Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 166 (2007) 111–125
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Using Microcomponents and Design Patterns to Build Evolutionary Transaction Services

Romain Rouvoy[1] Philippe Merle[2]

INRIA Futurs, JACQUARD Project,
LIFL - University of Lille 1,
59655 Villeneuve d'Ascq Cedex, France

Abstract

The evolution of existing transaction services is limited because they are tightly coupled to a given transaction standard, implement a dedicated commit protocol, and support a fixed kind of applicative participants. The next challenge for transaction services will be to deal with evolution concerns. This evolution should allow developers to tune the transaction service depending on the transaction standard or the application requirements either at design time or at runtime.

The contribution of this paper is to introduce the common approach that we have defined to build various evolutionary transaction services. This common approach is based on the use of microcomponents and design patterns, whose flexibility properties allow transaction services to be adapted to various execution contexts. This approach is applied in our GoTM framework that supports the construction of transaction services implementing several transaction standards and commit protocols. We argue that using fine-grained components and design patterns to build transaction services is an efficient solution to the evolution problem and our past experiences confirm that this approach does not impact the transaction service efficiency.

Keywords: Evolution, transaction service, microcomponent, design pattern, CBSE, Fractal component model.

1 Introduction

Current transaction services do not support evolution well. Among the possible evolutions of a transaction service (e.g., Commit Protocol, Transaction Model), this paper details our approach to address the evolution of the transaction standards and commit protocols supported by a transaction service.

Indeed, nowadays transaction standards are evolving more and more faster to fit with the middleware evolution. For example, the Web Service Atomic Transaction (WS-AT) specification has recently been released to provide a transaction support to Web Services [7]. But the implementation of such a transaction standard requires a legacy transaction service to be modified to support a new transaction API, model

[1] Email: Romain.Rouvoy@inria.fr
[2] Email: Philippe.Merle@inria.fr

doi:10.1016/j.entcs.2006.09.011

and propagation protocol. This is the reason why most transaction services are usually associated to only one transaction standard and the implementation of a new transaction standard results in the development of a new transaction service. In [21], we show that a transaction service can compose several transaction standards simultaneously. This transaction service is built depending on the transaction standards that are composed (e.g., OTS, JTS, and WS-AT). The resulting transaction service shares some common entities between the transaction standards (e.g., commit protocol, transaction status) and provides dedicated entities for the particularities of each transaction standard (e.g., synchronization objects, XA resources).

Similarly, we observe that transaction service implementations are tailored for particular application context. A transactional protocol is chosen and remains the same when the application context changes. This may lead to unexpected poor performances. Therefore, the evolution of the commit protocol is not supported by legacy transaction services. In [22], we show that a transaction service can switch dynamically over several 2 Phase-Commit protocols at runtime depending on the execution context of the application. The resulting transaction service selects the most appropriate protocol with respect to the execution context. It performs better than using only one commit protocol in an evolutionary system and that the reconfiguration cost is negligible.

The evolutionary transaction services described in [21,22] are built using a common approach. The contribution of this paper is to introduce this common approach that we have defined. In particular, we combine the use of microcomponents and design patterns to increase the adaptability of our approach. The notion of microcomponent can represent a pool of components, a policy of message propagation, or a command to execute. The microcomponents are implemented with the Fractal component model [5] and integrated in the GoTM framework. We argue that using fine-grained components and design patterns to build transaction services is an efficient solution to the evolution problem and our past experiences have confirmed that our approach does not impact the transaction service efficiency.

The paper is organized as follows. Section 2 introduces the Fractal component model and the concept of microcomponent. Section 3 illustrates the construction of design patterns using microcomponents. Section 4 discusses the benefits of our approach. Section 5 presents some related work. Section 6 concludes and gives some future work.

2 Microcomponents with Fractal

Our GoTM framework uses Fractal as its reference component model to build its microcomponents. This section first introduces the Fractal component model, then presents the concept of microcomponent in details, and illustrates the use of microcomponents to refactor a logging object.

2.1 Fractal Component Model

The hierarchical Fractal component model uses the usual component, interface, and binding concepts [5]. A component is a runtime entity that conforms to the Fractal model. An interface is an interaction point expressing the provided or required methods of a component. A binding is a communication channel established between component interfaces. Furthermore, Fractal supports *recursion with sharing* and *reflective control* [6]. The recursion with sharing property means that a component can be composed of several sub-components at any level, and a component can be a sub-component of several components. The reflective control property means that an architecture built with Fractal is reified at runtime and can be dynamically introspected and managed. Fractal provides an Architecture Description Language (ADL), named Fractal ADL [15], to describe and deploy component-based configurations automatically.

Figure 1 illustrates the different entities of a typical Fractal component architecture. Thick black boxes denote the controller part of a component, while the interior of the boxes corresponds to the content part of a component. Arrows correspond to bindings, and tau-like structures protruding from black boxes are internal or external interfaces. Internal interfaces are only accessible from the content part of a component. A starry interface represents a collection of interfaces of the same type. The two shaded boxes C represent a shared component.

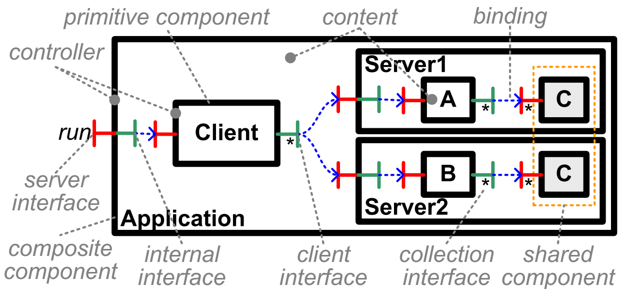

Fig. 1. The Fractal component model.

2.2 Microcomponents

The approach presented in this paper promotes the definition of microcomponents as units of design, deployment, and execution.

A microcomponent communicates with others via its microinterfaces. A microinterface identifies a function provided by a microcomponent. Microinterfaces are interfaces defining a very small set of operations (the empirical statistics performed on the code base show that GoTM microinterfaces define no more than 4 operations) where the operation signature is uncoupled from the operation semantics. If the interface contains too many operations, then it is split into several microinterfaces. The semantics of the microinterface depends on the semantics of the microcomponent that provides it. This approach makes easier the factorization and the reuse of microinterface operations. The definition of microinterfaces offers more modularities to compose microcomponents. Therefore dependencies between microcomponents are expressed in terms of functional dependencies. Then, the configuration operations usually available on an interface are reified as microcomponent attributes according to the separation of concerns property. If this attribute is not primitive or if it is shared by several microinterfaces, then the microcomponent attribute is isolated and reified as a microcomponent and composed with the other microcomponents.

This composition is achieved using an ADL. The composition concern has an important place in the concept of microcomponent. Indeed, microcomponents are not only defined by their microinterface but also by their composition with the other microcomponents. Microcomponents are identified to split a coarse-grained component into several fine-grained components. When composing them, the developer can provide various component semantics by changing only some of the microcomponents. Moreover, it appears that the architecture patterns used to compose the microcomponents can be identified. The remainder of this paper describes the design patterns that are used in the GoTM framework to build evolutionary transaction services. The ADL definitions provide the architectural definition of the design patterns. We show that by modifying the ADL definition, it is possible to make the design patterns evolve to handle different kinds of execution contexts.

2.3 The Logging Illustration

As an example, Figure 2 depicts the object LoggingImpl used by existing transaction services. This object implements an interface Logging containing two operations. The operation write stores in the logs (stable storage) the value of the parameter data. This operation is used by the coordinator and the participants of a transaction to log the progress of the commit protocol. This information is used by the recovery process if a failure crashes the system during the execution of a transaction. There exist two types of log writes: *force* and *non-force*. The force log writes are immediatly flushed into the log, generating a disk access. The non-force log writes are eventually flushed into the log. The use of force or non-force log writes is guided by the value of the parameter force. The operation read is used by the recovery process to analyze the progress of the transactions that were active when the transaction service crashed.

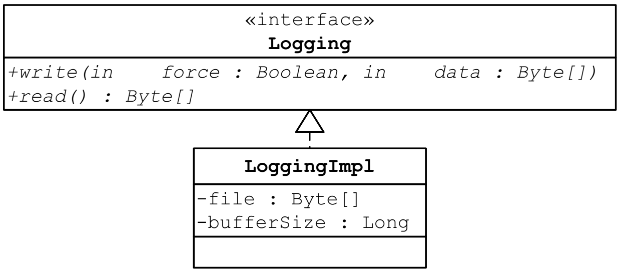

Fig. 2. The logging example.

When considering the semantics of the operations, the logging object can be refactored into several microcomponents and microinterfaces. The resulting microcomponent-based architecture is depicted in Figure 3. The operations write and read are used in different contexts. Therefore, the operations are split into two microinterfaces: LoggingWriter and LoggingReader. Then, the operation write semantics depends on the value of the parameter force. To uncouple the semantics and the operation signature, the parameter force of the operation write is removed. This parameter is replaced by two implementations of the interface LoggingWriter. The implementations correspond to the possible semantics *force* and *non-force*. The piece of code common to the implementations of the interface *LoggingWriter* are placed in a dedicated class LoggingProviderImpl, which implements the interfaces LoggingProvider and LoggingReader.

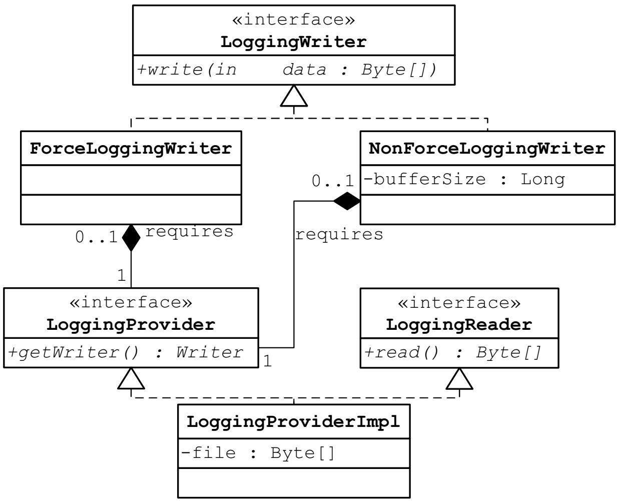

Fig. 3. The logging refactored.

The component Logging uses the sharing capability of the Fractal component model. The microcomponent LoggingProviderImpl is shared between three components: ForceLogging Policy that provides the microinterface *force write*, NonForceLogging Policy that provides the microinterface *non-force write* and Logging providing the microinterface *read*. The microinterfaces *write* provided by the microcomponents ForceLogging Policy and NonForceLogging Policy are exported via the collection interface *write*.

The composition of Figure 4 is described using Fractal ADL. Fractal ADL configurations of microcomponents are automatically generated by the Fraclet tool [20]. Therefore, the component Logging can be defined using the piece of configuration depicted in Figure 5. The definition Logging extends the definition LoggingReader to provide the interface *read* (Line 1). It defines the collection interface *write* with the signature LoggingWriter (Line 2). The microcomponent LoggingProviderImpl, named *provider*, is contained in the component Logging (Lines 3). This component *provider* is shared between the components ForceLoggingPolicy and NonForceLoggingPolicy (Lines 4-9). Finally, the component Logging exports the microinterfaces of the components ForceLoggingPolicy, NonForceLoggingPolicy, and LoggingProviderImpl using

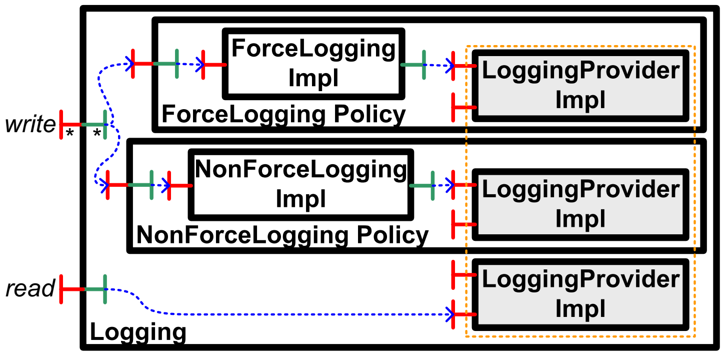

Fig. 4. The component Logging.

bindings (Lines 10-12).

```
 1 <definition name="Logging" extends="LoggingReader">
       <interface name="write" signature="LoggingWriter" cardinality="collection"/>
 3     <component name="provider" definition="LoggingProviderImpl"/>
       <component name="force" definition="ForceLoggingPolicy">
 5         <component name="provider" definition="./provider"/>
       </component>
 7     <component name="non-force" definition="NonForceLoggingPolicy">
           <component name="provider" definition="./provider"/>
 9     </component>
      <binding client="this.read" server="provider.read"/>
11    <binding client="this.write-force" server="force.write"/>
      <binding client="this.write-nonforce" server="non-force.write"/>
13 </definition>
```

Fig. 5. The Fractal ADL configuration Logging.

Thanks to this approach, it becomes easier to make the component Logging evolve. Indeed, additional microcomponents can be added to the Logging component to implement a new semantics (e.g., the *empty write* semantics). The semantics of the component can be dynamically changed by reconfiguring the microcomponents contained in the component. This approach has succesfully been applied to build component-based implementations of several well-known Two-Phase Commit protocols [22]. This allows developers to tune the implementation of components depending on the execution context targeted (e.g., fault tolerance, performance, etc.).

3 Revisiting Design Patterns with Microcomponents

This section introduces the design patterns used in the GoTM framework to build evolutionary transaction services. The configuration and the composition of the microcomponents define the semantics of the transaction service. Therefore, this transaction service can evolve when reconfiguring the assembly of microcomponents.

3.1 Design Patterns Overview

In this paper, we focus on five design patterns used in GoTM to build evolutionary transaction services: *Facade*, *Factory*, *State*, *Command*, and *Publish/Subscribe* [12].

These design patterns are the basis to build any evolutionary transaction service. We illustrate how the evolution of the transaction service is driven by the evolution of the design patterns.

A conceptual overview of an evolutionary transaction service built with GoTM is shown in Figure 6. The design patterns are shared out among the *static* and the *dynamic* parts of the evolutionary transaction service. The *static* part addresses the transaction service itself and supports the *Facade* and the *Factory* design patterns. The *dynamic* part handles the transactions created by the transaction service. This part uses the *Facade*, *State*, *Command*, and *Publish/Subscribe* design patterns. Each of these design patterns are implemented using several microcomponents that can be composed using Fractal ADL configurations.

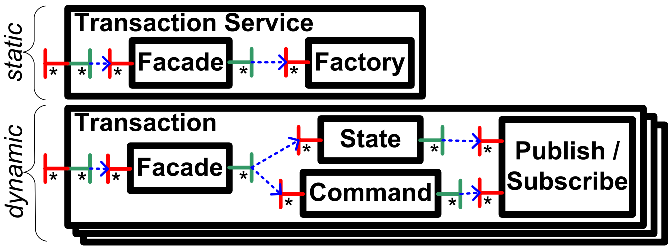

Fig. 6. Overview of the architecture.

3.2 Facade Design Pattern

The *Facade* design pattern provides a high-level unified interface to a set of interfaces in a subsystem to make it easier to use [12]. In GoTM, the *Facade* design pattern is used to conform to a particular transaction standard (e.g., JTS, OTS, WS-AT). The *Facade* design pattern converts the interfaces defined by the transaction standard to the microinterfaces provided by GoTM. Given that a transaction service is composed of a *static* and a *dynamic* part (see Section 3.1), the *Facade* design pattern is applied to the two parts using two components.

The evolution of this design pattern is related to the ability of providing the compliancy with various existing and future transaction standards. Using the *Facade* design pattern, new transaction standards can be easily taken into account. Indeed, this only requires to implement static and dynamic *Facade* components. In GoTM, the *Facade* components can be automatically generated using a model, presented in [19], that describes the mapping between the interfaces defined in a standard and the microinterfaces exported by the GoTM components.

Figure 7 focuses on the static part of the transaction service and depicts a component Facade that provides three facades. The component OTS Facade implements the OMG Object Transaction Service standard [17]. The component JTS Facade implements the Sun Java Transaction Service standard [8]. The component WS-AT Facade provides support for the Web Service Atomic Transaction standard [7]. All of these facades share the component Factory. The component instances created by the component Factory provide also three facades.

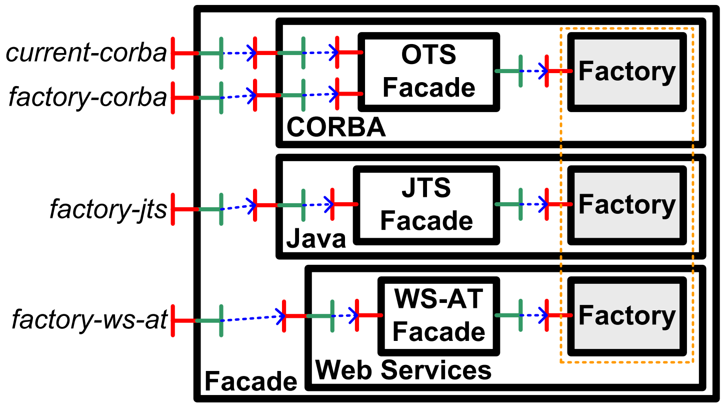

Fig. 7. The component Facade.

3.3 Factory Design Pattern

The *Factory* design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes [12]. In GoTM, this design pattern is used by the transaction service to build new instances of transactions at runtime.

The evolutionary dimension of this design pattern deals with the ability to handle crosscutting concerns, such as the *caching* and the *pooling* of the instances created by the *Factory*. Depending on the TX Model definition, the transaction service is able to create flat or nested transactions. In [22], the transaction factory evolves to provide self-adaptability and choose the Two-Phase Commit protocol that would complete faster depending on the current execution context.

Figure 8 depicts an example of a component Factory used to create new instances of transaction components. The component Factory provides a microinterface *factory* to support creation and destruction of transaction component instances. The microcomponent Basic Factory creates new instances of components using the component Tx Model. A component Tx Model represents a template of transaction components that can be cloned several times to produce instances of transaction components.

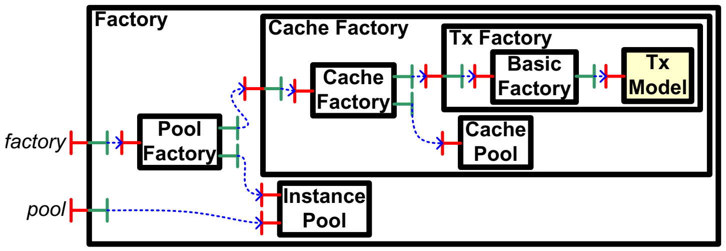

Fig. 8. The component Factory.

Moreover, the component Tx Model can be dynamically reconfigured to modify the architecture of future transaction components. The Cache Factory introduces a caching concern in the factory to reduce the cost of garbage collecting the references of transaction components. Instances of useless transaction components are stored in the component Cache Pool to be recycled by the Cache Factory. The Pool Factory registers the instances of transaction components created by the component Cache Factory. These instances are stored in the component Instance Pool and can be listed using the microinterface *pool* provided by the component Factory. This encapsulation of the components Factory forms a delegation chain [12].

3.4 State Design Pattern

The *State* design pattern allows an object to alter its behavior when its internal state changes [12]. In GoTM, this design pattern is used to represent and control the possible states of a transaction.

The evolutionary dimension of this design pattern deals with the capability of modifying the state automaton to support various transaction models. Using the component State, GoTM is able to implement state automatons conforming to the transaction standard specification. The *State* design pattern is revisited in this section using microcomponents to reify and configure the state automaton.

Figure 9 depicts a simple state automaton. The states `Inactive` and `Completed` are attached to the initial and the final states of the automaton, respectively. When receiving the event *start*, the system becomes `Active`. It can be `Suspended` when receiving the event *suspend*, and then moved back to `Active` via the event *start*. From the state `Active`, the system can be moved to the state `Completing` when receiving the event *complete*. Finally, the state `Completed` is accessible from the states `Active` or `Completing` when receiving the event *done*.

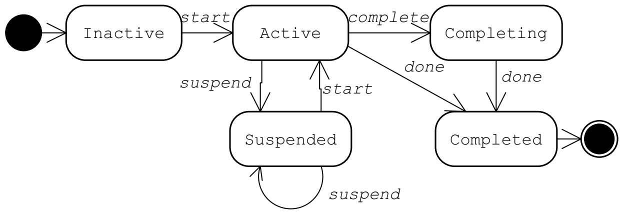

Fig. 9. The state automaton.

Figure 10 depicts a component State that implements the automaton depicted in Figure 9. The microcomponents Inactive, Active, Suspended, Completing, and Completed represent a state of the automaton. The bindings between the states represent the allowed transitions. The microcomponent State Manager manages the state automaton and allows the system to reset the state automaton at runtime via the microinterface *manager*. The exported microinterface *state* provides an access to the microcomponent reifying the current state.

3.5 Command Design Pattern

The *Command* design pattern encapsulates the concept of command into an object [12]. In GoTM, this design pattern is used to handle the different kind of participants registered in a transaction.

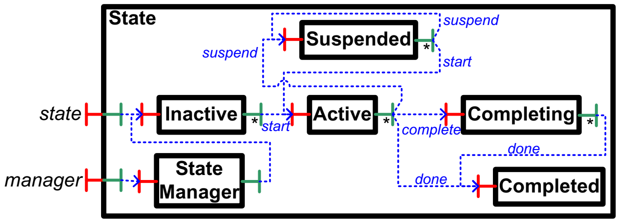

Fig. 10. The component State.

The evolutionary dimension of the *Command* design pattern deals with the list of commands that are executable on a transaction participant. Thanks to the application of the *Command* design pattern, GoTM can decline the *Command* component to support XA resources, Synchronization objects, CORBA resources, and Web Service participants depending on the content of the component *XA Commands* [21]. The *Command* design pattern is now revisited using microcomponents to easily configure the available commands.

Figure 11 illustrates the component Command. It encloses a variable number of XA participants on which commands can be applied [24]. The available commands are defined by the content of the component XA Commands. Each command is implemented by a microcomponent and conforms to the XA specification [24]. The participants enlisted in the system via the microinterface *register* are stored in the microcomponent Participant Pool. The policy used to send notify events to the enlisted participants can be configured. For example, the microcomponent Sequential Notify (resp. Parallel Notify) is responsible for notifying the participants sequentially (resp. in parallel) and executing the corresponding command. Thus, both the Participant Pool and the XA Commands components are shared between the Sequence Policy and Parallel Policy components.

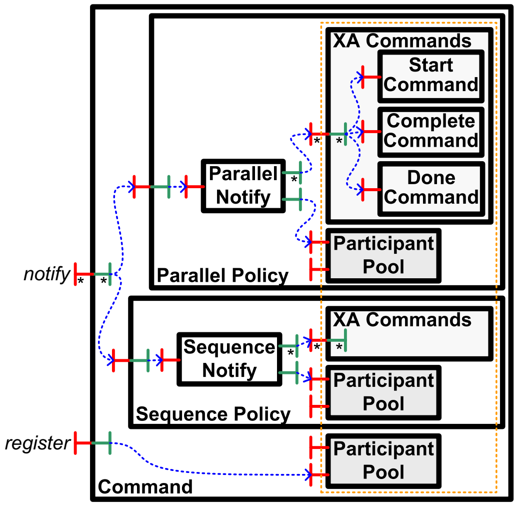

Fig. 11. The component Command.

3.6 Publish/Subscribe Design Pattern

The *Publish/Subscribe* design pattern[3] defines a one-to-many dependency between a publisher object and any number of subscriber objects so that when the publisher object changes state, all of its subscriber objects are notified and updated automatically [12]. In GoTM, the component Publish/Subscribe is used to synchronize the transaction participants during the execution of the Two-Phase Commit protocol [22].

The evolutionary dimension of this component consists in providing several publish policies. Additional publish policies are available in GoTM and provide *sequential propagation* or *pooled propagation* policies to ensure that no more than n messages are concurrently sent to the subscribers (n being the size of the pool).

In Figure 12, the architecture of the component Publish/Subscribe is similar to the architecture of the logging component one depicted in Figure 4. The microcomponent Subscriber Pool is shared between the components Synchronous Policy and Asynchronous Policy. The microcomponent Publish Synchronous guarantees that messages are correctly delivered and handled by the subscribers before returning. The microcomponent Publish Asynchronous sends messages to subscribers without waiting for. The State Checker microcomponent ensures that the published messages conform to the state automaton described in the shared component State (Section 3.4).

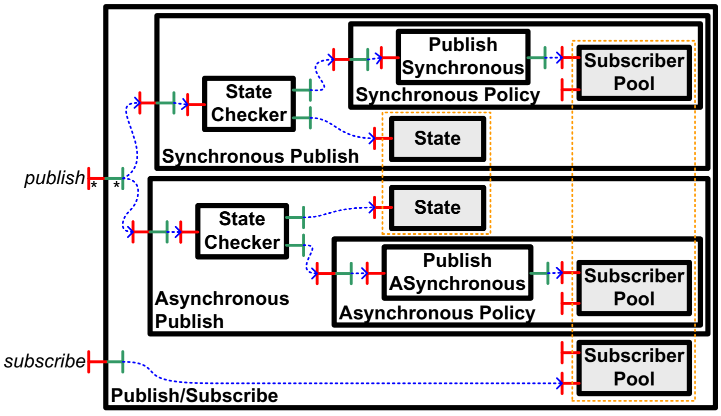

Fig. 12. The component Publish/Subscribe.

4 Discussion

Separation of Concerns. The definition of microinterfaces makes the composition of components more flexible. Microinterfaces factorize the definition of available operations and enforce their reuse by different microcomponents. The definition of microcomponents provides a better separation of concerns. This allows the developer to compose technical concerns, such as *Caching* or *Pooling*, independently.

[3] Derived from the *Observer/Observed* design pattern.

Microcomponents can be removed, replaced, or added depending on the architecture configuration of the component. This reconfiguration can be performed either while designing the transaction service using the Fractal ADL or at runtime using the reflective control capabilities of the Fractal component model. The Fraclet annotation framework drastically simplifies the definition of microcomponents while automatically generating the component glue and most of the Fractal ADL configurations.

Software Architecture Patterns. Once the microcomponents are defined, they can be easily composed using Fractal ADL. This composition relies mainly on the principles of *encapsulation* and *sharing*. Encapsulation reifies as a component the domain of application of a set of microcomponents. The sharing of microcomponents allows the components to collaborate transparently. The use of sharing favorises also the composition of orthogonal concerns to introduce additional functions (e.g., propagation policies). Basically, we define the *Sharing* architecture pattern as a Software Architecture Patterns [2]. This pattern allows a given component to be directly contained in several other components. This architecture pattern is used by the Publish/Subscribe, Command, and Facade design patterns. Based on the *Sharing* pattern, the *Encapsulation* architecture pattern extends the definition of a component using the delegation chain design pattern. Encapsulation is applied in the Factory and Publish/Subscribe design patterns. The *Policy* architecture pattern consists in sharing a core component between several policy components implementing the same interface. This architecture pattern is used by the Publish/Subscribe and Command design patterns. The *Pool* architecture pattern gathers components providing a common interface within a composite component. The pool pattern is used by the Factory, Publish/Subscribe and Command design patterns. The identification of such architecture patterns can help in providing a better evolution support to CBSE. Tools and rules can therefore be defined to control the evolution of component-based applications.

Performance of Transaction Services. Finally, considering the performance issue, our past experiences with GoTM have shown that using microcomponents and design patterns introduces no performance overhead to the transaction services [21,22]. Even better, it has shown that evolutionary transaction services built on top of GoTM can perform better than legacy transaction services [21].

5 Related Work

To achieve these goals, the approach used in GoTM takes advantages of several works related to CBSE, such as mixin-based approaches, component-based frameworks, microcomponents, and aspect-oriented design patterns:

Mixin-based approaches. Mixins [4] and Traits [10] provide a way of structuring object-oriented programs. Mixins are composed by inheritance to build an object that combine different concerns, each concern being implemented as a mixin. A trait is essentially a parameterized set of methods; it serves as a behavioral building block for classes and is the primitive unit of code reuse. Nevertheless, once mixed the object does not keep a trace of the mixins that compose it. This means that the object can not evolve to handle other concerns once it is mixed. In GoTM, we consider microcomponents as mixins that can be composed to build larger components. Once composed, the microcomponents are reified in a composite component to keep a clear view of the resulting architecture.

Component-based frameworks. The goal of Medor [1], OpenORB [3], Dream [14], and Jonathan [11] is to develop more configurable and re-configurable middleware technologies through a marriage of reflection, component technologies and component framework. These frameworks are based upon the lightweight and reflective OpenCOM and Fractal component models. For example, CORBA Object Request Brokers (ORB) have been implemented as a set of configurable and reconfigurable components in the context of the OpenORB and Jonathan frameworks. Nevertheless, these reflective adaptable middleware frameworks do not address the architecture of the component framework. Furthermore, they do not provide a methodology nor some evolutionary approaches to extend the possibilities of the component-based frameworks. While providing configurable properties equivalent to these frameworks, GoTM encloses also architectural patterns to support the evolution of transaction services.

Microcomponents. AsBaCo [16] and AOKell [23] introduce microcomponents to build the controller part of Fractal components as an evolutionary architecture. One of their contributions is a microcomponent model, which permits the capture of the structure of a component controller part; these frameworks enable the verification of the consistency of the controller configuration before launching the application. Since a microcomponent is, in a simplified view, an object with several provided and required services, the microcomponent model is applicable to Fractal implementations where the controller part consists of small object-like elements. The microcomponent models of AsBaCo and AOKell point out an interesting feature to build evolutionary middleware. Nevertheless, AsBaCo as well as AOKell do not provide any solution to the problem of evolutionary middleware architecture design. Based on a fine-grained component approach, GoTM addresses the evolution of middleware architectures either at design time or at runtime.

Aspects and Design Patterns. The combination of aspects and design patterns has been studied in several works [9,13]. The goal of this approach is to enforce the tracability, the modularity, and the reusability of design patterns using aspects. Aspect-oriented programming provides a way of tracking the design patterns, which tend to vanish in the code. Even if the design patterns are reified as aspects, this approach does not take into account the architectural dimension of an application. In particular, the design patterns are not reified in the architecture configuration to allow the application to evolve. Using microcomponents, GoTM reifies the design patterns as software architecture to make their configuration easier.

6 Conclusion and Future Work

This paper has introduced a component-based framework to build evolutionary transaction services. This framework, named GoTM, uses various design patterns to support evolution. These design patterns are implemented with microcomponents that can be composed following various Fractal ADL configurations. The use of microcomponents and design patterns to build transaction services provides better modularity properties and does not impact on the transaction service efficiency. Moreover, some architecture pattern can be extracted from this fine-grained architecture. Our previous experiences have shown that the transaction services built on top of GoTM are able to tune the transaction standards [21] and the commit protocols [22] supported.

Our future work will study additional technologies to provide further modularity to our framework using aspects and a higher level of abstraction for the design of transaction services using a model-driven approach.

Aspects and Microcomponents. We plan to investigate the use of an aspect-oriented framework to introduce some of the technical concerns presented in this paper. In particular, the *Fractal Aspect Component* (FAC) framework provides an interesting extension to support aspect-oriented programming at the component level [18]. For example, using FAC, the *Factory* design pattern (see Section 3.3) could be refactored to introduce the *Pooling* and the *Caching* concerns as aspect components rather than using encapsulation and sharing of components.

Model-Driven Engineering. We also intend to reify the software architecture design patterns identified in GoTM as template components to enforce and control their reuse. For example, we can define a software architecture pattern as an abstract component and use the extension mechanism of Fractal ADL to specify concrete components used to implement the design pattern. The concrete components could be generated using a Model-Driven Engineering (MDE) approach to complete the software architecture design patterns already defined in GoTM [19].

Availability. GoTM is freely available under an LGPL licence at the following URL: http://gotm.objectweb.org.

Acknowledgments. This work is partially funded by the national institute for research in computer science and control (INRIA), and the Region Nord - Pas-de-Calais.

References

[1] Alia, M., S. Chassande-Barrioz, P. Dechamboux, C. Hamon and A. Lefebvre, *A Middleware Framework for the Persistence and Querying of Java Objects*, in: *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*, LNCS **3086** (2004), pp. 291–315.

[2] Barais, O., J. Lawall, A.-F. Le Meur and L. Duchien, *Safe Integration of New Concerns in a Software Architecture*, in: *Proceedings of the 13th Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS)* (2006), to appear.

[3] Blair, G. S., G. Coulson, A. Andersen, L. Blair, M. Clarke, F. M. Costa, H. A. Duran-Limon, T. Fitzpatrick, L. Johnston, R. S. Moreira, N. Parlavantzas and K. B. Saikoski, *The Design and Implementation of Open ORB 2*, IEEE Distributed Systems Online **2** (2001).

[4] Bracha, G. and W. Cook, *Mixin-based inheritance*, in: *Proceedings of the International Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, SIGPLAN Notices **25** (1990), pp. 303–311.

[5] Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma and J.-B. Stefani, *An Open Component Model and Its Support in Java*, in: *Proceedings of the 7th International ICSE Symposium on Component-Based Software Engineering (CBSE)*, LNCS **3054** (2004), pp. 7–22.

[6] Bruneton, E., T. Coupaye and J.-B. Stefani, *Recursive and Dynamic Software Composition with Sharing*, in: *Proceedings of the 7th International ECOOP Workshop on Component-Oriented Programming (WCOP)*, Malaga, Spain, 2002.

[7] Cabrera, L. F., G. Copeland, M. Feingold, R. W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. Little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, T. Storey and S. Thatte, "Web Services Atomic Transaction (WS-AtomicTransaction)," 1.0 edition (2005).

[8] Cheung, S., "Java Transaction Service (JTS)," Sun Microsystems, Inc., San Antonio Road, Palo Alto, CA, 1.0 edition (1999).

[9] Denier, S., H. Albin-Amiot and P. Cointe, *Expression and Composition of Design Patterns with Aspects*, in: *Proceedings of the 2ème Journée Francophone sur les Développement de Logiciels Par Aspects (JFDLPA)*, RSTI L'Objet **11** (2005).

[10] Ducasse, S., O. Nierstrasz, N. Schärli, R. Wuyts and A. Black, *Traits: A Mechanism for fine-grained Reuse*, Transactions on Programming Languages and Systems (TOPLAS) (2005), pp. 46–78.

[11] Dumant, B., F. Horn, F. D. Tran and J.-B. Stefani, *Jonathan: An Open Distributed Processing Environment in Java*, Distributed Systems Engineering **6** (1999), pp. 3–12.

[12] Gamma, E., R. Helm, R. Johnson, J. Vlissides and G. Booch, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Westley Professional Computing, USA, 1995.

[13] Hannemann, J. and G. Kiczales, *Design Pattern Implementation in Java and AspectJ*, in: *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, SIGPLAN **37** (2002), pp. 161–173.

[14] Leclercq, M., V. Quéma and J.-B. Stefani, *DREAM: A Component Framework for Constructing Resource-Aware Configurable Middleware*, IEEE DS Online **6** (2005), pp. 1–12.

[15] Medvidovic, N. and R. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering **26** (2000), pp. 70–93.

[16] Mencl, V. and T. Bures, *Microcomponent-Based Component Controllers: A Foundation for Component Aspects*, in: *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC)* (2005), pp. 729–738.

[17] OMG, "Object Transaction Service (OTS)," Needham, MA, USA, 1.4 edition (2003).

[18] Pessemier, N., L. Seinturier, T. Coupaye and L. Duchien, *A Model for Developing Component-based and Aspect-oriented Systems*, in: *Proceedings of the 5th International ETAPS Symposium on Software Composition (SC)*, LNCS **4089** (2006).

[19] Rouvoy, R. and P. Merle, *Towards a Model Driven Approach to Build Component-Based Adaptable Middleware*, in: *Proceedings of the 3rd Middleware Workshop on Reflective and Adaptive Middleware (RAM)*, AICPS **80** (2004), pp. 195–200.

[20] Rouvoy, R., N. Pessemier, R. Pawlak and P. Merle, *Using Attribute-Oriented Programming to Leverage Fractal-Based Developments*, in: *Proceedings of the 5th International ECOOP Workshop on Fractal Component Model*, Nantes, France, 2006.

[21] Rouvoy, R., P. Serrano-Alvarado and P. Merle, *A Component-based Approach to Compose Transaction Standards*, in: *Proceedings of the 5th International ETAPS Symposium on Software Composition (SC)*, LNCS **4089** (2006).

[22] Rouvoy, R., P. Serrano-Alvarado and P. Merle, *Towards Context-Aware Transaction Services*, in: *Proceedings of the 6th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, LNCS **4025** (2006), pp. 272–288.

[23] Seinturier, L., N. Pessemier, L. Duchien and T. Coupaye, *A Component Model Engineered with Components and Aspects*, in: *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*, LNCS **4063** (2006).

[24] The Open Group, "Distributed Transaction Processing: The XA Specification," C193 edition (1992).