WEAVING ASPECTS TO SUPPORT HIGH RELIABLE SYSTEMS: DEVELOPPING A BLOOD PLASMA ANALYSIS AUTOMATON

Valérie MONFORT, Muhammad Usman BHATTI, Assia AIT ALI SLIMANE

Université Paris 1 Panthéon Sorbonne
Centre de Recherche en Informatique
90 rue de Tolbiac
75013 Paris, France
Valerie.monfort@univ-paris1.fr; muhammad.bhatti@malix.univ-paris1.fr; assia.ait-ali-slimane@malix.univ-paris1.fr;

Abstract

Among current architectures, Service Oriented Architectures aim to easily develop more adaptable Information Systems. Most often, Web Service is the fitted technical solution which provides the required loose coupling to achieve such architectures. However there is still much to be done in order to obtain a genuinely flawless Web Service, and current market implementations still do not provide adaptable Web Service behavior depending on the service contract. Therefore, our approach considers Aspect Oriented Programming (AOP) as a new design solution for Web Services. Based on both Web Service Description Language (WSDL) and Policy contracts, this solution aims to allow better flexibility on both the client and server side. In this paper, we aim to develop an automaton to analyze blood plasma; Web Services are used for software part of the automaton. Faced by the lacks of Web Services, we propose a concrete solution based on aspects.

Key Words

Web Services, Interoperability, Security, Aspects

1. INTRODUCTION

Companies have to communicate with distant IS, such as, suppliers, partners … they use to exchange data through workflows in heterogeneous contexts. The company for which we are working aims to develop automatons to analyze blood plasma, which means patient data information has to be highly reliable and correct. We are involved in the architecture definition and implementation of one of its automaton. In order to support consequent evolution and successive reutilization of the machines, this company decided to define and promote flexible and adaptable architecture according to the new emerging requirements. In this context, Web Service technology is asked to handle the same features as components from the DCOM, J2EE or CORBA worlds already handle. These features, such as security, reliability, or transactional mechanisms, can be considered as non-functional aspects. Obviously these aspects are crucial for business purposes and one cannot build any genuine Information System (IS) without consideration for them. However, managing these aspects is likely to involve a great loss in interoperability and flexibility. This effect has already been experienced with various middleware technologies. Mostly, middleware delegates these tasks to the underlying platform, hiding these advanced mechanisms from the developer, and then establishing a solid bond between the application and the platform. Thus, a great deal of work is required to make Web Service fully adapted for the industry. Especially, mechanisms in charge of handling non-functional tasks must preserve seamless interoperability.

In this article, we introduce the industrial context and technical choices for applications integration with Web Services. From the limitations of this solution, we propose a solution based on aspects and we explain how to apply this solution with a concrete implementation.

2. INDUSTRIAL CONTEXT

2.1. Description of the Automaton

Figure 1 shows some high level functional domains supported by the automaton, including software and firmware: arrows represent the communication flow. Application displays specific Human Machine Interface (HMI) according to profile and maturity level of the user. Access is allowed or denied according to user profile and protected from unauthenticated usage. However, it is possible to ask for analysis and to receive result with different media as mobile phone, PDA, and Web with specific passwords reserved for laboratory managers and doctors. Non functional services, such as security, reliability, persistency, archiving, multi tasking, and supervision, have to be defined and implemented. Automaton supports some business functions as patient data management and used consumables for plasma blood analysis. Using automaton involves data generation that is analyzed for preventive maintenance. Communication between software and firmware with specific protocol is implemented by using CAN bus [14]. Automaton allows handling of tubes containing the blood plasma. Automaton arms take the blood plasma and use reagents to test coagulation. With this system, blood disorders, such as hemophilia, can be detected.

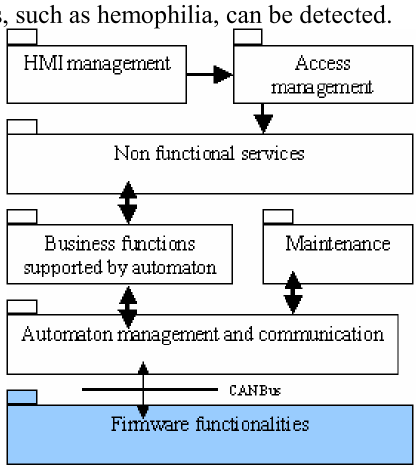


Figure 1: Functional Architecture

Communications between domains can be supported by Web services. Moreover, it might be necessary to exchange patient data and results between different hospitals or other Information Systems (IS). Infrastructures might be based on heterogeneous technologies. For instance, a laboratory uses IBM J2EE technologies and hospital uses Microsoft technologies. Thus, we invoke Web Services developed by different platforms supporting different technologies.

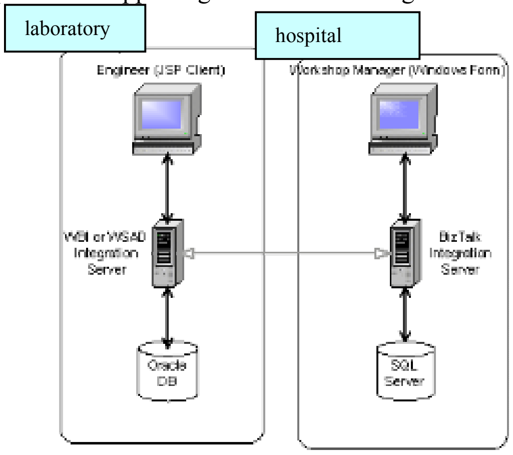

laboratory
hospital
Engineer (JSP Client)
Workshop Manager (Windows Form)
WBI or WSAD Integration Server
BizTalk Integration Server
Oracle DB
SQL Server

Figure 2: Technical Architecture

2.2. Using Web Services

DCOM, J2EE or CORBA don't scale to the Internet: their reliance on tightly coupling the consumer of the service to the service itself implies a homogeneous infrastructure. Web services use industry standard protocols to guaranty interoperability between IS. In order to provide the missing business features required to leverage Web Service technology, a first set of tools has emerged. Built on top of .NET and J2EE platforms, Microsoft and IBM have implemented their own toolkits based on the Web Service specifications. Web Services Enhancements for Microsoft .NET (WSE) [7] is a supported add-on to the Microsoft .NET framework providing developers the latest advanced Web Services capabilities such as security, security policy, addressing, routing, and attachments.

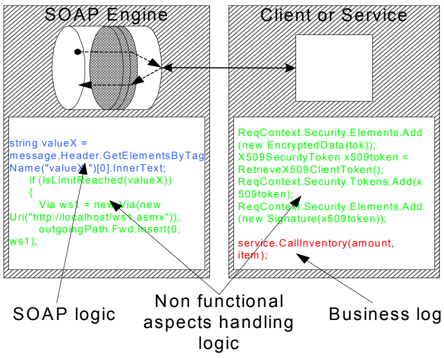
The Emerging Technologies Toolkit (ETTK) [8] is a software development kit for designing, developing, and executing emerging autonomic and Web Service technologies. It provides an environment in which to run emerging technology examples that showcase recently announced specifications and prototypes from IBM's emerging technology development and research teams. Based on Axis [9], ETTK processes messages through handlers in chain. One particular chain enables developers to insert their own message managers, such as security handlers. A MessageContext object is included in outgoing messages and is extracted from incoming messages. The handlers in charge of the transformations are specified in a Web Service Deployment Descriptor (WSDD) file. These toolkits look quite similar in the sense that they operate and compute messages. SOAP Engines are composed of filters (SOAP handlers) whose main role is to perform transformations on the SOAP message [6], depending on parameters included in the header. The SOAP headers are in charge of delivering the context of the message (authentication tokens, reliable messaging properties, etc.).

Our technical approach to current Web Service solutions enabled us to notice two major facts which are at the root of Web Service's lack of flexibility. First, there is no dynamic mechanism to bind policies and Web Service handlers. Secondly, there is no clean separation of concerns [5] between the functional and the non-functional code as well as between SOAP logic and non-functional logic within handlers, as figure 3 shows. Once the client or service is coded and the handlers are deployed, the Web Service cannot handle new features and, because the different logics are tangled up, it is not easy for another developer to reuse the application in a different context.

3. USING ASPECTS

3.1. Discovering aspects

Consequently, an appropriate way to deal with these crosscutting concerns [2] would be to use different units of modularization to encapsulate these logics [4]. Moreover, if these units of modularization could be managed by a dynamic mechanism, then the whole system would be able to dynamically reconfigure itself depending on the policies [1].


```
string valueX =
message.Header.GetElementsByTag
Name("valueX")[0].InnerText;
    if (IsLimitReached(valueX))
    {
      Via ws1 = new Via(new
Uri("http://localhost/ws1.asmx"));
      outgoingPath.Fwd.Insert(0,
ws1);
```
```
ReqContext.Security.Elements.Add
(new EncryptedData(tok));
X509SecurityToken x509token =
RetrieveX509ClientToken();
ReqContext.Security.Tokens.Add(x
509token);
ReqContext.Security.Elements.Add
(new Signature(x509token));

service.CallInventory(amount,
item);
```

Figure 3: Tangled Logic within SOAP Services

These requirements lead us to consider Aspects Oriented Programming (AOP), in the first step, as an answer to Web Services reusability issues [3]. AOP is one of the most promising solutions to the problem of creating clean, well-encapsulated objects without extraneous functionality. It allows the separation of crosscutting concerns into single units called aspects, which are modular units of crosscutting implementation. With AOP, each aspect is expressed in a separate and natural form, and can be dynamically combined together by a weaver. As a result, AOP widely contributes to increased reusability of the code and provides mechanisms to dynamically weave aspects [4].

Considering Web Services, non-functional aspects handling logic should be encapsulated within multiple aspects. Each aspect would be in charge of certain features, such as security, and would deal directly with well-defined objects like Kerberos tokens (security) or Shipping forms (reliable messaging). Pushing the non functional handling logic inside aspects means that handler's role has to be redefined, as they will only contain SOAP logic then. The idea is to replace the multiple specific handlers, which used to process SOAP messages depending on their own implementations, by a global handler whose role will be restrained to extracting non-functional data contained in incoming messages, and pushing it inside outgoing messages.

3.2. Weaving Process

At this point, we need to define where, when and how the aspects should be weaved. Let us answer these questions by considering the different opportunities for each of them. First, aspects could be weaved to the global handler, to the stub or to the service implementation itself. In fact, considering the global message path and process, choosing any of these entities does not really influence the mechanism. However, we found it more convenient to weave aspects to the stub since it provides a natural meta object to focus on the service itself [15]. Secondly, there are multiple choices for when to weave aspects. It could occur during compile time, deployment time, load time or run time. If the weaving were to happen at compile time or deployment time, it would not be possible to handle policy changes dynamically. Conversely, there is no need to weave aspects at runtime since the policy document will most likely not be changed after the service starts running. Thus, the ideal solution is to weave aspects when the service is loaded to enable one single yet sufficient analysis of the policies document for each new instance [11]. Thirdly, the weaver should be an application capable of reading the policy document, interpreting the policies, selecting the relevant aspects and finally mixing them with the plain stub, as can be seen on figure 4.

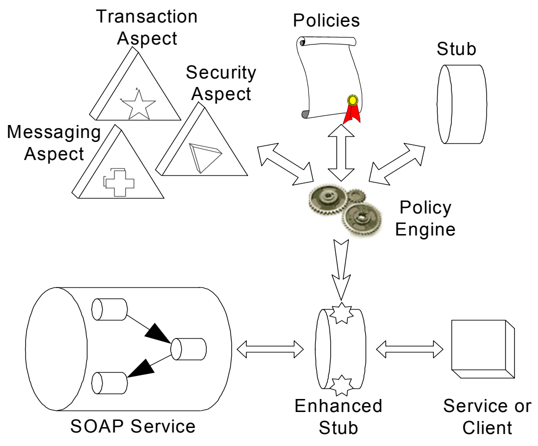


Figure 4. Aspects weaving at load time.

Transmitting non-functional data to aspects weaved to the stub at load time is one possible solution to achieve genuinely flexible Web Services. This mechanism allows Web Services to be reused more easily since each non-functional aspect is detached from both the service implementation and the handler. The Policy Engine inserts these aspects depending on the service contract requirements [16][7], which means that interoperability is preserved if, for instance, requirements from different clients vary.

We have seen how AOP can help to gain flexibility through a cleaner separation of logics and which mechanism can help to provide policy awareness among Web Services. We shall now present our concrete implementation of these concepts.

4. A CONCRETE SOLUTION

4.1. Structure of Axis

In our solution, we take advantage of multiple open source solutions already available for Java therefore we modify and assemble them easily. This way, we can start with a ready-to-use platform that we need to complete in order to obtain flexible Web Services. Thus, the Web Server and the SOAP Engine are constituted by the famous open source duo Tomcat-Axis. Basically, Axis plugs into the Tomcat Servlet Engine, meaning that it can be considered the same as any other Web Application. Web Services are hosted and managed by Axis in a transparent way for Tomcat as shown in figure 5. Axis is based on the concept of a chained message. The MessageContext object is a wrapper object for the request and the responses message and for contextual information about process, request, response, etc. In figure 5, Request and Response are handlers that manipulate the MessageContext.

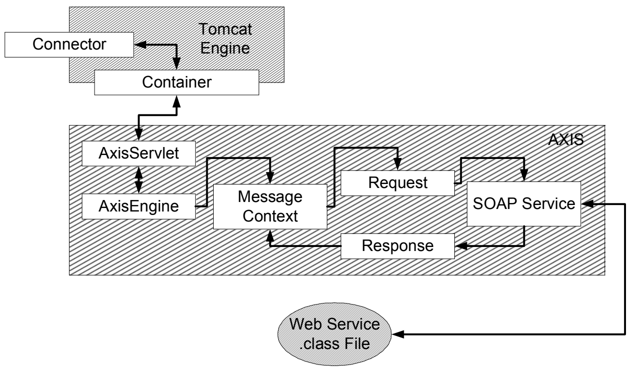


Figure 5. Axis Server-side Architecture.

Since these handlers can easily manipulate this object, it is quite natural to select these handlers to act like basic SOAP logic handler. For instance, if an incoming SOAP header contains data that says the body message is encrypted, then the Request handler needs to decrypt the body. But the genuine non-functional logic is hosted by the aspects, and non-functional data used by these aspects is transmitted by the provider. The provider is another handler that, when invoked, calls the stub corresponding to the service invoked. Once processed and transformed into appropriate objects, these data will be passed to the stub weaved with aspects.

4.2. Stub Bytecode Modifications

Let us now see how aspects are weaved to the stub. First, we need to understand how class loading works in Tomcat. Indeed, if we can modify the bytecode of the stub object when it is loaded into the Java Virtual Machine (JVM), then it will be possible to weave the aspects at load time. Tomcat uses multiple class loaders, which are java objects aiming to load resources (class or jar files). With Java 2, class loaders follow a delegation model, which means that if a class is asked to be loaded by a class loader, then this class loader will first ask its parent class loader to do so. If it cannot load the class, the initial class loader will search inside its own resources. All Tomcat class loaders follow this rule except Web Application class loaders, which are responsible for the loading of each class of the Web Application they are in charge of. Consequently, the idea is to modify the class loader in charge of Axis Web Application so we can reach any Web Service stub anytime it is loaded. To obtain such a class loader, we just need to reuse the code of the Axis regular WebAppClassLoader and specify that Tomcat has to use the ModifiedClassLoader when it loads Axis Web application, via the server.xml configuration file.

```
<Context docBase="C:\axis-1_1\webapps\axis"
path="/axis">

   <Loader loaderClass =
"org.apache.catalina.loader.ModifiedClassLoader"/>

</Context>
```

The next step is to use a tool which allows both introspection and reflection - the former to inspect the stub code when it is loaded and the latter to achieve the weaving of aspects. One particularly convenient answer to these requests is brought by Javassist [1]. Javassist is a class library for enabling structural reflection in Java, which is performed by bytecode transformation at compile time or load time. In order to modify bytecode at load time, Javassist performs structural reflection by translating alterations of structural reflection into equivalent bytecode transformation of the initial class file. After the transformation, the modified class file is loaded into the JVM by a special class loader. To bring this mechanism into our solution, the ModifiedClassLoader must adhere to three rules. First, it must encapsulate a Javassist.ClassPool object, which will act as a container for objects containing class files to be loaded. These objects derive from the CtClass class which is a convenient handle for dealing with class files (methods or fields adds or renames, etc.). Next, when the ModifiedClassLoader constructor is called, this ClassPool object must be instantiated with the Web Application class path so it can get the scope of the classes it can handle. Finally, whenever a class is to be loaded, the findClassInternal (String name) method is called and must contain the transformation logic which will affect the stub object anytime it is loaded. The code below shows these modifications inside of what used to be the regular WebAppClassLoader class.

```
public class ModifiedClassLoader extends URLClassLoader {
  protected ClassPool pool = null;
  public WebappClassLoader() {
    pool = ClassPool.getDefault();
    pool.insertClassPath(new LoaderClassPath(this));
  ...}
  /* Method called whenever a class is to be loaded */
  protected Class findClassInternal(String name) {
    ResourceEntry entry = findResourceInternal(name, classPath);
    Class clazz = entry.loadedClass;
    /* Javassist loader is invoked to get an easily modifiable CtClass
*/
    CtClass cc = pool.get(name);
    /* Class modifications according to the PolicyEngine */
    if(isStubClass("name"))
      PolicyEngine.Process(cc);
    byte[] b = cc.toBytecode();
    clazz = defineClass(name, b, 0, b.length);
    ...
    return clazz;
  }…
```

4.3. Policy Engine as a Weaver

Eventually, we shall define how the Policy Engine works. As explained before, Policies constitute the Service Contract and, thus, describe the requirements to establish communication. For instance, the <wsse:SecurityToken> element, as shown below, is used to describe which security tokens are required and accepted by a Web service. It can also be used to express which security tokens are included when the service replies.

```
<SecurityToken wsp:Preference="..." wsp:Usage="..." >
  <TokenType>...</TokenType>
  <TokenIssuer>...</TokenIssuer>
  <Claims>...Token type-specific claims...</Claims>
  ...  (TokenType-specific details)
</SecurityToken>
```

Once the PolicyEngine.Process(…) method is called, the engine gets a CtClass object containing the code of the stub. Because the name of this class is related to the name of the service itself, it becomes easy for the Policy Engine to locate the Policy contract and thus it can access the policy's requests. The next step for the engine is to fulfill each of these requests by inserting the appropriate aspects within the methods of the stub. This mechanism is almost equivalent for both client and service side. Eventually, the Policy Engine adds fields to the stub so it can obtain and set the non-functional data that the provider manages. At this point, the new "SOAP messages process" is effective and can be used to dynamically handle each of the functional aspects declared in the Policy document. Figure 7 below illustrates the global mechanism at runtime.

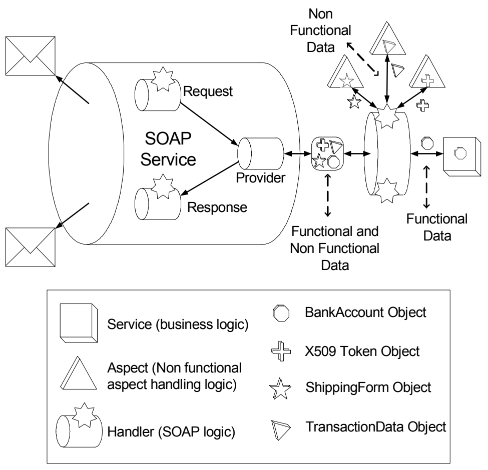


Figure 7. Functional, non-functional and SOAP logics.

5. RELATED WORKS

The Web Service Management Layer (WSML) [10] is an aspect based platform for Web Services allowing a more loose coupling between the client and server sides. The idea of this technology is to transfer the Web Service related code from the client code to this new management layer. The advantages are the dynamic adaptation of the client to find the most fitted Web Service, and it also deals with the non functional properties like Traffic Optimization, Billing Management, Accounting, Security, and Transaction. This work looks very similar to the solution we provide in the sense that it aims to gather the scattered code in aspects. However, our solution especially aims to target the norms of the Web Service Architecture, which are described in the policies. The Web Services Mediator (WSM) [11] is a middleware layer that sits above standard Web Services technologies such as Simple Object Access Protocol (SOAP) Servers. It aims to decouple an application from its consumed Web Service, and to isolate the application's characteristics (e.g., reliability, scalability, latency etc). Aspect-Oriented Component Engineering (AOCE) [12] has been developed to capture the cross-cutting concerns, such as transaction, co-ordination and security. To achieve this solution, the WSDL grammar has been extended by enriching it with aspect-oriented features so that it becomes better characterized and categorized. However, there are no universally accepted standards of the terminologies and the notations used in AOCE. On the whole, AOCE and our work seem to offer very similar approaches but, although just using the policies to select aspects might be restrictive, our strategy does not require developers to understand any vendor specific standard. The Web Service Description Framework (WSDF) [13] consists of a suite of tools for the semantic annotation and invocation of Web Services, by mixing both Web Service and Semantic Web communities. Instead of establishing a hard wired connection between the client and the service, by specifying the Web Services through addresses, WSDF enables the developer to formally specify a service using rules and ontological terms.

6. CONCLUSION

Service Oriented Architectures require loose coupling to access the services which will most likely be implemented with emerging Web Service technology. Using current SOAP toolkits, we noticed that interoperability between client and Web Service is damaged by non-functional aspects required by businesses (such as security, transaction, reliable messaging, etc). In fact, they require establishing a strong coupling between the service logic, the non-functional handling logic, and the SOAP logic. On top of this, there is no dynamic adaptation mechanism to bind the service contract requirements to the Web Service and client abilities. These facts significantly reduce Web Service flexibility and affect the loose coupling ability offered by Services. The solution that we are providing aims to offer a dynamic mechanism to compute the service contract on the fly, enabling Web Services to become fully aware of the business requirements. The main principle consists of using computational reflection [15] as a means to achieve separation of concerns and dynamic adaptability. Our new SOAP Service design provides a cleaner separation between the multiple logics weaved at load time. After analyzing the policies requirements, a Policy Engine is in charge of selecting the appropriate aspects to handle business mechanism like security, transactions, etc. This mechanism allows Services to gain in loose coupling.

Future works will consist of widening the application scope of this solution and validating the Web Services behavior in concrete Service Oriented Architectures. The main tasks will be to implement a library to handle the multiple WS-* norms and then develop a policies fully compliant Policy Engine.

7. REFERENCES

[1] F. Baligand, V. Monfort "A Pragmatic Use of Contracts and Aspects to gain in Adaptability and Reusability" The 2004 2nd European Workshop on Web Services and Object Orientation, EOOWS'04, ECOOP, June 14-18, 2004, Oslo, Norway

[2] M. N. Bouraqadi-Saâdani, R. Douence, T. Ledoux, O. Motelet, M. Südholt "Status of work on AOP at the OCM group, April 2001" , École des Mines de Nantes, technical report, no. 01/4/INFO, 2001 KW: AOP, execution monitoring, program transformation, interpreter

[3] Kiczales G. et al. "Aspect-Oriented Programming", in Proc of ECOOP'97. LNCS 1241, Spinger-Verlag, 1997

[4] Eric Tanter, Jacque Noyé, Denis Caromel, Pierre Cointe "Partial Behavioral Reflection: Spatial and Temporal Selection of Reification", 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003

[5] O. Barais, L. Duchien, R. Pawlak, "Separation of Concerns in Software Modeling: A Framework for Software Architecture" Transformation, IASTED International Conference on Software Engineering Applications (SEA), IASTED, USA, November 2003.

[6] visit web site http://www.w3.org/TR/SOAP

[7] visit web site http://msdn.microsoft.com/webservices/build-ing/wse/

[8] Visit web site http://www.alphaworks.ibm.com/tech/ettk

[9] Visit web site http://www.axis.com/

[10] Verheecke B., Cibrán M.A., "Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection," to be published in the proceedings of the European Conference on Web Services 2004 (ECOWS'04), Erfurt, Germany, September 2004.

[11] Visit web site http://javaboutique.internet.com/articles/WSApplications/

[12] Singh, S., Grundy, J.C., Hosking, J.G. Developing .NET Web Service-based Applications with Aspect-Oriented Component Engineering , In Proceedings of the Fifth Australasian Workshop on Software and Systems Architectures, Melbourne, Australia, 13-14 April 2004.

[13] A. Eberhart. Towards universal Web Service clients. In B. Hopgood, B. Matthews, and M. Wilson, editors, Proceedings of the Euroweb 2002.

[14] Visit web site http://www.ixxat.de/

[15] Chiba, S., "Load-time Structural Reflection in Java" in Proc. of ECOOP'2000, 2000, SpringerVerlag LNCS 1850

[16] D. Mandrioli, B. Meyer « Applying Design by contract » Interactive Software Engineering Inc editions Prentice Hall