# Adding high availability features to server applications using aspect oriented programming

RÓBERT FAJTA
*Nokia Research Center, Budapest, rfajta@gmail.com*
PÉTER DOMOKOS, ISTVÁN MAJZIK
*Budapest University of Technology and Economics, Dept. of Measurement and Information Systems*
*{pdomokos, majzik}@mit.bme.hu*

***Keywords: AOP, fault tolerance, legacy software, FT-scheme, AspectJ, HTTP server***

*The transformation of existing software to a fault tolerant one typically requires redesign and heavy modifications in the original source code. Aspect-oriented programming (AOP) is an emerging programming paradigm that promotes collecting features that are not related to the business logic, into crosscutting concerns, thus separates them from the original problem domain of the software. We analyzed how to use AOP to add fault tolerance to existing software by organizing the software into recovery blocks or N-version programming fault tolerance scheme. We gathered practical experiences by modification of a complex application software.* **(In: —)**

## 1. Introduction

Software fault tolerance (FT) techniques are designed to allow a system to tolerate software faults that remain in the system after its development. An important class of software fault tolerance techniques utilizes independently developed but functionally equivalent *software variants* (often called versions). Independent design and implementation of variants is aimed at reducing the probability that more variants contain the same error(s), since the variants will act as redundant components masking or replacing each other in case of an error is activated on a given input. Recovery blocks (RB) [1] and N-version programming (NVP) [2] are well-known software FT schemes. They are used in critical applications ranging from embedded control systems to availability- or security-critical server applications in carrier grade products.

• The RB scheme implements a passive redundancy approach. It contains at least two variants of the same functionality, an acceptance test and an executive. Receiving a request, the executive establishes a checkpoint by saving the state of the application, then executes the primary variant and invokes the acceptance test on the result of the variant. If the result is acceptable then this will form the output of the FT scheme. In case of a failure reported by the acceptance check, recovery (restoration of state) is performed, and the next variant is invoked. Variants are called in this way until a valid result is found. If no variant can provide acceptable result then a failure is reported.

• The NVP scheme implements an active redundancy approach. It contains at least three variants for the same functionality, a voter and an executive. The executive invokes all the variants in a parallel way forwarding to them the input data, then collects the output results and forwards them to the voter. If a majority result exists then it will be the result of the NVP scheme, otherwise a failure is reported. Accordingly, the majority of fault-free variants mask the output of an erroneous one.

There are several extra requirements towards the software that is to be used as a variant in an FT scheme. The communication with the environment, the use of global variables, the implementation of error handling etc. are restricted since *all input and output shall be strictly controlled by the executive* as presented above. Accordingly, variants shall be implemented by taking specific rules into account.

If a service implemented by an existing software (called in the following as *legacy software*) is to be transformed to a fault tolerant one then a natural idea is to *use the existing software implementation (or parts of it) as a variant in the FT scheme*. (Besides this, another variant(s) have to be implemented as well.) Unfortunately, legacy software typically does not satisfy the requirements mentioned above, thus *re-design or modification* become necessary. These modifications usually mix up the concerns of the original functionality (business logic) and fault tolerance, as source code snippets for error detection, fault handling and redundancy management have to be modified or inserted into the original code.

When transforming an existing service to a fault tolerant one, we face the following challenges:

• *Establishing rules to form a variant and modifying the legacy software to satisfy these rules.* To do this, a technique was needed that allows performing these modifications in such a way that is easy to review, verify and change if necessary. Direct modification of the source code was not feasible from this point of view, as clear *separation of the original business logic and the FT extensions* was needed.

• *Elaborating the core logic of the FT scheme* in such a way that is easy to re-use if another service is to be transformed to a fault tolerant one.

There are several approaches that aim at separating functional and non-functional (e.g. fault tolerance) aspects. Library calls to pre-defined mechanisms [3], reflection [4], and meta-object protocols [5] are mature and well-tried techniques that address the separation of func-

tional and dependability requirements. We opted for the emerging paradigm of *aspect-oriented programming* [6]) due to the following reasons:

• AOP provides a clear separation of non-functional activities (like redundancy management) by supporting the *modularized implementation* of the crosscutting concerns. In case of library calls, for example, the non-functional activities can be collected in a library, however, the calls to library functions are scattered in the original code.

• AOP has better support (considering programming languages, development and debugging environments) than reflection and meta-object protocols in our application environment.

• Moreover, the previous techniques do not allow fine grade parameterization of the modifications, e.g. by supporting name-based, property-based, location- or caller-specific modifications.

There are several approaches to AOP like the Multi-dimensional Separation of Concerns [7] supported by Hyper/J or the use of Composition Filters [8]. We follow the concepts and terminology of *AspectJ* [9] since in our case there is no need to handle multiple decompositions and the composability of aspects; only a modularized specification of the scattered behavior is needed.

The facilities of AspectJ AOP can be summarized as follows (the interested reader is referred to an introduction published in this journal [10]). The solution for fault tolerance as a crosscutting concern is provided by code snippets (*advices* in AOP terminology) that affect the behavior of the original software during execution.

These advices are applied at specific locations of the source code (called *join points*) as designated by AOP language expressions called *pointcuts*. Pointcuts can refer to call or execution of given methods, set or get of attributes etc. A `before` advice (`after` advice) may be executed before (after, respectively) a specified execution point (e.g. method call). An `around` advice may replace the original code at the join point. (Note that the original code can be executed by using a `proceed` statement in the `around` advice.) In this way, both the *extension of the behavior* and the *replacement or masking* of the original behavior is possible. *Aspects* modularize the pointcuts and advices, and can add members to existing classes/interfaces, as well.

For example, in *Figure 1* the call of the original *service1()* method (on the left) is designated by the *callService1* pointcut (on the right of the Figure). This pointcut is used in an `around` type of advice that replaces in run time the *service1()* method call with a different piece of code including a call to *otherService()*. The pointcut and the advice are modularized in an aspect called *MaskService*.

AspectJ as aspect language enables the entire feature set of the Java language and its libraries, while it adds the new language constructs. Its compiler (weaver) merges the aspects with the original code and generates ordinary Java class files. We examined the implementation of RB and NVP schemes on the basis of legacy software by using AspectJ AOP. After summarizing the general rules to form an FT variant (Section 2) our paper discusses the following:

• *Advantages and limitations of AOP to modify legacy software* to form a variant for an FT scheme (Section 3). AOP is used to insert additional functionality that is needed, and to replace existing behavior (code snippets) that are not allowed in order to satisfy the rules concerning the setup of FT variants. By using AspectJ AOP, the majority of the necessary modifications can be performed. However, it turned out that not all modifications can be implemented by applying the existing language constructs of AspectJ. In these specific cases the direct modification cannot be avoided.

• *Implementation of the core control logic of the FT scheme in a re-usable form* by using AOP (Section 4). Here the language constructs of the AOP are used to integrate the control logic (as a separate aspect) with the business logic of the application.

These results and experiences were gathered in a project in which a real-life application software, an HTTP service was transformed to an FT one. The details of this application are presented in Section 5.

## 2. Rules to be satisfied by a variant

The executive of the FT scheme shall effectively manage both the calling of the variant and the change of the global state outside of the variant. To be able to do

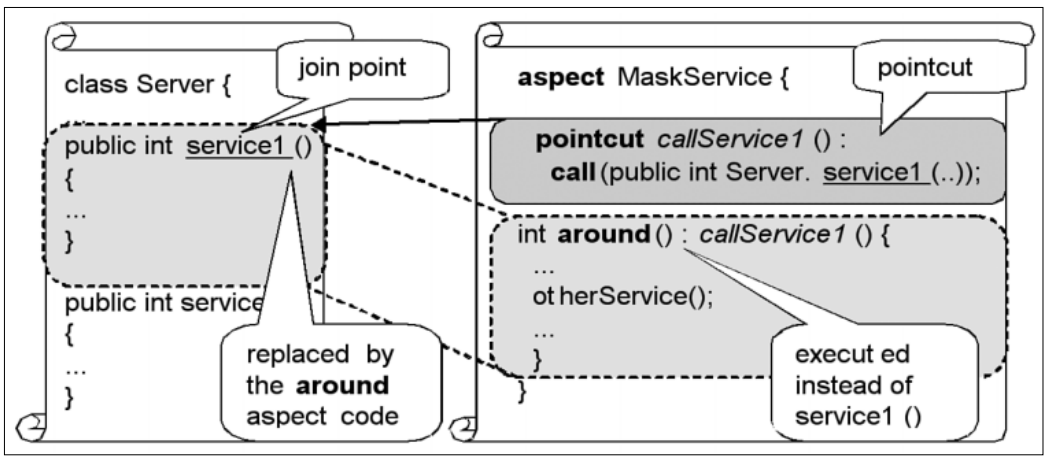


*Figure 1.*
*Masking a method by using AOP*

so, the functionality provided by a variant shall be available through an external interface in the form of a set of distinct methods. The service of the variant shall be provided by the return values of these methods. Moreover, the following side effects are not allowed:

– changing the global state of the application by modifying global (shared) variables,
– sending or receiving messages,
– using volatile or non-deterministic values,
– performing individual error handling (besides returning error codes or throwing exceptions).

The following section presents how these rules can be satisfied by applying AspectJ aspects to the legacy Java code. Naturally, the identification of the locations where these rules might be violated requires the availability of the program source code (the utilization of bytecode modifications is left for future work).

Two restrictions can be derived even before a detailed analysis:

– If the service to be made FT is not available by a set of methods but spreads across the software then the direct code modification cannot be avoided.
– AspectJ pointcuts are based on attribute operations and methods, and finer level join points cannot be specified. Accordingly, no modification is possible at the level of instructions and control-flow operations. If a rule is violated at this level then the AOP based modification is not possible.

## 3. Forming a variant from the legacy code

The violations of the rules presented in Section 2 can be handled by AOP in the following cases:

**Changing the global state.** There are two alternatives to avoid the potentially inconsistent change of global variables by the variants:

• *Emulation of a separate environment* for each variant, which contains a copy of the global variables that the variants would modify. Each modification of the original variables made in the variants must be redirected to the emulated variables by AOP advices. `Set` and `get` pointcuts referring to the original variables should be defined where a variant attempts to access the original variables. `Call` pointcuts should be used to pick out those points of the variant from where the original variables are accessed via internal setter or getter methods. Additionally, an `around` advice should be attached to these pointcuts to redirect the write and read operations to the emulated variable set. At the end of the successful execution, the emulated global variables of a variant that provided proper result must be copied to the original global variables.

• *Splitting up the variants into functionally equivalent blocks* that do not modify global variables, and performing a local adjudication process among the blocks before executing the modifications *(Figure 2)*. However, this splitting is not always possible, e.g. if the variants access the global variables in different order.

**Sending and receiving messages.** The legacy software may establish a connection to receive or send messages. These messages get out of control of the FT executive (e.g. by changing the state of the receiver).

The proposed solution is using aspects to redirect message readings and writings from/to the communication channel to buffers. Incoming messages have to be stored in these buffers by the executive and variants can receive messages from the buffer. The outgoing messages are written to the buffer and they are forwarded to the original target by the executive after the successful execution of the variants.

A possible realization with AspectJ is the definition of `call` pointcuts to pick out the method where the variant attempts to retrieve the input and output streams of the communication channel. `Around` advices are attached to these pointcuts to create the buffers in the advice and attach piped input and output streams to them, respectively. A thread is created to store the messages from the communication channel to the input buffer, and another thread is used to replay the contents of the buffer to the variant. The advices return with these piped

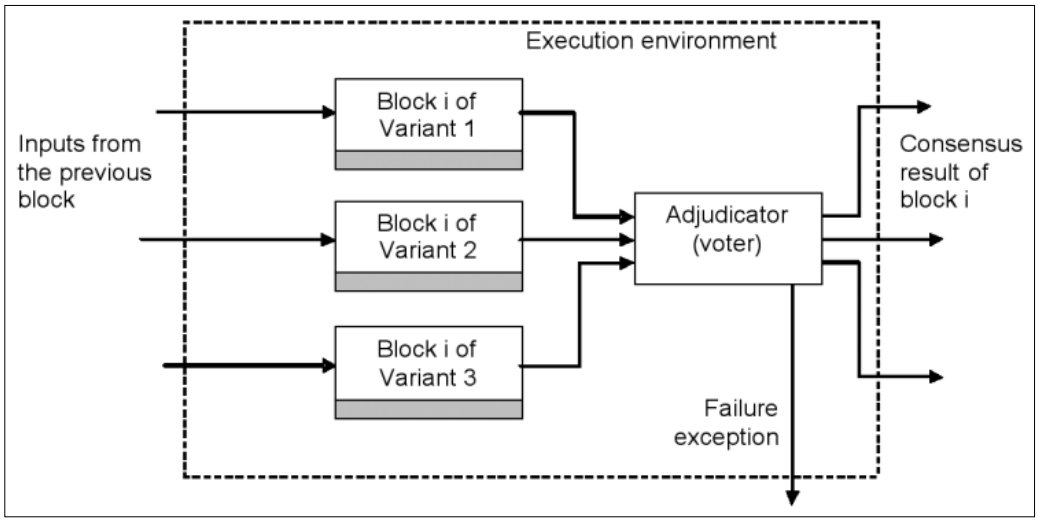


*Figure 2. Local adjudication among blocks in an NVP pattern*

streams. (Threads are necessary because piped input and output streams attached to each other must be handled in separate threads in order to avoid deadlock.)

If the original software expects an answer to a message, then the variants must be split into blocks (if possible) in which no message is sent/received. The executive applies local acceptance tests (in the case of RB) or voting (in the case of NVP, Figure 2) and then sends the messages and waits for the answer to be processed in the next block.

**Error handling including exceptions.** The legacy software may contain error handling code. This error handler shall be masked only if it prevents the proper execution of the FT scheme. E.g. an error handler that aborts the execution of the variant by returning null value or throwing an exception to indicate an error should not be modified. Instead, the executive can examine the return value or catch the exception to recognize that the variant failed. However, if the error handler takes actions that have effects outside of the variant (e.g. it sends messages or terminates the program execution), then the error handler must be masked by an advice as follows:

• If the error handler is located in a *distinct method*, then this method can be suppressed by picking it out using an `execution` pointcut to which an empty `around` advice is attached.

• If the error handler in the legacy code is realized as an *exception handler*, then a possible solution is attaching an `around` advice on each call that can throw an exception caught by this exception handler, and surrounding the `proceed` call in this advice by a *try-catch* block. Thus the exception does not reach the original exception handler.

## 4. Implementing the FT scheme

Conceptually, the FT executive and the wrapper code snippets that are applied to form a variant from the legacy code are implemented in aspects.

The executive of the FT scheme will represent the services of the legacy software (now forming one of the variants) towards the clients, i.e., the calls to the original legacy methods shall be redirected to the FT executive. To do this, these method executions are picked out by `execution` pointcuts that reveal their parameters. An `around` advice is attached to these pointcuts to mask the direct invocation of the original method and execute the control logic of the FT scheme instead. The original method is invoked from this advice by the `proceed` statement of AspectJ and the parameters are passed to it. If necessary, other variants and the adjudicator (voter or acceptance test) are called as well. The setup is presented in *Figure 3*.

In the following some specific problems of the implementation are summarized:

**Timeout checking.** If timeout checking of the call to the original method is implemented in the client then the executive shall translate this timeout value to timeout values for the individual variants that are executed and it shall check their timeliness.

**Termination support.** The variants shall implement a cooperative mechanism to allow stopping them in case of timeout, since Java does not support forced thread stopping (`Thread.stop()` became deprecated).

**Reporting the failure of the FT scheme.** If the failure of the variant(s) cannot be tolerated by the FT scheme then the executive has to report the failure to the caller according to the caller's expectation (e.g. by providing an error code or throwing an exception).

**Checkpointing and recovery.** In the case of the RB and the serially executed NVP schemes the legacy code must be analyzed to find out whether it modifies the state of its environment during execution. The state prior to modification must be saved and later restored.

**Design of adjudicators.** The acceptance test of the RB or the voter of the NVP shall not contain fault handling code (for example, exception throwing) since fault handling must be provided solely by the executive of the FT scheme. Existing fault handling code in a legacy adjudicator shall be masked by an AOP advice.

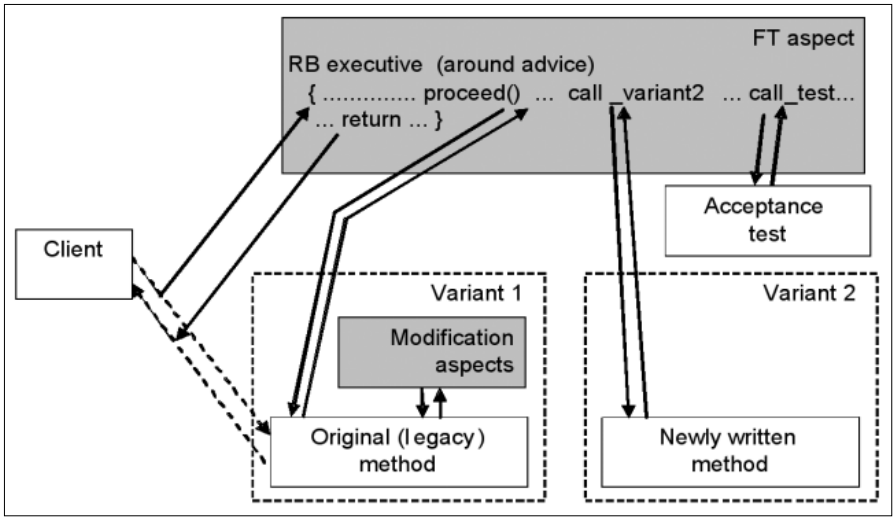


*Figure 3. Implementation of the FT scheme*

*Re-use of the aspect-based modularization* of the FT scheme is possible as follows:

- The core control logic implemented by the advice belonging to the executive is directly reusable.
- The application specific parts of the executive are formed by those functions which communicate with the client, call the original variant, handle checkpointing and recovery, and call the adjudicator. These parts must be adapted.
- Adjudicators are typically application-specific thus they cannot be re-used in original form.

The AOP based implementation of the FT scheme does not introduce significant extra cost besides the known costs of the FT solution (i.e., costs of developing additional variants and adjudicator, cost of increased execution time). Moreover, the same dependability bottlenecks are present as in the case of the traditional implementation: faults in the executive and the adjudicator shall be tolerated separately.

## 5. Implementation experiences

The concepts described in the previous sections were applied in the case of Sun Microsystems' Open Service Gateway Initiative (OSGi) [11] implementation, called Java Embedded Server (JES) that provides a framework for multiple applications (called bundles).

One of the applications is the HTTP Service. It can be used in web based servers as well as in mobile phones to provide extra services for configuration or download. The HTTP Service provides a registration service for other bundles in the framework to enable them to register their own resources and/or Java servlets. The HTTP Service acts as an HTTP server which analyzes the incoming request and maps it to a registered resource or servlet and delivers the resource or the result of the servlet to the client.

Our task was to make this OSGi HTTP Service fault tolerant by applying the RB scheme. The original HTTP Service implementation (referred to as *jesHTTP*, see *Figure 3*) is considered as one variant, and another implementation (referred to as *nHTTP*) is developed as the second variant.

The fault tolerant HTTP service implementation (referred to as *ftHTTP*) forms a single bundle. After starting the service, it is ready to accept servlet and resource registrations and unregistrations, and client connections. As it uses jesHTTP as a variant, there are several functions that the ftHTTP must implement, and some others that have to be masked in jesHTTP. The latter are not parts of the HTTP service itself, but are responsible for starting up and shutting down the bundle. Moreover, servlet initialization and destruction were implemented in the FT executive and were masked in the jesHTTP variant. They belong to the servlet's life cycle and must be performed exactly once. Note that failures occurring in the servlets as external units are out of the scope of the RB scheme.

During the modification of jesHTTP, the problems of changing the global state, and sending and receiving messages had to be handled.

**Changing the global state.** The global state of the HTTP server is represented by the registration database, the session data and the servlet context. In our implementation the registration database and the session data are in the scope of the variants, and their consistency is to be ensured by the executive.

In this way the complex operations of handling the registration database and the session data are performed under control of the RB scheme.

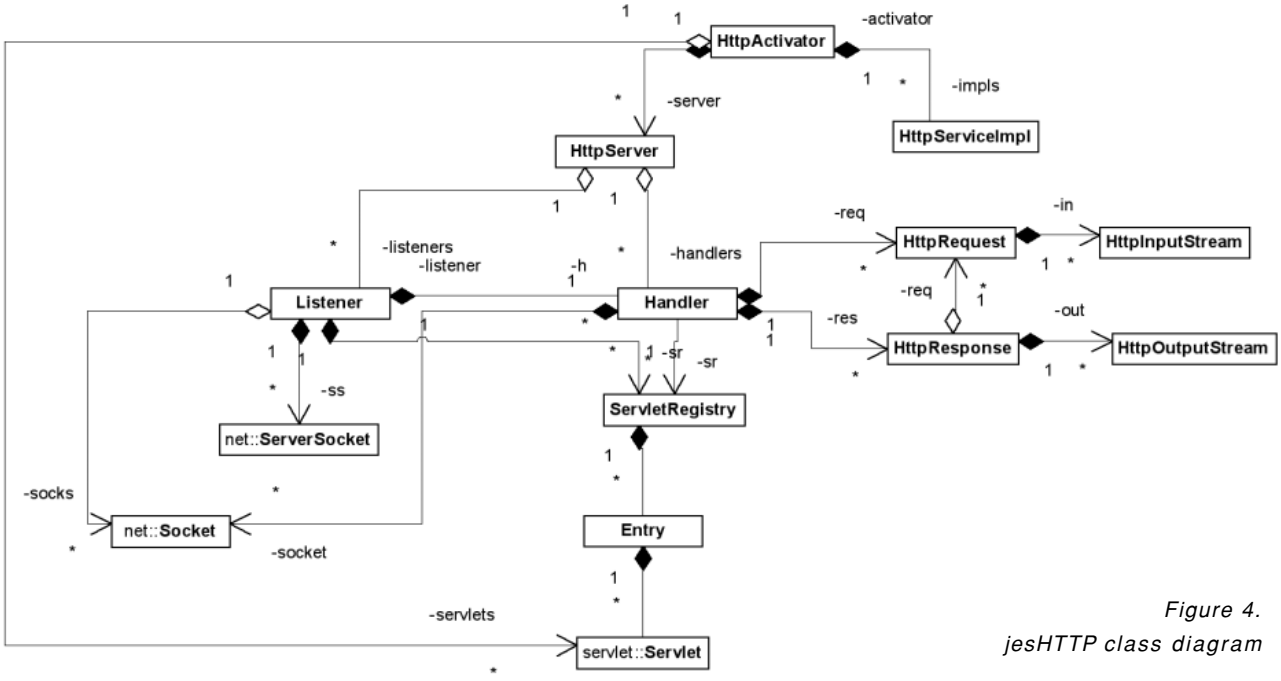


*Figure 4. jesHTTP class diagram*

*Registration database:* Each variant has to maintain its own registration database. Consistency is ensured by forwarding each registration related request to each variant *(Figure 4)*. If a fault occurs in a variant, e.g. it removes the wrong entry upon an unregistration request, it will not be able to serve the requests regarding the removed entry, instead, it will report an error. However, the other variant may be able to serve the request. In the case of a common registration database this kind of fault tolerance could not be achieved. The drawback of this solution is that the registration of a servlet or a resource becomes slower since multiple registrations are performed. However, this is a rare event under normal operating conditions.

*Session data:* Sessions are used to store client-related data between requests. They are manipulated by servlets (session data can be used for communication between servlets). Session data must be available to the servlets independently from the currently running HTTP server variant. Before the execution of the first variant, the session data is saved in a checkpoint. If the first variant succeeds, the checkpoint is discarded. If the variant fails, the savedsession data is reloaded into the next variant. This is the recovery step of the RB scheme. After the successful execution of this variant, its session data is used in the following. In fact, the session data is considered as global data that is passed to the variants upon execution (as in the case of the emulated environment).

*Servlet context:* It is part of the servlet configuration maintained by the HTTP server. If a servlet sets an attribute of the servlet context, it must be possible to retrieve this attribute later, even if the servlet is executed by another variant. The execution of the servlet is atomic from the viewpoint of the HTTP service. The implementation of the context management (setting and getting variables) is rather straightforward, therefore it was not put under control of the RB scheme. Although the servlet context belongs to global data, it is not involved in the checkpointing, because it is not the variant but the servlet that manipulates the context data.

The ftHTTP initializes the servlet with its own servlet configuration implementation that can store references to other servlet configurations. An advice is used to suppress the servlet initialization in the jesHTTP variant, and store a reference to the jesHTTP servlet configuration in the ftHTTP implementation.

If a variant retrieves the servlet configuration, an advice captures this request and it returns the object that was stored in the ftHTTP servlet configuration for the corresponding variant.

Access to the attribute handling methods of the servlet context contained by the servlet configuration of the variant is redirected to the common implementation. This way, if a variant adds extra functionality to the servlet configuration or the servlet context, it can continue to use the extra functionality; while attributes that are part of the servlet context specification remain common between all variants. However, if a variant adds extra functionality to the servlet configuration or the servlet context, it must be analyzed whether it interferes with the services defined by the interface. If it is so, then this interference must be managed.

**Sending and receiving messages.** The process of the original servlet registration is modified by an advice to suppress creating and starting a listener that would accept connections on a server socket. In the FT implementation, connections are accepted in the ftHTTP executive. The first step of serving a request is storing the request in a buffer, so that it can be replayed to the variants.

The implementation applies advices that replace the input and output streams in the jesHTTP variant (when they are retrieved from the socket) with piped input and output streams. The piped input stream reads from the

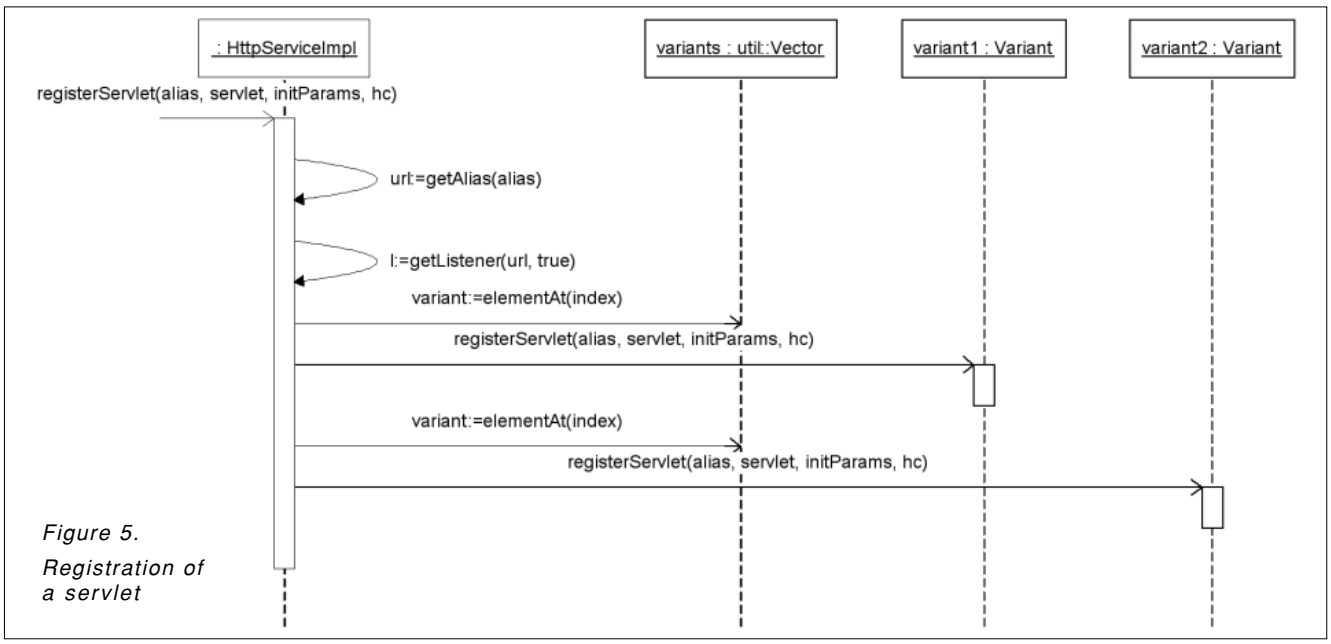


*Figure 5.*
*Registration of a servlet*

request buffer, the piped output stream writes to the response buffer, thus the variant does not use the streams of the socket directly. To compose the response, the variant invokes the servlet or retrieves the resource. The response of a successfully executed variant will be committed to the socket.

**Implementation of the acceptance test.** The acceptance test of the RB scheme examines the value of the response code returned by the HTTP server variant. If the response is "OK" then the variant can commit its result to the socket, thus the browser will receive the response to its request. Otherwise, the executive invokes the next variant that will compose its own response. Since the second variant is the last one, the HTML page generated on the basis of the response of the second variant will be forwarded to the client independently whether it is an error page or a proper web page.

**Summary of changes.** The original source code was changed at 19 points, where the visibility of classes and fields had to be made *public* for referencing them from aspects. These changes could not be made from aspects.

All other necessary changes, namely 5 structural and 15 behavioral modifications, were carried out by aspects. Typically, the aim of these modifications was either to prevent the execution of an action (e.g. initialization or destruction of a servlet in the jesHTTP variant), or to modify the data source for an action (e.g. redirecting socket read and write operations to an extra buffer).

The following peculiarities of AOP (that are not fully supported in the other reflective approaches) were utilized:

- Advices can introduce new attributes in legacy classes. E.g. a reference to the common servlet context was added in jesHTTP.
- The execution of an advice may depend on the *caller* method. E.g. initialization of a new ServerSocket was masked only from the legacy software. Note that reflective extensions are typically applied from the viewpoint of the *called* method.
- It is possible to designate the *call of a method* in an AOP pointcut. In this way those methods could also be masked that are not available in source code form (e.g. `System.exit`). In the case of the ServerSocket initialization, this `call` pointcut was parameterized with the name of the caller method.

## 6. Conclusions

The advantages of the AOP approach manifest directly if legacy systems or parts of legacy systems have to be made fault tolerant. In this case the necessary changes can be made and the selected FT scheme can be applied in a *modularized form by aspects*. They can be handled separately and implemented without drastically modifying the original source code.

In our paper we identified the conditions of applying AOP, discussed the problems that arise during the implementation, and proposed solutions. These results may be used to direct the attention of engineers to those points of the legacy code that has to be modified by AOP advices. In our future work we will investigate the insertion of aspects at Java byte-code level to implement the FT executive in case of commercial off-the-shelf (COTS) variants.

**References**

[1] B. Randell, J. Xu:
The Evolution of the Recovery Block Concept.
In M. Lyu (ed.): Software Fault Tolerance.
John Wiley & Sons Ltd, pp.1–22., 1995.

[2] A. Avizienis:
The Methodology of N-version Programming.
In M. Lyu (ed.): Software Fault Tolerance.
John Wiley & Sons Ltd., 1995.

[3] K. J. Birman:
Replication and Fault Tolerance in the Isis System.
ACM Operating System Review, Vol. 19, No. 5,
pp.79–86., 1985.

[4] G. Agha, S. Frolund, R. Panwar, D. Sturman:
A Linguistic Framework for Dynamic Composition of
Dependability Protocols. In DCCA-3,
pp.197–207., 1993.

[5] J.-C. Fabre, V. Nicomette, T. Pérennou,
R. J. Stroud, Z. Wu:
Implementing Fault Tolerant Applications using
Reflective Object-Oriented Programming.
In Proc. FTCS-25, pp.489–498., 1995.

[6] G. Kiczales et. al.:
Aspect-Oriented Programming.
In Proc. European Conf. on Object-Oriented
Programming (ECOOP). LNCS 1241,
Springer Verlag, 1997.

[7] H. Ossher, P. Tarr:
Using Multidimensional Separation of Concerns to
(Re)Shape Evolving Software. Comm. of the ACM,
(44)10, pp.43–50., 2001.

[8] L. Bergmans, M. Aksit:
Composing Crosscutting Concerns Using
Composition Filters. Comm. of the ACM, (44)10,
pp.51–57., 2001.

[9] I. Kiselev:
Aspect-Oriented Programming with AspectJ,
Sams Publishing, 2003.

[10] L. Lengyel, T. Levendovszky:
Introduction to Aspect-Oriented Programming.
Híradástechnika, Vol. LX, 2005/6, pp.18–23.

[11] Open Service Gateway Initiative,
http://www.osgi.org