Quick Adaptation of Web-Based Information Systems with Aspect-Oriented Features

Sašo Greiner, Simon Tutek, Janez Brest and Viljem Žumer

University of Maribor, Faculty of Electrical Engineering and Computer Science, Maribor, Slovenia

Aspect-oriented programming paradigm [4] has proven to be a viable approach to simplifying complex software systems. We are particularly concerned with systems where basic functionality interlaces with more specific and repetitive tasks such as exception handling, logging or message redirection. Aspect-oriented approach enables separation of concerns [19] which are better designed independently, but must operate together. We have extended this approach to distributed enterprise web-based information systems based on J2EE platform.

Keywords: information systems, J2EE, web applications, aspect-orientation, modification, adding functionality.

1. Introduction

Aspect-oriented programming (AOP) paradigm is an emerging discipline which has proven successful in simplifying complex software systems. AOP is a technique to express modular and orthogonal adaptations of existing software components [10]. The main idea behind aspects is separation of concerns in software design. Ideally, aspects are designed and implemented independently and then incorporated into a single environment to operate in a cooperative way. When software matures, it often gets more complicated as well. Modifying or adding new functionality to an existing large-scale system may be extremely difficult or even impossible at times. Aspects may be used to assist in situations like that. Additional functionality can be represented as an aspect which is then interwoven into existing software model.

We show how aspect orientation is used in an enterprise web-based information system [22] which we have been developing for the past two years. The system is built on top of J2EE application environment to achieve maximum scalability, portability, and efficiency. The evolution process of our system is like that of many industrial-strength production systems. Changes and additional demands which have not been present in the design phase, are occurring. Aspects come in where existing functionality must be altered in a transparent way to retain any dependencies between software modules. We have implemented almost all additional functionality through the use of aspects. These include authentication mechanisms, logging facilities, and some presentation aspects.

J2EE servlet techology provides an excellent way to apply aspects to existing functionality of web applications. This is achieved by input and output servlet filtering. A very simple pragmatic example is reroute aspect. If a multitude of web applications reference a single web application to acquire some service, and only some of those applications suddenly need a service with additional functionality, then the reroute aspect is applied in a transparent way only to those applications. Existing applications need not be modified. The HTTP GET request may be viewed as a method invocation where web page within the application represents the method's name, with page parameters as arguments.

J2EE servlet framework provides an efficient and more abstract mechanism for achieving aspects. Whereas aspects at language level may be

realised explicitly through method interposition [17], servlet filtering performs the same task in the background. Classic method interposition is promoted to HTTP request and response interposition. The mapping is almost 1:1, except that for standard method interposition proxy objects and reflection features are the key factors. In addition to HTTP manipulation, language level aspects [3, 6] may be used in conjunction. Aspect-oriented frameworks which enable dynamic weaving [8, 9] obviously provide much greater flexibility in defining and applying aspects. Servlet filtering allows chaining, which enables us to successively apply aspect modules which have been designed separately. This also means that we may insert new or remove existing aspect modules from operation.

We will present examples of problematic situations that frequently occur in real-time software systems and are difficult to solve. In our discussion we will remain oriented towards Java and J2EE software platform based on it.

In section 2 we present an architectural overview of J2EE servlet filtering and how it may be exploited to apply certain aspects to existing functionality. In section 3 we present an example of a real world enterprise information system which requires additional functionality, but cannot have its code changed. Section 4 gives some additional possibilites of using aspects and, finally, section 5 concludes with brief notions on related and future work.

2. Architectural View

J2EE is a software platform for large-scale enterprise applications which operate in a distributed object-oriented environment. The primary language of J2EE is Java. Architectural design enforced by J2EE has been successfully applied to many industrial environments with high demands for stability, scalability, and general efficiency.

We are concerned with web applications because the trends nowadays require access and platform transparency. The feature which is at the core of aspect-orientation is message interposition (interception). The message is usually a method invocation [5], but in the case of a web application the message is wrapped within the client request. In order to intercept a message certain mechanisms are needed. In Java, this may be achieved with AspectJ. In web environment, servlet request and response filtering [7] are used. Servlet filtering mechanism is closely related to composition filters [1, 2]. It should be stressed that request and response interception cannot realise all possible aspects, but they can be used to implement most of them. We have two join points – *before* and *after*, which are used to apply additional or alternative functionality (Figure 1).

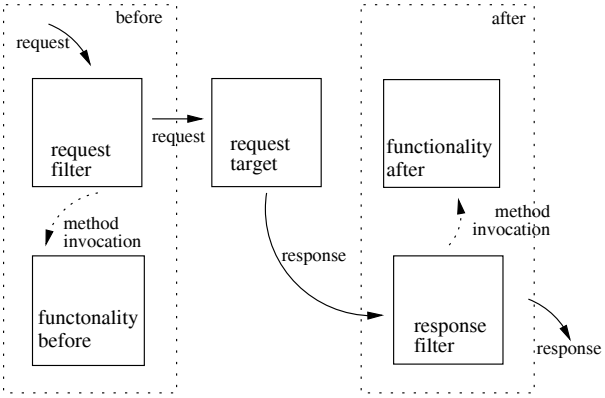

Fig. 1. Request/response-based join points for aspect application `before` and `after` target functionality.

Point *before* executes advice before the request reaches its target destination. Similarly, point *after* captures response after the request has been processed and just before it is sent to the client. Both points are realised within the `doFilter` method:

```
public void doFilter(ServletRequest r,
                     ServletResponse s,
                     FilterChain c)
       throws IOException,
              ServletException{

                // before join point

                chain.doFilter(r, s);

                // after join point

              }
```

To ensure simple and efficient management of software, the architecure must be designed in a component-oriented fashion. This is also reflected in the design and implementation of filters. Because filters may be very complex and diverse in their functionality, they should be designed and deployed as independent modules. To achieve successive applications of multiple filters, they are placed in a filter chain. If we represent filter with α, we may write filter chain with n filters as

$$\alpha_1 \oplus \alpha_2 \oplus \ldots \oplus \alpha_n$$

or shortly

$$\prod_{1 \leq i \leq n} \alpha_i$$

Filter chain does have some dynamics built into it, because it allows modification at run time. Existing filters may be removed or altered and new ones inserted (Figure 2).

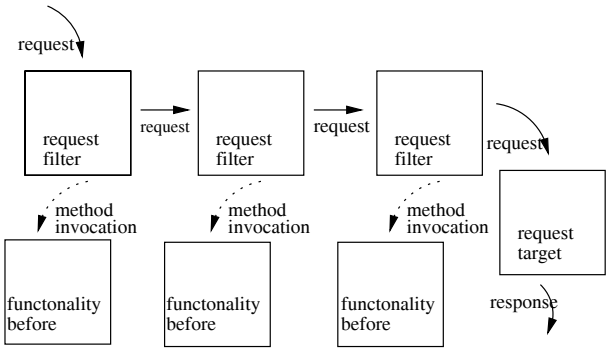

Fig. 2. Request-based filter chain.

Filters are used to model aspects. But, instead of subjecting all servlets and/or JSP pages within a single web application to a specific aspect, only certain servlets/JSP pages may have their advices realised. This is because not all pages in an application need the same treatment. Security aspect, for example, may be applied to the controller servlet in an MVC [20] application, but not to other JSP pages. Another example is the logging aspect which is applied only to action servlets, i.e. those that actually perform some action. It would make no sense to apply logging aspects to every single page or servlet in an application. Selection of destination where aspects will be applied is done through standardised web application configuration `web.xml`. Join point *before* may also be used to implement `instead-of` pattern that comes in handy at message redirection which may then be used in applying alternative functionality.

3. Rapid Adaptation with Aspects – a Practical Approach

3.1. Motivation

High availability of information infrastructure is cruical. Because the total cost of downtime in an enterprise environment is potentially devastating, the companies cannot afford such fallouts of business. When changes in existing functionality, either because of failures or system inefficiency are to be made, they should be applied as rapidly and transparently as possible. Availability is sometimes so important that adaptation must be carried out with system remaining in operation.

3.2. Implementation

To demonstrate how existing functionality may be altered we will present an example of insufficient authorisation mechanism. Authorisation rules of our current mechanism are group-based, which allows only very coarse access control. While designing certain application bundles, finer access control was required. As the information system has already been deployed in production, this demand would contradict our existing security architecture design. The amount of deployed applications would make any changes at basic levels of information system unacceptable. In order to retain the system as it is, we opted to implement additional authentication control as an aspect which is then applied on top of all other aspects. This is done with filter chaining. As more granular authentication scheme is necessary only for certain applications, this aspect is applied only to them. Web application configuration is as follows:

```
<web-app>
   <description>
     Storage application.
   </description>
   ....
   <!-- filters -->
   <filter>
      <filter-name>
         Authenticator
      </filter-name>
      <filter-class>
         system.auth.Authenticator
      </filter-class>
   </filter>

   <filter>
      <filter-name>
         StorageAuthenticator
      </filter-name>
      <filter-class>
         storage.auth.StorageAuthenticator
      </filter-class>
      </filter>

   <!-- mappings -->
   <filter-mapping>
      <filter-name>
         Authenticator
      </filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>

   <filter-mapping>
      <filter-name>
         StorageAuthenticator
      </filter-name>
      <url-pattern>/actions.do</url-pattern>
   </filter-mapping>

   <welcome-file-list>
      <welcome-file>
         actions.do?action=showMyStorage
      </welcome-file>
   </welcome-file-list>
   ....
</web-app>
```

Each filter is defined by putting its name and class between `filter` tags and its mapping between `filter-mapping` xml tags. Authentication mechanism uses the `before` join point of HTTP request. Current authentication is based on groups of people. This means that if a person belongs to a specific group and access rights of that group for a specific application are granted by the authenticator, that person is allowed access to application. Additional authentication is now transparently applied to a person that has already been authenticated by the global mechanism. Second authentication is user-based, which means that not all users may access full application functionality, even though they may belong to a group which is otherwise allowed access. If the authentication should fail, an exception is raised and message delegated to the error page. Yet additional authentication rules (location-based or time-based) could be implemented in a similar manner. It is significant to notice that this addition is applied transparently, without any intervention in existing code. Existing Java classes need not even be recompiled, because filtering does not modify code, but operates on request information instead.

4. Rapid Adaptation – Additional Possiblities

4.1. Manipulation and Validation of Request Parameters

Request signature consists of destination name and parameters. This is very similar to method signature in Java and other languages. The behaviour of method is usually controlled by the values of its parameters. The same applies to request parameters which are passed as `name=value` pairs. Filtering allows manipulation of parameter names and their values. This is again very useful in retaining consistency between disparate web applications which were designed in the past, but must now interoperate with additional functionality. Instead of altering the application itself, its request may be filtered and parameters manipulated to conform to newly defined signature.

If parameters may be modified, they can also be validated. This means validation of parameter names and their values.

4.2. Message Redirection

Message redirection is a well known pattern which proves to be very useful when one wants to transparently implement alternative functionality. This is achieved by rerouting aspect which maps original message to an alternative one.

4.3. Around Advice

Performance evaluator is an aspect that takes advantage of both `before` and `after` join points which can be used to form the `around` advice [21]. Performance is measured on account of

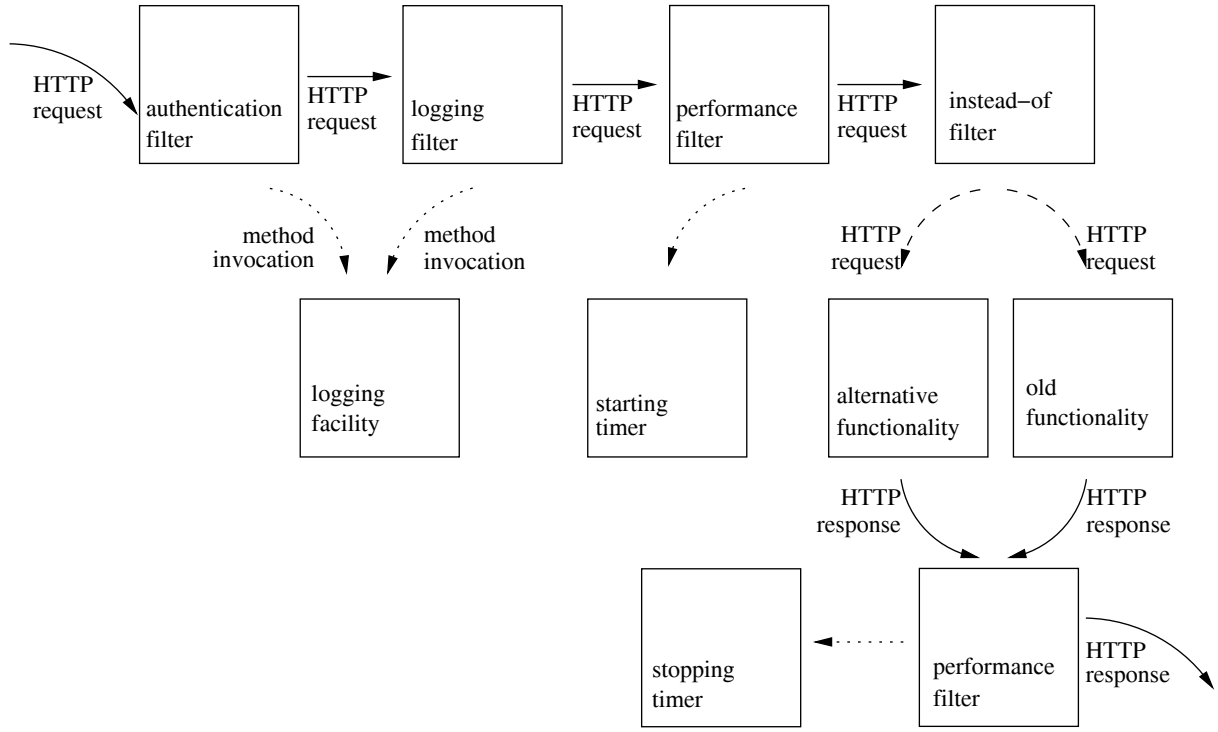

Fig. 3. Filtering facility for authentication, logging, and performance measurement.

time spent in processing the actual request. The design of filter servlet enables quick and effective implementation of such measurements, because single method (`doFilter`) is used for filtering both request and response.

4.4. Putting It All Together

To efficiently implement all desirable aspects, we first need to properly design the filtering scheme. This includes resolving any potential conflicts that might occur after filters become operational. Another factor that should be considered is overall efficiency. Because of additional functionality, filters may significantly increase response times when designed poorly.

Our design of all aspects put together is shown in Figure 3. First filter in the chain is the authentication filter which is responsible for allowing or rejecting users to specific application. Next is the logging filter which logs actions performed. Next is perfomance filter which implements the `around` advice to measure response times. This is used mainly during development stage. Fourth filter is implementation of `instead-of` pattern which enables switching between multiple functionalities. After the request is processed, the prerformance filter is reactivated. It stops the timer and calculates performance indicators.

5. Conclusion and Future Work

We have shown how aspect features may be employed as a mechanism for quick adaptation of existing functionality to an environment with new demands. Adaptation had to be carried out rapidly, without altering old code. Our real-life example demonstrates how separation of concerns using aspects not only gives cleaner approach to application development, but also provides more effective and modular adaptation.

Using aspect features in a web environment is very different from frameworks which operate at a language level, either through preprocessing or more dynamic mechanisms. HTTP is a stateless protocol, whereas ordinary method invocation may take advantage of all dynamics that language offers. Aspects have become popular with reflective features [12, 13] which are usually achieved with some sort of meta object protocol (MOP) [12, 11, 18] built into the language itself.

Nevertheless, aspect-orientation and its features have been used in web environments and distributed architectures. This is especially true for systems where security is among the principal concerns [16, 15].

Our future work will include design and implementation with even more dynamics in applying aspects to existing applications. This should obviate the need to write certain aspects from scratch, because they could be parametrised. This means that aspects are written in a more abstract way and then concretised with actual tasks. Dynamic features can also be added through the use of Java reflection mechanisms, which should allow true run-time modification of aspects themselves.

References

[1] M. AKSIT, L. BERGMANS & S. VURAL, An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach, *ECOOP '92*, LNCS 615, pp. 372–395, Springer-Verlag, 1992.

[2] M. AKSIT, B. TEKINERDOGAN & L. BERGMANS, Achieving Adaptability through Separation and Composition of Concerns, *Issues in Object-Oriented Programming*, pp. 12–23, 1997.

[3] G. KICZALES, E. HILSDALE, J. HUGUNIN, M. KERSTEN, J. PALM, W. G. GRISWOLD, *An Overview of AspectJ*, Lecture Notes in Computer Science, vol. 2072, pp. 327–355, 2001.

[4] G. KICZALES, J. LAMPING, A. MENHDHEKAR, C. MAEDA, C. LOPES, J. LOINGTIER, J. IRWIN, Aspect-Oriented Programming, *Proceedings European Conference on Object-Oriented Programming*, vol. 1241, pp. 220–242, Springer-Verlag, 1997.

[5] E. AVDICAUSEVIC AND M. LENIC AND M. MERNIK AND V. ZUMER, AspectCOOL: An Experiment in Design and Implementation of Aspect-Oriented Language, *ACM SIGPLAN Notices*, 36(12): 84–94, December 2001.

[6] R. HIRSCHFELD, AspectS-Aspect-Oriented Programming with Squeak, *Lecture Notes in Computer Science: International Conference NetObjectDays, NODe 2002*, Erfurt, Germany, October 7–10, 2002.

[7] S. ALLAMARAJU, C. BEUST, J. DAVIES, T. JEWELL, R. JOHNSON ET AL., Professional Java Server Programming: J2EE 1.3 Edition, Wrox, September 2001.

[8] A. POPOVICI, T. GROSS, G. ALONSO, Dynamic Weaving for Aspect-Oriented Programming, *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, April 2002.

[9] K. BÖLLERT, On Weaving Aspects, *International Workshop on Aspect-Oriented Programming at ECOOP99*, 1999.

[10] A. POPOVICI, T. GROSS AND G. ALONSO, Dynamic Homogenous AOP with PROSE, *Technical report*, ETH Zurich, Department of Computer Science, March 2001.

[11] S. CHIBA, A Metaobject Protocol for C++, *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10): 285–299, October 1995.

[12] G. KICZALES, J. RIVIRES AND D. G. BOBROW, The Art of the Metaobject Protocol, MIT Press, 1991.

[13] T. BREUEL, Implementing Dynamic Language Features in Java using Dynamic Code Generation, *In Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, TOOLS 39*, pp. 143–152, 2001.

[14] M. O. KILLIJIAN, J. C. RUIZ-GARCIA AND J. C. FABRE, Using Compile-Time Reflection for Objects'State Capture, *Lecture Notes in Computer Science*, vol. 1616, pp. 150–156, 1999.

[15] B. DE WIN, B. VANHAUTE AND B. DE DECKER, Security Through Aspect-Oriented Programming, *Network Security*, pp. 125–138, 2001.

[16] B. DE WIN, J. VAN DEN BERGH, F. MATTHIJS, B. DE DECKER AND W. JOOSEN, A Security Architecture for Electronic Commerce Applications, *SEC*, pp. 491–500, 2000.

[17] J. PRYOR AND N. BASTAN, A Reflective Architecture for the Support of Aspect-Oriented Programming in Smalltalk, *ECOOP Workshops*, pp. 300–301, 1999.

[18] N. BOURAQADI-SADANI, T. LEDOUX, F. RIVARD, Safe Metaclass Programming, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98*, ACM SIGPLAN Notices, vol. 33, no. 10, 84–96, October 1998.

[19] R. J. WALKER, E. L. A. BANIASSAD AND G. C. MURPHY, An Initial Assessment of Aspect-Oriented Programming, *International Conference on Software Engineering*, pp. 120–130, 1999.

[20] F. BUSCHMANN, R. MEUNIER, H. ROHNERT, P. SOMMERLAD, M. STAL, Pattern-Oriented Software Architecture: A System of Patterns, John Wiley and Sons, 1996.

[21] B. DE WIN, B. VANHAUTE AND B. DE DECKER, How Aspect-Oriented Programming Can Help to Build Secure Software, Informatica 26(2002), no. 2, pp. 103–243, 2002.

[22] S. GREINER, D. REBERNAK, B. BOŠKOVIČ, J. BREST, V. ŽUMER, Web Information System Based on Open Source Technologies, *ITI 2003: Proceedings of the 25th International Conference on Information Technology Interfaces*, June 16-19, Cavtat, Croatia, 2003.

Received: June, 2004
Accepted: June, 2004

Contact address:
Sašo Greiner
Simon Tutek
Janez Brest
Viljem Žumer
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17
2000 Maribor, Slovenia
Phone: ++386-2-220-7458
Fax: ++386-2-2511-178
e-mail: saso.greiner@uni-mb.si
simon.tutek@uni-mb.si

SAŠO GREINER graduated in 2002 and is currently a teaching assistant at the Faculty of Electrical Engineering and Computer Science of the University of Maribor. His research focuses on object-oriented programming languages, compilers, computer architecture and web-based information systems.

SIMON TUTEK has been employed as technical staff in the Laboratory for Computer Architecture and Programming Languages at the Faculty of Electrical Engineering and Computer Science of the University of Maribor since 2003. His fields of expertise are computer architecture, programming languages and web-based information systems.

JANEZ BREST graduated in 1995, received his M.Sc. in 1998 and his Ph.D. in 2001 from the Faculty of Electrical Engineering and Computer Science of the University of Maribor. He has worked in the Laboratory for Computer Architecture and Programming Languages since 1993. He researches web-oriented programming, parallel and distributed computing focused on task scheduling. His fields of expertise embrace also programming languages and optimisation research.

VILJEM ŽUMER is a full professor at the Faculty of Electrical Engineering and Computer Science of the University of Maribor. He is the head of Laboratory for Computer Architecture and Programming Languages. His fields of expertise are: programming languages, parallel and distributed computing and computer architecture.