Progressive Implementation of Distributed Java Applications

Paulo Borba* Saulo Araújo Hednilson Bezerra
Marconi Lima Sérgio Soares
Departamento de Informática
Universidade Federal de Pernambuco

1 Introduction

In this position paper we overview on-going research work aimed at defining, formalizing, and validating a method for the systematic implementation of distributed object-oriented applications. In particular, this method supports a progressive approach for object-oriented implementation, where distribution, concurrency, and persistence aspects are not initially considered in the implementation process, but are gradually introduced, preserving the application's functional requirements and semantics.

By initially abstracting from those subtle aspects, engineers can, for example, quickly develop and test a local, sequential, and non-persistent application prototype useful for capturing and validating user requirements. As requirements become well understood and more stable, that prototype can be used to derive a structured and functionally complete prototype, which is then progressively transformed into the final distributed, concurrent, and persistent version of the application, by carefully dealing with the aspects mentioned above, one at a time.

In this way we can significantly reduce the impact caused by requirements changes during development, since most changes will likely occur before the functionally complete prototype is transformed into the final version of the application, which is larger and much more complex than the prototypes. Furthermore, the progressive approach naturally helps to tame the complexity inherent to distributed systems, by supporting the gradual testing of the various intermediate versions of the application. For example, problems in the business logic layer can be isolated from problems in the persistence and communication layers.

Of course, a central assumption to our method—the Progressive Implementation Method (Pim)—is that it is possible to initially abstract from distribution and concurrency aspects. In fact, Pim considers that there are differences between the implementation of local and distributed objects, but that local objects can be gradually transformed into distributed ones. However, this might not be possible when distribution or concurrency is inherent to the application objects. Indeed, we believe that our method is not useful for all kinds of distributed applications, but it has been proven specially useful for the implementation of distributed information systems, which are usually distributed and concurrent for performance and fault tolerance reasons only.

As Pim is just an implementation or coding method, it should be integrated to design and testing methods in order to be used in practice. However, Pim relies on the use of specific architectural and design patterns for structuring object-oriented applications, so that it imposes constraints on design methods chosen for integration. Fortunately, those constraints

*Supported in part by CNPq, grant 521994/96–9. WWW: `http://www.di.ufpe.br/~phmb`. Electronic mail: `phmb@di.ufpe.br`. Telephone: +55 81 271 8430, extension 3323. Fax: +55 81 271 8438.

basically correspond to the use of simple or well known patterns useful for supporting the progressive introduction of distribution, concurrency, and persistence aspects.

This paper is organized as follows. We first explain how we formalize and justify Pim's activities and key tasks[1]. We then discuss in separate sections the introduction of distribution and concurrency aspects, in that order. Persistence aspects are not considered here for scope and space reasons. Examples are used to illustrate several aspects of Pim, including some refinement laws and the constraints imposed on design methods. The current version of Pim is specific to Java [4] and RMI [7], so that we use them throughout the paper. At the end we discuss the current status of this work and its limitations.

2 Laws of Progressive Implementation

Most tasks of Pim are just informally described and illustrated by examples. However, key tasks are formalized by semantic preserving refinement laws, which basically indicate that a class, a command, or a program can be safely replaced by another. Those laws are essential for precisely determining subtle constraints and code modifications associated to a given task. Moreover, the laws justify the soundness of Pim, assuring that the final version of a given application preserves the semantics of its functionally complete prototype, as long as database and distribution services work properly; for example, database and network connections are not indefinitely down.

As in [2], here we represent and formalize the laws by using class operators for building classes from smaller pieces, as if we were defining an algebra for classes. For example, the "$\langle \bullet \bullet \rangle$" operator constructs classes so that

$$\langle A \bullet M \bullet I \rangle$$

denotes the class formed by the private attributes[2] in A, the methods in M, and the initializers (constructors) in I. The "$\otimes$" operator puts attributes together, so the expression

$$\langle A \otimes B \bullet M \bullet I \rangle$$

denotes the class formed by the attributes in A and in B, besides the methods and initializers[3] respectively in M and I.

Using those operators, we can concisely formalize refinement laws. For example, consider methods m and m' having the same signature. Then, provided that m is refined by m', denoted $m \sqsubseteq m'$, we have that

$$\langle A \bullet M \oplus m \bullet I \rangle \sqsubseteq \langle A \bullet M \oplus m' \bullet I \rangle$$

This law basically indicates that method refinement implies class refinement. The same notation is used to formalize the laws of Pim, precisely indicating how the classes of application prototypes should be progressively transformed to consider distribution, concurrency, and persistent aspects.

3 Introducing Distribution

One of the constraints that Pim imposes on design methods is the use of the Facade design pattern [3]—which provides a unified interface for all services of a subsystem—in order to structure applications. Hence Pim assumes that the functionally complete and local prototype has *business facade classes* such as the following:

[1] For simplicity, we consider that a method just consists of activities, which are described by a group of tasks.

[2] Hereafter we consider that classes have only private attributes; so we omit the word "private" for brevity.

[3] Usually called "constructors" in Java.

```
class MyBusinessImplementation implements MyBusiness {
   private CustomerFile customers;
   private ProductList products;
   ⋮
   void addCustomer(Customer customer) {
      customers.add(customer);
   }
   void setPrice(ProductCode code, double price) {
      products.setPrice(code,price);
   }
}
```

where the classes `CustomerFile` and `ProductList` provide services for the insertion, updating, querying, and deletion of customer and product records[4].

The objects of business facade classes are precisely the ones that should be distributed or available for remote access. So Pim also assumes that the functionality of those objects is abstracted by corresponding *business facade interfaces*:

```
interface MyBusiness {
   void addCustomer(Customer customer) throws CommunicationException;
   void setPrice(ProductCode code, double price)
      throws CommunicationException;
   ⋮
}
```

which explicitly indicate that subsystem services might not be available due to problems in the communication or distribution infrastructure and basic services. This is necessary because business facade interfaces abstractly represents the services of a subsystem, which might be local or remote to the clients of those services.

Fortunately, in order to transform the local prototype into a distributed one, we do not need to change the business facade classes and interfaces. Pim simply suggests the use of object adapters [3] for them. The adapters basically encapsulate the RMI code that is necessary for allowing the distributed or remote access of business facade objects. In this way, the business logic layer becomes totally independent from the RMI communication layer, so that changes in the latter layer do not impact the former layer.

There are two kinds of adapters: *source adapters* and *target adapters*. Roughly, the latter wrap business facade objects in the places where they are located, and the former represent those objects in remote locations. In a typical client-server system, user interface objects would request the services of a source adapter located in the client machine. The source adapter would then request the services of a corresponding target adapter located in the server machine. Finally, the target adapter would request the services of a business facade object also located in the server side.

3.1 Target Adapters

A target adapter contains a reference to a `MyBusiness` object—likely a business facade object—and methods that simply invoke corresponding methods on that object. For example, the following is a target adapter classs for any class that implements `MyBusiness`:

[4]For brevity, we omit the "`public`" qualifier from type and method declarations. We also assume that the illustrated methods raise no exceptions.

```
class MyBusinessTargetRMIAdapterImplementation
   extends UnicastRemoteObject implements MyBusinessTargetRMIAdapter {
   private MyBusiness myBusiness;
   ⋮
   void addCustomer(Customer customer)
      throws CommunicationException, RemoteException {
      myBusiness.addCustomer(customer);
   }
}
```

where the code inherited from the RMI class `UnicastRemoteObject` allows target adapters to be remotely accessed, and `RemoteException` is raised by RMI to notify communication or configuration problems during the call of a remote method [7].

As target adapters shall be used as remote objects, we must also have an interface with the same signature as the target adapter class:

```
interface MyBusinessTargetRMIAdapter extends Remote {
   void addCustomer(Customer customer)
      throws CommunicationException, RemoteException;
   ⋮
}
```

where `Remote` is an RMI interface used to identify remote object types; in particular, `Remote` is indirectly implemented by `UnicastRemoteObject` [7].

Note that the methods of target adapters may indirectly raise `CommunicationException` because they invoke methods of `MyBusiness` objects, instead of a particular implementation of that interface. This gives an extra flexibility, as discussed later.

3.2 Source Adapters

A source adapter contains a reference to a target adapter and basically calls the methods of the second, catching RMI specific exceptions and replacing them by `CommunicationException`. For example, observe the following source adapter class for any class that implements the interface `MyBusiness`:

```
class MyBusinessSourceRMIAdapter implements MyBusiness {
   private MyBusinessTargetRMIAdapter myBusiness;
   ⋮
   void addCustomer(Customer customer) throws CommunicationException {
     try {
        myBusiness.addCustomer(customer);
     } catch (RemoteException exception) {
        throw new CommunicationException(exception.getMessage());
     }
   }
}
```

This class implements `MyBusiness` so that source adapters effectively represent the services provided by their corresponding business facade objects. For instance, user interface classes typically declare variables of type `MyBusiness`. In the functionally complete and local prototype, those variables are instantiated with `MyBusinessImplementation` objects. In the distributed prototype, those variables are instantiated with source adapters.

In fact, note that a target adapter might refer either to a business facade object or to a source adapter. This gives us extra flexibility in such a way that a source adapter sa_1 might refer to a target adapter ta_1 that refers to a source adapter sa_2 and so on, until we finally have that ta_n refers to a business facade object. In this way we can easily support various configurations for a distributed system.

3.3 Distribution Tasks

In summary, Pim's tasks for transforming a local prototype into a distributed prototype include the following: create specific source and target adapter classes; replace, where appropriate, instances of business facade classes for source adapters; configure and execute target adapters as server processes; properly handle communication exceptions raised by source adapters. Furthermore, as RMI remote method calls actually copy argument and result objects, we must declare the parameter types and result types of remote methods as subtypes of `Serializable`, a Java interface that guarantees class serializability. In addition, we must insert update operations in order to guarantee that updates to object copies are reflected in the original objects. For example, clients of `MyBusiness` typically write code such as the following:

```
Customer customer = myBusiness.fetchCustomer(customerId);
customer.setName(...);
customer.setAddress(...);
```

But, if `myBusiness` holds a reference to an RMI source adapter, `customer` will not hold a reference to an original customer. So, in order to reflect the changes in the original customer, we should add the following method call at the end of that code:

```
myBusiness.updateCustomer(customer);
```

where the `MyBusiness` method `updateCustomer` is responsible for fetching and updating the original customer with the attribute values of its argument.

4 Dealing with Concurrency

After performing the tasks presented in the previous section, the business facade objects are ready to be distributed, but are not ready for concurrent access, which likely comes as a consequence of distribution. In fact, the concurrent access to the methods of business facade objects might generate a vast range of undesirable interferences, since the distributed prototype code uses no synchronization mechanisms.

In order to avoid undesirable interferences and preserve the semantics of the original functionally complete prototype, Pim suggests tasks for bridging the gap between Java applications developed to be used in sequential and concurrent environments [1]. Those tasks are based on practical guidelines for safely introducing, moving, and removing synchronization mechanisms without leading to deadlock, livelock, or to undesirable interference. Such guidelines are, for example, derived from common design patterns for structuring concurrent object-oriented programs [6].

Pim's synchronization laws (see Section 2) formalize those guidelines. Each law typically helps to correctly increase either liveness or safety. For example, a useful law for increasing liveness establishes that we can remove the Java `synchronized` qualifier[5] from a method as long as some constraints are satisfied:

[5] The methods qualified as `synchronized` cannot be executed while there is another `synchronized` method being executed in the same object [4].

$$\langle A \otimes b \bullet M \oplus \texttt{synchronized}(m) \bullet I \rangle \sqsubseteq \langle A \otimes b \bullet M \oplus m \bullet I \rangle$$

provided that

- *body*(m) is in the form "`b.n(`$\overline{\texttt{e}}$`)`", where *name*(b) is `b`, `n` is any method name, and $\overline{\texttt{e}}$ is an arbitrary list of expressions formed only by variables and constants[6]; and
- *type*(b) is a fully synchronized class—its methods are all synchronized, and it has no public instance variables [6].

Roughly, this law establishes that it is sound to remove synchronization from a method m that simply invokes another method that is accessed in a serial way. The justification is that m will be serially accessed anyway.

Pim's tasks for transforming a sequential prototype into a prototype adapted to be used in a concurrent environment are roughly the following: using synchronization laws, introduce `synchronized` qualifiers to the methods of business facade classes, and also to the methods of *business collection classes* such as `CustomerFile`; whenever possible, remove the `synchronized` qualifiers from the methods of business facade classes, using the law illustrated in this section, for example; apply other synchronization laws to further improve liveness.

By using Pim, the business facade class presented in the previous section would first receive `synchronized` qualifiers:

```
class MyBusinessImplementation {
   private CustomerFile customers;
   private ProductList products;
   ⋮
   synchronized void addCustomer(Customer customer) {
      customers.add(customer);
   }
   synchronized void setPrice(ProductCode code, double price) {
      products.setPrice(code,price);
   }
}
```

which can be safely introduced without leading to deadlock because this facade class satisfies some constraints. In fact, we can guarantee that the objects of the class above are safe; they have the same observational [8, 9] behaviour no matter if executed in a sequential or concurrent environment. Moreover, this behaviour is equivalent to the behaviour of the original business facade objects (defined on Section 3) when they are executed in a sequential environment.

After achieving safety, we could benefit from the synchronization law presented in this section for optimizing `MyBusinessImplementation`. The `synchronized` qualifiers would be safely removed, assuming that the classes `CustomerFile` and `ProductList` serialize the insertion, updating, querying, and deletion of customer and product records.

5 Conclusion

We have given an overview of the Pim method for the systematic implementation of distributed object-oriented applications. Pim supports the progressive transformation of an structured and functionally complete application prototype into a distributed, concurrent, and persistent

[6]We assume that method bodies are well typed, so that we do not need to impose further constraints on `b`, `n`, and $\overline{\texttt{e}}$.

version of the application. We have concentrated here on distribution and concurrency aspects because of scope and space reasons. Persistence aspects are considered elsewhere [10].

Pim's progressive approach can significantly reduce the impact caused by requirements changes during development, since most changes likely occur before distribution, concurrency, and persistence aspects are introduced to the application. Furthermore, Pim naturally helps to tame the complexity inherent to distributed systems, by supporting the gradual testing of the various intermediate versions of the application. In this way, we can, for example, isolate problems in the business logic layer from problems in the persistence and communication layers.

Pim is not useful for any kind of application. It is only useful when concurrency and distribution are not inherent to the application, but are necessary for performance and fault tolerance reasons only. For example, Pim has been quite useful for the implementation of distributed information systems in different domains. In particular, Pim has been partially used to implement the following applications: a customer management system for a telecommunications company; a retail store system; and a mobile dairy sales management system. From our limited practical experience so far, managers and engineers are initially skeptical about the method, mainly because it implies rework on some classes. However, after using Pim they usually get convinced of its benefits.

As Pim is just an implementation or coding method, it should be carefully integrated to design and testing methods in order to be used in practice. In the experiences reported above, Pim has been integrated either to the Rational's Objectory process [5] or to *ad hoc* processes. Also, besides fully describing the method, we should provide tool support (via component and IDE wizards) for Pim, since most tasks are tedious and can be largely automated. The current version of Pim is specific to Java and RMI, but we intend to generalize it so that we can support other middleware technologies such as CORBA and DCOM.

References

[1] Paulo Borba. Systematic development of concurrent object-oriented programs. In *US–Brazil Joint Workshops on the Formal Foundations of Software Systems*, volume 14 of *Electronic Notes in Theoretical Computer Science*. Rio de Janeiro, Brazil, 5-9th May, 1997, and New Orleans, USA, 13-16th November, 1997.

[2] Paulo Borba. Where are the laws of object-oriented programming? In *I Brazilian Workshop on Formal Methods*, pages 59–70, Porto Alegre, Brazil, 19th–21st October 1998.

[3] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[5] Ivar Jacobson. *The Unified Software Development Process*. Addison-Wesley, 1999.

[6] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.

[7] Sun Microsystems. *Remote Method Invocation Specification*, 1998.

[8] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[9] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[10] Euricélia Viana and Paulo Borba. Integrando Java com bancos de dados relacionais. In *III Brazilian Symposium on Programming Languages, submitted*, 1999.

Contents

1 Introduction **1**

2 Laws of Progressive Implementation **2**

3 Introducing Distribution **2**

3.1 Target Adapters 3

3.2 Source Adapters 4

3.3 Distribution Tasks 5

4 Dealing with Concurrency **5**

5 Conclusion **6**