JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 2
Special issue: .NET: The Programmer's Perspective: ECOOP Workshop 2003

Dynamic Component Composition in .NET

Anis Charfi, David Emsellem, Michel Riveill, Laboratoire I3S, Bâtiment ESSI, BP 145, 06903 Sophia Antipolis CEDEX, France

Components have brought with them the notion of services which let the programmer concentrate on the business behavior of his component while the non functional part (i.e. the services) is the responsibility of the platform provider. Thus services are not reusable throughout the different platforms; the mechanism used to integrate them in the component is totally platform dependant. In this paper we propose a model to define the integration of services and describe its implementation in the .NET framework. We also discuss the facilities offered by the .NET platform in comparison with the first implementation of this model which was in Java.

1 INTRODUCTION

Nowadays, adding and composing non-functional requirements at deployment time or at runtime have become a ubiquitous way to deal with service integration such as transaction, security, replication and other high-level features.

Deployment in component models. To partially achieve the static service integration, component models such as CORBA Component Model (CCM) and Enterprise Java Beans (EJB) have emerged. Those standards specify how some services can be statically plugged into components. One of the most important contributions in component models is to separate application programming from deployment. Indeed, deployment descriptors allow component programmers to give information about which services to use. Then the deployer has to customize the deployment descriptor in order to adapt the component to the specificity of the runtime environment (transaction, persistency, security, database support, etc.). According to a deployment descriptor, generators provided by platforms generate adequate interposition code. So, the integration of services by the deployer is basically done through a parameter file. As the definition of the services is integrated into the platform, their evolution and composition is handled by the platform. Consequently, we could suppose that the deployer does not have to deal neither with the application code nor with the generated code.

Deployment drawbacks in component models. However, the code generators support only services provided by the platform. For using new services like replicated EJB for example, the deployer must either modify the generated code or specialize the generator if it is open-source [6], [5]. So, it is a difficult task, because service calls must be plugged either in the generated code or in the code to generate. This makes maintenance, evolution and service composition quite impossible to manage. Moreover, as the code integration is predefined, the deployer has no high level way to adapt the code generation nor to control the composition of existing services. This is needed for example in order to modify or to trace access to persistent data. Finally, these component models do not allow neither dynamic service integration nor dynamic customizations of a single component instance.

Reflection and dynamic service integration. Run-time reflection is a powerful technique that is successfully used to implement non-functional requirements such as load balancing, fault tolerance and security [2], [9], [10], [11]. The code that realizes the non-functional requirements is expressed as meta-programs using a meta-object protocol to reflect the base-level behavior. Generally, these works apply to expert programmers able to deal with reflection. So they are often used by the platform providers, who hide these techniques inside the provided libraries. But we have shown that the deployer still needs to partially do the same job to integrate new services for example. However he/she is not necessarily an expert in reflective systems. Aspect oriented approach [**?**], [8], [11] proposes to define code integration as "aspects", providing a way to meta-program in a declarative form

2 OUR APPROACH: DYNAMIC CONFIGURATION BASED ON INTERACTION PATTERNS.

We propose to define dynamic services integration using a rule language. We have chosen to present it throughout examples. Our goal is to intercept invocations at least with similar controls as the ones generated by the Jonas EJB platform, in order to bear out our approach. So, we have defined integration of several services [6]. At the class level, interaction patterns describe all the possible connections among classes of the application. At the instance level, reified interactions represent actual connections among instances. An interaction pattern is expressed by rules (written in a specific language, based on Interaction Specification Language ISL [4]). An ISL interaction rule describes how the behaviour of an object should change when it interacts with another one. It consists of two parts: the left side is the notifying message and the right side is the reaction to that message. In object-oriented languages, behaviours correspond to class methods.

To show how interactions are created and used, we will take as an example the connection between a component and a security manager. Several mechanisms could be used to reach secure execution of an application. As a first step, we have chosen, to check the validity of invocation by a call to a security manager, which raises an exception if the invocation is not allowed. Figure 1 shows the interaction pattern describing such integration. The interaction modifies the behaviour of the JBean component. When this component is accessed by a controlled method and

```
interaction SecurityPattern( JBean B, SecurityManager S) {
    B.* ->  if (S.checkSecurity(_call))
               B._call
            else
               exception("unauthorized user")
}
```

Figure 1: Interaction pattern for dynamic integration of the security service

the security manager forbids the execution an exception is raised, otherwise the call is executed.

The interaction pattern security, when plugged on components, will control any call that can be "unified" with one of the operations to control. The "." operator in left part of the interaction rule denotes the message reception. The operator * stands for any message(method calls) and the operator `->` expresses that the code in right part of the interaction rule is executed as a reaction to the notifying message (i.e. the call to the business method). The method call is reified as an object, which is designed by the operator *_call*.

During execution, an end-user may decide to use this interaction pattern to connect an account instance of JBean (say, MyAccount) to an instance of Security-Manager (say LocalManager) and dynamically integrate the security service on the component MyAccount.

Interaction patterns represent models for component interactions. They contain one or more interaction rules expressing the control that should be executed on the connected components. An interaction rule consists of two parts: the left side is the notifying message and the right side is the reaction. Both sides are separated by the `->` operator. Interaction patterns are specified in the Interaction Specification language ISL. The ISL language allowsthe specification of interactions independently of the application language. It defines many operators such as the conditional operator (if . . . then . . . else . . . endif), the sequence operator (;), the concurrency operator (//), waiting operators, exception handling and others.

Implementation of the interaction model. We have adapted our prototype build for Java component model [1], [3], [4] to the component model offer by the CLI to allow dynamic service integration using interaction rules for all languages supported by the CLI platform. We also attempted to define interactions across different platforms, so that we can connect a .NET component to a Java component by means of interaction patterns.

3 PORTING AN INTERACTION SERVICE FROM JAVA TO .NET

Structure of the interaction service

The interaction service can be split into two main parts. The Interaction Server on the one hand and the component management services on the other hand.

The Interaction Server acts as a central repository for interaction patterns and provides methods for pattern instantiation as well as for rule merging. Rule merging is required when two interaction rules with the same notifying message are applied to a component. The component management services include the execution of interaction rules, the management of interactions (adding, removing, call redirection), inter-component communication (using proxies) and code instrumentation tools such as GenInt.

Our purpose was to extend the interaction model and support .NET components without rewriting the interaction server. Therefore we only ported the component management services to the .NET platform and we reused the Java interaction sever (called Noah).

Independently of the targeted platform interaction rules should be represented as an abstract tree. The tree comprises several node types respectively to the actions that can be specified in ISL (Concurrency, Sequence, if then else etc). Another constraint is communication between .NET components and the java interaction server. It is required during two phases of the interaction lifecycle.

The first time is when an interaction pattern is instantiated. The server needs somehow to talk to the target components and hand them the respective interaction rules. Those rules are available at the server as tree of java objects. They need to be packed appropriately before accessing the .NET world. We have chosen to serialize them in XML. This is a universal format which can be easily handled in both Java and .NET.

The second time communication is needed is when more than one interaction rule is applied to the same notifying message of a component. In this case the rules should be merged so that we have only one reaction for each notifying message. As we said before rule merging belongs to the tasks of the interaction server. Since it is already implemented in Java we wanted to reuse it. Therefore we exposed the merging service to .NET components using a webservice.

Component code instrumentation

An interacting component modifies its behaviour dynamically according to the current interaction rules. This ability has to be acquired by the component and therefore it must be prepared to manage interactions and execute them. The interaction server needs among other things to instantiate and remove interaction rules from the component. Moreover the component should store the interaction rules that affect it. This transformation makes a component ”Noah compliant”. The component class is modified in such a way that it provides an interaction management interface (addRule, removeRule, getBehaviour) as well as wrappers for the business methods (call interception) and additional fields to store the interaction rules.

In Java this task is accomplished by the GenInt tool. It instruments the class bytecode using the BCEL library. We developed a similar tool for .NET components which instruments .NET assemblies.

4 IMPLEMENTING THE INTERACTION SERVICE IN .NET

In this part will we discuss some technical details concerning the implementation of *the component management services* in .NET compared with Java . In addition we show to which extent the .NET platform helped us. We also address some aspects where we think the .NET platform should provide more support to the programmer.

Threading

An ISL tree represents the reaction to a notifying message. Each interacting .NET component should be able to execute reactions. A reaction is an abstract ISL tree with several types of nodes e.g. notify call, global call, assignment, sequence etc. The execution of the tree is multithreading and requires therefore thread synchronisation.

When the executor thread comes to a concurrency node (reaction with several parallel sub-reactions) it starts a new child thread for each sub-reaction. The parent thread blocks and waits until all child threads terminate. The .NET framework provides the Join() method in the class System.Threading.Thread which makes the current thread wait till another thread exits. Java does not provide the Join functionality. For this reason we used the methods wait() and notify() of the class java.lang.Object to get this functionality. We derived a class ReactionThread from java.lang.Thread. A ReactionThread has a reference to its parent thread. This reference is needed to wake up the waiting father thread (call notify()) at the end of the run() method.

ISL trees also include qualified message nodes and waiting message nodes. A qualified message is a labelled message e.g. [1] obj1.method1(). The label enables other nodes to reference the qualified message node and in particular allows waiting message nodes to block till the execution of the qualified message exits.

When the executor comes to a waiting message node e.g. obj2.method2() @X it first checks if the message with label [X] has been executed. If not the executor waits till it gets notified of the end of the execution of the message labelled by X.

In .NET we used the ManualResetEvent class for inter-thread communication. This class notifies one or more waiting threads that an event has occurred. We associate a ManualResetEvent instance with each qualifier (label) in the ISL tree. When the executor visits a waiting message it retrieves the corresponding ManualResetEvent instance and calls the WaitOne() method on it. This results in blocking the executor thread so long as the respective qualified message has not been executed. When the executor visits a qualified message it calls the Set() method which sets the state of the ManualResetEvent object to signalled and releases all waiting threads. Hence the execution of the waiting message resumes.

Java does not provide a similar concept to synchronisation events. For this reason we created our own. The class MessageMutex is used for thread synchronisation in Java; it holds a vector of waiting threads. A MessageMutex object is initially locked. The unlock() method releases the MessageMutex and notifies all waiting threads.

In conclusion the System.Threading namespace in .NET provides many useful classes that considerably reduce the work for the programmer. However these concepts (monitor, lock, synchronisation events, join, . . .) can be also implemented in Java with some additional coding.

Reflection

We used code instrumentation in order to make a class ”Noah compliant”. We developed therefore the previously mentioned tool GenInt. It is available for both Java and .NET. In Java GenInt works at the bytecode level whereas it operates on the MSIL level in .NET. GenInt inserts new fields, methods and constructors to a class and modifies some of its methods and Constructors. The basis for GenInt is the Reflection mechanism.

Compared to Java the .NET framework provides a more powerful reflection API. It is sort of a ”read-write” API in .NET while a ”read-only” API in Java. The System.Reflection.Emit namespace in the .NET framework contains several classes that allow programmers to dynamically create new types at runtime. The TypeBuilder class defines and creates new instances of classes during execution while the classes FieldBuilder and MethodBuilder create new class members. We can even dynamically create code at runtime using the class ILGenerator.

In Java reflection only allows programs to interrogate objects at runtime about their members, their access modifiers and their parent class. Dynamic method invocation is also provided but all the classes and the methods must be defined at compile-time. This means we can neither dynamically create a new class nor even add a field into a given class. Nevertheless, The emitting functionality is not really part of the reflection in the OOP sense but it is rather a bonus in .NET.

On the other hand,we missed some important methods in the .NET reflection API such as a GetMethodBody method in the class MethodInfo that returns an array of IL instructions representing the body of a method. We also expected to retrieve somehow information about exceptions. With .NET reflection we can not find out which exceptions a given method catches or may throw.

```
// a method wrapper in .NET
public double foo (int num, Object obj) {
    ...
    if (rule) {
        //int num is automatically converted to an object
        return reactionExecutor.execute (‘‘foo’’, new object[]{num,obj});
    } else {
       foo_INITIAL (num,obj);
    }
}
```

Figure 2: Type unification in .NET

Another problem is related to .NET Emitting. In fact,when emitting new types is not possible to create a new Type starting from another type. Let us examine this through an example: we have defined a class Foo and we want to dynamically create a new class FooName which is the same as Foo except that it has a string field name more. We want to tell the TypeBuilder object ”do not start from scratch but start from Foo”. Unfortunately this is not possible. Instead we have to traverse all members of Type Foo using reflection (fields, methods, events, properties, constructors and their access modifiers), then we create a TypeBuilder for FooName and consequently add all members of Foo into the TypeBuilder object. Thereby it is easy to copy the field members and method headers but not the bodies of constructors or methods. To achieve this we used a PE file Reader library. We hope that in the next release, PE Reader/Writer classes will be integrated to the .NET framework.

Type Unification

The code instrumentation tool GenInt creates wrappers for the component's business methods. A wrapper intercepts the method call and checks whether any interaction is applied to that method. If yes the reaction to the rule should be executed. That means the method parameters should be passed to the reaction executor (as an array of objects). In .NET every thing is an object. The type system unification provides value types with the benefits of object-ness and thus bridge the gap that exists in many other languages such as Java. This means ValueTypes as well as ReferenceTypes are derived from the ultimate base class System.object. In situations where value types need to be treated as objects, the CLR automatically converts them to objects. This process is called boxing. The reverse process is called unboxing. Both transformations are totally transparent to the programmer.

Java has another approach on data types. It differentiates between primitive types and classes. Primitive types are not inherited from the java.lang.Object class and must therefore be treated specially. Unlike C#, wrapping and unwrapping in

```
public double foo (int num, Object obj) {
    ...
    if (rule) {
        //does the method return a double or a Double
        Class returnClass = double.class;

        //wrap int num to an Integer
        Integer numInt = new Integer (num);

        //method based on reflection
        Object objet = reactionExecutor.execute ("foo", new Object{numInt,obj});

        //get the class of the return object: Double
        ...

        //what shall we return a Double or a double ?
        Double retDouble = (Double) objet;

        //returnClass is double.class so unwrap and return a double
        return retDouble.doubleValue ();
    } else {
       foo_INITIAL (num,obj);
    }
}
```

Figure 3: Wrapping and Unwrapping in Java

java must be managed by the programmer using wrapper classes such as Integer, Double, Boolean etc. Further more if a method (such as the reflection-based invocation method: Object invoke(String methodName, Object[] parameters) returns an Object of class Double for instance, we should be smart enough to know if this Double is a real Double or a primitive double. We have therefore some overhead because we must store somewhere the real return type that we expect.

Language Interoperability

The common language runtime CLR provides the necessary foundation for language interoperability by specifying and enforcing a common type system and by providing metadata [7]. Language interoperability is a great advantage we had while porting the interaction platform from Java to .NET. We did the code instrumentation at the MSIL level and thus we could seamlessly support many languages such as VisualBasic, C#, Eiffel, Cobol and others. This is due to the fact that all languages targeting the CLR follow the common type system rules for declaring and using types. The common type system plays a similar role to the IDL in Corba or type libraries in COM. In fact it is also possible to compile many programming languages to Java bytecode but they can not really share and extend each others libraries simply because Java has no match to the Common Type System.

5 CONCLUSION

This paper describes our experience with the .NET platform. We partially ported an Interaction Service(originally implemented in Java) and now are able to provide that service for .NET components too. This experience was very valuable for us at least in two respects.

First we gained insight into many interesting aspects provided by the .NET environment such as remoting, threading, reflection and Web Services. The rich set of capabilities the .NET framework made our task easier to achieve. In particular the reliable language interoperability enabled us to target many languages such VB.NET, C# and Cobol.

Secondly, by porting the service into a new platform, we worked out a set of core functionalities that must be ported in order to support other component platforms. Thus we have a kind of cookbook that can be used to extend the interaction model to Corba Component Model for example.

REFERENCES

[1] L. Berger. *Mise en œuvre des interactions en environnements distribués, compilés et fortement typés: le modèle " MICADO ".* PhD thesis, Université de Nice - Sophia Antipolis, 2001.

[2] E. Bergmans and M. Aksit. Constructing reusable components with multiple concerns using composition filters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice.* Kluwer Academic Publishers, 2000.

[3] L. Bussard. Towards a pragmatic composition model of corba services based on aspectj. In *ECOOP's Workshop on Aspect and Dimensions of Concerns*, Cannes, France, 2000.

[4] M. Fornarino, A.-M. Pinna, and S. Moisan. Distributed access knowledge-based systems: Reified interaction service for trace and control. In *International Symposium on Distributed Object Applications (DOA 2001)*, Roma, Italy, 2001.

[5] JBoss. http://www.jboss.org/.

[6] JOnAS. Javatm open application server. http://www.objectweb.org/jonas/jonasHomePage.htm.

[7] MSDN library. http://msdn.microsoft.com.

[8] H. Ossher, W. Harrison, F. Budinsky, and I. Simmonds. Subject-oriented programming: Supporting decentralized development of objects. http://www.research.ibm.com/sop/.

[9] R Pawlak, L. Duchien, and G. Florin. An automatic aspect weaver with a reflective programming language. In *Workshop on Meta-Level Architectures and Reflection, Reflection'99*. Springer Verlag, LNCS 1616, 1999.

[10] B. Robben, B. Vanhaute, W. Joosen, and P. Verbaeten. Non-functional policies. In *Workshop on Meta-Level Architectures and Reflection, Reflection'99*. Springer Verlag, LNCS 1616, 1999.

[11] P. Tarr, M. D'Hondt, L. Bergmans, and C. V. Lopes, editors. *ECOOP's Workshop on Aspects and Dimensions of Concern: Requirements on, Challenge Problems For, Advanced Separation of Concerns*, Cannes, France, 2000.

ABOUT THE AUTHORS

Anis Charfi Anis Charfi is a PhD student at the Darmstadt University of Technology. During his master thesis within the Rainbow team he implemented the interaction model in .NET. He can be reached at charfi@informatik.tu-darmstadt.de.

David Emsellem is a research engineer at CNRS/I3S Laboratory, University of Nice. He can be reached atemsellem@essi.fr.

Michel Riveill is professor of computer science at the Université de Nice - Sophia Antipolis. He heads the Rainbow project at the Laboratoire I3S (http://www.i3s.unice.fr). Previously, he was successively Professor of Computer Science at Université de Savoie, Institut National Polytechnique de Grenoble since 1993.He can be reached at riveill@essi.fr. See also http://rainbow.essi.fr/riveill.