Design Guidelines for Coordination Components

Sander Tichelaar, Juan Carlos Cruz and Serge Demeyer
Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland.
E-mail: {tichel,cruz,demeyer}@iam.unibe.ch. *WWW:* http://www.iam.unibe.ch/~scg/.

Keywords: coordination component, object-orientation, contracts, transactions, design guidelines

ABSTRACT

The distributed nature of a typical web application combined with the rapid evolution of underlying platforms demands for a plug-in component architecture. Nevertheless, code for controlling distributed activities is usually spread over multiple subsystems, which makes it hard to dynamically reconfigure coordination services. This paper investigates *coordination components* as a way to encapsulate the coordination of a distributed system into a separate, pluggable entity. In an object-oriented context we introduce two design guidelines (namely, "*turn contracts into objects*" and "*turn configuration into a factory object*") that help developers to separate coordination from computation and to develop reusable and flexible solutions for coordination in distributed systems.

1 INTRODUCTION

With the coming of the world-wide web, more and more software systems must be adapted to incorporate facilities for inter- and intra-nets. Large portions of the software industry have tried to tackle this market with component technology: desktop publishing software provide components that generate HTML and PDF; database vendors sell components that translate "search and query applications" into HTML forms; Java component environments provide support for building attractive user interfaces using native widgets. And this list continues to grow.

One of the main reasons why web applications embrace component technology is the ability for dynamic reconfiguration by means of *plug-in components* [16]. Indeed, since web applications operate in heterogeneous contexts, they need to encapsulate all platform dependent behaviour into separate components. And since there is no way to freeze the internet while reconfiguring, web applications must plug-in new functionality at run-time.

Despite the increasing maturity of middle-ware standards like CORBA and Microsoft's ActiveX, component technology has not been applied to one of the most critical aspects of web-applications, namely the coordination of the distributed activities [13]. The reason for this is twofold: first, coordination is difficult to shrink-wrap into an off-the-shelf component [23], implying that application developers will not easily be able to buy "coordination components" like they buy GUI-components. Second, code for coordinating distributed activities is typically spread over all subsystems that make up the web-application, implying that dynamic reconfiguration of coordination policies is usually impossible.

Since dynamic reconfiguration of coordination policies is often necessary (e.g., for load-balancing) and since it is unlikely that one will be able to buy off-the-shelf solutions, application developers are obliged to implement their own coordination components. Yet, implementing your own solution has considerable drawbacks and must —among other things— be guided by solid design guidelines. This paper presents two design guidelines in object-oriented framework technology (i.e., *turn contracts into objects* and *turn configuration into a factory object*), which provide support for encapsulating coordination into special-purpose plug-in components that can be dynamically exchanged. The paper introduces the design guidelines by means of an example (a web-application for managing bank accounts), gradually adding requirements to show how the resulting coordination component allows for dynamic system reconfiguration.

2 AN EXAMPLE: INTERNET BANKING

Consider the example of a bank that provides its customers with internet banking services. Customers use a web browser to consult the balance on their accounts and may transfer money from one account to another. The requirements we impose on our example application are:

- **Fundamental Requirements**
  1. *Security.* Only authorised users should have access to an account.
  2. *Reliability.* Any operation must leave the system in a consistent state, i.e. while seeing the balance of an account, or while transferring money from one account to another, no money may disappear. Thus, the sum of the balances of all accounts remains constant over time.

- **Additional requirements for a dynamic environment**
  3. *Performance Tuning*. For optimal throughput, the system should be able to switch policies, for instance between an optimistic or pessimistic locking protocol [17].
  4. *Replication*. To be able to handle lots of requests in parallel, it should be possible to replicate the web server on different machines.
  5. *Dynamic Reconfiguration.* In the above two cases all reconfiguration of the web server should be dynamic. Thus, switching between policies and adding extra servers, should be possible without terminating the system.

This set of requirements has the following implications: firstly, to provide reliability for multiple concurrent clients coordination of actions is needed. Secondly, the additional requirements demand for a way to easily exchange solutions to this coordination problem.

We start with an initial design and implementation that fulfils the two fundamental requirements (section 2.1 and section 2.2). This elementary version doesn't, however, satisfy the additional requirements for a dynamic environment. To resolve that we apply two design guidelines (section 2.3) including the additional technology we need to be able to apply them (section 2.4).

2.1 An Initial Design

An initial design for our banking system is shown in Figure 1, where we see that a `customer` owns a number of `accounts` within a `bank`, and that a customer may request the bank to `see` the balance of an account and to `transfer` money from one account to another. When the bank object is answering such requests, it guarantees the "security" and "reliability" requirements, phrased in the "account access contract"[1] on the class `Bank`. That contract states (i) as invariant that the sum of the balances of all accounts remains equal; (ii) as precondition for both the `transfer` and `seeBalance` operation that the requesting customer is authorised to issue the request; (iii) as postcondition for the `transfer` operation that the balances of the involved accounts have been updated accordingly.

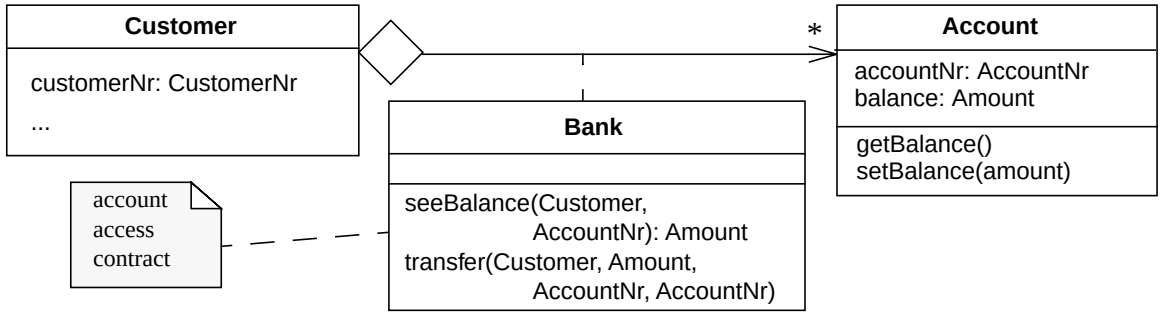

Figure 1 Initial design for an internet banking system, including the account access contract

2.2 Naive Implementation

To satisfy the "Reliability" requirement the internet banking system must incorporate concurrency control facilities, thus a per-account locking mechanism. A naive implementation —i.e. one that does not have to deal with the "Dynamic Reconfiguration" requirement— might achieve this via some additional locking code on the class `Account` and some transaction management code on the class `Bank`. Typical code —adapted from [17]— is shown in Figure 2. To meet the "Security" requirement the call `isAuthorised` is added.

A first observation to be made at this point is that concurrency control adjusts the public interface of the domain objects. In particular, the `getBalance` and `setBalance` operations require an extra parameter (the transaction identifier), and the class `Account` contains additional operations (`lock`, `commit`, `abort`). A second observation is that we are forced to wrap a considerable amount of code around the domain specific functionality to meet the non-functional requirements (i.e., the greyed-out code in Figure 2). Finally, this design does not cope well with the "Dynamic Reconfiguration" requirement. Adding extra servers requires a new implementation of the `Account` class that is impossible to load at run-time without halting the system. And once the replicated servers are running, dynamic switching of the locking policy is impossible, because it requires loading new versions of the `Bank` and `Account` classes on all replicas simultaneously.

The diagnosis of the problem —as should not come as a surprise— is an incorrect separation of concerns: functional as well as non-functional behaviour is mixed into the same class. To remedy this situation, we must factor out the state and behaviour for the transaction from the domain-specific code. In section 2.3 through section 2.6 we show how a redesign leads to coordination components that encapsulate non-functional behaviour and allow for dynamic reconfiguration.

2.3 Design Guidelines

To tackle the problem of dynamic reconfigurability we apply -- similar to what is proposed in [10] -- two design guidelines, namely *"turn contracts into objects"* and *"turn configuration into a factory object"*. The guidelines provide a step-by-step recipe to introduce components that encapsulate coordination, as proposed in [8]. Applying the guidelines to coordination is not trivial as we need to deal with concurrent interdependent accesses to the domain objects (namely, the different transactions

1. In this paper, we use the notion of contracts as explained in "Design by Contract"[19].

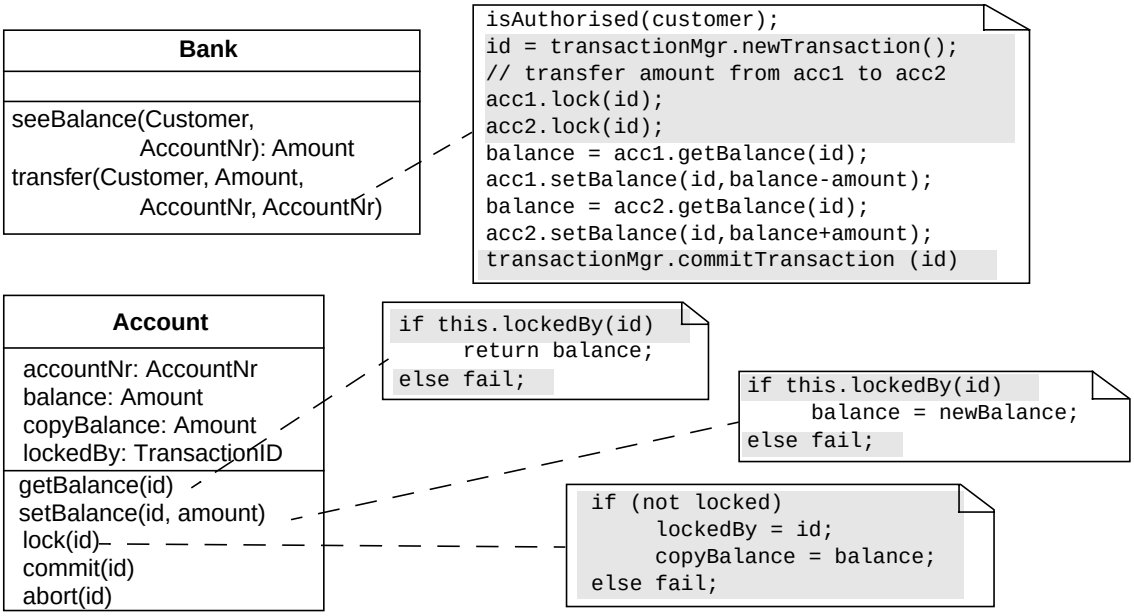

Figure 2 Naive implementation violating the "separation of concerns" principle. Grey regions denote non-functional code.

accessing the accounts). Therefore, in addition to what is proposed in [10], we need to wrap the domain objects individually to (1) control the access to these domain objects and (2) control the state of the domain objects in the presence of multiple transactions.

Following the first guideline, we introduce an explicit representation of the "Account Access" contract (see Figure 3). This explicit contract object checks the pre- and postconditions of the contract with the `pre` and `post` operations. Following the second guideline, we introduce an explicit factory object (cf. the "Abstract Factory" design pattern [12]) that is responsible for supplying an appropriate set of domain objects according to the system configuration.

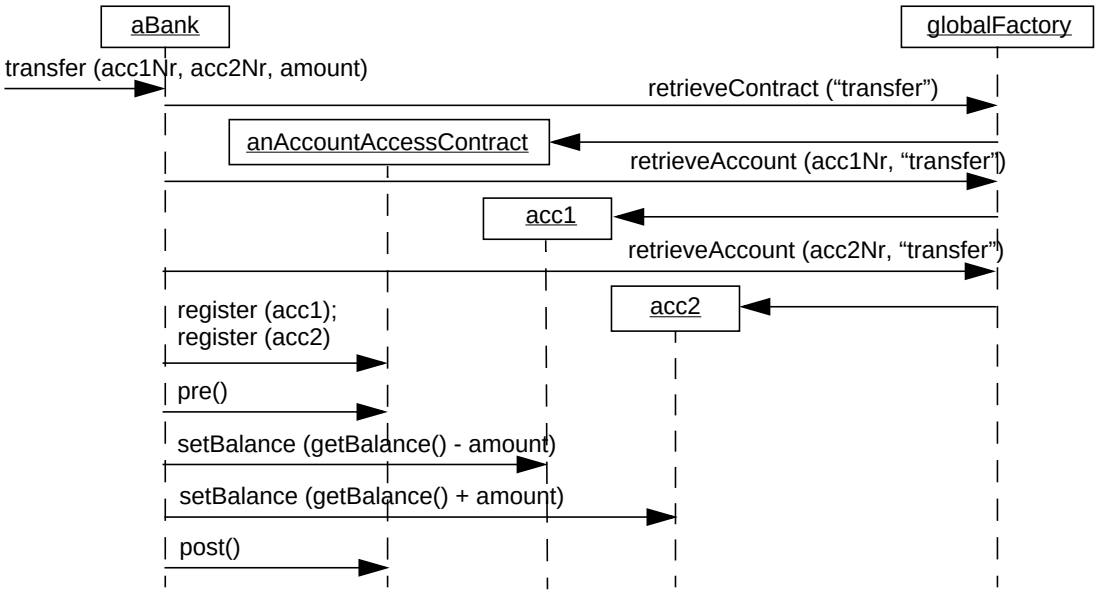

Figure 3 Sequence diagram for "transfer" operation using explicit contract and factory objects.

The interaction diagram in Figure 3 illustrates the effect of introducing an explicit contract and a configuration object on the `transfer` operation. Basically, whenever a bank receives a request (`transfer` or `seeBalance`), it first asks the global factory to supply the necessary objects (using `retrieveContract` and `retrieveAccount`). Next, it registers the participating objects on the contract (using `register`); requests the contract object to check the precondition (using `pre`); then does whatever is required to satisfy the actual request (a sequence of `getBalance` and `setBalance`) and finally checks the postcondition (using `post`).

An important property of the *turn contracts into objects* and *turn configuration into a factory object* design guidelines is that these objects never introduce any behaviour outside the domain. However, they do introduce a number of hook methods (i.e., `retrieveContract`, `retrieveAccount`, `pre` and `post`) that enable us (i) to separate transactional state from domain specific state and (ii) to wrap additional transactional behaviour around the domain specific operations. In the next sections we show how to make use of these hook methods to deal with the additional requirements for a dynamic environment.

2.4 Wrapping the Domain Objects

To address the three requirements for a dynamic environment, we encapsulate the per-account locking mechanism into a set of collaborating wrapper objects. The transaction mechanism requires additional behaviour on `pre` and `post` (to initialize and terminate the transaction) and on `getBalance` and `setBalance` (to check the lock before performing the actual get- or set operation). This behaviour is added by wrapping the contract object and the account objects. Wrapping allows us to leave the protocol inside the existing transfer operation untouched. Figure 4 shows the details: the greyed code shows the wrappers call-

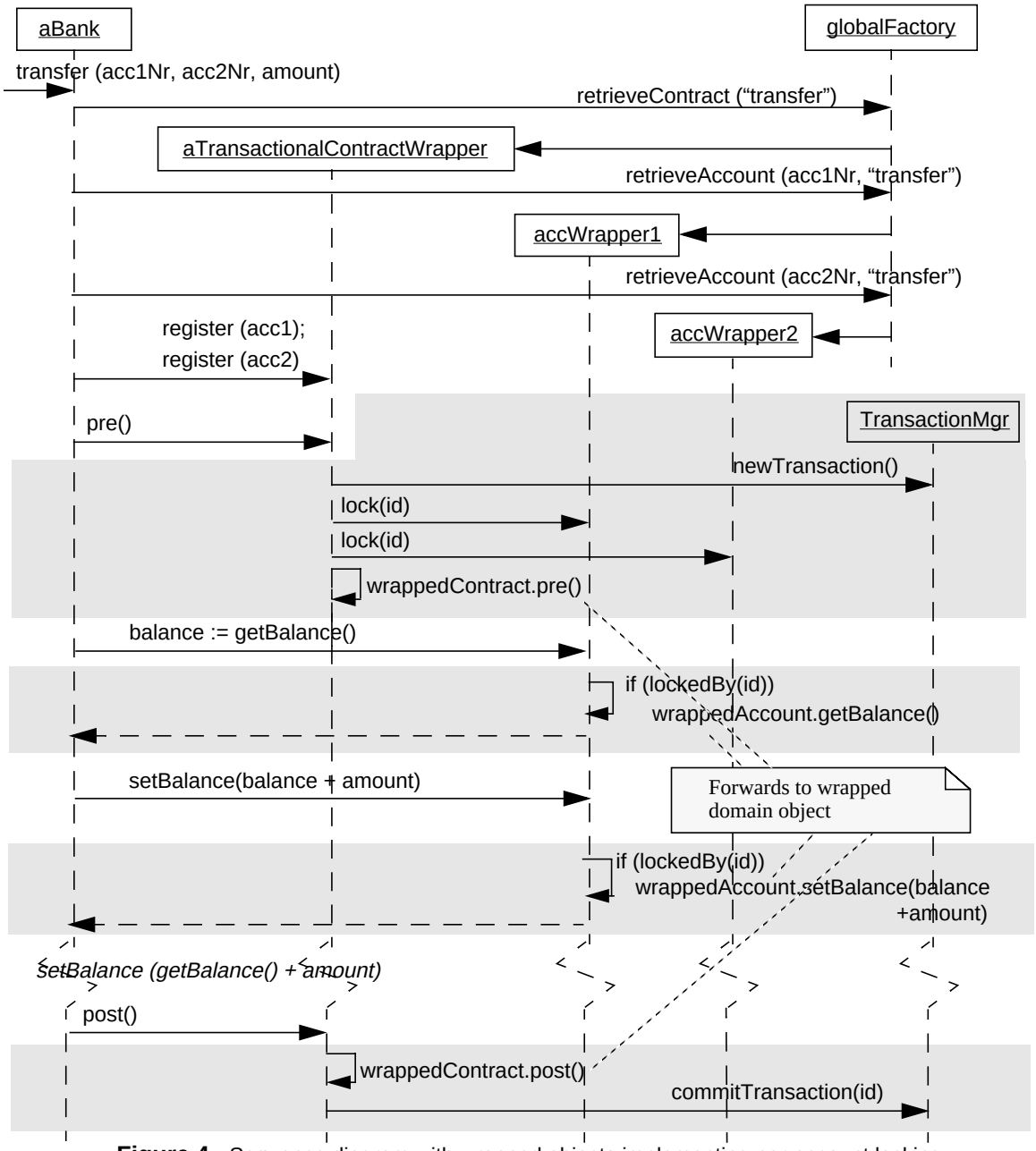

Figure 4 Sequence diagram with wrapped objects implementing per-account locking. The grey areas correspond to added details with respect to Figure 3: wrappers calling each other to provide coordination of the wrapped objects.

ing each other (to provide the coordination) or the action they wrap. The details are explained in the following list:

1. The `retrieveContract` operation wraps the original contract object to patch the `pre` and `post` operations.
2. The `retrieveAccount` operation returns wrapped account objects with the original `getBalance` and `setBalance` operations patched.
3. The patched `pre` operation initializes the transaction by issuing `newTransaction` on a global transaction manager and locking the accounts. Finally, it forwards the call to the wrapped contract object, to check the original preconditions as well.
4. The patched `getBalance` and `setBalance` operations participate in the transaction by checking the lock and forwarding to the wrapped account object.
5. The patched `post` operation forwards to the wrapped contract objects and then issues the final `commitTransaction`.

Finally, a note on how a transaction may abort. This may happen after the `lock` or `commitTransaction` requests because the transaction system could not satisfy the request because of collisions with other transactions. Aborts may also happen just after the forwarded `pre` or `post` operations return, because the contract object decides that the pre or post conditions are not satisfied. In all these cases, an exception is raised which handler gracefully cleans up all resources and terminates the transaction, leaving all the domain objects in their original state.

2.5 Reconfiguration of Locking Protocol

Once we have introduced the per-account locking mechanism using the protocol in Figure 4, it is possible to switch from optimistic to pessimistic locking transparently. Indeed, the only difference lies in the implementation of the `lock` operation: optimistic locking aborts immediately when it can not acquire a lock; pessimistic locking waits until the lock comes available [17]. Consequently, it suffices that the globalFactory instantiates the appropriate wrapper class --either `OptimisticLockAccount` or `PessimisticLockAccount` (see Figure 5)-- to make the accounts lockable. The contract wrapper then provides all transaction related state needed by the lockable accounts (namely the transaction id). Figure 5 shows a possible class hierarchy to implement such locking functionality, similar to what is described in [24].

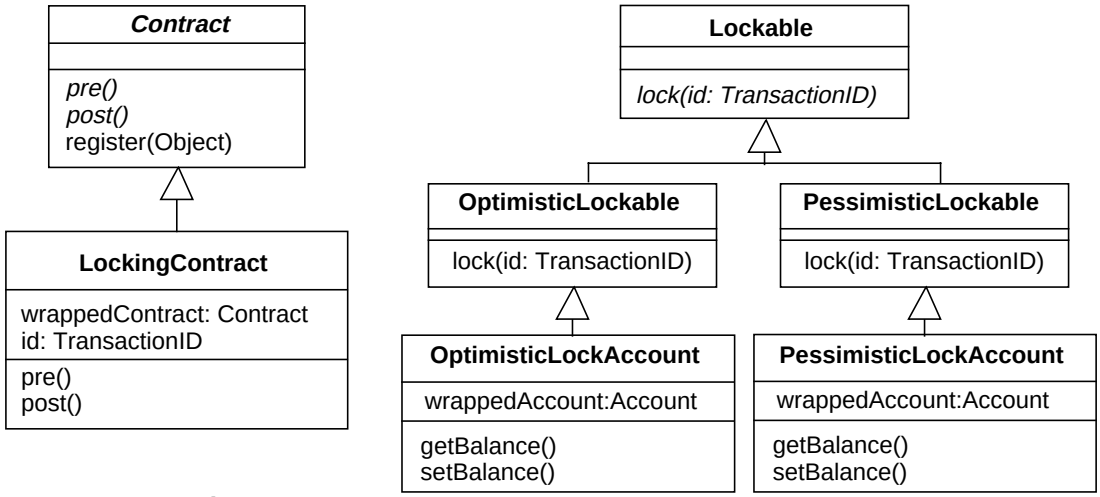

Figure 5 Class hierarchy for reconfigurable locking protocols

Note that the wrappers around the domain objects expand the basic interaction protocol, as they interact with each other to provide the coordination service. Therefore, they are not independent and can only be used as a group. Consequently, we have but one global factory object (`globalFactory`) which is responsible for instantiating the appropriate set of wrappers.

2.6 Replication of Services

With the per-account locking protocol from Figure 4, it is possible to replicate the individual account objects within several servers. To achieve this, all replicated account objects are wrapped into one `CompositeLockAccount` using a variant of the "Composite" design pattern [12] (see Figure 6). On receiving a `lock` or `setBalance` operation, the composite forwards to all contained accounts, while the `getBalance` operation is forwarded to a single account object only. Again, by instantiating the appropriate wrapping objects into the factory object, the replicated services of our system can be dynamically reconfigured.

3 DISCUSSION

To achieve the goal of dynamic reconfigurability of non-functional behaviour we have introduced explicit contracts and configuration objects as a way to encapsulate coordination behaviour into a special-purpose component. The example of the internet banking illustrates the typical steps involved in designing coordination components:

1. Start with an initial design for the domain, including contracts that state the obligations of the participating classes. Define the contracts according to "De-

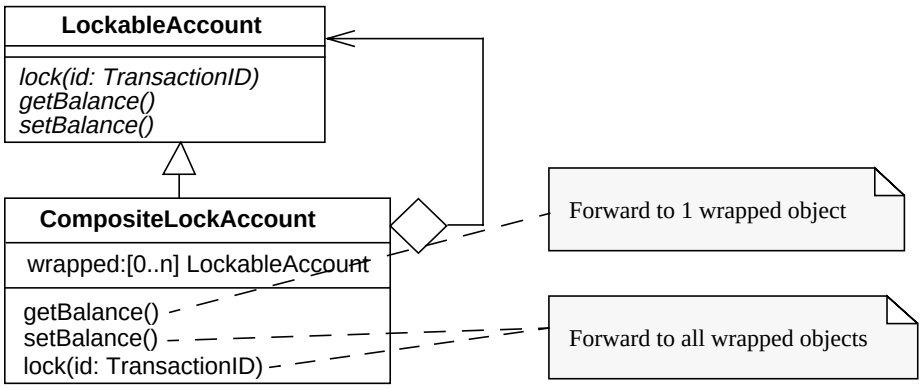

Figure 6 Class hierarchy for replicated services

sign by Contract" principle [19], i.e. using pre- and postconditions.

2. Apply the design guideline "*turn contracts into objects*". Thus, define a separate object per contract, providing `pre` and `post` operations for checking the contractual obligations. Also, have the domain objects invoke these `pre` and `post` operations.
3. Apply the design guideline "*turn configuration into a factory object*". Thus, introduce an object that knows which wrappers and domain objects to create for a certain configuration.
4. Add non-functional behaviour by providing wrappers around the original domain objects. The wrappers instrument the domain-specific operations with the non-functional coordination behaviour. A particular configuration of wrapper objects is then a *coordination component*.

The fact that the explicit representations of the contracts provide the necessary hooks for coping with non-functional requirements is *not* coincidence. Indeed, a contract object forces the designer to make the important sequence of domain operations explicit, thus providing the ideal place to factor out non-functional state and wrap non-functional behaviour.

We call a configuration of wrapper objects containing a particular locking or replication policy a *coordination component*. According to the definition "components are static abstractions with plugs" ([20], p.5), this corresponds indeed to a component. The configuration is (a) static because it can be stored inside a component-base to be reused across different applications; it's an (b) abstraction because it encapsulates well-defined coordination behaviour and it has (c) plugs by means of the wrapped operations.

Although the solution provides for dynamic reconfiguration of coordination aspects of a system, it also makes the system inherently more complex. Instead of only having the domain objects altered with some coordination code, the management of coordination is delegated to a set of specialized objects which can be exchanged at run-time. This implies that the design steps should only be applied when dynamic reconfiguration is a requirement.

Finally, dynamically exchanging coordination protocols may be eased by reflection support in the underlying programming language. In languages such as CLOS [3] or Smalltalk [11], domain specific operations can be explicitly manipulated to wrap additional state and behaviour. We have used the wrappers together with the factory object as a kind of "poor men's reflection", to achieve the necessary method instrumentation in mainstream object-oriented languages such as C++ and Java.

Past, Present and Future

The initial ideas on coordination components appeared in [25], later summarized in [8]. On the other hand, the role of explicit contract and configuration objects in framework design has been explored in [9], later refined in [10]. In the paper you are reading now, both the ideas on pluggable coordination policies and the explicit objects have been combined into a series of design steps that lead to coordination components.

A prototype with the presented design has been successfully built. Indeed, switching the banking system from a non-transactional system to a transactional system requires only to make the factory provide the right set of wrappers. Similarly, switching between transaction policies is done by making the factory provide a different set of transactional wrappers.

Future work includes extending the framework to use CORBA's and COM's transactional systems and test the applicability of the approach in a real-world context. In particular, more work needs to be done on examining the effects of dynamically changing the policies. Future work also includes a survey of contracts as the basis for the development of a coordination framework. Special emphasis within this framework is on coordination contracts as the means for governing the collaborations between objects or components in a software system. A coordination contract makes explicit what the minimal and sufficient conditions are for the different parts to work together.

4 RELATED WORK

This work is located in the area between coordination and object-oriented framework technology. Therefore, we discuss related work out of both these areas. Note that we build upon established ideas from well-known concurrent systems technology as well, but we do not refer to these.

Coordination languages

Coordination languages, such as Gamma [4] and all kinds of Linda-flavours [7], typically provide a single paradigm to separate the coordination part of an application from the computa-

tional part. For instance, Gamma supports a chemical reaction model for the definition of programs without artificial sequentiality, while Linda provides tuple-spaces for generative communication with some basic synchronization.

The idea of coordination components is not new, as illustrated by work such as Manifold [1]. This language coordinates the global behaviour of a number of black box components by connecting their respective communication ports. Systems can be reconfigured by dynamically rewiring its constituting components.

The TOOLBUS coordination architecture [6] provides a communication bus to connect components in a distributed environment. The communication bus is controlled by process-oriented scripts, which formalize the interaction between the components. The approach provide a clean separation of concerns, as the components only compute and the scripts describe the interaction.

Like promoted by coordination languages, our approach provides explicit separation of coordination and computation. However, we do not investigate a particular paradigm or formalization of coordination. We focus on wrapping technology in mainstream object-oriented languages. Consequently, our design guidelines are directly applicable in object-oriented applications that need dynamic reconfiguration of coordination policies.

What we don't address in this paper is the coordination problem of inter-activity coordination as for instance known in workflow systems. Examples of coordination research in this area are CLF [2] and Sonia [4].

Object-Oriented Framework Technology

Today, a few commercial frameworks deal with coordination issues, most notably CORBA implementations and Microsoft's ActiveX. These frameworks provide basic services for coordinating distributed communication, e.g. the CORBA transaction service. Our approach is *complementary* to these, in the sense that --like promoted by coordination languages-- we advocate for a separation of coordination from computation. As such, the guidelines help to encapsulate the dependencies on such commercial services, so that we for instance would be able switch transparently and dynamically between a CORBA and DCOM implementation.

The ADAPTIVE Communication Environment (ACE) [22] of Doug Schmidt implements a set of design patterns for concurrent event-driven communication software. It simplifies the development, configuration and reconfiguration of distributed applications and services that use interprocess communication, event demultiplexing, explicit dynamic linking and concurrency.

In [24], a transaction framework is presented that provides ways to dynamically adapt the transaction semantics of a system for optimal transactional behaviour at all times. This work shows the feasibility of dynamic exchange of transaction policies.

Coda [14] has been used to open up the implementation of Smalltalk message passing and is able to add meta-level infrastructure to Smalltalk objects, so that additional behaviour such as concurrency or distribution can be added. The Coda experiment is especially important as it shows that a meta-level is "just another application" and that traditional software design techniques such as abstraction and decomposition remain valuable.

Implementational Reflection and Aspect Oriented Programming

A major source of inspiration for our work, called *implementational reflection*, is presented in [21]. It is an approach that "opens up" implementations by exposing their meta-level. On one hand a system has a base-level interface which is the common interface for such a system, where on the other hand the system has a meta-level interface that reveals how some aspects of the system are implemented. The meta-level interface provides the possibility to change the default base-level behaviour to behaviour that differs in semantics and/or performance. In [21] the approach is illustrated by a windowing system. In the context of this approach our work can be viewed as providing a meta-level for coordination where coordination policies can be switched.

In Aspect Oriented Programming [15],[18], systems are viewed as a set of components and aspects. Components are properties of systems that are easily encapsulated in a generic way, and aspects are properties that effect many other components and therefore cannot be cleanly encapsulated. Both these kinds of properties will cross-cut each other in a system's implementation. The solution they propose is to describe the components and the aspects in their own languages, thereby ensuring separation of concerns, and then mix the properties using a special language processor. Although the solution provides a clean separation of concerns, it doesn't support the coordination of off-the-shelf components. Aspects and components are mixed at compile time, leaving out the possibility to plug-in other components afterwards.

5 CONCLUSION

The very nature of web-like environments --with its rapid evolution of standards and protocols-- demands for dynamic reconfiguration. Component technology as a means to dynamically plug in new functionality is becoming increasingly important. Yet, truly distributed web systems are scarce, partly because component technology has not yet been able to deliver reusable coordination abstractions.

In this paper we have shown that it is possible to provide reusable coordination solutions as dynamically pluggable components. Explicit contract and factory objects form the key to the solution: they provide the right set of hooks to wrap additional non-functional coordination behaviour. This in turn gives rise to what we call a coordination component. The explicit contract and factory objects follow naturally from applying two framework design guidelines: "*turn contracts into objects*" and "*turn configuration into a factory object*".

ACKNOWLEGDEMENTS

We would like to thank all members of the SCG who have carefully reviewed earlier drafts of this paper. Furthermore, this

work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT programme Project no. 21975.

REFERENCES

[1] Arbad, A., "The IWIM Model for Coordination of Concurrent Activities," *Proceedings of COORDINATION'96*, Ciancarini, P. and Hankin, C. (Eds), LNCS 1061, Springer Verlag, April 1996.

[2] Andreoli, J.-M., Freeman, S. and Pareschi, R., ''The Coordination Language Facility: Coordination of Distributed Objects,'' TAPOS, vol. 2, no. 2, 1996, pp. 635-667.

[3] Attardi, G., Bonini, C., Boscotrecase, M. R., Flagella, T. and Gaspari, M., ''Metalevel Programming in CLOS,'' *Proceedings ECOOP'89*, S. Cook (Ed.), Cambridge University Press, Nottingham, July 10-14, 1989, pp. 243-256.

[4] Banâtre, J.-P. and Le Métayer, D., "Gamma and the Chemical Reaction Model," *Proceedings of the Coordination'95 Workshop*, IC Press, London, 1995.

[5] Banville, M., ''Sonia: an Adaptation of Linda for Coordination of Activities in Organizations,'' Proceedings of COOORDINATION'96, Ciancarini, P. and Hankin, C. (Eds), LNCS 1061, Springer-Verlag, Cesena, Italy, April 1996, pp. 57-74.

[6] Bergstra, J. A. and Klint, P., ''The ToolBus Coordination Architecture,'' *Proceedings of COORDINATION'96*, Ciancarini, P. and Hankin, C. (Eds), LNCS 1061, Springer-Verlag, Cesena, Italy, 1996, pp. 75-88.

[7] Carriero, N. and Gelernter, D., "Linda in Context," *Communications of the ACM*, vol. 32, no. 4, April 1989, pp. 444-458.

[8] Cruz, J. C. and Tichelaar, S., ''Managing Evolution Of Coordination Aspects In Open Systems,'' *Ninth International Workshop On Database And Expert Systems Applications*, Wagner, R. R. (Ed.), IEEE Computer Press, Vienna, Austria, August 1998, pp. 578-582.

[9] Demeyer, S., *ZYPHER Tailorability as a link from Object-Oriented Software Engineering to Open Hypermedia*, Ph.D. Thesis, Vrije Universiteit Brussel, Brussels, Belgium, July, 1996.

[10] Demeyer, S., Meijler, T. D., Nierstrasz, O. and Steyaert, P., "Design Guidelines for Tailorable Frameworks," *Communications of the ACM*, vol. 40, no. 10, October 1997, pp. 60-64.

[11] Ducasse, S., ''Evaluating Message Passing Control Techniques in Smalltalk,'' *Journal of Object-Oriented Programming (JOOP)*, vol. 12, no. 6, SIGS Press, June 1999.

[12] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns*, Addison- Wesley, Reading, MA, 1995.

[13] Guerraoui, R. and Fayad, M. E., "Object-Oriented Abstractions for Distributed Programming," *Communications of the ACM*, vol. 42, no. 8, August 1999, pp. 125-127.

[14] McAffer, J., ''Meta-level Programming with CodA,'' *Proceedings ECOOP'95*, Olthoff, W. (Ed.), LNCS 952, Springer-Verlag, Aarhus, Denmark, August 1995, pp. 190-214.

[15] Kiczales, G. et al., "Aspect-Oriented Programming," *Proceedings ECOOP'97*, Aksit, M. and Matsuoka, S. (Eds), LNCS 1241, Springer-Verlag, Jyväskylä, June 1997, pp. 220-242.

[16] Laddaga, R. and Veitch, J., ''Dynamic Object Technology,'' *Communications of the ACM*, vol. 40, no. 5, ACM Press, May 1997, pp. 36-38.

[17] Lea, D., *Concurrent Programming in Java -- Design principles and Patterns*, The Java Series, Addison-Wesley, 1996.

[18] Lopes, C., "Aspect Oriented Programming", *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, Demeyer, S. and Bosch, J. (Eds), LNCS 1543, Springer-Verlag, July 1998, pp. 394-443.

[19] Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, 1997, second edition.

[20] Nierstrasz, O. and Tsichritzis, D., *Object-Oriented Software Composition*, Prentice Hall, 1995.

[21] Rao, R., "Implementational Reflection in Silica," *Proceedings ECOOP'91*, America, P. (Ed.), LNCS 512, Springer-Verlag, Geneva, Switzerland, July 1991, pp. 251-267.

[22] Schmidt, D. C., "The ADAPTIVE Communication Environment: An O.O. Network Programming Toolkit for developing Communication Software," Technical Report, Department of Computer Science, Washington University, 1994.

[23] Schmidt, D. C., and Fayad, M. E., "Lessons Learned Building Reusable OO Frameworks for Distributed Software," *Communications of the ACM*, vol. 40, no. 10, *October 1997*, pp. 85-87.

[24] Tekinerdogan, B., "An Application Framework for Building Dynamically Configurable Transaction Systems," OOPSLA'96, Development of Object-Oriented Frameworks Workshop, San Jose, USA, 1996.

[25] Tichelaar, S., *A Coordination Component Framework for Open Distributed Systems*, Master's Thesis, University of Groningen, May 1997.