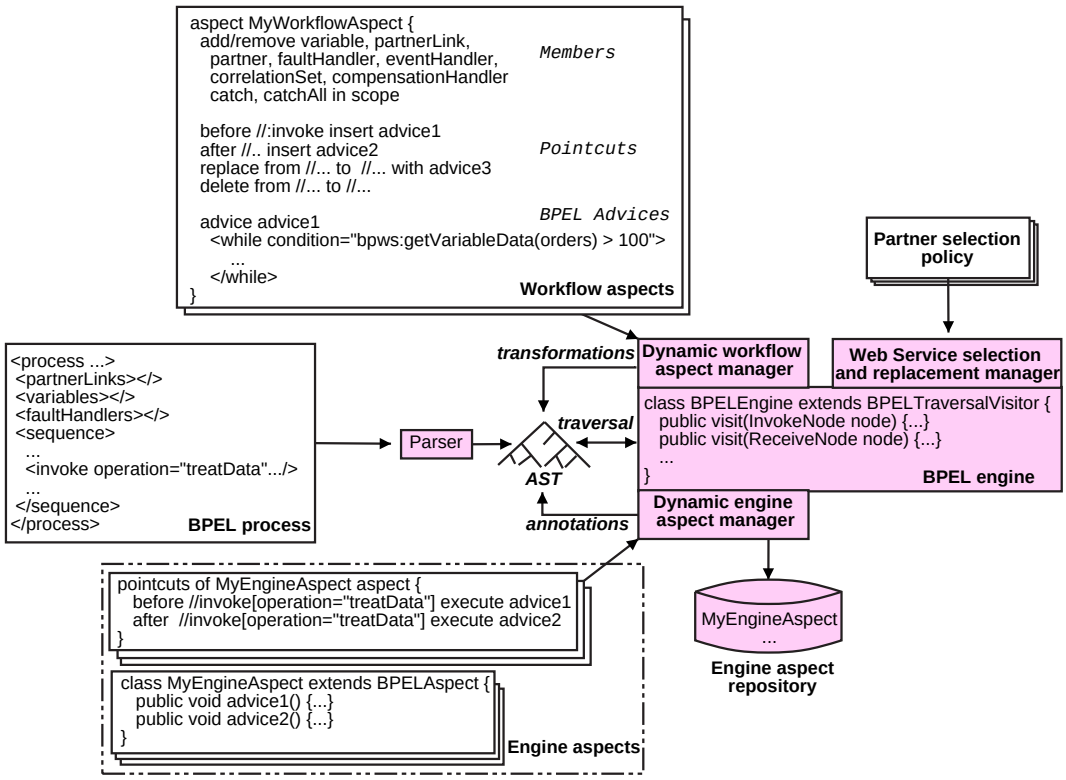


Figure 1: Overview of the BPEL engine architecture

engine makes it possible to separate, in a modular way, the different concerns, to focus only on its core logic in the first place, and to rapidly integrate unforeseen concerns into it in a non-invasive way. For greater flexibility, we have chosen to have dynamic aspects to be able to weave and unweave them into/from at runtime.

We also argue that dynamic aspect techniques can not only be used in the engine itself but also on business processes to address the well-known problems of Web Service hot deployments and hot fixes. Additionally, we believe that the BPEL engine should be customised with different selection policies as Web Service selection should be done after deployment, and with Web Service replacement capability.

We have started the development of our system, using SmartTools [13], a DSL (*Domain-Specific Language*) development environment, to quickly prototype tools for our different languages (BPEL, the engine aspect language, and the workflow aspect language). Later, we will need to refine our aspect languages by identifying which pointcuts are needed for advices either in BPEL (for the hot fixes such as the choreography interface compositions) or in Java (for orthogonal concerns).

5. ACKNOWLEDGEMENTS

The authors want to thank David Lesaint and George Papamargaritis from BT Exact for the fruitful discussions, as well as Ben Butchart from UCL.

6. REFERENCES

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, and I. Trickovic. Business Process Execution Language for Web Services version 1.1. Technical report, BEA, IBM, Microsoft, SAP, Siebel Systems, May 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[2] B. Bachmendo and R. Unland. Aspect-Based Workflow Evolution. In *Tutorial and Workshop on Aspect-Oriented Programming and Separation of Concerns*, Lancaster, UK, August 2001. http://www.comp.lancs.ac.uk/computing/users/marash/aopws2001/papers/ba

[3] J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In *First International Conference on Aspect-Oriented Software Development*, pages 86–95, Enschede, The Netherlands, April 2002. ACM. http://www.cs.utah.edu/~wilson/papers/handiwrap-aosd02.pdf.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *Proceedings of OOPSLA'98*, Vancouver, Canada, October 1998. ACM Press. http://www.cis.unisa.edu.au/~pizza/gj/Documents/gj-oopsla.pdf.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub Co, January 1995. ISBN 0201633612.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knuden, editor, *Proceedings of European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–355, Budapest, Hungary, June 2001.

http://www.cs.ubc.ca/~gregor/kiczales-ECOOP2001-AspectJ.pdf.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira, and J.-M. Loingtier. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.
http://www.cs.ubc.ca/~gregor/kiczales-ECOOP1997-AOP.pdf.

[8] D. J. Mandell and S. A. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *Proceedings of the Second International Semantic Web Conference*, number to appear in LNCS, Sanibel Island, USA, October 2003. Springer-Verlag.
http://www.ksl.stanford.edu/people/sam/iswc2003sam-djm-FINAL.pdf.

[9] T. D. S. C. D. Martin, M. Burstein, G. Denker, J. Hobbs, L. Kagal, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. DAML-S: Semantic Markup For Web Services 0.9, 2003. white paper available online at
http://www.daml.org/services/daml-s/0.9/daml-s.pdf.

[10] N. Mukhi. Reference guide for creating BPEL4WS documents. Technical report, IBM, November 2002.
http://www-106.ibm.com/developerworks/webservices/library/ws-bpws4jed/.

[11] A. Olivia and L. E. Buzato. The Design and Implementation of Guaraná. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, San Diego, USA, May 1999.
http://www.ic.unicamp.br/~oliva/guarana/docs/desimpl.ps.gz.

[12] J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998.
http://www.cs.ucla.edu/~palsberg/paper/compsac98.pdf.

[13] D. Parigot, C. Courbis, P. Degenne, A. Fau, C. Pasquier, J. Fillon, C. Held, and I. Attali. Aspect and xml-oriented semantic framework generator: Smarttools. In M. van den Brand and R. Lämmel, editors, *ETAPS'2002, LDTA workshop*, volume 65 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, Grenoble, France, April 2002. Elsevier Science.
http://www.elsevier.nl/gej-ng/31/29/23/117/52/33/65.3.009.pdf.

[14] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns: Third International Conference, Reflection'01*, volume 2192 of *LNCS*, pages 1–24, Kyoto, Japan, September 2001.
http://jac.aopsys.com/papers/reflection.ps.

[15] A. Popovici, G. Alonso, and T. Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 100–109, Boston, USA, March 2003. ACM Press.
http://www.lst.inf.ethz.ch/research/publications/publications/AOSD_2003/AOSD_2003.pdf.

[16] Y. Sato, S. Chiba, and M. Tatsubori. A Selective, Just-in-Time Aspect Weaver. In Springer-Verlag, editor, *Proceedings of Generative Programming and Component Engineering (GPCE'03)*, number 2830 in LNCS, pages 189–208, Erfurt, Germany, September 2003.
http://www.research.ibm.com/trl/people/mich/pub/200306_gpce2003.pdf.

[17] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *ICEIS2003, Web Services: Modeling, Architecture and Infrastuture workshop*, Angers, France, April 2003. ICEIS Book.
http://www.mindswap.org/papers/composition.pdf.

[18] V. Tosic, B. Pagurek, and K. Patel. WSOL - A Language for the Formal Specification of Various Constraints and Classes of Service for Web Service. In *The International Conference On Web Services, ICWS'03*, pages 375–381, Las Vegas, USA, June 2003. CSREA Press.
http://www.sce.carleton.ca/netmanage/papers/TosicEtAlResRepNov2002.pdf

[19] J. Whaley. Joeq: A Virtual Machine and Compiler Infrastructure. In *The Workshop on Interpreters, Virtual Machines, and Emulators*, pages 58–66, San Diego, USA, June 2003. ACM SIGPLAN 2003.
http://www.stanford.edu/~jwhaley/papers/ivme03.pdf.

A case study of separation of concerns in compiler construction using JastAdd II

Torbjörn Ekman
Department of Computer Science, Lund University, Sweden
torbjorn.ekman@cs.lth.se

Abstract

This paper presents a case study of separation of concerns in compiler construction using the JastAdd II compiler compiler. A domain-specific specification language, Rewritable Reference Attributed Grammars (ReRAGs), is combined with Java to implement compilers in a high-level declarative and modular fashion. Three synergistic mechanisms for separations of concerns are described: inheritance for model modularisation, aspects for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. Each technique is presented using a series of simplified examples from static semantic analysis for the Java programming language.

1 Introduction

We present a case study of separation of concerns in compiler construction using the JastAdd II compiler compiler. Simplified examples from static semantic analysis for the Java programming language [GJSB00] are used to demonstrate the mechanisms for separation of concerns provided in JastAdd II. This work is part of a larger project where the entire static semantics of Java 1.4 have been implemented. We believe that Java is a suitable language implementation for experimenting on language modularisation because of its advanced scope rules with nested types and inheritance as well as need for reclassification of contextually ambiguous names during name analysis.

JastAdd II uses a declarative compiler specification in the form of Rewritable Reference Attributed Grammars (ReRAGs) [EH04] combined with imperative Java code. ReRAGs provide three synergistic mechanisms for separations of concerns: inheritance for model modularisation, aspects for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. This allows compilers to be written in a high-level declarative and modular fashion.

The rest of this paper is structured as follows. Section 2 describes JastAdd II and its specification language. The three mechanisms for separation of concerns are demonstrated in sections 3, 4, and 5. Section 6 discusses how the three mechanisms are used to deal with interaction between aspects. Section 7 points out some related work and Section 8 concludes this paper and discusses some future work.

2 JastAdd II Background

JastAdd II is an aspect-oriented compiler compiler tool using declarative Rewritable Reference Attributed Grammars (ReRAGs) and Java as its specification languages. The grammars define attributes and equations to specify computations and information propagation in the abstract syntax tree (AST). The formalism is object-oriented viewing the grammar as a class hierarchy and the AST nodes as instances of these classes. Behavior common to a group of language constructs can be specified in a common superclass and specialized or overridden for specific constructs in the corresponding subclasses. Often the most appropriate AST structure can only be decided after parial attribution of the AST. Rewrites allow restructuring of the tree to simplify the specification of the remaining attribution. The following sections give an introduction to JastAdd II compiler specifications.

2.1 The AST class hierarchy

The nodes in an Abstract Syntax Tree (AST) are viewed as instances of Java classes arranged in a subtype hierarchy similar to the Interpreter pattern, [GHJV95]. An

AST class correponds to a nonterminal or a production (or a combination thereof) and may define a number of descendents and their declared types, corresponding to a production right-hand side. In an actual AST, each node must be *type consistent* with its ancestor according to the normal type-checking rules of Java. I.e., the node must be an instance of a class that is the same or a subtype of the corresponding type declared in the ancestor. Shorthands for lists, optionals, and lexical items are also provided. All node types implicitly inherit the common ancestor type ASTNode that support generic access to node children. This is particular useful for generic tree traversals. An example definition of some AST classes is shown below.

```
// Expr corresponds to a nonterminal
ast Expr;

// Add corresponds to an Expr production
ast Add : Expr ::= Expr leftOp, Expr rightOp;

// Id corresponds to an Expr production
// id is a token
ast Id : Expr ::= <String id>;
```

2.2 Reference Attributed Grammars

ReRAGs are based on Reference Attributed Grammars (RAGs) which is an object-oriented extension to Attribute Grammars (AGs) [Knu68]. In plain AGs each node in the AST has a number of *attributes*, each defined by an *equation*. The right-hand side of the equation is an expression over other attribute values and defines the value of the left-hand side attribute.

Attributes can be *synthesized* or *inherited*. The equation for a synthesized attribute resides in the node itself, whereas for an inherited attribute, the equation resides in an ancestor node. Note that the term *inherited attribute* refers to an attribute defined in the ancestor node, and is thus a concept unrelated to the inheritance of OO languages. In this article we will use the term *inherited attribute* in its AG meaning, unless explicitly stated otherwise.

Inherited attributes are used for propagating information downwards in the tree, e.g. propagating information about declarations down to use sites, whereas synthesized attributes can be accessed from the ancestor and used for propagating information upwards in the tree, e.g. propagating type information up from an operand to its enclosing expression.

RAGs extend AGs by allowing attributes to have reference values, i.e., they may be object references to AST nodes. AGs, in contrast, only allow attributes to have primitive or structured algebraic values. This extension allows very simple and natural specifications, e.g., connecting a use of a variable directly to its declaration, or a class directly to its superclass. Plain AGs connect only through the AST hierarchy, which is very limiting.

In the JastAdd II implementation of RAGs attributes can be seen as methods where the method declaration and method body may be separated. Inherited attributes have their method body that defines the behavior in an ancestral node. An inherited attribute equation defines the behavior for a corresponding declaration of the same attribute in the subtree where the targeted equation node is the root. That way the only dependency on tree structure for that attribute is that the node holding the equation must be an ancestor to the node holding a declaration.

Aspects can be specified that define attributes, equations, and ordinary Java methods of the AST classes. An example is the following aspect for very simple type-checking.

```
// Declaration of an inherited attribute env
// of Expr nodes
inh Env Expr.env();

// Declaration of a synthesized attribute
// type of Expr nodes and its default equation
syn Type Expr.type() = TypeSystem.UNKNOWN;

// Overriding default equation for Add nodes
eq Add.type() = TypeSystem.INT;

// Overriding default equation for Id nodes
eq Id.type() = env().lookup(id()).type();
```

The notation for method invocation is used when accessing descendent nodes like `leftOp` and `rightOp`, tokens like `id` and user-defined attributes like `env` and `type`. This API can be used freely in the right-hand sides of equations, as well as by ordinary Java code.

2.3 Rewrite rules

ReRAGs extends RAGs with rewrite rules that automatically and transparently rewrites nodes. The rewriting of a node is triggered by the first access to it. Such an access could occur either in an equation in the ancestor node, or in some imperative code traversing the AST. In either case, the access will be captured and a reference to the final rewritten tree will be the result of the access. This way, the rewriting process is transparent to any code accessing the AST.

A rewrite step is specified by a rewrite rule that defines the conditions when the rewrite is applicable, as well as the resulting tree. After the application of one rewrite rule, more rewrite rules may become applicable. This allows complex rewrites to be broken down into a series of simple small rewrite steps.

A rewrite rule for nodes of class *N* has the following general form:

```
rewrite N {
  when {cond}
  to R result;
}
```

This specifies that a node of type *N* may be replaced by another node of type *R* as specified in the result expression *result*. The rule is applicable if the (optional) boolean condition *cond* holds. Both the rewrite rule application order and the tree traversal order are implicitly defined by attribute dependences. A thorough description of ReRAGs implementation and application will appear in [EH04].

3 Inheritance for model modularisation

The subtype hierarchy generated from the grammar production rules provide excellent support for model modularisation. Generic behavior is defined in the possibly abstract node types and then specialized in the concrete node types. A small example adding a reference attribute to each expression referencing its corresponding type declaration node is shown below. The production rule hierarchy is in itself specialized in multiple steps, e.g binary operands, arithmetic expressions, and addive expressions are all succesive specializations from the generic language element expression. The type reference is defined to be boolean for all relational types while the type of arithmetic expressions is the widest type of both operands. The approach is generic in the sense that adding another arithmetic expression, e.g. subtraction, does not affect type propagation but merely requires implementation of the unique behavior, e.g. code generation.

```
ast Expr ;
ast BinOp : Expr ::= Expr left, Expr right ;

ast ArithmeticExpr : Binop ;
ast AddExpr : ArithmeticExpr ;

ast RelationalExpr : Binop ;
ast LessThanExpr : RelationalExpr ;

syn Decl Expr.type() ;
eq ArithmeticExpr.type() =
    widestType(left().type(), right().type());
eq RelationalExpr.type() = TypeSystem.BOOLEAN;
```

4 Aspects for cross-cutting concerns

The examples shown so far are actually feature aspects where attributes that cross-cut the AST subtype hierarchy are grouped into separate modules. This technique is very similar to static introduction techniques used in AspectJ [KHH+01], Hyper/J [OT01], and Multi Java [CLCM00].

The example below is a simple name binding module that binds a use-site to its declaration site through the inherited attribute `bind` taking a name as its parameter. A block of statements is modeled as a list of statements and a list of declarations for simplicity. Each block introduces a new scope to search for declarations and there are nested scopes since each statement in a block can be a block itself. The inherited attribute `bind` must thus have an equation in each scope, i.e. the `Block` node, and if a matching declaration is not found the search must be delegated to the surrounding scope.

```
ast Block : Stmt ::= Stmt stmt*, Decl decl*;
ast Name : Expr ::= <String name>;
ast Decl ::= <String name>;

protected inh Decl Name.bind(String name);
protected inh Decl Block.bind(String name);

eq Block.stmt().bind(String name) {
  for(int i = 0; i < numDecl(); i++)
    if(decl(i).name().equals(name))
      return decl(i);
  return bind(name);
}

public syn Decl Name.decl = bind(name());
```

To limit coupling between aspects such as name binding and type checking it is useful to limit visibility of certain attributes outside the defining aspect. The only attribute that needs to be exported outside a name binding aspect is for instance the binding from a use-place to its declaration, e.g. `decl` in `Name`. Attributes that define scope rules, e.g. `bind`, only affect the name binding and should thus be private to name binding modules.

Aspects have proven a very powerful technique to implement design pattern roles, [HK02], [NK01]. The same technique can be used in JastAdd II to implement reusable modules, illustrated below where the name binding approach described above is generalized in a generic module for nested scopes. The involved actors are nodes that need to lookup declarations and nodes that define new scopes. These actors are specified as interfaces and later used to tag each tree node that takes the role of an actor defined in the module. These interfaces also specify the equations that the implementors must supply to define non-generic behavior, e.g. finding declarations in its scope that matches the provided name. In the example, `Scope` represents nodes that defines a new scope and the non generic behavior is to match a name to a declaration while `Bind` represents the node that receives a reference to a declaration.

```
aspect NestedScopes {
  interface Scope {
    protected syn Decl lookup(String name);
  }

  interface Bind  {
    protected inh Decl bind(String name);
  }

  eq Scope.child().bind(String name) =
    lookup(name) != null ?
       lookup(name) : bind(name);
}
```

The module is generic in the sense that the only requirement on the AST structure is that an enclosing scope is defined by an ancestral node. It can be further generalized by adding more scope types, e.g. inheritance from super classes, and declare before use. Below is a name binding module that uses the module with the previously defined concrete node types `Block` and `Name`. The only behavior that needs to be implemented is the matching attribute `lookup` in `Block` and the use of the provided attribute `bind` in `Name`. In a Java several nodes implement a scope, e.g. block, class, interface, and for statement, and thus share common properties.

```
aspect NameBinding extends NestedScopes {
  declare parents: Block implements Scope;

  declare parents: Name implements Bind;

  eq Block.lookup(String name) {
    for(int i = 0; i < numDecl(); i++)
      if(decl(i).equals(name))
        return decl(i);
    return null;
  }

  public syn Decl Name.decl = bind(name());
}
```

5 Rewrites to create the most suitable model

Rewrites can improve separation of concerns by allowing computations to be expressed on the most suitable model. The information acquired during the early stages of static semantic analysis can be used to rewrite the model to make that information explicitly visible in the model structure for later stages.

We use an example from Java name analysis to demonstrate the technique. When parsing an expression containing qualified names, e.g. `java.lang.System.out`, it is syntactically undecidable if a part of a name is a reference to a package, type, field, or variable unless their context is taken into account. In the above example, `java` is most often a package, but only as long as there is no variable-, field-, or type-declaration named `java` that would shadow the package according to the Java scope rules. Thus, a context-free grammar can only build generic name nodes that capture all cases. The attribution will need to handle all these cases and therefore becomes complex. To avoid this complexity we would like to do *semantic specialization*, i.e. we would like to replace the general name nodes with more specialized ones. Other computations, like type checking, optimization, and code generation, can benefit from this rewrite by specifying different behavior in the specialized classes rather than having to deal with all the cases in the general name node.

An aspect that models Java names and resolves syntactically ambiguous names as described is shown below. There are two different types of names in Java from a syntactic point of view, simple names and qualified names. A simple name is a single identifier and a qualified name consists of a name, a `"."` token, and an identifier. During parsing a context-free grammar is used and thus general unbound names has to be build during AST creation. Semantic specialization is used to rewrite these general nodes into more specific ones, e.g. variable- or type-names. The ast-declarations in the aspect below model the described name structure.

Semantic specialization is implemented using a rewrite that rewrites an ambiguous `UnboundName` node into a `VariableName`-node or `TypeName`-node depend-

ing on the type of the binding received from the name binding module. Finally the `QualifiedName` nodes changes the scope rules for its right child to search the type of its left child to provide, e.g. when trying to bind out in the `System.out` expression the class `System` should be searched for a field named `out`.

```
ast Name : Expr;

ast SimpleName : Name ::= ID id;
ast QualifiedName : Name ::=
   Name left, SimpleName right;

ast UnboundName : SimpleName ;
ast VariableName : SimpleName ;
ast TypeName : SimpleName ;

// Resolve names depending on bound entity
rewrite UnboundName {
  when (bind().isVariableDecl())
  to SimpleName new VariableName(id());
  when (bind().isTypeDecl())
  to SimpleName new TypeName(id());
}

// The left name in a QualifiedName changes
// the scope for the name to the right
eq QualifiedName.right().bind(String name) {
  if(left() instanceof TypeName)
    return left().decl().lookup(name);
  if(left() instanceof VariableName)
    return left().type().lookup(name);
}
```

6 Aspect interaction

While the aspects demonstrated to far define static features we also use more pluggable aspects, e.g. a declare before use aspect to complement the name binding module in Section 4 and optional code optimization aspects. Pluggable aspects define rewrites that change a run-time node instance to a subtype node with extended behavior. That way an aspect can be added to the system in a way transparent to other aspects.

The examples demonstrated so far deal with equations that cross-cut the type hierarchy only and not cross-cutting concerns within equations. To override and extend attribute equations we use inheritance of the model structure in combination with rewrites that change the type of a node instance at run-time. I.e. we may have different/extended equations for UnboundName and VariableName defined in the example in Section 5. This technique, using run-time rewriting and inheritance, is more powerful than static compile-time point-cuts within equations in that it may take run-time information into account but less powerful in that each node may only be changed by a single aspect. Therefore it would be interesting to combine the current approach with more fine-grained static point-cuts within equations.

7 Related work

The introduction of attribute definitions and equations to an exisiting class hierarchy in a modular fashion used in JastAdd II is very similar to static introduction in AspectJ [KHH+01], hyperslices in Hyper/J [OT01], and open classes in MultiJava [CLCM00]. A functional approach to attribute grammar aspects using the same technique is presented in [dPJV00] where aspects are first-class objects that can be freely combined using a combinator library in Haskell. ReRAGs further improved modularisation support in that the current model instance may be rewritten during run-time to a more suitable model allowing each computation to be expressed on the most suitable model and more fine-grained separation of concerns within equations.

The Visitor pattern, [GHJV95], is often used in compiler construction for separation of concerns when using object-oriented languages. Visitors can only separate cross-cutting methods while static introductions can be used for fields as well. AOP implementations of the Visitor pattern need not rely on a delegation mechanism resulting in a cleaner more intuative implementation, [HK02]. ReRAGs aspects differs from AOP implementations of the Visitor pattern in that an explicit traversal strategy in the form of a Visitor is not specified but merely implicitly defined by attribute dependences. Rewrites further improves modularisation in that the underlying structure may change during run-time to better fit the current computation.

Higher order attribute grammars (HAGs) [VSK89] adds tree nodes computed reading the partially attributed AST at run-time and can thus provide a more suitable model. The process is, however, not transparent to other computations and is thus less flexible from a separation of concerns view. The use of attribute grammars and forwarding for modular language implementation is discussed in [VWMBK02]. Forwarding overrides attribute equation dynamically at run-time and forwards equation to a different part of the tree. Since it is based on HAGs the target tree can be computed at run-time and the approach is thus similar to semantic specialization.

8 Conclusions and future work

We have demonstrated three synergistic mechanisms for separations of concerns supported by ReRAGs in the JastAdd II compiler compiler: inheritance for model modularisation, aspects for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. Examples inspired by static semantic analysis of the Java programming languages have been used to illustrate and motivate each technique. We believe that this allows compilers to be written in a high-level declarative and modular fashion.

Our experiences indicate that the implementation leads to flexible solutions to several traditional compiler construction problems, and we hope to generalize some of these techniques and document them as design patterns or frameworks for compiler construction using ReRAGs.

We would also like to investigate the interaction between pluggable aspects and also how to better support fine-grained cross-cutting within equations combining AspectJ-like point-cuts with run-time rewriting implemented using ReRAGs in JastAdd II.

References

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of OOPSLA 2000*, volume 35(10), pages 130–145, 2000.

[dPJV00] Oege de Moor, Simon Peyton-Jones, and Eric Van Wyk. Aspect-oriented compilers. *Lecture Notes in Computer Science*, 1799, 2000.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, 2004. Accepted for publication.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of OOPSLA-02*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 161–173, New York, November 4–8 2002. ACM Press.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[NK01] N. Noda and T. Kishi. Implementing design patterns using advanced separation of concerns. In *OOPSLA2001 workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[OT01] Harold Ossher and Petri Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *Proceedings of the 23rd international conference on Software engineering*, pages 821–822. IEEE Computer Society, 2001.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 131–145. ACM Press, 1989.

[VWMBK02] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Grenoble, France, April 8-12, 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002.

The Proxy Inter-Type Declaration

Michael Eichberg
Software Technology Group,Dept. of Computer Science
Darmstadt University of Technology, Germany
eichberg@informatik.tu-darmstadt.de

ABSTRACT

Aspect-oriented programming [16] with its support for modularizing crosscutting concerns opens up the vision that component middleware can be replaced by sets of collaborating aspects that implement the infrastructural services. In this way, application servers could be customized on a per-project basis, as they are no longer monolithic applications.

In order for this vision to become reality it should be possible to implement services offered by current containers as aspects. In particular services targeting scalability and performance issues, such as passivation and instance pooling must be available. Such services build upon the concept of virtual instances [25]. They are realized in the current application servers by means of the (virtual) proxy pattern [11].

Implementing virtual instances represents a crosscutting concern, as we will discuss in the paper. Unfortunately, no current AOP language or framework has explicit support for modularizing this concern. We propose to support the generation of proxies as an inter-type declaration and argue that it is essential to allow the development of scalable and efficient component containers. It would be a significant improvement when compared with the current offered possibilities.

1. MOTIVATION

Aspect-oriented programming [16] with its support for modularizing crosscutting concerns opens up the vision that component middleware (e.g., application servers for Sun's EJB [9] or CORBA components [20] models) can be replaced by sets of collaborating aspects that implement the infrastructural services. Before this can be done the question arises how to achieve performance and scalability. Infrastructural services such as instance pooling and passivation of components are prominent solutions exploited by current application servers. The implementation of these services requires the use of *virtual instances* [25]: Instead of a component instance a virtual instance is exposed to the client (clients are all users of a component). To implement virtual instances the proxy design pattern [6, 11] is used [25]. One proxy object represents one virtual instance and the same proxy might refer to distinct physical component instances (one at a time) during its lifetime. All physical instances referred to by one (virtual) proxy object represent the same logical instance.

The concept of virtual instances is also used by the Enterprise JavaBeans component model [9]; the services which must be implemented by every EJB compliant container require that a component (an Enterprise JavaBean (EJB)) is not directly accessible; otherwise the services could be bypassed and the container would be compromised. Proxy objects automatically generated by the container, called `EJBObject`s, are the concrete realization of the virtual instances concept. An instance of an `EJBObject` always represents the same logical component regardless of its physical instance. It executes the explicitly requested services (in the deployment descriptor), such as security, transactions, etc., as well as implicitly provided services such as passivation, pooling, etc.

Although component passivation and pooling are services that span all components of the system in the same way and hence appear to be typical crosscutting concerns, we claim in this paper that it is not well supported by common AOP frameworks [3, 4] or AOP languages [15, 19]. Reports are indeed available about applying AspectJ to resource pooling, such as pooling JDBC connections [17,18]. However, pooling of resources is different from pooling of components in two significant ways:

1. A pooling aspect for resources is specialized to exactly one type of resource (e.g. database connections using JDBC) which is implemented by a fixed set of well-known classes. Thus, it is easy to determine the correct join points, i.e. to write the pointcut to be used by the aspect in order to integrate pooling. A pooling aspect for components, on the other hand, must be generic to handle multiple different types of components, e.g., different types of entity beans as different as `Order`, `Book`, `Customer`, etc., which are furthermore unknown at aspect development time. That is, we should be able to write the pooling functionality independently of the concrete type of the components that will be pooled. We will argue that this is not well supported by current mainstream AOP languages.
2. Pooled resources can be directly reused, as they are not client specific. On the contrary, when recycling physi-

cal components we need to keep their logical identity, so that the pooling is transparent to the client. So, if we want to reuse a pooled component, it is necessary to (re-)assign it the logical identity that is hold by the client. This forces us to integrate virtual instances in a consistent way all over the system.

To implement these services the ability to fully control all references to a component is required; a component can not be pooled as long as at least one other component has a direct reference to it that we cannot control. To implement the concept of virtual instances we can use proxy objects if and only if we simultaneously make sure that the proxy object can not be bypassed. Only under this condition is the proxy guaranteed to be the only one ever holding a reference to a component. The proxy can then be used as a virtual instance of a component.

To achieve the "non-bypassable" feature of a proxy the corresponding component needs to be transformed so that the reference to itself (`this`) is never passed to another component. Additionally, we have to check that `this` is not otherwise available e.g. via a public field. So, the following code:

```
1 class Order{
2   public void setCustomer(Customer c){ ...
3     c.addOrder(this);
4 } }
```

needs to be transformed into something similar to:

```
1 class Order{
2   public void setCustomer(Customer c){ ...
3     c.addOrder(getMyProxy());
4 } }
```

Also all component creations:

```
1     new Order();
```

needs to be transformed:

```
1     new OrderProxy(new Order());
```

This kind of transformation can be automatically performed if we can distinguish between components and normal classes. However, using standard pointcut and advice two solutions can be considered to simulate the effect of the transformation. The first "solution" would be to manually scan all component implementations and to write appropriate advice which passes the proxy instead of `this`. The second solution is to write one general advice which checks if a passed object is `this` and if so returns the proxy instead; given that the advice is generic, which means it simply advices all method calls to any component in the same way, it is necessary to check every passed parameter to every method call at runtime if the parameter is `this`. The effort to implement the first solution would be very high and also it would be tedious and error prone since an ignored statement that passes `this` to another component must not immediately result in a compile-time or runtime error. The inefficiency of the second solution would render any other optimization useless. So, the implementation of virtual instances is not well supported; the modularization of this cross-cutting concern with the current techniques offered by AOP frameworks or languages is hardly possible.

The reminder of this paper is organized as follows. In Sec. 2 we present our proposal by means of an example of a passivation service and indicate that what we are proposing is not well supported currently. In Sec. 3, we go into more technical details of the proposal. Then we discuss related work and conclude this position paper with a short summary.

2. THE DECLARE PROXY CONSTRUCT

We first give an overview of the syntax and semantics of the declare proxy construct by the example of the passivation service. Next, we go into some more details about the semantics of the proposed construct. Implementation details are out of scope for this position paper. In the following listings we use annotations and generics as they will be available in Java 1.5. However, we do not make any assumptions about their support in future AspectJ versions.

2.1 Overview of the Proposal

To support virtual instances, we propose a new construct for inter-type declarations in AspectJ: the `declare proxy` statement. The proposed syntax of the statement is as follows:

```
1 declare proxy
2   extends AProxy :
3   implements AnInterface;
```

The effect is that proxy classes are generated that extend the specified proxy class (line 2) and implement the specified interface (line 3). The interface (line 3) determines the components for which proxies are to be used: Proxies of type `AProxy` are generated for all classes that directly or indirectly implement `AnInterface`. This means the interface specified in the `implements` clause in line 3 has a selection functionality picking the set of components to be equipped with virtual instances, roughly comparable to a pointcut, and also determines the super-type of the proxies and components. The specified class (line 2) must not be final and must extend (directly or indirectly) the system type `Proxy`.

To illustrate the syntax and semantics of the `declare proxy` construct we consider its use within the implementation of a passivation service as shown in the snippet from the `Passivation` aspect below (line 2-4). Line 4 builds the bridge between the generated proxies and the components for which these proxies are to be used. On the one hand the interface determines the common super-type of the component and the proxy and on the other hand it determines for which components a `PassivationProxy` (line 2) is to be generated and used.

```
1 public aspect Passivation {
2   declare proxy
3     extends PassivationProxy :
4     implements SessionComponent;
```

To provide an accessible set of join points, the generated classes not only implement the specified interface, they also inherit a user supplied proxy class. In line 3 we additionally specify that the generated proxies have to inherit from the class `PassivationProxy`. This enables the definition of pointcut and advice in relation to a specific proxy and its subclasses and not only in relation to all classes that implement the specified interface.

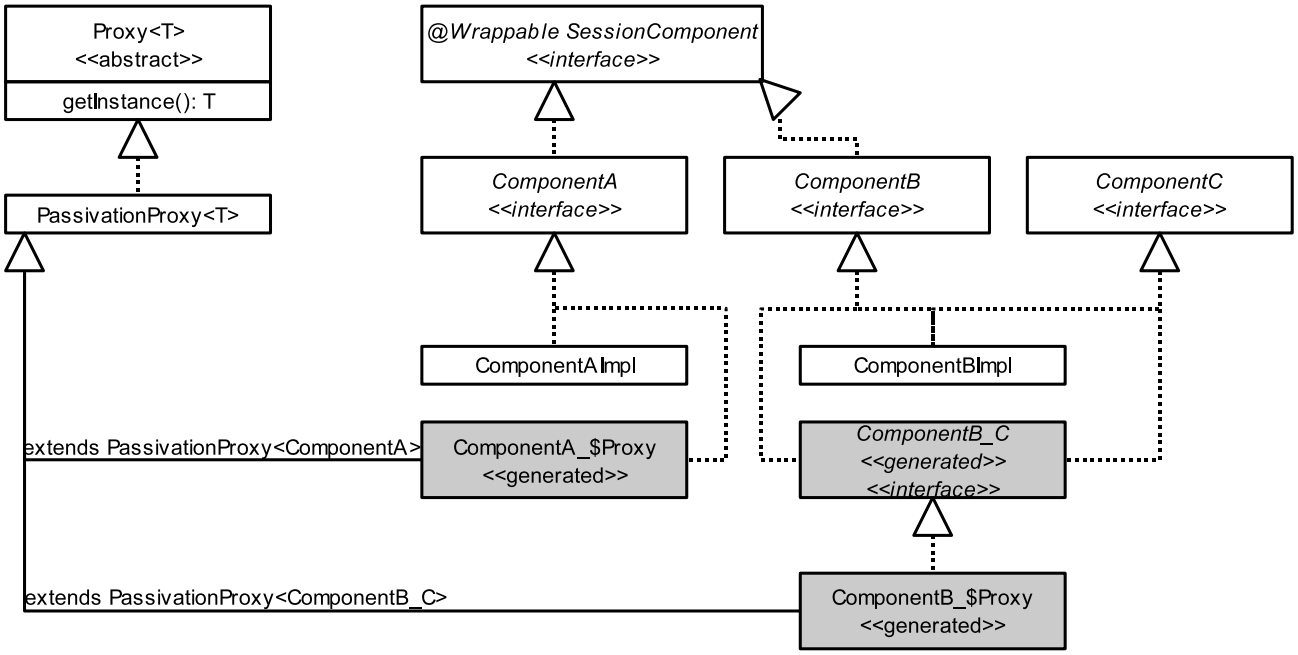


Figure 1: The effect of declare proxy: "`declare proxy extends PassivationProxy: implements SessionComponent;`"

Figure 1 shows the class diagram after the generation of the proxies, as the result of compiling the above declare proxy statement. For the component `ComponentAImpl` a proxy class (`ComponentA_$Proxy`) is generated that implements the interface of the component (`ComponentA`) and extends the specified `PassivationProxy`. Assuming that a component is always accessed via its interface, it is type safe to use a proxy instance instead of a component instance.

As seen in Figure 1, the proxy classes do not only implement the specified interface, they actually implement all interfaces of a component. The proxy class for `ComponentBImpl` must implement both interfaces (`ComponentB` and `ComponentC`) even though `ComponentC` does not extend `SessionComponent`. This is necessary to preserve the validity of the program. Code that casts between the component's interfaces is valid and must remain valid; this can only be assured if a proxy implements all interfaces. As a technical necessity, for every component which implements multiple interfaces an artificial interface has to be generated that extends all of them; this interface is then used to correctly bind the type parameter `T`. `T` determines the concrete interface type of a component (the interface `ComponentB_C` in Figure 1 is an example).

The interface (line 4) in the `implements` clause must be annotated `Wrappable` (see figure 1 for an example). For every class inheriting the `Wrappable` annotation proxy classes can be generated. The annotation with `Wrappable` enables us to distinguish between "normal" classes and components and to validate the later (see section on Implementation Restrictions). All component classes are transformed to no longer expose `this` unless no proxy is generated for the component. Further, it is assured that the program does not give rise to errors by the declaration and use of a simple forwarding proxy (calls to the proxy are directly forwarded to the corresponding method of a wrapped instance). To make this possible the interface type of a component is always to be used to access components. In addition, all created components instances are automatically wrapped by an associated proxy. This means every:
`new ComponentAImpl()`
expression is transformed into:
`new ComponentA_$Proxy(new ComponentAImpl())`.
This requires that all classes of a project needs to be recompiled whenever a `declare proxy` statement changes.

An important property of the generated proxy classes is that they are *anonymous*, Implying that we cannot write a pointcut that selects a joinpoint in a specific proxy (e.g. in `ComponentA_$Proxy`; the concrete type of a proxy is unavailable during the implementation of an aspect). This is due to the fact that the interface specified in the declare proxy statement does not directly correspond to one component; instead it determines a set of components that are related by the implementation of the interface. However, we can still refer to the proxies by their parent types as we will illustrate in the following by the advice of the passivation aspect.

Passivation (line 10) of components is executed asynchronously (line 3,15) if a component has been idle for the specified time (line 5, 8 and 9).

```
1  Set<PassivationProxy> proxies
2    = new WeakHashSet<PassivationProxy>();
3  { Thread thread = new Thread(
4      new Runnable(){
5        final long x = ...;
6        final long t = ...;
7        public void run(){
8          for (PassivationProxy proxy : proxies){
9            if (System.currentTimeMillis() −
10             proxy.lastAccess > t){
11           passivate(proxy);
12       } }
```

```
13              thread.sleep(x);
14            }
15          }
16        );
17        thread.start ();
18    }
```

In the default implementation of the proxies each method call is directly forwarded to the corresponding component method. Before this is done we want to make sure (as shown by the following code snippet) that the component is not passivated (lines 1-4) and after processing of the call by the component the time of the last access (lines 5-8) is updated.[1]

```
1   before (PassivationProxy proxy) : execution (..)
2     && !within(PassivationProxy) && this(proxy) {
3     if (isPassivated(proxy)) activate(proxy);
4   }
5   after (PassivationProxy proxy) : execution (..)
6     && !within(PassivationProxy) && this(proxy) {
7     proxy.lastAccess = System.currentTimeMillis();
8   }
```

The aspect additionally defines the methods to passivate and activate a component. The implementation details are omitted because they do not further contribute to the discussion of the `declare proxy` statement. The signatures are:

```
1   boolean isPassivated(PassivationProxy proxy){...}
2   void activate(PassivationProxy proxy){...}
3   void passivate(PassivationProxy proxy){...}
```

The last piece in the implementation of the `Passivation` aspect is the `PassivationProxy` class. In this implementation the proxy object is not only a virtual instance it also implements part of the logic of the service. The proxy class has a field to store the time of the last access (line 3) and also registers the proxy at the passivation service (line 6).

```
1  abstract class PassivationProxy<T>
2  extends DefaultProxy<T>{
3    long lastAccess;
4    public PassivationProxy(T instance) {
5      super(instance);
6      Passivation.aspectOf().proxies.add(proxy);
7    }
8  }
```

2.2 Proxies – Design Space and Decisions

Multiple possibilities exist for the generation of proxy classes. In the following we discuss them to determine the semantics of our declare proxy statement.

A proxy class can be (1) a subclass of a component or (2) an implementation of the interface of a component. The first possibility does not impose any particular requirements on the design of an application in order for it to be "decorated" by proxies; it simply requires that a component is not final, has no final methods and does not declare any non-private fields, otherwise the proxy could be bypassed. The problem with this alternative is that every instantiation of a proxy also instantiates the superclass (the component). So, a proxy object is in a sense also a component. This is not feasible if a proxy is supposed to be a (temporary) replacement of a component, e.g., to perform optimizations such as passivation, pooling and lazy initialization. The second solution assumes a particular design – every component must be implemented against an interface and the component's type is not allowed to be used directly (details are given later). However, this is only a small restriction since it is also considered good design to implement toward interfaces. Further, a proxy object is not a component, a proxy instance is fully independent from a component instance. So, we basically consider this solution the only feasible one. However, an equivalent alternative to the second solution is to require that every component fulfills implementation restrictions so that we are able to generate the necessary interfaces on demand and transform the program appropriately. This approach is taken by Caesar [19]. While this solution might me more convenient, it reliefs the developer from the burden of implementing the interfaces on its own, it is from a technical point-of-view no different.

A proxy can either be *open* or *closed*. A closed proxy, on the other side, is the only class which ever holds a reference to a component and no aliasing [14] of the component references takes place. In the following we use the term wrapper and proxy as a synonym for *closed proxy*.

Further proxies can either be replacing or forwarding. A replacing proxy processes a method call on its own and never relies on a component instance. A forwarding proxy executes functionality before or after forwarding the call to a component instance. We support both replacing and forwarding semantics. For this, the proxy class specified in the `extends` clause of the declare proxy statement must implement either a constructor with the following signature:

```
1   public MyProxy(Constructor c,Object [ ]args){...}
```

or with:

```
1   public MyProxy(T instance){...}
```

The constructor determines whether the proxy replaces (first case) or wraps the component instance (second case). In the first case the parameters of the component constructor are put into an object array and are passed to the proxy along with the original constructor. The constructor is necessary (e.g. for lazy initialization) if the object array alone would not allow the determination of the correct constructor (e.g. if a value is `null` the type can no longer be determined).

Additionally, a proxy must not only be able to wrap a component but also any other proxy generated for the same component. Otherwise, it would not be possible to chain proxies generated for a component by two or more declare proxy statements.

To sum up the generative semantics of our `declare proxy` construct (the properties are named **R1** - **R4**): generated proxies (**R1**) are closed, (**R2**) implement the interface of the component, (**R3**) must support the wrapping of other proxies as well as components and (**R4**) can be either replacing or forwarding.

2.3 Implementation Restrictions

Implementation restrictions are imposed on components annotated as `Wrappable` in order to ensure the following properties:

P1 it is possible to control the aliases of a component; that

[1] Thread synchronization is not shown in the listings because it is unrelated to the `declare proxy` statement.

the proxy is the only owner of a reference to a component.

P2 the use of proxies does not give rise to type errors at run time.

These properties depend on each other. We can control the references to a component if we are able to always use a reference to a proxy instance instead of a reference to a component instance. In this case, it is possible to transform the component such that it always returns its associated proxy instead of `this`. If now a newly created component instance is immediately wrapped by a proxy we can ensure that only the proxy holds a reference to the component. Note that we do not require any kind of alias protection mechanism [24, 8, 12, 7]. We only need to make sure that we are able to *confine* the reference to a component to a single proxy if needed; in other words, that we can control the aliases for components.

To guarantee the properties **P1** and **P2** the following restrictions need to be imposed:

- the type of the component must not to be used directly; neither in field-, local variable- or method declarations nor in type checks or type casts. ⇒ except for component creation, the only way to access a component is via its interface type(s).
- The supertype of a component must also be a component. Otherwise it would be possible to cast from a component interface type to the type of the superclass and invoke methods, which violates **P1**.
- Every subclass of a component is also considered a component and the first two restrictions apply.

To enforce these restrictions the IRC tool [10] can be used. It allows the definition of restrictions as checks on Java bytecode. We consider it important that these implementation restrictions are explicitly checked before the creation of proxies. Since all restrictions can be checked in one pass and no full program analysis is necessary the compile time overhead should be acceptable. Provided that a component complies with all implementation restrictions the generated proxies fulfill the requirements **R1-R4**

3. RELATED WORK

To the best of our knowledge, there has been no attempt to implement services such as passivation and pooling of components. So we can discuss only related implementation techniques.

The Dynamic Proxies [23] feature of Java can be used to create proxies. But, its usage would have the following drawbacks which makes it a "no-go": (1) **It leads to a tighter coupling between aspects.** The advantages of using proxies especially if they are used for optimizations vanish if each service that requires a proxy generates its own one. So, an aspect that generates a proxy and makes it available for other aspects would be required in order to enable the integration of multiple services using one proxy instance. In the proposed model this would happen implicitly if the `declare proxy` statements are identical. Further, the functionality to define precedence of aspects needs to be duplicated since the "normal" `declare precedence` functionality does not impose an order on the execution of `InvocationHandler`s (implementing the advice) associated with dynamic proxies. (2)**The generated dynamic proxies are open**. So, their use to simulate virtual references is very limited – as discussed especially in the motiviation. (3) **It forces to use reflection.** Even though the support for and performance of reflection is improving the source code remains harder to read and maintain and is also more error prone when compared with the proposed solution. This discussion applies equally well to AOP frameworkds based on the interceptor pattern [22] (e.g.: [3, 4, 21]) or native AOP languages (e.g.: [15, 19])

Hibernate [2] generates proxies at runtime using CGLib [1]. CGLib is more powerful than Java Dynamic Proxies – it provides default implementations for different types of proxies. Nevertheless, the base mechanism is comparable and thus suffers from the same problems as described above.

Generic Wrappers [5] is an language integrated approach to generate wrappers / proxies at runtime. E.g. the statement `class PassivationProxy wraps SessionComponent` generates wrappers for `SessionComponent`s and all subtypes of it. However, two main differences exist: (1) wrappers are not automatically used; to create a wrapper for a component it is necessary to explicitly instantiate a wrapper (e.g. `PassivationProxy p = new PassivationProxy<new` *AComponent()*`>()`) while in our case the proxies are automatically used. (2) The generated wrappers are subtypes of the wrapped objects - which means subclasses of components in our terminology. This is in our case not generally applicable – as discussed in the section on the design space of proxies.

Parametric Introductions [13] enable the parameterized introduction of e.g. methods and fields in existing classes. The introductions can be parameterized with the static information which will be available when the introduction is actually carried out, such as the type of the target class, the name of methods and their parameters, etc.. With a so-called unnamed introduction it would be possible to introduce complete sets of methods in a specific class without explicitly enumerating them. This mechanism could be used to introduce generic method implementations in proxy classes. However, as the name suggests parametric introductions can only be used to "introduce" code into an existing class or interface. The generation of completely new classes (the concrete proxy classes in our case) and the transformation of the components is out of scope for this approach. Nevertheless, it could supplement the generation of the proxy classes.

4. SUMMARY AND FUTURE WORK

In this position paper we have proposed an *inter-type declaration* to declare that specific classes are wrapped by automatically generated proxies. This gives us the ability to implement infrastructural services in a more straightforward manner than supported by current approaches. Further, the implementation is very robust since it only relies on types and not on names. The aspect's functionality is directly woven in the generated proxies by using standard pointcuts and advice. Further, it is shown which programming restrictions

needs to be imposed on the implementation of components and how to check them.

In future work we are going to investigate how proxies can help in the implementation of other infrastructural services. Additionally we are going to investigate if other mechanisms are needed to implement infrastructural services.

Acknowledgment

The author would like to thank the anonymous reviewers, Christoph Bockisch, Mira Mezini, Klaus Ostermann and Thorsten Schäfer for comments on earlier versions of this paper.

5. REFERENCES

[1] Code Generation Library (cglib). http://cglib.sourceforge.net/.

[2] Hibernate2 Reference Documentation. http://hibernate.bluemars.net/. Version 2.1.1.

[3] JBoss AOP. http://www.jboss.org/developers/projects /jboss/aop.

[4] Jonas Bonr and Alexandre Vasseur. Aspectwerkz. http://aspectwerkz.codehaus.org, 2003.

[5] Martin Büchi and Wolfgang Weck. Generic wrappers. In Elisa Bertino, editor, *Proceedings of ECOOP 2000*, volume 1850 of *Lecture Notes in Computer Science*, pages 201–225. Springer Verlag.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – a System of Patterns*. John Wiley & Sons, 1996.

[7] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: deployment-time confinement checking. In *Proceedings of OOPSLA 2003*, pages 374–387. ACM Press.

[8] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of OOPSLA '98*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York. ACM Press.

[9] Linda G. DeMichiel. *Enterprise JavaBeans^{TM} Specification, Version 2.1*. Sun Mircosystems, 4150 Network Circle, Santa Clara, California 95054, U.S.A, November 2003.

[10] Michael Eichberg, Mira Mezini, Thorsten Schäfer, Claus Beringer, and Karl Matthias Hamel. Enforcing system-wide properties. Melbourne, Australia. IEEE Computer Scociety. Proceedings of ASWEC 2004 (to appear).

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995.

[12] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of OOPSLA 2001*, pages 241–255. ACM Press.

[13] Stefan Hanenberg and Rainer Unland. Parametric introductions. In *Proc. of AOSD 2003*, pages 80–89. ACM Press.

[14] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355, Budapest,Hungary. Springer.

[16] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of ECOOP 1997*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer.

[17] Ivan Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams, July 2002.

[18] R. Laddad. *AspectJ in Action*. Manning, 2003.

[19] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *Proceedings of AOSD 2003*, pages 90–99. ACM Press.

[20] OMG. *CORBA Components*. Object Management Group, June 2002. Version 3.0, formal/02-06-65.

[21] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Grard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of REFLECTION 2001*, volume 2192 of *Lecture Notes in Computer Science*, Kyoto, Japan. Springer.

[22] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*. Software Design Patterns. John Wiley & Sons, 2000.

[23] Sun Microsystems. Dynamic Proxy Classes. java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[24] Jan Vitek and Boris Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.

[25] Markus Völter, Alexander Schmid, and Eberhard Wolff. *Server Component Patterns*. Software Design Patterns. John Wiley & Sons, 2002.

Applying Aspect Orientation to J2EE Business Tier Patterns

Therthala Murali
murali_therthala@yahoo.com

Renaud Pawlak
pawlakr@rh.edu

Houman Younessi
houman@rh.edu

Rensselaer Polytechnic Institute- Hartford Graduate Campus
275 Windsor St, Hartford, CT 06120 USA

ABSTRACT

J2EE Design Patterns [1] offer flexible solutions to common software problems encountered in the design and construction of distributed systems for the J2EE platform. A number of J2EE patterns involve crosscutting structures in the relationship between the roles in the pattern and classes in each instance of the pattern, thus making the resulting components increasingly complex. This complexity is at odds with one of patterns' key goals - to make it easier to build simple, elegant and high-quality systems that work. This paper analyzes the problem of crosscutting within the implementation of J2EE patterns in the Business Tier and demonstrates how Aspect-Oriented techniques can be used to generate improvements within the business layer components from the perspective of better code locality, reusability, composability and (un)pluggability.

1. INTRODUCTION

Software Patterns are designed to communicate expert knowledge about system construction. Useful Patterns address structural problems and are carefully written to be readable.

Prior research [2] shows that aspect-based implementations of the GoF design patterns showed modularity improvements in 17 of 23 cases. These improvements were in terms of better code locality, reusability, composability and (un)pluggability. These results suggest that it would be worthwhile to undertake the experiment of applying aspect-oriented techniques to J2EE pattern implementations.

Constructing an application under the J2EE platform involves the assembly/composition of prefabricated, reusable and independent components. The J2EE design patterns [1] offer flexible solutions to construct high-quality, reusable, evolvable components for the J2EE platform. While a lot of the J2EE pattern literature is focused on highlighting the benefits of the J2EE applications constructed using the patterns, there is hardly any discussion of how the patterns have introduced code tangling and code scattering within the core functionality of the J2EE components.

In our study, we highlight the problems caused by code scattering and code tangling within the J2EE business tier due to the implementation of the patterns. We develop and compare Object-Oriented and Aspect -Oriented implementations of the J2EE patterns for this tier. We retain the purpose, intent and applicability of the J2EE patterns but only allow the solution structure and solution implementation to change.

The rest of the paper is organized as follows. Section 2 highlights some of the problems created within the business tier due to the implementation of J2EE design patterns and present them. Section 3 introduces the format of study that we have undertaken. In section 4, we present our AspectJ implementations for some patterns and highlight the improvements we observed. Section 5 summarizes our work.

2. CHALLENGES

2.1 Established Challenges

The three major problems [2] in systems realized using patterns are related to pattern implementation, pattern documentation and pattern composition.

Implementations of patterns are often governed by the instance of use and context owing to the fact that pattern implementations heavily influence system structures and vice versa [3]. This makes it hard to distinguish between the pattern, its concrete instance and the pertinent object model [4]. Changes to a pattern within a system are often invasive and tedious. Consequently, while the design pattern is reusable, its implementations usually are not [2].

As stated in [2], the impacts of design patterns on programs are of two different natures. In the first case, they can *superimpose roles*. An initial functional class could be enhanced to define a role in the design pattern. In the second case, they could add new classes to the program that are independent from the initial functional program and *define new roles*. In both the cases, the design patterns are not completely modular. In the first case, the design pattern implementation is invasive since it modifies a class of the initial program. In the second case, the newly created role has to be used eventually by a class of the functional program and this reflects the existence of an associated superimposed role.

Thus, non-modularization in the business layer of the J2EE applications due to patterns introduces code scattering and code tangling within the program. Code Scattering is caused because several instances of the patterns or of a given role will be used within several classes of the program. Code tangling occurs when several pattern or role instances overlap in a single class. This last effect is particularly troublesome because, when a particular class is involved in more than one pattern, it becomes difficult to compose the patterns together because the structure of the application becomes less straightforward. Moreover, documenting the patterns and their participants within the application also becomes cumbersome.

2.2. Crosscutting in J2EE Patterns

This section presents the standard J2EE design patterns and discusses the problems caused by their implementation in terms of crosscutting.

2.2.1 Business Delegate

The Business Delegate hides the underlying implementation details of the business service, such as lookup and access details of the EJB Container and JNDI Directory Services and thereby reduces the coupling between presentation-tier clients and business services. However, interface methods in the Business Delegate may still require modification if the underlying business service API changes. The reference to the Business Delegate layer, within every client that accesses the business services layer, is a crosscutting concern. While location transparency is one of the benefits of this pattern, a different problem may arise due to the developer treating a remote service as if

it was a local one. This may happen if the client developer does not understand that the Business Delegate is a client side proxy to a remote service. Typically, a method invocation on the Business Delegate results in a remote method invocation under the wraps. Ignoring this, the developer may tend to make numerous method invocations to perform a single task, thus increasing the network traffic.

It would be worth exploring if there exists a way to leverage the advantages offered by the business delegate layer, without implementing the business delegates and eliminating the coupling with the client.

2.2.2 Service Locator

The Service Locator pattern reduces the client complexity that results from the client's need to perform lookup of distributed services and their creation, which are resource-intensive. However clients that use the Service Locator are faced with a plethora of crosscutting problems.

A client of the Service Locator such as a Business Delegate has to explicitly reference the interfaces (`javax.ejb.EJBHome` and `javax.ejb.EJBLocalHome`) within the javax.ejb package and the exceptions within the `javax.ejb`, `java.rmi` and the `javax.naming` packages. The clients ought to capture these exceptions and handle them appropriately. The references to the interfaces and classes are a crosscutting concern.

A reference to the Service Locator within the client that needs to lookup services is in itself a crosscutting concern. The Service Locator is an implementation of the GoF Singleton pattern and has a private constructor. Hanneman et al [2] have demonstrated how a plain old java object can be turned into a Singleton by weaving into it, the Singleton Protocol via an aspect. It would be worth applying this idea to the Service Locator to see if it offers any advantages within the J2EE world.

2.2.3 Transfer Object

When clients require more than one value from the business services layer, it is possible to reduce the number of remote calls to the Session Façade and to avoid overhead by using Transfer Objects to transport the data from the enterprise bean to its client.

In order to be transportable over the wire via Java's Remote Method Invocation (RMI), the Transfer Objects have to implement the `java.io.Serializable` interface. If a client that is located within the same virtual machine as the Session Facade, desires to invoke the same business service,the client need not invoke the service via RMI and hence the implementation of `java.io.Serializable` by the Transfer Object becomes redundant.

The Client and the Session Facade that use Transfer Objects reference these objects within their implementations. Thus it would be worth investigating whether the benefits offered by Transfer Objects can be obtained, without them implementing the `java.io.Serializable` and also not cross-cutting the client and Session Facade implementations.

2.2.4 Session Facade

The Session Facade in a J2EE application is usually a Session Enterprise Bean that manages the business objects, and provides a uniform coarse-grained service access layer to the clients. The benefits of a facade have been highlighted in the GoF literature and also in Core J2EE Patterns [1]. The Session Facade bean ought to implement the `javax.ejb.SessionBean` interface.

It would be worth exploring whether the Session Facade can be made to leverage the features of the EJB Container by realizing it as a plain old java object (POJO) and without implementing the `javax.ejb.SessionBean`.

2.2.5 Transfer Object Assembler

The Transfer Object Assembler can be a POJO or a Session Facade. If the Transfer Object Assembler is implemented as a Session Facade, then the problems discussed in section 2.2.4 for the Session Facade would apply.

2.2.6 Value List Handler

The Value List Handler can be a POJO or a Stateful Enterprise Session Bean. In either of the implementations, the Value List Handler is coupled to the Value List Iterator interface. If a Session Bean, the Value List Handler becomes tied to the `javax.ejb.SessionBean` interface and the problems discussed in section 2.2.4 for the Session Façade would apply.

2.2.7 Composite Entity

The Composite Entity's implementation of the Entity Bean interface is a crosscutting concern and is not beneficial from a system adaptability standpoint. It would be worth pursuing the realization of the Composite Entity as a POJO, without implementing the `javax.ejb.EntityBean` but still leveraging the container managed persistence features.

2.2.8 Application Service

The Application Service is usually a POJO and is implemented either as a Command pattern or as a Strategy pattern. The problems of Command and Strategy have been highlighted [2] and we shall implement the Application Services using the AspectJ versions of Command and Strategy as demonstrated in [2].

2.2.9 Business Objects

The Business Objects are usually implemented either as POJOs or as Enterprise Entity Beans. When realized as POJOs, they are implemented by composing any of the GoF patterns depending on the problem domain. In such a scenario, their AspectJ implementations could be realized as outlined in [2]. When realized as Enterprise Entity Beans, the BusinessObject has to implement the `javax.ejb.EntityBean` interface. Hence if the BusinessObject is to be reused in another J2EE application that does not use Entity Beans, the BusinessObject becomes useless and needs to be converted to a POJO.The implementation of the javax.ejb.`EntityBean` interface by the BusinessObject is a crosscutting concern and does not facilitate seamless component adaptation. It would be worth pursuing the realization of the BusinessObject as a POJO, without implementing the `javax.ejb.EntityBean` interface and yet leveraging the EJB's container-managed persistence features.

3. STUDY FORMAT

The methodology for study involved the design and implementation of a contrived distributed application in accordance with the J2EE specification on a J2EE platform using J2EE patterns, first using the classical Object-Oriented approach and later employing aspects using AspectJ 1.1.4. The core business model of the application provides a Currency component that performs conversion between currency values as shown in the interface listing below.

```
public interface ICurrency {
  public double dollarToPound(
    double aDollarValue) throws
      TooLargeValueException,RemoteException;
  public double dollarToEuro(
    double aDollarValue) throws
      TooLargeValueException,RemoteException;
  public CurrencyTO getCurrencyTable()
    throws RemoteException;
  public CurrencyTO getCurrencyByCountry()
```

```
    throws RemoteException;
  public String getUsCurrency()
    throws RemoteException;
  public String getUkCurrency()
    throws RemoteException;
  public String getFranceCurrency()
    throws RemoteException;
  public String getPolandCurrency()
    throws RemoteException;
}
```

The application's business tier is fronted by EJB Session facades while the client tier consists of java application clients. The application was packaged and deployed on Sun ONE Application Server. The Java implementations correspond to the samples presented in the Core J2EE Patterns book [1]. Each J2EE pattern has a number of implementation variants and alternatives. If a pattern offered more than one possible implementation, we picked the one that seemed the most widely used. Our modularization goals in implementation of J2EE patterns using AspectJ were consistent with those in [2]. In this paper, we will mainly focus on the aspectized implementation of the Business Delegate, the Service Locator, and the Transfer Object patterns.

4. RESULTS

This section presents a comparison of the aspect-oriented and pure object-oriented implementations of concrete instances of the J2EE Business Tier patterns. We focus on the Business Delegate, the Service Locator, and the Transfer Object patterns.

4.1 Business Delegate and Service Locator

In the classical implementation, the Business Delegate pattern manages the complexity of distributed component lookup and exception handling for the calling client, yet the reference to the delegate within the client's implementation is a crosscutting concern. The delegate's presence is truly valuable only when invoking a remote service.

The following code shows the implementation of a typical client. It explicitly uses the Business Delegate that uses the Service Locator pattern.

```
public class TestClient {
  public static void main(String[] args) {
    try  {
      // delegate is used here
       CurrencyDelegate delegate =
        new CurrencyDelegate();
      logger.debug(
        delegate.dollarToPound(10.0) + "GBP");
      // a transfer object is used here
      // it reduces network traffic
      CurrencyTO to =
        delegate.getCurrencyByCountry();
      logger.debug(" US Currency -> " +
        to.getUsCurrency());
    } catch(Throwable t) {
        [...]
```

The following code sample shows the implementation of Currency Delegate using regular Object-Orientation. It has to lookup distributed services by using the Service Locator and deal with the exceptions that can be thrown by the invocation of remote services. In a real application, several delegates are created, usually one per facade. This introduces crosscutting within the clients and a dependence on the JNDI (Java Naming and Directory Interface) and EJB technologies that reduces the adaptability of the application.

```
public class CurrencyDelegate {
  private static ServiceLocator locator;

  private void init() throws SystemException {
    try {
      locator =  ServiceLocator.getInstance();
    } catch(NamingException ne) {
      throw new SystemException(
        ne.getMessage());
    }
  }

  private Currency getServiceFacade()
    throws SystemException {
    Currency currency = null;
    try {
      CurrencyHome home = (CurrencyHome)locator
        .lookupHome(Currency.class);
      currency = home.create();
    } catch(ClassNotFoundException cne){
      throw new SystemException(
        cne.getMessage());
    } catch(NamingException ne) {
       throw new SystemException(
         ne.getMessage());
    } catch(CreateException ce) {
       throw new SystemException(
         ce.getMessage());
    } catch(RemoteException re) {
      throw new SystemException(
        re.getMessage());
    }
    return currency;
  }

  public CurrencyDelegate()
    throws SystemException {
    if(locator == null)
    init();
  }

  public double dollarToPound(double aValue)
    throws SystemException {
    Currency currency = getServiceFacade();
    try {
      return currency.dollarToPound(aValue);
    } catch(RemoteException re) {
      throw new SystemException(
        re.getMessage());
    } catch(TooLargeValueException te) {
      throw new SystemException(
        te.getMessage());
    }
  }
  // same principle with other delegating
  // methods
  [...]
```

The code snippet below shows how the `ClientAspect` and the `LocatorAspect` combine to make the Business Delegate obsolete. The pointcuts and their corresponding advices provide the necessary J2EE plumbing that enables a plain java client to invoke the business services offered by components within the EJB container.

```
public class TestClient {
  public static void main(String[] args) {
    try {
      TestClient client = new TestClient();
      ICurrency currencyService =
        (ICurrency)client.getServiceFacade(
           Currency.class);
      logger.debug("GB POUNDS -> " +
        currencyService.dollarToPound(10.0));
      logger.debug("GB CURRENCY -> " +
        currencyService.getUkCurrency());
      logger.debug("US CURRENCY -> " +
        currencyService.getUsCurrency());
    } catch(Throwable t) {
      t.printStackTrace();
    }
  }

  public Object getServiceFacade(Class aClass)
    throws SystemException {
    // empty method that is automatically
    // implemented by the client aspect
    return null;
  } [...]
```

As is evident from the discussion, the Client does not need to (i) use a Business Delegate, (ii) provide an implementation of the `getServiceFacade` method. The `LocatorAspect` introduces into

the client the implementation for the `getServiceFacade` method which is used to lookup the Service Facade. This facade directly implements the business component's interface and appears to be co-located with the client. This technique offers two main advantages. Firstly, it simplifies the overall design by removing the delegate in most cases (usually, delegates have the same interfaces as the facade they delegate to – note that the use of a specific delegate is still possible). It makes the code more local because as seen in the implementation of the `LocatorAspect`, all the delegating code is confined to a unique aspect. Secondly, the code has no distributed semantics.

The final code is:

- less technology dependent – it can use the EJB component model or any other distributed computing technology or none at all.
- independent of deployment semantics– in case the client is finally deployed in the same virtual machine as the server, then the `getServiceFacade` implementation can be easily changed to return a direct reference to a local object and not the delegate.

The following sample code shows the main parts of the Locator aspect.

```
public aspect LocatorAspect {
  public static final String CURRENCY_SERVICE =
  "edu.rh.cs.j2ee.business.Currency";
  private EJBServiceLocator ejbLocator;
  private JDBCServiceLocator
    jdbcConnectionLocator;
  private JMSServiceLocator jmsObjectLocator;

  // pointcut to capture calls made to
  // getServiceFacade.
  pointcut ejbservice(Class aClass):
    call(* *.getServiceFacade (Class))
    && args(aClass);

  // same principle for databases
  pointcut connectionservice(
    String aDataSource):
    call(* *.getDatabaseConnection(String))
    && args(aDataSource);

  // same principle for JMS
  pointcut jmsservice(String aJMSObject):
    call(* *.getJMSObject(String))
    && args(aJMSObject);

  // EJB service locator -> EJBHome
  Object around(Class aClass)
    throws SystemException:
    ejbservice(aClass) {
      Object service =null;
      try {
        if(ejbLocator == null)
          ejbLocator = new EJBServiceLocator();
        Object home =
          ejbLocator.lookup(aClass);
        // all the lookups can be centralized
        // right here...
        if(aClass.getName()
            .equals(CURRENCY_SERVICE)) {
          CurrencyHome currencyhome =
            CurrencyHome)home;
            service = currencyhome.create();
        }
      } catch (NamingException ne) {
          throw new SystemException(
            ne.getMessage());
      } catch (ClassNotFoundException cne) {
          throw new SystemException(
            cne.getMessage());
      } catch(CreateException ce) {
          throw new SystemException(
            ce.getMessage());
      } catch (RemoteException re) {
          throw new SystemException(
            re.getMessage());
      } catch (Exception e) {
          throw new SystemException(
            e.getMessage());
      }
      return service;
    }

    // -> java.sql.Connection
    Object around(String aDataSource)
    throws SystemException:
    connectionservice(aDataSource) {
      [...]
    }

    // -> JMS Object
    Object around(String aName)
    throws SystemException:jmsservice(aName) {
      [...]
    }

    public pointcut exception():
      call(* edu.rh.cs.j2ee.business..*+.*(..)
           throws *Exception)
      && !within(LocatorAspect);

    // soften thrown exceptions
    declare soft:RemoteException: exception();
    declare soft:TooManyItemsException:
      exception();
    declare soft:TooLargeValueException:
      exception();

    Object around():exception()   {
      Object value = null;
      try {
        value = proceed();
      } catch(Exception e) {
        throw new RuntimeException(
          e.getMessage());
      }
      return value;
    }
}
```

The `LocatorAspect` removes the Client's need to reference the Service Locators explicitly to lookup objects and locate services. This task is seamlessly done within the advices for the pointcuts `ejbservice`, `connectionservice`, and `jmsservice`. The `Locator Aspect` also introduces into the client, the reference to the Service Locator and the references to the classes and interfaces within the `java.rmi` and `javax.naming` packages. The remote exceptions are captured and logged by the `LocatorAspect` (see the `exception()` pointcut). If the application requirement is such that a certain exception is to be handled consistently for all incoming client transactions, then the exception handling can be implemented within the `LocatorAspect` itself and a user friendly error message can be encapsulated within a generic runtime exception (`CompositeRuntimeException`) and passed onto the client. If the application requirement is such that a certain exception is to be handled differently depending on the type of incoming client transaction, then the `LocatorAspect` can pass the exception onto the client, by wrapping it within the `CompositeRuntimeException`. The client layer can then choose to handle the exception appropriately. Finally, the `LocatorAspect` also introduces into the client, the reference to the classes and interfaces within the `javax.ejb` package.

The actual remote invocation performed previously by the delegate is performed within the `ClientAspect`. The code below shows the parts of the `ClientAspect` responsible for the invocation of the `dollarToPound` method of the remote service. Note that it uses `getServiceFacade`, which is implemented by the `LocatorAspect`.

```
public aspect ClientAspect {
  [...]
  // should write a more general pointcut
  pointcut currencyConversion(double aValue):
  call(* edu.rh.cs.j2ee.business.ICurrency+
       .*(*))
  && args(aValue)
  && !within(ClientDataTransferAspect);
```

```
  [...]
  double around(double aValue)
  throws java.rmi.RemoteException:
  currencyConversion(aValue) {

    Signature sig =
      thisJoinPointStaticPart.getSignature();
    String name = sig.getName();
    ICurrency currency = null;
    if(name.equals("dollarToPound")) {
      try {
        currency = (ICurrency)
          getServiceFacade(Currency.class);
      } catch(SystemException se) {}
      return currency.dollarToPound(aValue);
    } else if [...] // other methods
  }
  [...]
```

The Client implementation is conscious only of the Business interfaces and knows nothing about any of the classes or interfaces within the `javax.ejb` package. The client layer is EJB technology agnostic. So if an existing EJB based J2EE solution is implemented using `the LocatorAspect` and `ClientAspect` and, if later there arises a need to convert the existing EJB based implementation to a non EJB solution, then the conversion can be accomplished effortlessly by simply not weaving the aspects into the client, during the compilation phase.

The Service Locator's implementation as a Singleton is based on the techniques outlined in [2]. The Service Locator can be instantiated like a POJO using the new constructor instead of using a factory method like getInstance. However this feature can lead to some confusion among J2EE developers. A factory method makes it clear that the Service Locator is a singleton but the new constructor does not. So it is conceivable that the Singleton might extend another class that can be cloned and developers might call the clone method of the Singleton. In order to prevent the cloning of the Singleton, the `SingleProtocol` aspect's Singleton interface has been modified as shown below.

```
public Object SingletonProtocol.Singleton.clone()
throws CloneNotSupportedException {
   throw new CloneNotSupportedException();
}
```

So if an attempt is made to clone the Singleton, a `CloneNotSupportedException` is thrown.

4.2 Transfer Object

The implementation of the `TestClient` in the typical J2EE application references the transfer object `CurrencyTO`. The Transfer Object reduces the network traffic by carrying multiple data items. The Aspect-Oriented version of the `TestClient` does not use the Transfer Object. As shown in the implementation below, the `Client Aspect` captures the join points of a logical set of remote calls made by the client to the business service within a pointcut. The advice to this pointcut allows the remote invocation during the first call and fetches all the data for the remainder of the invocations within a transfer object. The `ClientAspect` caches the transfer object locally to service subsequent client requests. Thus it eliminates the crosscutting within the client due to the Transfer Object pattern.

```
public aspect ClientAspect {
  public static final String CURRENCY =
    "CurrencyTO";
  private HashMap transferObjectMap =
    new HashMap();

  [...]
  pointcut currencytransfer():
    call(* edu.rh.cs.j2ee.business.ICurrency+
         .get*Currency())
    && !within(ClientAspect);

  Object around()
  throws java.rmi.RemoteException:
  currencytransfer() {

    Signature sig =
      thisJoinPointStaticPart.getSignature();
    String name = sig.getName();
    CurrencyTO to =
      (CurrencyTO)transferObjectMap
       .get(CURRENCY);

    // if the cached TO is null, fetch it
    if(to == null) {
      to = new CurrencyTO();
      ICurrency currency = null;
      try {
        currency = (ICurrency)
        getServiceFacade(Currency.class);
      } catch(SystemException se) {}
      CurrencyTO fetched =
        currency.getCurrencyByCountry()
      to.setUsCurrency(
        fetched.getUsCurrency());
      to.setUkCurrency(
        fetched.getUkCurrency());
      to.setFranceCurrency(
        fetched.getFranceCurrency());
      to.setPolandCurrency(
        fetched.getPolandCurrency());
      transferObjectMap.put(CURRENCY,to);
    }

    // get the data from the cache
    if(name.equals("getUsCurrency"))
      return to.getUsCurrency();
    else if(name.equals("getUkCurrency"))
      return to.getUkCurrency();
    else if(name.equals("getFranceCurrency"))
      return to.getFranceCurrency();
    else if(name.equals("getPolandCurrency"))
      return to.getPolandCurrency();
    return null;
  }
  [...]
```

The `ClientAspect` also provides a cache invalidation pointcut. For our simple case, it invalidates the cache (removes the transfer object from the hash map) when the program returns from the main method. The cache invalidation pointcut is application dependent and can be quite complex in real applications.

4.3 Session Facade

The Aspect version of the pattern uses the `SessionBeanProtocol` and the `FacadeAspect`, to introduce the `javax.ejb.SessionBean` interface within the session facade. The `CurrencyBean` session façade is a POJO that is business -functionality centric and is oblivious to the `javax.ejb package`. The facade is reusable and adaptable within a non EJB environment.

```
public interface SessionBeanProtocol
    extends javax.ejb.SessionBean {}

public aspect FacadeAspect {
  // ICurrency is due to the BusinessInterface
  //pattern and not the
  //Remote interface
  declare parents: CurrencyBean implements
     SessionBeanProtocol,ICurrency;
  public void SessionBeanProtocol.ejbCreate()
    throws CreateException {}
  public void SessionBeanProtocol.ejbRemove() {}
  public void SessionBeanProtocol.ejbActivate() {}
  public void SessionBeanProtocol.ejbPassivate() {}
  public void SessionBeanProtocol
        .setSessionContext(SessionContext sc) {}
}
```

4.4 Code Improvement Evaluation

4.4.1 Business Delegate, Service Locator, and Transfer Object patterns

The Aspect-Oriented implementation of the Business Delegate, the Service Locator and the Transfer Object patterns has the following closely related modularity properties:

Locality – All the code that implements the Business Delegate functionality and the associate service lookup is in the `ClientAspect` and the `LocatorAspect` and none of it is in the participating client classes. For each kind of service lookup, the code is within the advice of the `LocatorAspect`. The packaging of all related data for a transfer object is localized within the `ClientAspect`. The participant clients are entirely free of the Business Delegate and Transfer Object pattern contexts and as a consequence there is no coupling between the participants. Potential changes to the pattern instance are confined to one place. All the Singleton related code is within the `SingletonProtocol` and the ServiceLocator is a POJO.

Reusability – The core pattern code is abstracted and reusable. The implementation of the `getServiceFacade` method within the `Client` via the `LocatorAspect` generalizes the overall pattern behavior. The interface can be reused and shared across multiple pattern instances. The implementation of the Transfer Object is limited to the clients that need the same Transfer Object. The `SingletonProtocol` aspect can be reused to create several types of Singeltons.

Composition transparency – Since the Client implementation is not coupled to either of the patterns, it can participate in other kinds of pattern relationships and the resulting code does not become more complicated. Since the ServiceLocator is oblivious to the Singleton pattern's context, it could participate in another pattern context seamlessly.

(Un)pluggability – Since the Client need not be aware of its role in any of these pattern instances, it is possible to switch effortlessly between using the Business Delegate pattern and Transfer Object pattern, and not using them in the system. It is possible to add and remove the Singleton property to the ServiceLocator easily.

4.4.2 Session Facade pattern

The Aspect-Oriented implementation of the Session Facade pattern has the following closely related modularity properties:

Locality – All the code that implements the Session Facade functionality is within a POJO and the Session Bean contract is introduced via a protocol and an aspect. For each kind of Session Facade, we only need to extend the `SessionBeanProtocol` and supply an implementation for the Session Bean's methods via the aspect. The participating facade is entirely free of the pattern context, and as a consequence is EJB agnostic. Potential changes with the EJB container's contract are confined to the aspect.

Reusability – The core pattern code present within the Session Bean interface methods (`ejbCreate()`, `ejbActivate()`…) is abstracted and reusable.

Composition transparency – Since the facade implementation is not coupled to the pattern, it can participate in other kinds of pattern relationships and the resulting code does not become complicated.

(Un)pluggability – Since the facade need not be aware of its role in this pattern instance, it is possible to switch effortlessly between using the EJB Component Model and not using it in the J2EE application.

4.4.3 Transfer Object Assembler, Value List Handler, and Composite Entity

The modularity advantages discussed for a Session Facade are also applicable to the Transfer Object Assembler and the Value List Handler.

The `CompositeEntityBean` is implemented as a POJO and the `javax.ejb.EntityBean` interface is weaved into the Composite Entity via the `EntityAspect` and the `EntityBeanProtocol`. The Composite Entity implementation purely manages the inter-entity relationships and is unaware of the `javax.ejb` package. The POJO entity is reusable and adaptable within a non EJB environment.

4.4.4 Application Services and Business Objects

The Application Services and the Business Objects are usually implemented using any of the GoF patterns, and as discussed in [2], there may or may not be significant modularity benefits after applying aspects, depending on their implementation.

5. ANALYSIS AND CONCLUSIONS

In this paper we have presented and compared a J2EE application built using EJB Component Software Engineering techniques and using Aspect-Orientation. We have demonstrated and explained how a more flexible, adaptable and reusable component based J2EE system can be built using Aspect-Oriented techniques.

The improvements from using AspectJ in J2EE business tier pattern implementations are closely tied to the presence of crosscutting in the structure of the patterns. Crosscutting in pattern structure is caused by roles [2] and their collaboration with participant classes. We notice great improvements in those patterns where a single module of abstraction handles the original behavior and the pattern specific behavior. In such patterns, the roles cut across participant classes and conceptual operations crosscut methods and constructors. Patterns having shared participants can also crosscut each other. The improvements in the J2EE world are apparent as a set of properties associated leading with modularity. The J2EE pattern implementations are more localized and reusable and hence the system is more adaptable. Localization enhances the documentation. AspectJ implementations of J2EE business tier patterns are composable because there is a better alignment between the dependencies in the code with dependencies in the participant structure.

Our results suggest that Aspect-Orientation should strongly be considered in the design and implementation of J2EE applications.

REFERENCES

[1] Deepak Alur (Author), John Crupi (Author), Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall PTR; 2nd edition (June 2003)

[2] Hannemann and Kiczales. *Design Pattern Implementation in Java and AspectJ*, Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161-173, November 2002.

[3] Florijn, G., Meijers, M., Winsen, P. van. *Tool support for object-oriented patterns*. Proceedings of ECOOP 1997

[4] Soukup, J. *Implementing Patterns*. In: Coplien J. O., Schmidt, D. C. (eds.) Pattern Languages of Program Design. Addison Wesley 1995, pp. 395-412

Aspect-Oriented Design and Implementation of a Java Bytecode Analyzer Framework

Susumu YAMAZAKI*
Fukuoka Laboratory for Emerging & Enabling Technology of SoC
Fukuoka Industry, Science & Technology Foundation
816-8580, JAPAN
yamazaki@fleets.jp

Michihiro MATSUMOTO†
Fukuoka Laboratory for Emerging & Enabling Technology of SoC
Fukuoka Industry, Science & Technology Foundation
816-8580, JAPAN
michim@fleets.jp

Tsuneo NAKANISHI‡
Graduate School of Information Science and Electrical Engineering
Kyushu University
816-8580, JAPAN
tun@f.csce.kyushu-u.ac.jp

ABSTRACT

We propose a new type of Java bytecode analyzer framework based on aspect-oriented design and programming. We also observe that aspect-oriented design and programming improve separation of concerns of many of the characteristics of the design, including extensibility, type safety, and execution efficiency of its design and implementation, when compared to existing analyzer frameworks based on object-oriented design and programming. This paper reports how the following concerns are separated in our framework: the extension of elementary objects, the separation of parser and instruction set, the Visitor Pattern, binary operations and non-functional concerns such as verification.

1. INTRODUCTION

A Java bytecode analyzer framework has a wide range of applications, including bytecode-level optimizing compilers, ahead-of-time compilers, verifiers, aspect weavers. One of the most widely used frameworks is Soot [10], which has been created using extensible object-oriented design and implementation. As a result, it experiences some problems in separation of concerns, type safety, execution efficiency, *etc.*

Therefore, we propose a new type of Java bytecode analyzer framework based on aspect-oriented design and programming using AspectJ [5] (this framework uses Javassist [1] as a bytecode reader and writer). We observe that aspect-oriented design and programming improved separation of concerns of many characteristics of design, including extensibility, type safety, and execution efficiency of design and implementation of the framework, when compared to the existing analyzer frameworks based on object-oriented design and programming.

*Graduate School of Information Science and Electrical Engineering, Kyushu University

†Graduate School of Information Science and Electrical Engineering, Kyushu University

‡System LSI Research Center, Kyushu University

To appear in the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), March, 2004, Lancaster, UK.

1.1 Framework Overview

Java bytecode [7] is a variable length code based on the stack machine model. In our framework, a parser first translates the bytecode into a sequence of objects corresponding to each instruction. The instruction object is also based on the stack machine model. Although we do not currently support translation into 3-address code or the static single assignment (SSA) form, it may be supported later if necessary.

Our framework contains several analyzers. The most central of these is the dataflow analyzer, which can be easily customized. Our framework also contains interprocedural analyzers such as class hierarchy analyzers. Moreover, we can create a composite analyzer, consisting of other analyzers that are called when the composite analyzer is called.

1.2 Contributions and Organization

Through the design and implementation of the analyzer framework, we observed many advantages, both general and application specific. We also observed limitations in the current AspectJ. The rest of this paper outlines these advantages and limitations in the following order:

- We discuss the structured extension of elementary objects maintaining type safety and execution efficiency in Section 2.
- We propose the separation of an extendable bytecode parser from instruction sets in Section 3.
- We propose a simple process description of each instruction using the *Smart Instruction Visitor* in Section 4.
- We propose simple and extendable binary operations in Section 5.

- We discuss problems of description of a verifier as an aspect in Section 6.
- We discuss some related works in Section 7.

2. EXTENSIONS TO ELEMENTARY OBJECTS

It is often remarked that aspect-oriented programming improves separation of concerns. We point out that the most effective example of this is that of elementary objects of a framework, such as instruction objects.

For example, consider adding a new feature to an analyzer derived from a framework. A simple approach is to add fields or methods to the elementary objects in order to store the necessary information.

However, traditional object-oriented programming languages cannot add fields or methods to elementary objects structurally, and so they tend to 'bloat' chaotically. If a structure is enforced, the class hierarchy can become deep. In either case, maintainability and readability are degraded.

Within a framework, the extension of elementary objects is realized by using an indirect approach such as table or the Visitor Pattern [3], rather than by direct addition of fields or methods. For example, in Soot elementary objects are extended by adding tags. Tags are named, and may be requested by searching a table using this name.

The above techniques may sacrifice type safety or execution efficiency. Soot sacrifices both of them: the retrieved tag sacrifices type safety and must be cast downward before it may be used, and execution is inefficient because of the need for searching the table.

AspectJ can define fields and methods directly with classifications, as an aspect using an inter-type member declaration. For example, if we add a field or a method necessary for an analysis, we can define it structurally in an aspect concerned with the analysis.

Indirect extension mechanisms such as tags in Soot, are no longer needed. The type system of AspectJ ensures the type safety of added fields and methods, and because they are woven into classes directly, execution efficiency is improved when compared to indirect extension.

```
public class Parser {
    public static class Instruction {
        void setSucc
          (Instruction[] table, int pc) {}
        ...
    }
    public static interface Terminator {}
    public static interface Branch {...}
    public static interface Switch {...}
    static aspect addNextToSucc {
        pointcut addNextToSucc(Instruction inst,
          Instruction[] table, int pc)
            : call(void Instruction.setSucc
                   (Instruction[], int))
            && target(inst) && args(table, pc)
            && !target(Terminator);

        before(Instruction inst,
            Instruction[] table, int pc)
            : addNextToSucc(inst, table, pc) {
            ...
        }
    }
    static aspect addBranchTargetToSucc {
        pointcut addBranchTargetToSucc
          (Instruction inst,
           Instruction[] table, int pc)
            : call(void Instruction.setSucc
                   (Instruction[], int))
            && target(inst) && args(table, pc)
            && target(Branch);

        before(Instruction inst,
         Instruction[] table, int pc)
            : addBranchTargetToSucc(inst,
                  table, pc) {
            ...
        }
        ...
    }
}
```

Figure 1: Bytecode Parser using AspectJ

3. SEPARATION OF THE BYTECODE PARSER AND INSTRUCTION SETS

A bytecode parser scans binary class files, generates instruction objects corresponding to the byte sequences, and inserts labels. It also sets the relationships between instructions, for example using a succeed set, which is a set of instructions that may be executed after other instructions except those throwing exceptions in a manner similar to that of a Factored Control Flow Graph [2].

We now focus on setting succeed sets, which depend on the class of an instruction. Instructions are divided into non-terminator and terminator categories: a succeed set of a non-terminator includes the next instruction, while a succeed set of a terminator does not.

Instuctions may also be divided into non-branch, branch and switch categories: a succeed set of a non-branch instruction does not include any special jump target; a succeed set of a branch instruction includes one jump target; and a succeed set of a switch instruction includes two or more jump targets.

A succeed set can be determined by the classification, rather than by the instruction set, but the instruction set determines how a concrete instruction class is classified. In addition, another process, such as one detecting PEIs (potential exception-throwing instruction) [2], may require another classification. Therefore, because Java is a language that supports single inheritance and multiple supertypes, we must realize such a classification using `interface`.

However, Java does not allow `interface` to have concrete methods, so the process of setting succeed sets is distributed among code sections containing concrete instructions.

AspectJ solves this problem. Firstly, we provide two aspects to the parser. The first aspect is `addNextToSucc`, which adds the next instruction to the succeed set if the current instruction is a non-terminator. The second aspect is `addBranchTargetToSucc`, which adds the target instruction(s) to the succeed set if the current instruction is a branch or a switch. Secondly, we make a concrete instruction class implement the `interface` corresponding to the classification. Lastly, if the order of the succeed set is important, we can set the priority order using the `precedence` declaration between `addNextToSucc` and `addBranchTargetToSucc`. Figure 1 and Figure 2 show example code of a parser and a instruction set.

```
import Parser.*;

public class Aload extends Instruction {...}
public class Return extends Instruction
    implements Terminator {...}
public class Ifeq extends Instruction
    implements Branch {...}
public class Goto extends Instruction
    implements Terminator, Branch {...}
public class Tableswitch extends Instruction
    implements Terminator, Switch {...}
...
```

Figure 2: Java Bytecode Instruction Set Example

In general, if there are classifications into some given classes, and if the classifications determine the corresponding processes, we can write simple code so that the classifications and the processes are represented using `interface` and aspects, respectively.

Although multiple-inheritance has similar effects, this approach using aspects has two advantages: we can add a classification without modifying existing code, and we can also avoid the method confliction problem. For example, an instruction that is both a non-terminator and a branch is realized easily in AspectJ but cannot be realized naturally using multiple-inheritance.

4. THE SMART INSTRUCTION VISITOR BASED ON THE STACK-MACHINE MODEL

An operation corresponding to a given instruction often includes common processes. For example, because Java bytecode is based on the stack machine model, operations such as push or pop are commonly included in the operations corresponding to each instruction.

Therefore, we provide a Smart Instruction Visitor as part of our framework, based on the Java bytecode model, which is a domain-specific variant of the Visitor Pattern [3]. The programmer has access to four basic operations (`push`, `pop`, `load`, `store`) and the processes corresponding to each instruction. The programmer does not need to write all of these and can override only those necessary.

Because Java bytecode is a typed language, we provide variations of basic operations corresponding to different types. For example, `pushInt` corresponds to the type `int`. We also provide `push` and `pop` operations that handle values using the appropriate types for `getfield`, `putstatic`, *etc.* Moreover, we provide variations for basic operations corresponding to 32- and 64-bit types to satisfy the Java bytecode specification. Finally, we provide variations of `push` and `pop` corresponding to either two 32-bit types or one 64-bit type, for `dup2`, *etc.*

In our framework, we describe the process corresponding to each instruction as a method that is supplied an instruction object and zero or more arguments, and that returns zero or one result. For example, a process corresponding to the instruction `idiv` is defined as a method that is given an instruction object and two division values, and returns a result value object.

It is effective to define pointcuts for methods corresponding to instructions that have common features. This enables the methods to be defined structurally from various viewpoints.

```
public abstract class InstructionVisitor {
    ... // S1
    protected void push(Object value) {}
    protected void push2(Object value) {
        push(value);
    }
    ...
    protected void pushInt(Object value) {
        push(value);
    }
    ...
    protected void pushDouble(Object value) {
        push2(value);
    }
    ...
    protected Object pop() {
        return null;
    }
    ...
    protected void store
      (int index, Object value) {}
    ...
    protected Object load(int index) {
        return null;
    }
    ... // S2
    public static abstract aspect Pointcuts {
        pointcut intBinaryOperator
          (InstructionVisitor v,
           Instruction inst,
           Object value1, Object value2)
            : execution
              (Object InstructionVisitor+
               .at(Instruction+, Object, Object))
              && target(v)
              && args(inst, value1, value2)
              && args(Idiv, Object, Object)
              && ...;
        ...
    } // S3
    protected Object at
      (Iload inst, Object loadedValue) {
        return loadedValue;
    }
    protected Object at
      (Idiv inst,
       Object value1, Object value2) {
        return null;
    }
    ... // S4
    static aspect InsertCode {
        private abstract void Instruction.at
          (InstructionVisitor v);
        ...
        private void Iload.at
          (InstructionVisitor v) {
            Object value
              = v.loadInt(this.index);
            value = v.at(this, value);
            v.pushInt(value);
        }
        private void Idiv.at
          (InstructionVisitor v) {
            Object value2 = v.popInt();
            Object value1 = v.popInt();
            Object result
              = v.at(this, value1, value2);
            v.pushInt(result);
        }
        ...
    }
}
```

Figure 3: The Smart Instruction Visitor

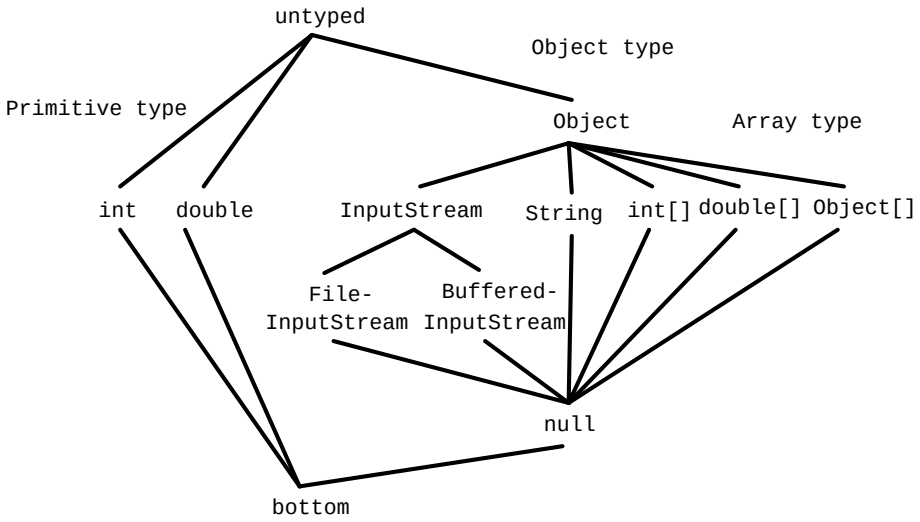


Figure 4: The Type Property Space for Java

Figure 3 shows the implementation of the Smart Instruction Visitor. The basic operations, various pointcuts, processes corresponding to instructions and inner processes, are defined from `S1`, `S2`, `S3` and `S4`, respectively.

The basic behavior is as follows: Methods receiving a Visitor are first defined using inter-type method declarations (`S4`). The corresponding basic operations and processes are called in these methods. For example, the inner method of `idiv` calls `popInt`, twice. The process corresponding to `idiv` is called with the instruction object and the returned values, and `pushInt` is called with the returned value.

We provide default implementations of basic operations and processes corresponding to each instruction. Relationships between the variations of basic operations are represented as an invocation from more constrained variation methods to the less constrained (`S1`). Therefore, all a programmer must do is to override the necessary methods.

5. SIMPLE AND EXTENSIBLE BINARY OPERATION

To realize a data flow equation as a framework, we need to implement the binary operation of properties. In type checking, for example, we must calculate the least upper bound ($\cup$) of the type property at the merge points [8].

Figure 4 shows a lattice representing the property space for type checking [6]. Bottom $\bot$ represents an initial value, so the least upper bound of property P and $\bot$ is P ($\bot \cup P = P \cup \bot = P$). Top $\top$ in type checking means untyped, and the least upper bound of property P and $\top$ is $\top$ ($\top \cup P = P \cup \top = \top$).

Next, the least upper bound of the same primitive type, such as `int`, is the type and the least upper bound of a different primitive type is untyped. For example, $P_{\texttt{int}} \cup P_{\texttt{int}} = P_{\texttt{int}}$, $P_{\texttt{int}} \cup P_{\texttt{float}} = \top$. Note that the rule of top and bottom precedes this rule, *i.e.* $P_{\texttt{int}} \cup \bot = \bot$.

Next, the least upper bound of the object type is a common ancestor. For example, $P_{\texttt{FileInputStream}} \cup P_{\texttt{BufferedInputStream}} = P_{\texttt{InputStream}}$. Note that the rules for bottom, top, and primitive types precede this rule. Moreover, the least upper bound of an object type is a class with zero or more interfaces, because Java provides single inheritance of class but multiple subtyping of interfaces.

Last, the least upper bound of `null` and the primitive type is untyped, and the least upper bound of `null` and the object type is the object type. Note that the rule of bottom precedes this rule.

Consider the implementation of binary operations with a least upper bound based on these rules. A naive implementation may use `instanceof`. For example, Figure 5 shows the implementation of a primitive type property, but this implementation is less extensible and maintainable. If we add a new type property, we must modify all `meet` methods, which calculate the least upper bounds. Moreover, if we change the order of precedence of the rules, we must swap the order of `if` in some methods.

Next, consider the implementation using double-dispatch, as shown in Appendix A. The advantage of this technique is that maintainability is improved because each method is simplified. However, the problems remain, as we must modify all property classes to add a new type property. We also must modify many classes to swap the precedence order of the rules. Moreover, we need to write the same process as many methods, such as the implementation of `Bottom` and `Untyped`.

AspectJ solves these problems simply (see Figure 6). The actual processes are implemented using `around` without `proceed` in the coordinator aspect. For example, `BottomCoordinator` describes the process of involving bottom and something else. Moreover, the process of combining different types is described in a *combination coordinator*. For example, a process of involving a combination of object types and primitive types is described in `ObjectAndPrimitiveCoordinator`.

Next, sort coordinators in `precedence` in topological order of lattice from the bottom. A combination coordinator precedes the coordinator of each property. The content of the method `meet` in the class `Property` is meaningless except when throwing an exception, when it is called with an

```
public class PrimitiveType extends Property {
    public Property meet(Property p) {
        if(p instanceof Bottom) {
            return this;
        }
        if(p instanceof Untyped) {
            return p;
        }
        if(p instanceof ...) ...
        ...
    }
}
```

Figure 5: The Implementation of the Primitive Type Property using `instanceof`

unexpected combination of properties.

This implementation solves the above problems. If we add a new type property, we must only write a coordinator and give the appropriate precedence order. If we must write a special behavior for combination with other properties, we must only write an appropriate combination coordinator. If we change the order of precedence, we must only modify the `precedence`. Moreover, we do not need to write the same process in many methods.

6. THE VERIFIER AS AN ASPECT

Our framework provides a bytecode verifier using a parser and a type checker. One of the advantages of aspect-oriented programming is the ability to unify the cross-cutting concern of a non-functional features such as verification. We have actually implemented the verifier in this way. An overview of our current implementation of the verifier is shown in Appendix B. During implementation, we found that there are two limitations in current AspectJ.

Firstly, AspectJ is not expressive enough to structure aspects. In our implementation, the verifier depends strongly on the inner structure of the parser and the type checker. So, not only must we modify the verifier whenever we modify the parser or type checker, but we cannot reuse the verifier with, for example, another instruction set. This problem is partially solved by using aspect structuring, *i.e.* dividing the verifier into parts that are dependent and independent of instruction sets. However, AspectJ cannot currently separate the verifier in this manner. Abstract pointcuts are useful, but insufficient to perform this separation.

It may not be possible to provide advice with information only from a pointcut. For example, we cannot provide advice to detect the overflow of the operand stack naturally, because its pointcut gives only an operand stack object as a parameter, and the object does not provide a method to retrieve the maximum stack size defined in each method.

This may be solved by defining advice and an inter-type field declaration by adding information about the corresponding method to the stack object. However, this is a specific and *ad hoc* approach.

7. RELATED WORK

Joeq [11] is an extensible virtual machine and compiler infrastructure. Of course, it can be used as a bytecode analyzer framework, and it has many sophisticated features.

Joeq provides the Visitor framework, enabling a simple analyzer implementation. Joeq can realize an analyzer by

```
public aspect Coordinator {
    declare precedence: BottomCoordinator,
        ...
        ObjectAndPrimitiveCoordinator,
        ObjectTypeCoordinator,
        PrimitiveTypeCoordinator,
        ...
        UntypedCoordinator;
}
public abstract class Property {
    public Property meet(Property p) {
        throw new RuntimeException
          ("unsupported property:"
           + this + ", " + p);
    }
}
public abstract aspect PropertyCoordinator {
    pointcut meet(Property p1, Property p2)
        : execution(
          Property Property+.meet(Property+))
        && target(p1) && args(p2);
}
public class Bottom extends Property {}
public aspect BottomCoordinator
  extends PropertyCoordinator {
    Property around(Property p)
        : meet(Property, p)
        && target(Bottom) {
        return p;
    }
    Property around(Property p)
        : meet(p, Property)
        && args(Bottom) {
        return p;
    }
}
...
public class ObjectType extends Property {}
public aspect ObjectTypeCoordinator
  extends PropertyCoordinator {
    Property around
      (ObjectType p1, ObjectType p2)
        : meet(Property, Property)
        && target(p1) && args(p2) {
        // calculate least upper bounds on types
    }
}
...
public aspect ObjectAndPrimitiveCoordinator
  extends PropertyCoordinator {
    Property around
      (ObjectType p1, PrimitiveType p2)
        : meet(Property, Property)
        && target(p1) && args(p2) {
        return new Untyped();
    }
    Property around
      (PrimitiveType p1, ObjectType p2)
        : meet(Property, Property)
        && target(p1) && args(p2) {
        return new Untyped();
    }
}
```

Figure 6: The Implementation of the Type Property using AspectJ

overriding the defined methods *in advance* for some situations, such as field accesses in an instance so, a programmer cannot unify arbitrary methods with the same behavior. On the contrary, our Smart Instruction Visitor can realize the analyzer using pointcuts, which can be defined freely by a programmer without performance penalty.

Joeq also provides a dataflow framework, where the binary operations of properties are defined in the centralized dataflow problem class. Though Whaley does not show an implementation detail for binary operations, it would be complicated for some analyzers, such as a type analyzer.

Though OVM [9] focuses on the virtual machine, its design policy can apply a bytecode analyzer. The main advantage of OVM is its memory efficiency; the OVM intermediate representation (OvmIR) uses the Flyweight Pattern [3]. Our current implementation, on the other hand requires more memory than OVM.

OVM also adopts the Runabout Pattern [4], making it more extensible, but with worse execution performance than the Visitor Pattern approach. This tradeoff is unavoidable when using Java and Java-based languages such as AspectJ. OVM focuses on the customization of the intermediate representation, so OVM has opted for extensibility and the Runabout approach but, because the main target of our framework is Java bytecode, we choose to optimize performance by using the Visitor approach.

Ideally, our approach should be mixed: in the early stage of development, we should take the Runabout approach, and when the specifications of the intermediate representations are almost fixed, we should switch to the Visitor approach. To make the switch easier, we will need an automatic code translator to convert from the Runabout to the Visitor.

8. CONCLUSIONS AND FUTURE WORK

We have built a Java bytecode analyzer framework that uses aspects, and have observed five advantages. Firstly, we realized extensions of elementary objects structurally and maintained type safety and execution efficiency. Secondly, we implemented a bytecode parser that is independent of any single concrete instruction set. Thirdly, we simplified the description of processes for each instruction using the Smart Instruction Visitor based on the stack machine model. Fourthly, we realized binary operations that are simple, extensive, and easy to maintain. Finally, we unified the description of a cross-cutting concern of a wide ranging non-functional features such as verification.

However, we also observed that AspectJ currently has two limitations: it is not expressive enough to structuralize aspects deeply on the basis of their inner structure; and it does not provide a general approach to write advice that cannot be described with its pointcut only.

In the future, we will build a bytecode translator framework based on aspect-oriented software development, enabling us to build many applications such as a bytecode-level optimizing compiler.

9. ACKNOWLEDGMENTS

This research was partly supported by a grant from the Cooperative Link of Unique Science and Technology for Economy Revitalization (CLUSTER) by Ministry of Education, Culture, Sports, Science and Technology (MEXT).

10. ADDITIONAL AUTHORS

Additional authors: Teruaki KITASUKA (Kyushu University, email: `kitasuka@f.csce.kyushu-u.ac.jp`) and Akira FUKUDA (Kyushu University,
email: `fukuda@f.csce.kyushu-u.ac.jp`).

11. REFERENCES

[1] S. Chiba. Load-time structural reflection in Java. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2000), Sophia Antipolis and Cannes, France.*, pages 313–336. Springer-Verlag, June 2000.

[2] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 21–31, Sept. 1999.

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

[4] C. Grothoff. Walkabout revisited: the Runabout. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany.*, pages 103–124. Springer-Verlag, July 2003.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2001), Budapest, Hungary.*, pages 327–353. Springer-Verlag, June 2001.

[6] X. Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999. Second Edition.

[8] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, Berlin, 1999.

[9] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME'03)*, pages 67–76. ACM Press, June 2003.

[10] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – a Java optimization framework. In *CASCON'99*, Sept. 1999.

[11] J. Whaley. Joeq: A virtual machine and compiler infrastructure. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME'03)*, pages 58–66. ACM Press, June 2003.

APPENDIX

A. DOUBLE-DISPATCH FOR BINARY OPERATION

Figure 7 shows a binary operation using double-dispatch. The behavior of this operation is somewhat complicated.

When the method `meet` is called, it calls the method `with*` corresponding to the class of the receiver `this`. For example, if the receiver is `Bottom`, it calls the method `withBottom`. Note that the receiver and the argument of the call is swapped. This realizes binary operations by defining processes corresponding to each class of receiver and the argument of `meet`.

B. OUR CURRENT IMPLEMENTATION OF THE VERIFIER

Figure 8 shows a section of the verifier code.

We can divide this into parsing-time verification and type-checking-time verification subsections. The former subsection includes the pointcut `insertLabel` and the `after` advice of `insertLabel`. The parser calls the method `insertLabel` when it finds a branch instruction. If the branch refers to a location outside the bounds of the code, `insertLabel` throws an `IndexOutOfBoundsException`. The advice of the verifier catches the exception, and rethrows a `VerifyException`.

The latter subsection includes the `stackUnderFlow` and `stackOverFlow` parts. The `TypeChecker` class extends our dataflow analyzer framework, and uses `LinkedList` in the Java class library as the operand stack.

We designed our framework to separate an analyzer from the target bytecode, *i.e.* the analyzer should not hold any analysing state information about the target bytecode, and the target bytecode should hold all of this analyzing state information. The operand stack then belongs to the target.

The `LinkedList` throws a `NoSuchElementException` when the list is empty and the method `removeFirst` is called, so the pointcut and the advice of `stackUnderFlow` catches the exception and rethrows a `VerifyException`.

In contrast, the implementation of `stackOverFlow` experiences a problem when attempting to retrieve the maximum stack size. The advice of `stackOverFlow` can access the operand stack and this join point. We may extract information about the type checker classes from this join point. According to our design policy, however, the type checker classes do not hold any code information, such as the maximum stack size.

On the other hand, the operand stack originally does not hold the maximum stack size because it is an instance of `LinkedList` in the Java class library. If we wish to add the maximum stack size to the stack, we must establish the maximum stack size of the list in advance. It is difficult to ensure this setting for general cases.

```
public abstract class Property {
    public abstract
      Property meet(Property p);
    protected abstract
      Property withBottom(Bottom p);
    protected abstract
      Property withUntyped(Untyped p);
    protected abstract
      Property withPrimitiveType
        (PrimitiveType p);
    ...
}
public class Bottom extends Property {
    public Property meet(Property p) {
        p.withBottom(this);
    }
    public Property withBottom(Bottom p) {
        return p;
    }
    public Property withUntyped(Untyped p) {
        return p;
    }
    public Property withPrimitiveType
      (PrimitiveType p) {
        return p;
    }
    ...
}
public class Untyped extends Property {
    public Property meet(Property p) {
        p.withUntyped(this);
    }
    public Property withBottom(Bottom p) {
        return this;
    }
    public Property withUntyped(Untyped p) {
        return this;
    }
    public Property withPrimitiveType
      (PrimitiveType p) {
        return this;
    }
    ...
}
public class PrimitiveType
  extends Property {
    public Property meet(Property p) {
        p.withPrimitiveType(this);
    }
    public Property withBottom(Bottom p) {
        return this;
    }
    public Property withUntyped(Untyped p) {
        return p;
    }
    public Property withPrimitiveType
      (PrimitiveType p) {
        ...
    }
    ...
}
```

Figure 7: The Implementation of the Type Property using Double-Dispatch

```
public aspect Verifier {
  pointcut insertLabel(Instruction inst, int pc)
    : call(void Instruction
      .insertLabel(Instruction[], int))
    && target(inst) && args(Instruction[], pc);
  after(Instruction inst, int pc)
    throwing (IndexOutOfBoundsException e)
    : insertLabel(inst, pc) {
      throw new VerifyException(
        "The target branch is out of bounds: "
        + pc + ":" + inst);
  }
  pointcut stackUnderFlow()
    : call(Object LinkedList.removeFirst())
    && within(TypeChecker);
  after() throwing (NoSuchElementException e)
    : stackUnderFlow() {
      throw new VerifyException
        ("stack under flow");
  }
  pointcut stackOverFlow(LinkedList stack)
    : call(void LinkedList.addFirst(Object))
    && target(stack)
    && within(TypeChecker);
  before(LinkedList stack)
    : stackOverFlow(stack) {
    int maxStack = ...; // how can we get?
    if(stack.size() >= maxStack) {
       throw new VerifyException
         ("stack over flow:" + maxStack);
    }
  }
}
```

Figure 8: The Implementation of the Verifier

Software Connectors in the COSA Approach

Adel Smeda, Tahar Khammaci, and Mourad Oussalah
LINA, Université de Nantes
2, Rue de la Houssinière, BP 92208
44322 Nantes Cedex 03, France
Tel: +332 51125963, Fax: +332 51125812
Email: {smeda, oussalah, khammaci }@lina.univ-nantes.fr

Abstract. Connectors are very important modeling entities which unfortunately are not sufficiently dealt with by the models of conventional components. Indeed, the majority of describing or programming languages for component-based systems do not offer any means of expressing the connectors explicitly at the level of implementation. In general, they introduce types of predefined connectors (if they propose them any way) or they obligate connectors to be programmed within the components or even sometimes within the application programs. In any case, these languages do not provide mechanisms that permit a user to define new types of connectors with different semantics. In our work we present an approach to model and to describe the architecture of component-based systems, in which connectors are defined as first-class entities.

Keywords: architectural description, components, connectors, configuration, definition.

1. Introduction

Architecture Description Languages (ADL) [1] describe systems as a collection of components that interact with each other using connectors. They define components explicitly, however most of them leave the definition of interactions implicit. Interactions are defined through include files and import and export statements (the connectors are buried inside the components). This implicitly of describing interactions (connectors) makes it difficult to build heterogeneous systems that provide complex functionalities and enroll in complex relations.
In our research we are developing an approach to describe the architecture of component-based systems based on software architecture and object-oriented modeling, it separates components from connectors. This approach is called Component-Object based Software Architecture (COSA [2]). Its basic elements are: components, connectors, interfaces, configurations, constraints, and properties. COSA describes systems in terms of types and instances. Components, connectors, and configurations are types that can be instantiated to build different architectures. In this paper we present the definition of connectors in COSA.

2. Motivations of defining connectors as first-class entities

Obliging components to communicate via connectors has number of significant benefits including: increasing reusability (the same component can be used in a variety of environments, each of them providing specific communication primitives), direct support for distribution, location transparency and mobility of components in a system, support for dynamic changes in the system's connectivity, improving system's maintenance, etc.
In additional, various applications can be modeled more easily by using an approach in which components and connectors explicitly separated. Among those which seem to us most important, we quote:

Configuration management: Components can be organized in several ways:

1. they can be composed of other components (hierarchy of composition)
2. they can exist in several versions (hierarchy of derivation)
3. they can have several representations or point of views.

The combination of these various hierarchies often requires complex propagation mechanisms. For example, If a composite component becomes invalid the subcomponents also should become invalid. Moreover, the creation or destruction of a version of a component can involve the creation (or the destruction) of a new version of the composite component.

Distributed systems: If components are localized at different nodes of a distributed system, the cooperation of the distant components is determined by the semantics of the connectors, which connect them.

Subsystems Coupling: If subsystems need to be coupled, connectors must be defined among the components of the subsystems. The updating or changing operated in one of the subsystems must then be propagated via these connectors to the other subsystems. Hence as a result: The consequences of the activities (functionalities) of a component are not specific properties to this component, but

properties of the connectors that connect these components. The interactions among components (i.e. connectors) often need to adapt to the requirements of a specific environment.

In addition, we think that not taken into account the explicit definition of connectors in components oriented languages, and leaving them tangled as the case of object oriented systems relations could lead to various problems among which we can quote:
Lack of abstraction: the lack of abstraction of the mechanisms used to model connectors does not permit connectors to have a true place in the paradigm of components. Moreover this absence of abstraction prevents connectors from updating and evolving their concepts and their code. Therefore it is often required to understand the whole functionalities of a component to distinguish the implicit connectors, which are buried in the component (connectors and their semantics are mainly based on components).
Increase in the complexity of a component: the existence of many interactions among components in a system contributes to a significant increase in the complexity of the components. And each new interaction adds more complexity to the components.
Lack of reusability: as components and connectors are amalgamated, it becomes difficult to identify the behavior intrinsic of a component, and this harms the reuse and the evolution of components and connectors.
Difficulty in implementation: the implementation of connectors (in the form of properties of components) is proved to be difficult. First of all, the developer must have thought of the interactions in which its instances could take part. Second, the dynamic creation of the connectors is difficult, and it is to be stressed that certain interactions have complex dynamic behaviors. Hence it is the responsibility of the developer to implement these mechanisms.

3. Connectors in COSA

A COSA connector is mainly represented by an *interface* and a *glue* specification [3]. In principle, the *interface* shows the necessary information of the connector, including the number of *roles*, service type that a connector provides (communication, conversion, coordination, facilitations), connection mode (synchronous, asynchronous) , transfer mode (parallel, serial) etc. The interaction points of an interface called *roles* A *role* is the interface of a connector intended to be tied to a component interface (a component's port). In the context of the frame, a *role* is either a *provide role* or a r*equire role*. A *provide role* serves as an entry point to a component interaction represented by a connector type instance and it is intended to be connected to the *require* interface of a component (or to the *require* role of another connector). Similarly, a *require role* serves as the outlet point of a component interaction represented by a connector type instance and it is intended to be connected to the *provide* interface of a component (or to the *provide* role of another connector). The number of roles within a connector denotes the *degree* of a connector type. For example in a client-server a connector type representing procedure call interaction between client and server entities is a connector with degree of two. More complex interactions among three or more components are typically represented by connector types of higher degrees. The interface is the visible part of a connector, hence it must contain enough information regarding the service and the type of this connector. By doing this, one can decide whether or not a given connector suits its qualifications by examining its interface only.
The *glue* specification describes the functionality that is expected from a connector. It represents the hidden part of a connector. The *glue* could be just a simple protocol links the roles or it could be a complex protocol that does various operations including linking, conversion of data format, transferring, adapting, etc. In general the *glue* of a connector represents the connection type of that connector. Connectors can also have an internal architecture that includes computation and information storage. For example a connector would execute an algorithm for converting data from format A to format B or an algorithm for compressing data before it transmits them. Hence the service provided by a connectors is defined by its *glue*, the services of a connector could be either communication service, conversion service, coordination service, or facilitations service.
In case of composite connectors the subconnectors and subcomponents of the composite connector must be defined in the glue, as well as the binding among the subconnectors and subcomponents. Figure 1 presents a meta-model illustrates the structure of the COSA connector. Meanwhile Figure 2 illustrates a client-server architecture described using COSA, the figure shows only one architecture (arch-1), more architectures could be instantiated.

```
Class Configuration client-server {
   Interface {External {External-protocol;} }
   Class Component server {
      Interface{
        Port provide {provide-protocol;}
        external-port{ External-port-protocol ;}}
      Properties { connection-mode=sync;
                   data-type =format-1;
                    max-clients=2;}  }
   Class Component client {
      Interface{
        Port request {sent-request;}}
      Properties { data-type =format-2;}
      Constraints {max-roles =2;} }
```

```
Class Connector RPC{
   Interface {
     Roles {participator-1, participator-2 }
     Service-type = conversion;
     Connection-mode = asynchronous;
     Properties {throughput=10kb;}
     Constraints {no-of-roles <= 2;}
   Glue {
     Define-Service{
        Conversion{
          read participator-1;
          convert from format-1 to format-2;
          write participator-2;}
     Properties{bidirectional;} }
     }
  Binding { server.external-port to External;}
  }
Instance client-server arch-1 {
       Instances {
          S1: server;
          C1: client;
          C1-S1: RPC; }
       Attachments {
          C1.request to C1-S1. participator-1;
          S1.provide to C1-S1. participator-2;
          } }
```

Figure 2. Describing a client-server using COSA.

4. Conclusion

Building heterogeneous systems based on off-the-shelf reused components requires not only well-defined components (with well-defined interfaces) but also well-defined connectors. Hence defining connectors as first-class entities, therefore raising them to the level of components, helps us in reusing them effectively. In this article we define connectors the same way components are defined by separating their interfaces from their behaviors. A connector now is represented by *roles* and *glue* specification. By defining connectors as first-class elements we can specify mechanisms for their reuse and evolution (instantiation, inheritance, template, composition, and refinement [4]).

5. References

[1] N. Medvidovic, R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, Vol. 26, 2000, pp. 70-39.

[2] A. Smeda, M. Oussalah, T. Khammaci "A Multi-Paradigm Approach to Describe Software Systems", *In Proceedings of 3rd WSEAS Int. Conf. On Software Engineering, Parallel and Distributed Systems*, Salzburg, Austria, 2004.

[3] M. Oussalah, A. Smeda,, and T. Khammaci, "An Explicit Definition of Connectors for Component-Based Software Architecture", *In Proceedings of the 11th IEEE Conference on Engineering of Computer Based Systems (ECBS 2004)*, Brno, Czech Republic, May 24-27, 2004.

[4] M. Oussalah, A. Smeda, T. Khammaci, "Software Connectors Reuse in Component-Based Systems", *Proceedings of the 2003 IEEE International Conference on Information Reuse and Integration*, Las Vegas, Nevada, October, 2003.

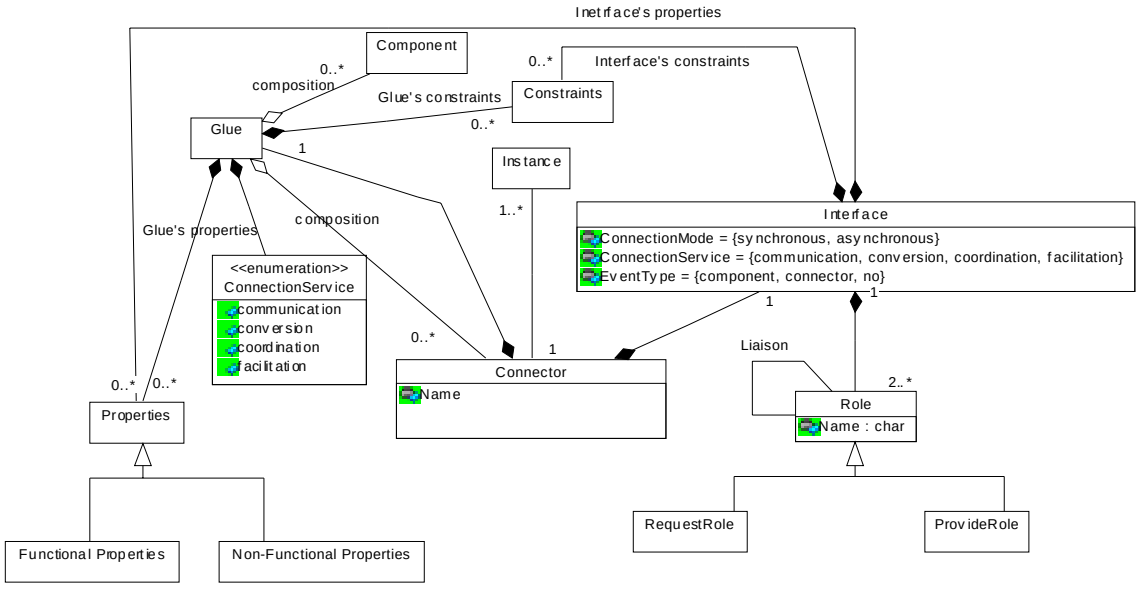


Figure 1. structure of COSA connectors

Addressing Ubiquitous Software Complexity with Mobile Containers

Vasian Cepa
cepa@informatik.tu-darmstadt.de
Darmstadt University of Technology

1 Using Containers to Structure Mobile Applications

We present here an overview of the idea of applying the software containers model to mobile applications using generative programing techniques. We also present an evaluation prototype called MobCon.

A software container is a wrapper component that offers services nearly transparently to other components that '*live*' inside it. The wrapped components contain the functional logic of the application. The services offered by the container are secondary to the logic of the application, but are nevertheless necessary to get an application running. The container services, known as technical concerns [14], include data persistence, logging etc. Components that live inside a container are usually written in a more restricted way that normal components. This restrictions are compensated by automatic usage of container services. The container itself is an abstraction of the surrounding middleware that an application uses. The container concept is made known by its original usage in enterprise application frameworks like EJB [13] and COM+ [6].

A container offers services that are usually cross cutting concerns with the rest of application code. Enterprise containers are specialized for e-commerce business applications. We can reuse the container concept and specialize it to other classes of applications. We are interested to apply this concept to mobile applications. Using a container bring well-known benefits like the natural separation of programmer roles that write functional and technical code. Once the technical concerns are identified, they are coded and debugged and made of the container logic. This reduces the bugs and increases productivity. Apart of this the container idea is especially benefiting for mobile applications. The container hides the middleware related API-s from the rest of the application in a centralized way. Ideally porting a family of mobile applications to a new middleware platform would require only porting the container. This is very important considering the speed the mobile middleware [4] changes today. Even different versions of the same middleware may expose different API-s. The container abstraction can be used to address such mobile software problems in a structured way.

There are several reasons, however, why the enterprise container idea cannot be directly adopted for mobile applications. First, since we are addressing a new domain of application we need to find and parameterize the technical concerns of the mobile applications to be addressed. Second, and more importantly, the design issues involved in building a container have different constrains in the enterprise as compared to mobile applications. For enterprise applications scalability is the main issue. For mobile applications the main issue is performance of the container itself: Every time we introduce a new abstraction, we introduce new layers. Having many layers of abstractions is however unacceptable for mobile applications that run on resource limited devices. Last but not least, current enterprise container technology has several drawbacks [16], which are also unacceptable for the domain of mobile applications, including the inability of such technology to be tailored to application specific needs, its complexity due to lack of tight language integration, as well the implied dependency from a particular component middleware platform, a serious limitation in the mobile domain, where middleware changes rapidly.

Generative techniques in the idea of OMG MDA [5] combined with taking into consideration the cross-cutting nature of technical concerns that a container addresses can be used to implement a mobile container framework. The current research in MDA tries to find ways how to specify such transformations so that they are done automatically. While nowadays we are still far from modeling an entire application instead of writing its code, the MDA ideas are still valid, if we try to organize the source code of an application in a platform independent way that enables automatic transformations. This way we can focus on the main functionality and introduce the technical concerns automatically later. To achieve this we need language support for MDA concepts such as marking with tags and AOP like [1] cross cutting code decoration techniques. Languages used for mobile applications are usually restricted dialects of mainstream languages. Tools that transform mobile source code may not be directly available. This means that we should use techniques that can be easily introduced to any language [1].

[1] For example we cannot use AspectJ [7] with J2ME MIPD, a Java dialect for mobile applications that we use in MobCon prototype.

2 MobCon: A Mobile Container Framework

To demonstrate the feasibility of the proposal, a mobile container prototype called MobCon is being implemented. In the following, we only outline the structure and features of this prototype; more details can be found in [12].

Based on the ides of marking in MDA and presence attributes in several languages like .NET we presented the idea of Generalized and Annotated AST - GAAST languages, that can be used to facilitate marked transformations at the source code level. GAAST idea requires two features to be supported by a language technology (a) support for annotations (marking in MDA) of arbitrary program elements with user-defined tags which are first-class program units with well-defined semantics, and (b) support for explicit meta-representation of programs that is accessible in a programmatic way. A GAAST proposal evaluation in the form of the CT-AST API prototype is developed as part of the MobCon transformer framework.

We selected Java 2 Micro Edition Mobile Information Device Profile (MIDP) 2.0 [3] for our prototype because is relatively simple (we can focus on the container concept not in the technology) and hardware independent and it is well supported. The process of identifying technical concerns for a set of mobile applications is similar to defining software product lines [8]. Various technical concerns have been addressed until now in the MobCon prototype such as data persistence, screen management, logging, image adaptation, encryption, session and context. We use a source code template based approach. Our templates are implemented as Velocity [9] scripts.

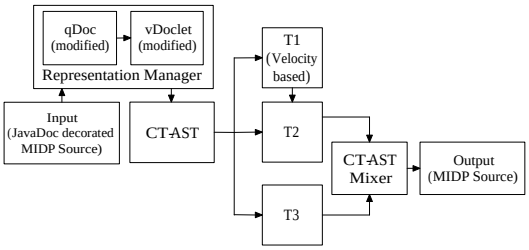


Figure 1: MobCon Transformers

Fig. 1 schematically shows the MobCon transformer framework. As part of this framework, we have implemented a MIDP-based prototype language technology with explicit support for tag-based model-driven development (represented by the Representation Manager (implemented with customized versions of QDox [15] and vDoclet [17]) and the Class Template AST (CT-AST) in Fig. 1). Tags allow us to associate new semantics with language entities. JavaDoc tags are used in our prototype to simulate source code entity annotations, since MIDP lacks such support. The representation manager processes annotated source code, producing a CT-AST representation of it, which serves as the input for the transformers. Only three different such (Velocity-based) transformers (Ti) and their interactions as they process the input introducing the technical concerns directed by the annotations of the input CT-AST-s are shown. Some concerns like image adaptation require code to be placed on the server side. The MobCon framework generates Java code for such concerns, for both mobile side and the server side. The network communication is handled by the framework. Messages are handled to the appropriate application using bookkeeping data, part of session and context concerns. Three types of uses are identified: (a) those that use predefined concerns (tags); (b) those that can append and modify concerns dealing with MobCon transformer framework; (c) users that port the framework to a new set of middleware. The MobCon framework itself is independent of J2ME MIDP and is written in Java. Only transformer scripts written in Velocity are platform dependent.

Generation can be used successfully to implement container-like approaches for small systems. The work of Voelter [11] targets generative component infrastructures for embed systems. Unlike MobCon the idea is that the entire operating system can be customized for the application. PicoContainer [10] is a generic container project based on constructor parameters that tries to define a minimalistic container to be used also in client applications. This project focuses on containers in general, while MobCon is specialized for mobile applications, trying to reduce abstraction layers via generation. The work of Popovici et al [2] investigates the application of so-called *spontaneous containers* in mobile clients. The idea is that the container ideally changes itself to be adapted to environment changes at run-time without stopping the application. MobCon on the other hand addresses ways to structure mobile applications so that they can be maintained easier. Also run-time dynamic AOP requirements of spontaneous containers require more powerful computing devices that those addressed by MobCon framework.

References

[1] A. Rashid A. Colyer, G. Blair. Managing Complexity In Middleware. *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2003.

[2] A. Popovici, et al. Spontaneous Container Services. *ECOOP*, 2003.

[3] C. Bloch and A. Wagner. *MIDP 2.0 Style Guide for the Java 2 Platform, Micro Edition*. Addison-Wesley, 2003.

[4] W. Emmerich C. Mascolo, L. Capra. Middleware for Mobile Computing. *In Advanced Lectures on Networking - Networking 2002 Tutorials, Springer Verlag, LNCS 2497*, pages 20–58, May 2002.

[5] D. S. Frankel. *Model Driven Architecture - Applying MDA to Enterprise Computing*. Wiley, 2003.

[6] T. Ewald. *Transactional COM+: Building Scalable Applications*. Addison-Wesley, 2001.

[7] J. Hugunin M. Kersten J. Palm W. G. Griswold G. Kiczales, E. Hilsdale. An Overview of AspectJ. *In Proc. of ECOOP '01, Springer-Verlag, LNCS 2072*, pages 327–353, 2001.

[8] J. Bosch. *Design and Use of Software Architectures, Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2002.

[9] J. Cole J. D. Gradecki. *Mastering Apache Velocity*. John Wiley & Sons Inc, 2003.

[10] PicoContainer. *http://www.picocontainer.org/*, 2003.

[11] M. Voelter. A Generative Component Infrastructure for Embedded Systems. *Position Paper at Reuse in Constrained Environments Workshop at OOPSLA 03*, 2003.

[12] MobCon: A mobile container framework prototype for J2ME MIDP. *http://www.st.informatik.tu-darmstadt.de/static/pages/projects/mobcon/index.html*, 2003.

[13] R. Monson-Haefel. *Enterprise JavaBeans*. Addison-Wesley, 2000.

[14] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 1972.

[15] QDox Java Tag Parser. *http://qdox.codehaus.org/*, 2003.

[16] R. Pichler, K. Ostermann, M. Mezini. On Aspectualizing Component Models. *Software Practice and Experience, Volume 33, Issue 10, pp. 957-974*, 2003.

[17] vDoclet Java Code-Generation Framework. *http://vdoclet.sourceforge.net/*, 2003.

ACP4IS Workshop at AOSD'04

Towards Integrating Aspects and Components

Houssam Fakih[1,2] fakih@ensm-douai.fr
Noury Bouraqadi[1] bouraqadi@ensm-douai.fr
Laurence Duchien[2] duchien@lifl.fr

March 22, 2004

Integrating aspects and components can be important for both AOSD and CBSD. On the one hand CBSD suffers from crosscutting and tangling code [7]. On the other hand, actual AOSD technologies are not mature enough to enable aspect reuse [6, 4]. So, each paradigm can resolve other's paradigm limitations.

Problem Statement

The integration of AOSD and CBSD is a complex task that can be subdivided into three facets.

1. Facet 1 consists in componentizing aspects. That means representing each aspect as a single reusable component. Basic characteristics of a component include attributes, provided and required services. The challenge is to map these characteristics on aspects. We have also to explore the applicability of related concept such as connector, composite and sub-component on aspects.

2. Facet 2 consists in aspectizing a component-based software. Nowadays, AOSD is used in conjunction with object oriented or procedural languages. Base code is expressed using either an object oriented language or a procedural one. The second facet allows extending this list with component based languages. Set differently, the second facet consists in defining aspects that act on base code expressed in terms of components and related concepts. In this context, we have to define weaving and join points on execution flow and structure of components and related concepts.

It is worth noting that there is a variety of component models. Thus, redefinition of AOSD concepts will certainly vary according to the model used to implement base code. A solution proposed for a flat component model (*i.e.* a model without composite concept) will probably not be applicable to a hierarchical model (*i.e.* a model with composite concept). However, we believe that some solutions could be transposed between some models. In the case of component model with explicit connectors there would be an extra relationship : the one between connectors and aspects. But, this relationship can be transposed to a component-aspect relationship. Indeed, we agree with Sacha Krakowiak that components and connectors are two entities of the same nature (i.e. structure/behavior) but with different roles [5] . So, connectors can be considered as components dedicated to connection.

3. Facet 3 is a merge of the two previous facets. It consists in unifying aspects and component-based software by defining a general enough component model to encompass not only "traditional" CBSD concepts, but also AOSD concepts. This unification should lead to a single definition that should apply for both aspects and components. In this context, weaving aspects with a base code consists of assembling components from a base code with components representing aspects. We identify two differences between assembly and weaving mechanisms.

- First in CBSD, all participating components in an assembly are aware of their assembly points and the provided or required services. On the contrary in AOSD only aspects are aware of assembly points (join points) and services they provide to change or adapt the base code normal execution.
- Second, the assembly mechanism is not intrusive like weaving. The former keeps components intact while the latter often changes base-code structure and behavior.

Note that as for facet 2, solutions will probably vary according to concepts provided by the chosen component model.

First steps towards the integration

1. Provided and required services are among components characteristics. In a componitizied aspect, provided services include advices. Indeed, advices should be triggered in order to execute. Introductions are also part of provided services of a componitizied aspect. This is because their execution can be

[1] Ecole des mines in Douai. France. *http://csl.ensm-douai.fr/research/*

[2] Lille University Of Sciences and Technology. France. *http://www.lifl.fr/GOAL/*

triggered. Introductions are performed when code elements to extend/change are given. One possible solution for facet 1 consists in using the concept of contract [1]. Contracts seem to be applicable for provided services corresponding to both advice and introductions. Syntactic contracts (types) for advice enforce the type of acceptable join points (e.g. message to be sent or to be received, field access, ...). While behavioral contracts for advices check their invariants and pre-post conditions (e.g. change of a log file for a logging aspect). For an introduction, a syntactic contract corresponds to the kind of constructs (class, method, ...) to which the introduction is applicable.

2. One possible solution for facet 2 consist in defining entry points on components [3] that allow to change its internal behavior. Thus, aspects are plugged on these entry points. The definition of join points depends on component model. We can identify some basic join point families common to all component models such as actions related to the component state (creation, initialization, ...) or provided or required component services (call of services, connecting or disconnecting components,...)

3. One viable solution for Facet 3 could be to define a reflective component model (figure 1). We distinguish two kinds of components : a base component defines application business features while a meta-component defines how to perform these features (aspects) [2].

Each component should have an extra-functional interface (an extra-functional interface corresponds to an entry point) allowing it to be connected to a meta component. A meta-component controls one or several base components. A componentizied aspect can be made up of one or several meta-components. So it can be considered as a single (Aspect 2 and Aspect 3) or a composite component (Aspect 1) as we show in figure 1. Composite components assemble more than one meta-component

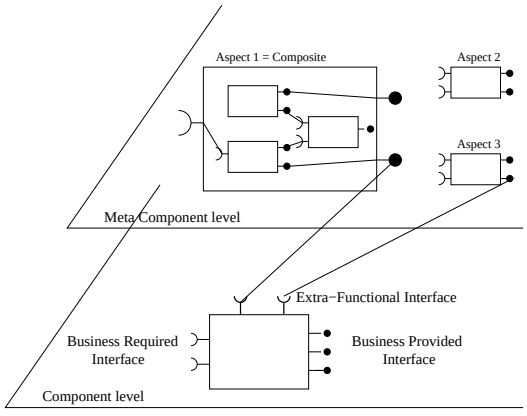


Figure 1: a reflective component model

in order to represent a single aspect. It manages also the visibility of provided and required meta-component interfaces. Representing aspect as a set of meta-components has the advantage to make no difference between assembly components and weaving componentizing aspects mechanisms. In cases of one component controlled by several meta-components. We have to manage potential conflicts among meta-components. One solution consists of either using a chain of responsibility. The ideal solution is to give users the possibility to change or manage conflicts. We could do it by using a meta-component dedicated to conflict resolution. The same strategy can be used in case of conflicts among componentizied aspects. Conflicts can be addressed defining an adapter component that link componentizied aspects to base components.

References

[1] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, jul 1999.

[2] N. Bouraqadi and T. Ledoux. *Aspect-Oriented Software Development*, chapter 11 – Supporting AOP using Reflection. Addison-Wesley, 2003.

[3] Patrice Gahide, Noury Bouraqadi, and Laurence Duchien. Promoting component reuse by integrating aspects and contracts in an architecture model. In Yvonne Coady, editor, *Proceedings of the First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 51–55, Enschede, The Netherlands, April 2002. University of British Columbia.

[4] Stefan Hanenberg and Rainer Unland. Using and reusing in aspectj. *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, October 2001.

[5] Sacha Krakowiak. Patrons et canevas pour les intergiciels. Talk given at 4$^{\text{th}}$ Summer school on distributed systems in Autrans, France, August25 2003.

[6] Karl Lieberherr, David H. Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA 02115, March 1999.

[7] Roman Pichler, Klaus Ostermann, and Mira Mezini. On aspectualizing component models. *Software Practice and Experience*, 33:957–974, 2003.

JAsCoAP: Adaptive Programming for Component-Based Software Engineering.

Wim Vanderperren
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 62
wvdperre@vub.ac.be

Davy Suvée
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 65
dsuvee@vub.ac.be

1. INTRODUCTION

Adaptive Programming [3] aims at providing support for a very different kind of crosscutting concern than the ones tackled by classical aspect-oriented approaches. When an operation involves a set of cooperating classes, one can either localize the operation in one class or split up its logic among the set of involved classes. Localizing the operation in one class causes hard-coded information about structural relationships, this way violating the Law of Demeter [1]. Spreading the operation among the set of involved classes conforms to the Law of Demeter, but gives raise to crosscutting concerns which obstruct evolution. In order to cleanly encapsulate an operation that involves several cooperating classes, Adaptive Programming introduces adaptive visitors that allow visiting the objects contained within an application, without explicitly specifying the structural relationships among these objects.

JAsCo [6] on the other hand, is an aspect-oriented extension for Java which is especially tailored to be employed in the context of Component-Based Software Engineering (CBSE). CBSE advocates low coupling between components and high cohesion of single components [7]. The JAsCo language introduces two new entities, namely *aspect beans* and *connectors*. An aspect bean allows describing crosscutting concerns independently of concrete component types and APIs. JAsCo connectors on the other hand are used for deploying one or more aspect beans within the concrete application at hand. JAsCo connectors also allow managing the combined aspectual behavior of the instantiated aspect beans in a fine grained manner.

Although Adaptive Programming is originally designed for Object-Oriented Software Engineering, its ideas can also be reused within a Component-Based context. Currently available Adaptive Programming realizations, such as DJ [4], DemeterJ [2] or DAJ [5] however are not very suitable to be employed within CBSE. Even though adaptive visitors are independent of the architecture of the application at hand, they still refer to specific component types and APIs, rendering a visitor not as reusable as required by CBSE. To this end, we propose to recuperate the context-independency idea promoted by JAsCo within adaptive programming, this way making adaptive visitors suitable to be employed within CBSE. Furthermore, currently available Adaptive Programming realizations provide little support for specifying complex combinations among several collaborating adaptive visitors in order to execute their behavior simultaneously. Also here, the JAsCo ideas are able to contribute, as JAsCo allows expressing complex combinations among independently specified aspect beans.

In the next section, we show how the ideas of Adaptive Programming and JAsCo can be combined in order to make Adaptive Programming fit into the Component-Based world. We illustrate how adaptive visitors can be implemented by means of JAsCo aspect beans in order to improve their reusability. Afterwards, we demonstrate how the behavior of several adaptive visitors can be combined making using of JAsCo precedence and combination strategies. Finally, we present our conclusions.

2. JASCOAP

2.1 Aspect beans as adaptive visitors

An adaptive visitor is very similar to a set of related advices as it is able to group several before, after and around methods that need to be executed whenever a corresponding component type is visited. Therefore, it seems natural to employ a regular JAsCo aspect bean as a kind of abstract and loosely coupled adaptive visitor. Figure 1 illustrates the implementation of the *DataStorePersistence* aspect bean, which allows capturing an incremental backup of data objects.

```
1  class DataStorePersistence {
2
3    hook Backup {
4
5      Backup(triggeringmethod(..args)) {
6        execute(triggeringmethod);
7      }
8
9      isApplicable() {
10       //returns true if changed since last visit
11     }
12
13     before() {
14       ObjectOutputStream writer = …
15       writer.writeObject(getDataMethod(calledobject));
16     }
17
18     public abstract Object getDataMethod(Object c);
19   }
20
21 }
```

Figure 1: DataStorePersistence AspectBean that specifies a reusable backup aspect

The aspect bean contains one hook, the *Backup* hook (line 3 till 19). The constructor of this hook (line 5 till 7) specifies that the behavior of the hook should be performed whenever the concrete method, bound to the *triggeringmethod* abstract method parameter, is executed (line 6). An *isApplicable* method is employed (line 9 till 11) which returns true if the state of the object, on which *triggingmethod* is executed, has changed since it was last visited. The before method (line 13 till 16) serializes the visited object to file using the abstract method *getDataMethod*. This abstract method (line 18) is responsible for fetching the data

from the current object related to the method call and needs to be implemented in the connector. Notice that the *DataStorePersistence* aspect bean does not refer to specific component types and APIs. As a result, the aspect bean remains completely independent and reusable.

```
1  traversalconnector BackupTraversal(
2    "from system.Root to *") {
3
4    DataStorePersistence.Backup hook = new
5      DataStorePersistence.Backup(visiting DataStore) {
6
7        public void getDataMethod(Object obj) {
8          return (DataStore)obj,getData();
9        }
10   };
11
12   hook.before();
13
14 }
```

Figure 2: BackupTraversal traversal connector.

In order to deploy the *DataStorePersistence* aspect bean as an adaptive visitor, a new kind of connector is introduced: a *traversal connector*. A traversal connector instantiates one or more aspect beans as adaptive visitors onto a traversal strategy. Figure 2 illustrates a traversal connector that instantiates the *DataStorePersistence* aspect bean (line 4 till 10) upon the "*from system.Root to **" traversal strategy (line 1 till 2). The *visiting* keyword allows declaring on which specific type of objects, encountered during the traversal, the behavior of the hook needs to be performed, in this case, *DataStore* objects. As a result, the object structure of an application is traversed as specified by the traversal strategy "*from system.Root to **" and the *before* advice of the *Backup* hook is triggered each time a *DataStore* object is encountered. Likewise to a regular JAsCo connector, the *getDataMethod* abstract method is implemented in order to fetch the data from the visited *DataStore* objects (line 7 till 9).

```
1  public void backup(system.Root mySystemRoot) {
2
3    Connector myBackup = BackupTraversal.getConnector();
4    myBackup.traverse(mySystemRoot);
5
6  }
```

Figure 3: Invoking the BackupTraversal connector.

Traversal strategies need to be invoked explicitly in order to start the traversal. Figure 3 illustrates how the traversal specified in the *BackupTraversal* connector is explicitly invoked (line 4).

2.2 Combinations among Adaptive Visitors

Currently available Adaptive Programming realizations provide little support for specifying complex combinations between several collaborating adaptive visitors. The precedence and combination strategies offered by the JAsCo connector language can however be employed when adaptive visitors are implemented as aspect beans.

```
1  traversalconnector BackupFileLoggerTraversal (
2    "from system.Root to *") {
3
4    DataStorePersistence.Backup hook1 = …
5    Logger.FileLogger hook2 = …
6
7    logger.before();
8    backup.before();
9    addCombinationStrategy(new TwinComb(hook1,hook2));
10
11 }
```

Figure 4: Connector with explicit combinations.

Figure 4 illustrates the implementation of a traversal connector that instantiates the *Backup* hook and the *FileLogger* hook, which logs backup actions, upon the same traversal strategy. The *BackupFileLoggerTraversal* traversal connector allows to explicitly control the precedence of both hooks when they visit the same object. In this case, the before behavior method of the *logger* hook is triggered prior to the before behavior method of the *backup* hook (line 7 till 8). In addition, a log should only be kept of those objects that have been saved to file. For these specific kinds of interactions between hooks, combination strategies can be employed. A combination strategy is a kind of filter which acts on the list of applicable hooks. In this case, the *TwinComb* strategy specifies that the behavior of the *FileLogger* hook should only be performed, if the behavior of the *BackUp* hook was also executed.

3. CONCLUSIONS

This paper illustrates how the ideas behind Adaptive Programming and JAsCo can be combined in order to make Adaptive Programming fit into the Component-Based world. Adaptive visitors are described by means of traditional, context-independent JAsCo aspect beans which are deployed making use of JAsCo traversal connectors. As a result, adaptive visitors, implemented as aspect beans, are now truly reusable as no specific component types and APIs are hard coded into the visitor itself. In addition, the behavior of several adaptive visitors, implemented as aspect beans, can easily be combined into one traversal connector in order to visit the same traversal strategy simultaneously. JAsCo precedence and combination strategies can be employed here in order to describe complex interactions between several adaptive visitors applied upon the same traversal strategy.

4. REFERENCES

[1] Lieberherr, K. and Holland, I. *Assuring Good Style for Object-Oriented Programs*. IEEE Software, pages 38-48., September 1989.

[2] Lieberherr, K and Orleans, D. *Preventive Program Maintenance in Demeter/Java*. In Proceedings of International Conference of Software Engineering (ICSE), pp. 604-605, 1997.

[3] Lieberherr, K., Orleans, D. and Ovlinger, J. *Aspect-Oriented Programming with Adaptive Methods.* Communications of the ACM, Vol. 44, No. 10, October 2001.

[4] Orleans, D. and Lieberherr, K. DJ: *Dynamic Adaptive Programming in Java.* In Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September 2001.

[5] Sung J. and Lieberherr, K. DAJ: *A Case Study of Extending AspectJ.* Northeastern University Technical Report NU-CCS-02-16, 2002. Available at: http://www.ccs.neu.edu/research/demeter/biblio/DAJ1.html

[6] Suvee, D., Vanderperren, W. and Jonckers, V. *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[7] Szyperski, C. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, ISBN 0-201-17888-5, 1998.

Infrastructural support for data dependencies in data-centered software systems

Lieven Desmet, Frank Piessens, Wouter Joosen
DistriNet, Dept. of Computer Science, Katholieke Universiteit Leuven

Lieven.Desmet@cs.kuleuven.ac.be

1 Introduction

The identification of key concerns is crucial for a good application of the separation-of-concerns principle [5, 3]. However, an exhaustive list of all important non-functional concerns and the correct decomposition of software into those concerns is still an open question. Moreover, we believe that some of the important key concerns are application domain or software architecture specific.

Therefore, we argue that in order to provide better infrastructural support, the infrastructure must take into account this architectural correlation. The infrastructure must provide explicit support for describing and enforcing implicit application information, that is specific to the software architecture.

In this paper, we illustrate this idea for the concern of data dependencies in data-centered software systems.

2 Data flow dependencies

In the data-centered architectural style [6], a system consists of a central data structure (representing the state of the system) and a set of separate components interacting with the central data store. The components of a data-centered software system describe a *required* and *provided* dataset, specifying the set of data that a component fetches from or puts onto the shared repository. A correct composed data-centered application is a collection of separate components and a shared repository, with respect to functional data dependencies: every required data item of a component is provided by another component by means of the shared repository.

Figure 1 illustrates a simple, servlet-based [4, 2] e-commerce web application. Three different services are identified within the application: adding a product item to the personal shopping basket, the payment of the shopping order and searching through the website. Each box represents a functional task implemented as a servlet, and the services are pipe-and-filter compositions of several independent tasks. The functional data dependencies are depicted by dotted lines. For each website user, for example, a personal shopping basket is saved at server-side, in the session

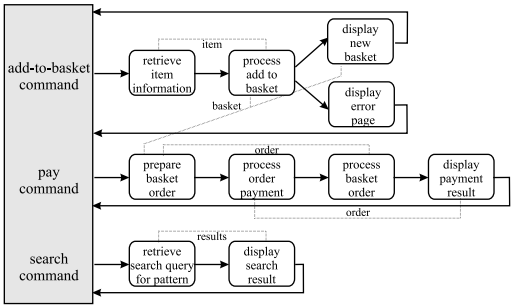


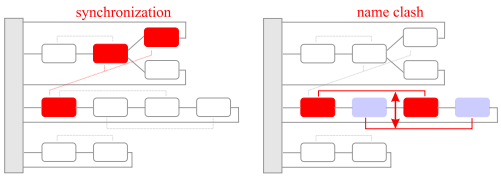
Figure 1: A small e-commerce web application

scope of the shared repository. The personal basket is created at a user's first visit and it is used by three different servlets, entitled in the figure as 'process add to basket', 'display new basket' and 'prepare basket order'.

Besides functional data dependencies, also non-functional requirements on the dataflows may exist. In other words extra constraints on the dataflows through the shared repository may be expressed. These constraints can address for instance the authenticity of the dataflow, confidentiality or synchronization. For instance, in order to prevent race conditions, the composed application needs also extra synchronization support for the shopping basket (figure 2(a)).

Also more general constraints, such as splitting up the shared repository in several disjunct logical repositories, or protecting the repository against name clashes between several dataflows are possible extra composition requirements in data-centered applications. For example, within the payment service, two dataflow dependencies are present. Since the servlets of this service are developed separately (e.g. the payment component is typically a third-party component), a conflict exists in the naming of the shared data. Therefore, extra support is needed to prevent the name clash (figure 2(b)).

From our point of view, data dependencies should be explicitly modelled by the software composer in data-centered application to enforce correctness. Furthermore, data dependencies should be clearly separated from the core functionality in order to improve reusability, adaptability and



(a) synchronization (b) name clash

Figure 2: Data dependency constraints

manageability. In the following section, appropriate infrastructural support for data dependencies is outlined.

3 Infrastructural support

We believe that an infrastructure must provide explicit support for describing and enforcing implicit constraints and dependencies. Therefore, in order to introduce specific support for implicit data dependencies, we have identified the following approach in which three requirements can be defined. Firstly, functional components within the application need an explicit specification extension of the required and provided data items for each component. Secondly, composing an application requires a declarative policy of the functional and non-functional data dependencies between the components. Finally, the enforcement of this declarative policy is needed, either at deployment time or at run-time.

Extended specification Traditionally, an operation is syntactically and semantically specified based on the operation's name and its input and output. In order to express data dependencies, this specification must be extended with extra information about the data items that are provided or required from the shared repository by the component's operation. Extending the specification with repository interactions can be either done manually by the component's designer or implementor, or can be generated by tools based on the component's implementation.

Declarative policy The composition of an application with a shared repository requires more than defining the functional components within the application and the corresponding control flow. The application composer also needs to define the dataflow by means of functional data dependencies and non-functional constraints on the dependencies. Moreover, to enhance adaptivity and manageability, the dataflow should be described in a separate, declarative policy.

Policy enforcement To enforce the dataflow policy at the infrastructure, additional support is needed for controlling the access to the shared repository. As such, the shared repository can be extended with an enforcement engine. Alternatively, a wrapper with a built-in enforcement engine can encapsulate all access to the shared repository. The enforcement engine decides whether a component is allowed or denied access to a data item on the shared repository (functional data dependency). In addition, the enforcement engine ensures the non-functional constraints on the considered dataflow.

In the presented approach, both functional and non-functional data dependencies are clearly separated from core functionality and existing specification of the different components. Furthermore, the data dependency concern is easily adaptable through the use of the declarative policy, and together with the enforcement engine a high cohesion and low coupling is achieved.

4 Current status

Currently, the approach for describing and enforcing data dependencies has been validated in simple servlet-based applications such as the one in Figure 1. Hereby, the data dependencies are identified as an important non-functional, cross-cutting concern and are cleanly separated from the core functionality. The specification and policy language used however, is still in development.

Next to the presented case study, the approach of making dataflow dependencies explicit has also been validated in the component-based protocol stack framework DiPS. In DiPS [1] functional components are chained into a pipe-and-filter structure and components can share data anonymously along the pipe by means of a shared repository. Similar results were achieved within this case study.

Future work will expand the current approach to other implicit constraints and dependencies. Target tracks are the study of dataflow dependencies in Message Driven Beans, and implicit invocations in event-based systems.

References

[1] K.U.Leuven DistriNet Research Group. DiPS home page. http://www.cs.kuleuven.ac.be/cwis/research/-distrinet/projects/DIPS/.

[2] J. Hunter and W. Crawford. *Java Servlet Programming*. O'Reilly, second edition, April 2001.

[3] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[4] Java servlet technology. http://java.sun.com/products/servlet/.

[5] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[6] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an emerging discipline*. Prentice-Hall, 1996.