



# White Paper

## Using Model-Driven Architecture™ to Develop Web Services

By David Frankel, Chief Consulting Architect, IONA  
Technologies,

and John Parodi, Principal Writer, IONA Technologies

**IONA Technologies PLC**

**Second Edition**

**April, 2002**

Portions adapted, with permission, from "Model-Driven Architecture" by David Frankel, to be published by the John Wiley & Sons OMG Series in 2002.

**iPortal Application Server is a Trademark of IONA Technologies PLC.**  
**IONA e-Business Platform is a Trademark of IONA Technologies PLC.**  
**IONA Enterprise Integrator is a Trademark of IONA Technologies PLC.**  
**IONA Mainframe Integrator is a Trademark of IONA Technologies PLC.**  
**Adaptive Runtime Technology is a Trademark of IONA Technologies PLC.**  
**Total Business Integration is a Trademark of IONA Technologies PLC.**  
**IONA SureTrack is a Trademark of IONA Technologies PLC.**  
**IONA XMLBus is a Registered Trademark of IONA Technologies PLC.**  
**Orbix is a Registered Trademark of IONA Technologies PLC.**  
**Orbix 2000 Notification is a Registered Trademark of IONA Technologies PLC.**  
**Orbix/E is a Registered Trademark of IONA Technologies PLC**  
**E2A is a Registered Trademark of IONA Technologies PLC**  
**"End 2 Anywhere" is a Registered Trademark of IONA Technologies PLC**

**Object Management Group is a trademark of the Object Management Group**  
**OMG is a trademark of the Object Management Group**  
**CORBA is a Registered Trademark of the Object Management Group**  
**Model-Driven Architecture is a trademark of the Object Management Group**  
**MDA is a trademark of the Object Management Group**  
**Unified Modeling Language is a trademark of the Object Management Group**  
**UML is a trademark of the Object Management Group**

**Design by Contract is a trademark of Interactive Software Engineering**

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA Technologies PLC shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

**COPYRIGHT NOTICE**

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this white paper. This publication and features described herein are subject to change without notice.

Copyright © 1999-2002 IONA Technologies PLC. All rights reserved.

All products or services mentioned in this white paper are covered by the trademarks, service marks, or product names as designated by the companies that market those products.

David Frankel is IONA's Chief Consulting Architect, and has worked in the area of distributed computing for many years. Mr. Frankel consults with IONA customers and internal groups and is a member of the Object Management Group (OMG) Architecture Board. He has been involved in the formulation and evolution of a number of OMG standards, including COM-CORBA Interworking and UML™.

John Parodi is a Principal Writer in IONA's Information Design and Development group. He has more than twenty years of experience in technical communications and has produced trade press articles, technical overviews, presentations, and award-winning user documentation on topics that include programming, security, architecture, and enterprise-scale integration.

## Executive Summary

This paper describes how the principles of the Object Management Group's Model Driven Architecture™ (MDA™) can be used to develop Web services in a way that improves programmer productivity and avoids rapid obsolescence of the services. MDA uses the Unified Modeling Language™ (UML™, a widely-accepted standard managed by the OMG) to capture and express the essence of information and services, and is thus the basis for MDA.

In developing Web services, technology is secondary to the information and the services that use and create the information. Our goal should be to provide an environment in which those who produce individual Web services can do so in a way that is as independent of specific Web service implementation technologies as possible. The purpose of such an approach is to protect the investment in Web services as the underlying technologies change.

From the perspective of Web services, MDA involves using UML to specify services precisely and in a technology-independent manner. MDA promotes Web services that are based on traceable business requirements. It also sets the stage for automatic generation of at least part of the XML and code, such as Java code, that implements the services. Finally, it makes it easier to re-target the services to use different Web services implementation technologies when required.

The design and implementation of Web services should be approached within the context of a multi-tiered enterprise architecture that supports multiple access channels to the services. B2B business document (XML) exchange, fat clients, browsers, and wireless devices are the most common access channels.

The automation of business processes and "choreographies" of business-to-business interactions is at the frontier of MDA. Defining and generating such cooperative processes is much more difficult than simply defining and generating Web services, but progress is being made on this front.

Overall, MDA is still in its early stages. There are significant benefits to be gained from MDA now, but its full potential will be reached only after a number of years of evolution.

# Table of Contents

Executive Summary .....	iii
1 Introduction .....	1
1.1 First- and Second-Generation Web Services Integration .....	2
1.2 Raising the Level of Abstraction .....	3
1.3 Web Services and Enterprise Architecture.....	4
2 The Unified Modeling Language™ .....	6
2.1 Informal vs. Formal Models .....	6
2.2 Generating Implementations from Formal UML Models.....	12
2.2.1 Mapping Business Information Models to XML.....	12
2.2.2 Mapping Business Service Models to WSDL.....	14
2.3 UML Profiles .....	14
2.4 Relevant Standards.....	16
3 Automating Business Processes and Choreographies.....	16
4 Model-Driven Architecture: The Big Picture .....	19
5 Conclusions.....	20
6 References.....	21
7 Further Reading.....	22
8 Contact Details.....	23

# 1 Introduction

Consider what might be called the “Sisyphus Syndrome” in IT. Homer tells the story of Sisyphus, who was fated to repeatedly push a rock up a hill, whereupon the rock would roll down the hill and force Sisyphus to push it up the hill again.

It is easy to view an IT manager as a modern day Sisyphus in her efforts to maintain the viability of applications in the face of fast-changing technologies and business requirements. Thus the critical goal is to reduce the amount of labor-intensive work that developers have to do in reaction to such changes.

Web services can be developed in a way that will result in Sisyphean labours, or we can take this opportunity to do better. Consider the following statement from a prestigious trade journal:

“There is no question that if Web services are to take off as smoothly as vendors hope, a significant chunk of the more than 20 million programmers in the world will have to write to UDDI, WSDL, XML, and SOAP.”

—from “Web Services,” *InfoWorld*, vol. 23, issue 11,  
March 12, 2001, page 39

We at IONA question this assertion. The approach will not scale, and is not viable in the long term, for several reasons.

First, the technologies for Web services are in flux, and there are also several different ways of stacking these technologies, as we shall see later. For example, the original eXtensible Markup Language (XML) specification has evolved from DTDs to XML Schemas, and we know there will be further evolution. The Simple Object Access Protocol (SOAP), Universal Description, Discovery and Integration (UDDI), and the Web Services Description Language (WSDL) are new and still evolving.

The technologies implementing Web services depend on Port 80. We must ask whether Port 80 will remain what it was originally intended to be—that is, a secure firewall “peephole”—when we begin multiplexing huge numbers of Web service messages through that port.

The point is that these technologies are bound to change. Having Web services developers program directly to these technologies invites rapid obsolescence and is also far too labor-intensive.

MDA lets us design Web services at a more abstract level than that of technology-specific implementations. Systems built using MDA exhibit more flexibility and agility in the face of technological change—as well as a higher level of quality and robustness, due to the more formal and accurate specification of requirements and design.

We in the technical community have a tendency to fall in love with technology, in this case SOAP, WSDL, and so on. Thus we may fail to see

the forest for the trees. We must step back from the technology in order to understand that it is the information, and the services that use and create the information, that are of primary importance.

The breakthrough concept of Web services is that this information can be exposed and accessed programmatically over the Internet. While Web service implementation technologies are important, they must be kept in perspective, for they will change while the breakthrough concept survives.

Our goal should be to provide an environment in which production of individual Web services can be done in a way that is as independent of these particular technologies as possible. The purpose of such an approach is to protect the investment in Web services as the underlying technologies change. Further, this approach promotes the ability to generate technology-specific Web service implementations automatically or semi-automatically, rather than forcing programmers to construct them entirely by hand.

## 1.1 First- and Second-Generation Web Services Integration

The first generation of Web services integration has resulted in products that, in essence, project existing objects and components (for example, CORBA®, Java, or COM constructs) as Web services. There are tools that generate the required WSDL, SOAP, and other XML files, and that also generate code (such as Java code) that binds the Web services to the intermediate and back-end tiers.

These tools relieve the programmer of much tedious work, and they meet the requirement of avoiding intensive hand coding of technology-specific artifacts. Programmers “turn the crank,” so to speak, and the tool generates a great deal of the necessary XML, WSDL, SOAP, and Java artifacts.

However, Web services will be (and must be) coarser-grained than many of the already-existing objects and components, because the business functions we need to expose to other businesses generally contain much more application logic than is present in any single existing object or component. Web services will present abstractions of finer-grained functions that already exist; in other words, Web services will be compositions of more primitive functionality.

Therefore we must actually *design* Web services. Second-generation Web services integration will entail Web services whose design is driven by business requirements and by the need to minimize network traffic from fine-grained interactions. We cannot limit ourselves to a mechanical approach that exposes as Web services the objects that we are already using, because those individual objects are not likely to provide the level of function needed.

Once a specific Web service has been identified according to these criteria, designers need to answer the following questions:

- What is the information that needs to be manipulated?
- What is the functionality that must be provided?

If we can answer these questions in a formal way, the answers can be used to generate the artifacts that implement the services over some set of technologies. The Web services design vocabulary should let us describe the information and services in ways that are entirely independent of XML, WSDL, SOAP, UDDI, Java, and other Web service implementation technologies.

The trend is toward tools that can automate production of XML, WSDL, SOAP, UDDI, and implementation code from design input. But in order for this approach to be viable, tools must also allow fine-tuning by engineers who are intimately familiar with these Web services technologies.

One approach for supporting fine tuning is to provide configurable generators that present engineers with a range of options as to how to generate the necessary artifacts. Another approach, which is controversial, is to allow engineers some freedom to modify generated artifacts. The latter approach requires tools to be smart enough not to overwrite engineer's modifications when the design is enhanced and the generator executes again.

In sum, the tools must provide a way to capture the answers that knowledgeable business people supply to the above questions about required information and functionality. But engineers also need to have some control over the code that is generated based on the answers that business experts provide.

## 1.2 Raising the Level of Abstraction

The fundamental principle here is raising the level of abstraction. This is part of a general trend that is already well established for dealing with the front and back ends of systems. On the front end we already do "what-you-see-is-what-you-get" (WYSIWYG) modeling for graphical user interfaces (GUIs), and on the back end we do data modeling.

Thus, hand coding is no longer the predominant way we produce GUI and database systems. However, we do allow qualified personnel to fine-tune the behavior of these systems, via supplemental hand coding on the front end and via various tuning parameters and stored procedures for the back-end databases. Thus we achieve a higher degree of development process automation, yet still allow a necessary degree of flexibility.

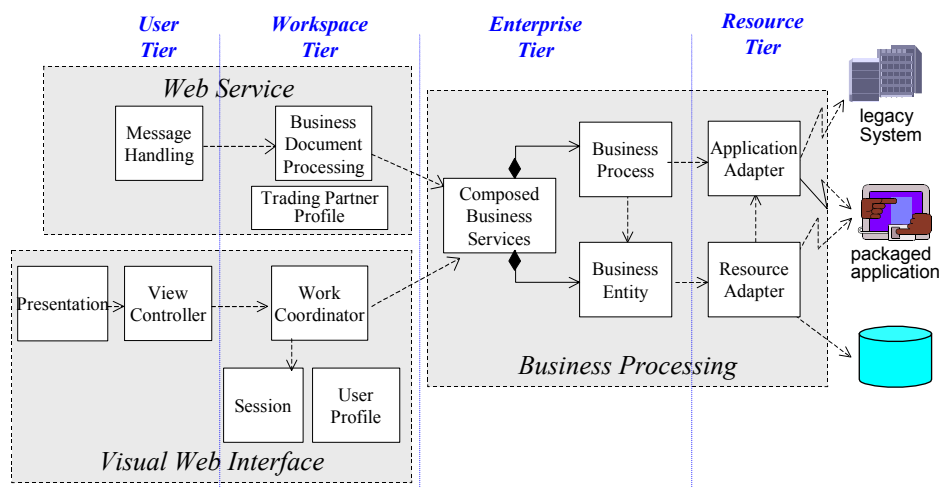
Hence the principle of raising the level of abstraction above that of 3GL code is already well established for some kinds of systems. We would

never consider going back to hand coding enterprise database systems and GUIs, because we rely on the productivity benefits of being able to produce all or most of our GUI and database systems from higher level models.

The MDA approach offers similar productivity gains for Web service development. In raising the level of abstraction, it also lessens or avoids disruptions to the system in the face of change. We can apply MDA to Web services in order to increase the resilience of our implementations as Web services technologies evolve.

### 1.3 Web Services and Enterprise Architecture

Figure 1 depicts a four-tier architecture for distributed systems<sup>1</sup>. These four tiers are an evolution of the familiar three-tier distributed system architecture. A three-tier architecture has a front end that is concerned with presentation, a back end that is concerned with data and legacy systems, and a middle tier that provides business functions.



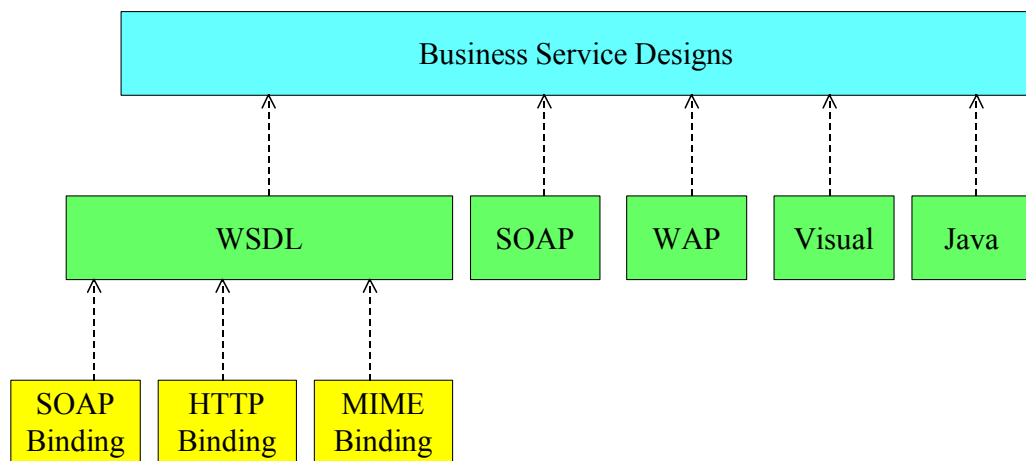
**Figure 1: Web Services Integration Architecture**

The difference between traditional three-tier architecture and the four-tier architecture shown here is that the presentation tier is divided into a user tier and a workspace tier. This is done for reasons of architectural separation; the device-dependent aspects of presentation are placed in the user tier and the device-independent aspects are placed in the workspace tier.

<sup>1</sup> This figure is an adaptation of a figure from a Webcast and white paper by Mike Rosen, IONA's Chief Enterprise Architect. See [HGR]. The fundamental concepts behind four-tier architecture are derived from Herzum and Sims' groundbreaking work [HS].

Another important point shown here is that the business services, which are composed from lower-level, finer-grained business functions and information entities, are separated from how they are presented. That is, the same business service might be presented as a Web service, as a Web page, as an application screen, or as a message to a wireless hand-held device. Because we want to reuse these composed business services in these different contexts, we locate their core logic in the enterprise tier (which we used to call the “middle tier”). This core logic is independent of the implementation technologies used to expose the service, such as WSDL, HTML, WAP (Wireless Access Protocol), and so on.

When we decide to expose the business service as a Web service, there are a number of technologies to choose from. We might expose it via WSDL bindings. WSDL itself has the ability to bind to SOAP, simple HTTP, or MIME, so there are several variants of WSDL-based implementation from which to select. We might decide not to use WSDL at all and instead to expose the service via SOAP directly. Regardless of which technology stack we use, we would still like to use the same enterprise tier business services, as illustrated by Figure 2.



**Figure 2: Maximizing Reuse Over Different Technology Stacks**

Early Web applications tended to wire Web front ends directly onto system back ends, and in that way some companies have avoided building intermediate tiers. However, Web services and B2B require one or more intermediate tiers to expose coarse-grained, reusable business services. By encapsulating these services in a fashion that is independent of the technical mechanisms we use to expose them, we make it possible to reuse them in any number of contexts.

This separation of the executable service functionality from the technologies used to access them dovetails with the requirement to

formally capture the essence of a service in a fashion that is independent of WSDL, SOAP, and so forth. Let us now examine how to meet this requirement.

## 2 The Unified Modeling Language™

The Unified Modeling Language (UML) provides a medium to create designs that are precise enough to drive code generators and yet abstract enough to remain independent of technology.

UML is a widely accepted standard managed by the Object Management Group (OMG). It is suitable for capturing and expressing the essence of information and services, and is thus the basis for OMG's Model-Driven Architecture (MDA).

### 2.1 Informal vs. Formal Models

To date, much of the use of UML has been for informal modeling, in which UML is used to sketch out the basic concepts of a system. For informal modeling, UML has significant advantages over typical "box and line" diagrams, because the UML notation—its shapes and line types—have very specific meanings. In contrast, the meaning of the boxes and lines in a typical "system diagram" is subject to the interpretation of the reader.

Informal modeling is a fine use for UML, but informal models cannot drive code generators or dynamic execution engines. An analogous situation exists with text; an informal textual description describing the functions of a program cannot be compiled and executed the way corresponding formal source code can be.

In contrast to informal UML models, formal UML models are precise and computationally complete. But one might ask, if the models are supposed to be abstract, what does it mean to make them precise?

First, we must understand that precision and detail are *not* the same thing. For example, imagine that I have an electric mixer. Inside there is an electric motor, gears, and so forth, but none of these are visible from the outside because of the mixer's casing. Abstraction is "the suppression of irrelevant detail" [ISO 10746-2]. One can suppress the detail of the motors and gears inside, leaving a more abstract view from the outside.

The external, more abstract view might be very precise. For example, one might measure the circumference of the casing to a millionth of an inch. In such a case we would say that the outside view of the mixer is extremely precise, but it is not particularly detailed because you can't see what is inside the casing. On the other hand, one can imagine a view of the mixer that lets you see the details of the inside, namely the

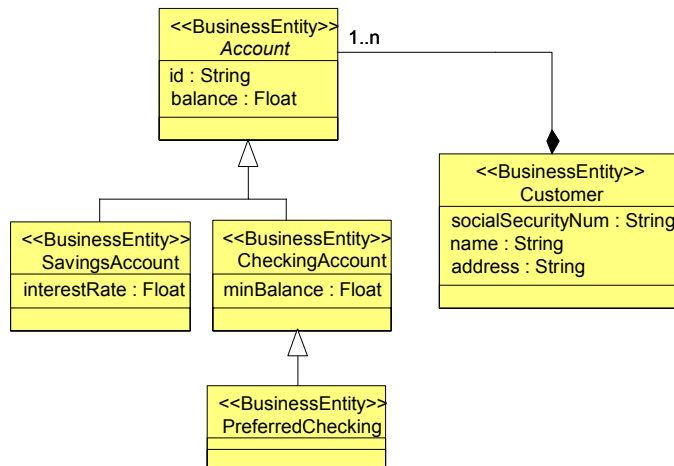
motors and gears, but where the measurements of the inside items are very rough. This is a view that is detailed but imprecise.

Thus we see that detail and precision are different. In a formal UML model, the visible elements are, and must be, precise enough to drive code generation. Let us consider what “precise” means in this context.

A formal UML model must be syntactically complete. In an informal UML model it is common, and accepted, to leave out certain properties, such as the multiplicities for some of the associations among model elements. Or an attribute may be defined to be of a certain type, such as Date, but the informal modeler may not have taken the trouble to ensure that the Date type was defined somewhere in the model.

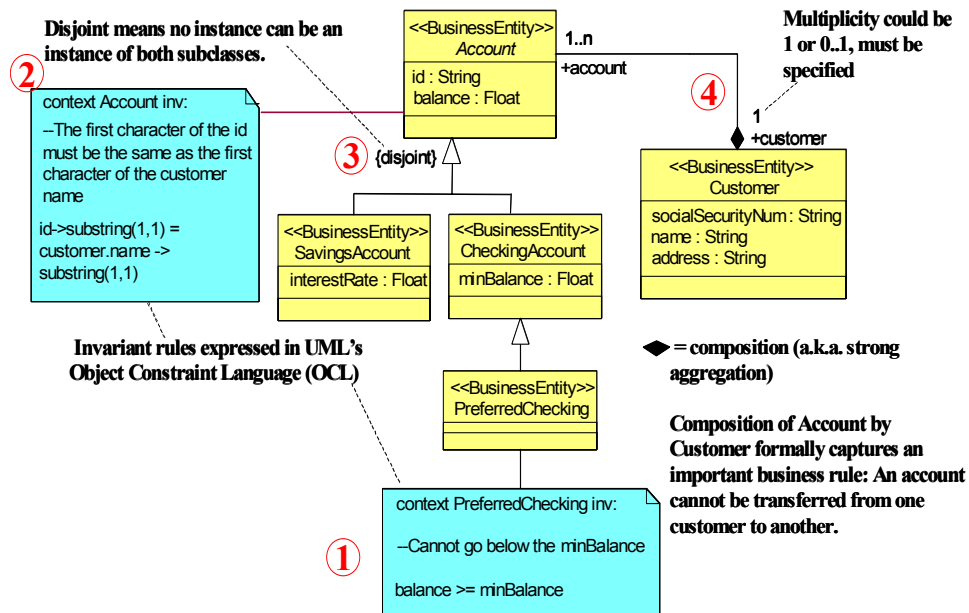
In a formal UML model, this kind of looseness is not acceptable. Syntactical incompleteness makes it impossible for a code generator to process the model. This behavior is similar to what we see in traditional programming, where an undeclared identifier or an incomplete expression causes a compiler to reject a program.

Figure 3 shows a UML business information model for a bank. This model shows the notion of an account, some subtypes such as savings account and checking account, and a subtype of checking account called preferred checking.



**Figure 3: Business Information Model—Imprecise and Incomplete**

This model may look fairly precise and complete at first glance, but Figure 4 shows the same model in a form that is more precise and complete.



**Figure 4: Business Information Model—More Precise and More Complete**

The box (1) attached to the preferred checking class declares a constraint called an *invariant* or *invariant rule*. An invariant expresses an assertion about a class that must always be true in order for an instance of the class to be well-formed.

The invariant attached to preferred checking states that the balance for a preferred checking account is not permitted to go below the minimum balance established for the account. The invariant is expressed in English as well as in Object Constraint Language (OCL), which is part of UML. The identifiers `balance` and `minBalance` are inherited from ancestor classes. The properties that the OCL expression references must be defined in the model or else a generator will not be able to process the OCL successfully.

Another invariant rule (2) is attached to the `Account` supertype class. It says that the first character of the account's ID must be the same as the first character of the customer's name.

Note that, without a formal model, it may be very difficult to discover these invariants. They may exist only deep within code or, if we are fortunate, informal textual descriptions of the system requirements may be available. By contrast, the invariants shown in Figure 4 are explicit and formal, and expressed independently of specific implementation technologies. Implementation and technology-dependent aspects of the system are suppressed at this level of abstraction.

Defining the rules in a way that is independent of implementation technology makes it easier for business experts to validate them and to

re-implement the same logic over different implementation technologies.

We have all seen cases where business rules have been captured only in the code. In some cases, the programmer has made a unilateral decision, when that decision should have been made in consultation with business experts. Even if the programmer consults a business expert before making such an implementation decision, and implements the function according to the expert's wishes, it is still very difficult to find the "audit trail" for that decision. This is a serious problem in enterprise development, where it is critical to maintain the traceability of requirements. If you don't know the requirement that a piece of code addresses, it becomes difficult to maintain that code if the requirement changes.

Another difference between the imprecise and precise models is the constraint (3) on the subtyping of `Account` into `SavingsAccount` and `CheckingAccount`. In Figure 4, the constraint says explicitly that the partitioning of `Account` is *disjoint*, meaning that you cannot have an instance that is both a `SavingsAccount` and a `CheckingAccount`. A violation of this constraint is in effect a violation of an important business rule.

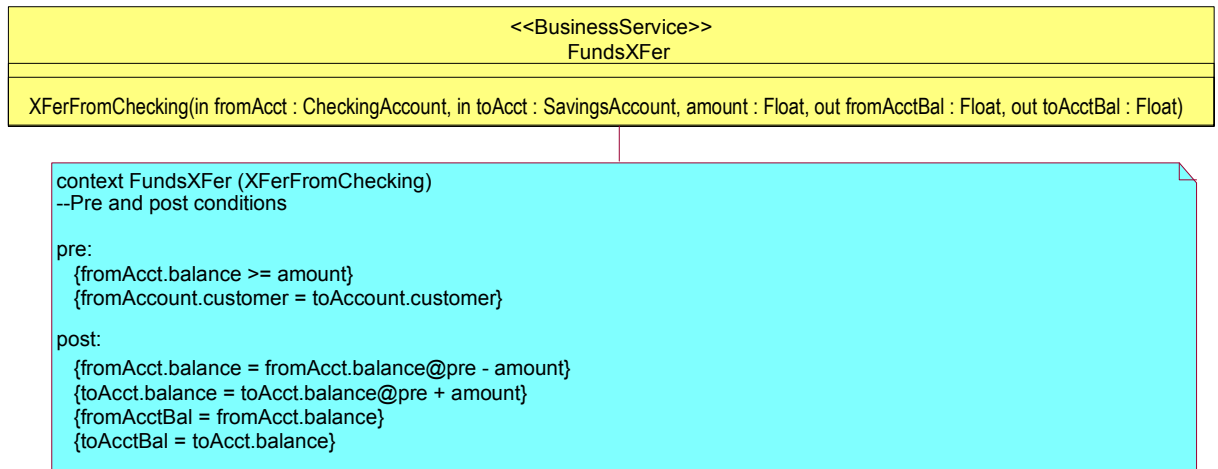
Another thing left unspecified in the less-precise model is the `Customer`-side multiplicity (4) of the association between `Customer` and `Account`.<sup>2</sup> Code generators make important inferences based on these multiplicities. Note that the fact that this association is denoted as a composite aggregation is not simply a technical choice made by a technical person. It captures an important business rule, which is that an account cannot be transferred from one customer to another. That account, for its lifetime, is bound to one specific customer.

Figure 5 illustrates one way to define a business service in UML. The service is called `FundsXfer` (funds transfer), and it has an operation called `XferFromChecking` (transfer from checking). This UML operation has three input parameters. One is the checking account from which

---

<sup>2</sup> In UML notation, a black diamond denotes a strong form of aggregation called *composite aggregation* or *composition* for short. Omission of the multiplicity on the aggregate side of a composite aggregation association is a common mistake; modelers often are under the impression that the multiplicity on the aggregate side of the association is always 1 (that is, 1..1) by definition, and thus don't bother to fill it in. In our particular case, the multiplicity is explicitly 1, which means that under no circumstances can there be an account that stands alone without a customer who owns it. However, there are cases where the multiplicity on the composite side of such an association can be 0..1, so the modeler must specify this multiplicity—as is the case for the multiplicities on both sides of *all* associations in a formal UML model.

the transfer is made, another is the savings account to which the transfer is made, and the third is the amount to transfer from the checking to the savings account. The operation also defines two output parameters that represent the ending balances of the checking and savings account, respectively.



**Figure 5: Business Service Model Using Design By Contract™**

There are other approaches to creating this specification. For example, one might reify the operation into a class, and then make the parameters attributes of the class. There are advantages and disadvantages to each approach.

In either case, simply specifying the signature of the operation—that is, the names and data types of the parameters—is not enough. In order to specify the characteristics of this service more precisely, we declare *pre-conditions* and *post-conditions*. A pre-condition is a constraint that asserts something that must be true in order for the operation to start executing. A post-condition is a constraint that asserts something that must be true when the operation finishes executing.

The example shown in Figure 5 shows only the OCL constraints and does not show them in English. There are two pre-conditions:

- The balance in the source checking account must be greater than or equal to the amount to be transferred to the target savings account. Note that the source account's type is `CheckingAccount`, which leverages our information model from Figure 4. The `CheckingAccount.balance` identifier in the constraint explicitly references the property `balance` of `CheckingAccount` in that information model.
- Both accounts must be owned by the same customer. When this constraint specifies `fromAccount.customer` and `toAccount.customer`, `".customer"` traverses the association

between `Account` and `Customer` shown in Figure 4. In other words, the customer is being treated as a property of the account.

Figure 5 also shows four post-conditions, that is, conditions that must be true when the operation has finished executing:

- The source account's balance must be equal to what the balance was before this operation occurred, minus the amount of the transfer. The `from.Account.balance` identifier to the left of the equal sign refers to the balance of the account after the operation executes; `from.Account.balance@pre` refers to the balance of the account before the operation executes.
- The target account balance must be equal to what the balance was before the operation, plus the amount of the transfer.
- The final two post-conditions define the semantics of the two output parameters.
  - The first says that the output parameter `fromAcctBal` must equal the balance of the account from which funds were transferred, that is, `fromAcct.balance`. This makes the purpose of the output parameter explicit: to provide the ending balance of the account after the operation is complete.
  - Similarly, the other post-condition says that the output parameter `toAcctBal` must equal the ending balance of the account to which funds were transferred, that is, `toAcct.balance`.

Invariant rules and pre/post-conditions, such as the ones that we have just examined, are the essential elements of an approach to rigorous design called *Design by Contract™ (DBC)*. The basic foundations of DBC were formulated by E. W. Dijkstra in the 1970's [DIJ]. Later day advocates such as Bertrand Meyer [MEY] and Haim Kilov [KR] built upon his work, which they readily acknowledge.

DBC and other formal engineering disciplines are often overlooked in the software industry because of relentless short-term production pressures. One can hardly blame development managers, whose resources are stretched thin by aggressive schedules, for this behavior. DBC advocates respond that DBC takes more time up front but saves time in the long run because it tends to produce better quality software. But this argument does not help the manager who has to make short term milestones in order to survive to the long term.

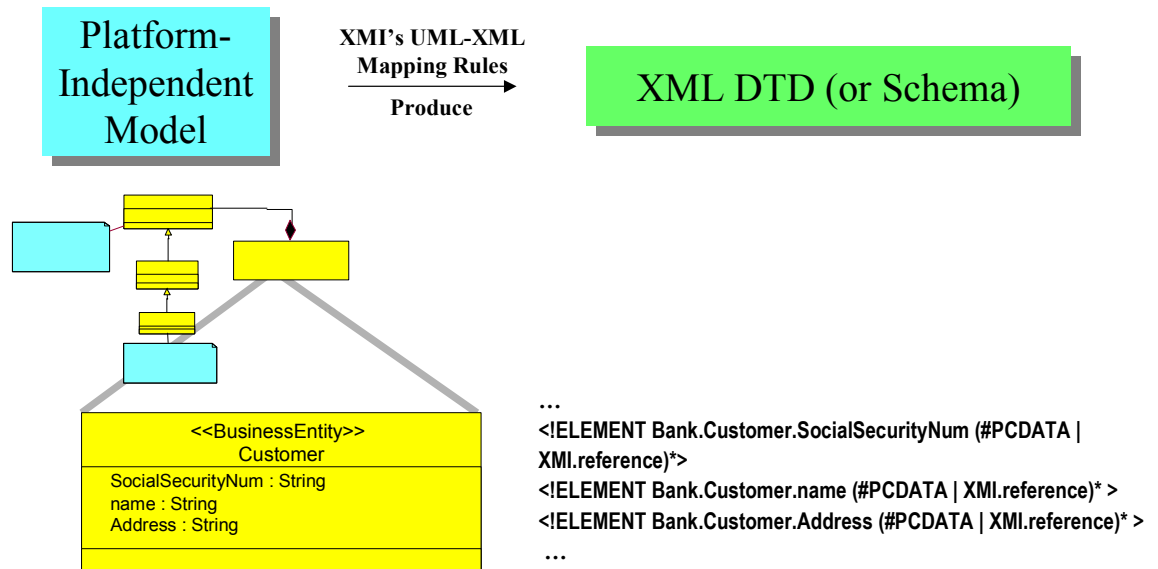
However, the combination of MDA and DBC promises time savings right from the start, because succinct invariants and pre/post-conditions can drive code generators that relieve the programmer of having to write a lot of code. Tools that parse OCL and generate code from it are beginning to be available. Examples include Sun's NetBeans Metadata Repository [NB] and the Adaptive Repository [ADAP].

## 2.2 Generating Implementations from Formal UML Models

### 2.2.1 Mapping Business Information Models to XML

There are a number of different approaches to mapping formal UML models to implementation technologies. One approach is embodied in an OMG standard called XML Metadata Interchange (XMI). This standard defines mapping rules that specify how to generate an XML Document Type Definition (DTD) or Schema from a class model. The generated DTD or Schema defines a format for representing instances of the classes in XML documents [XMI].

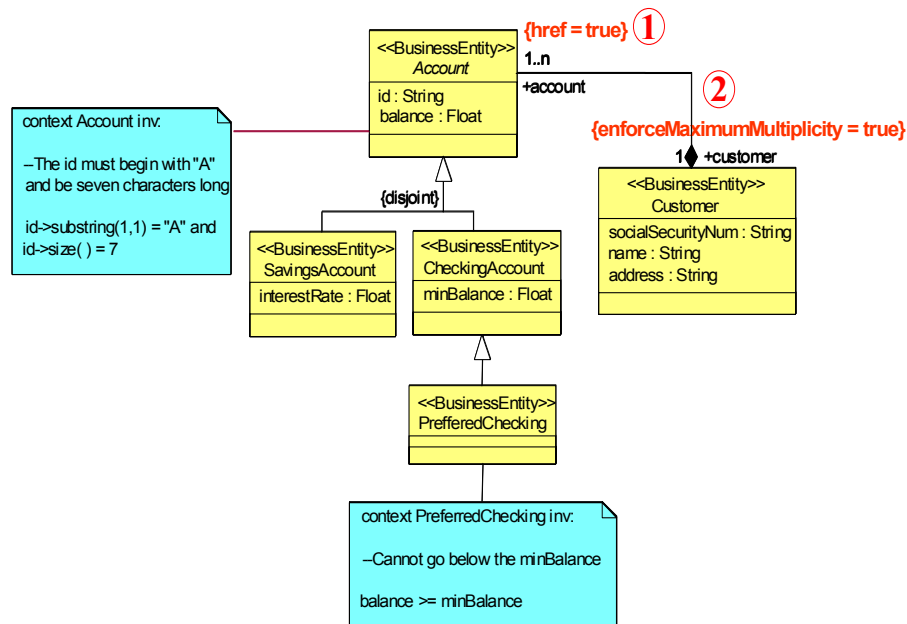
Figure 6 shows the results of running the class model shown in Figure 4 through an XML generating tool. The result is a DTD, and the lower right portion of Figure 6 shows a fragment of that generated DTD which corresponds to the `Customer` class. It shows XML DTD definitions for the three attributes of `Customer`, namely `Social Security Number`, `Name`, and `Address`. Although it isn't evident from the small fragment of the generated DTD shown here, the various semantic elements of the model, such as the properties of associations, affect how the DTD is generated.



**Figure 6: Mapping the Business Information Model to XML**

As mentioned above, the notion of generating concrete software artifacts from abstract models is viable only if it allows engineers who understand the concrete technologies to fine-tune the generation rules. XMI gives the engineer a number of choices governing XML production. Each of these choices is expressed as a parameter. A mapping that

provides such choices is called a *parameterized mapping*<sup>3</sup>. The engineer indicates her choices by decorating the model with actual values of those parameters, using UML tagged values.



**Figure 7: Fine-Tuning XML Generation Using XMI**

XMI defines a number of parameters that can be added to a model to control how XML generation is done. Figure 7 shows two such parameters, affixed to the ends of the `Account/ Customer` association. These two parameters are just a small sample of those available.

The parameter (1) on the `account` end of the association `{ref = true}` specifies that, when an XML `Account` element references `customer`, HREFs (rather than IDREFs) are used.

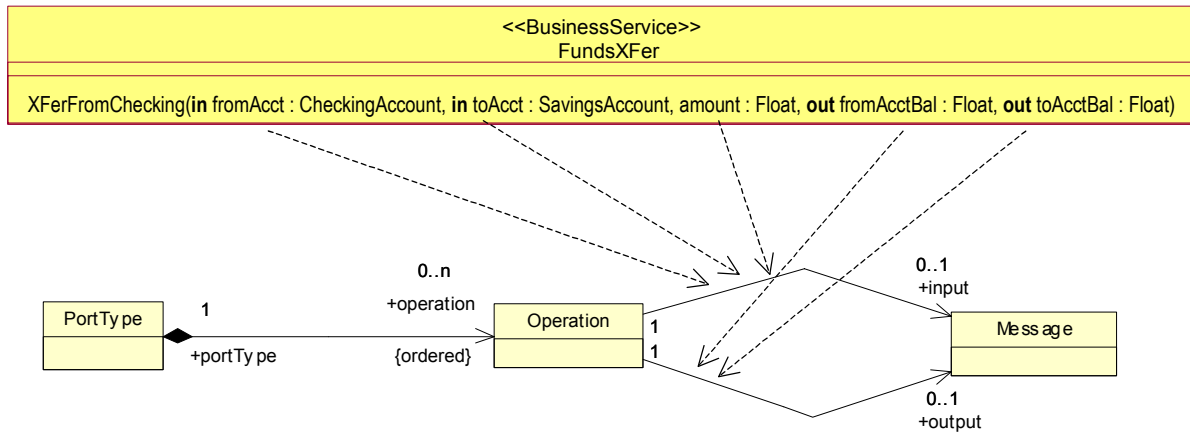
The parameter (2) on the `customer` end of the association `{enforceMaximumMultiplicity = true}` means that, in the generated XML Schema, the maximum number of customers for an account (which is 1) should be declared. If the value of this parameter is false, the generated XML Schema declares the cardinality<sup>4</sup> as unbounded rather than as 1.

<sup>3</sup> Computer Aided Software Engineering (CASE) tools are an example of previous attempts at raising the level of abstraction in system development. CASE tools have not been widely successful, and a significant factor in this lack of success was that they do not provide engineers with the level of control that parameterized mappings offer.

<sup>4</sup> UML *multiplicity* and XML Schema *cardinality* are similar concepts.

## 2.2.2 Mapping Business Service Models to WSDL

Mapping a business service model to WSDL is an exercise in mapping the input parameters to WSDL input messages and the output parameters to WSDL output messages, as illustrated in Figure 8.



**Figure 8: Mapping the Business Service Model to WSDL**

The boxes in the lower part of Figure 8 correspond to fragments of a formal model of WSDL itself. Here we see that WSDL defines port types, which own operations. Operation definitions reference, but do not own, message definitions, with some messages playing the role of input parameters for the operation, and some playing the role of output parameters.

The XML formats of the message payloads can be derived by applying an information model mapping (such as XML), since all of the input and output parameter types of the business service's UML definition are defined by the information model of Figure 4.

Eventually there will be generators that automatically implement mappings to WSDL, but for now they are done manually. However, the tool that automatically produced the DTD from a class model, as shown in Figure 6, exists today.

## 2.3 UML Profiles

A UML profile is a UML dialect defined via UML's built-in extension mechanisms. The UML specification [UML] explains how to define a profile; it involves selecting a subset of UML and then using the UML extension facilities to extend that subset.

For example, the information model shown in Figure 4 uses a *stereotype* `<<Business Entity>>` to indicate that certain classes represent business information entities (the double angle brackets denote a stereotype). So here we have extended the notion of a UML class to create a new type

of class that represents business information entities. Stereotypes can be used to extend not only the UML class construct, but also to extend any of the other UML modeling constructs such as attribute, operation, association, and so on.

Similarly, Figure 5 uses a stereotype `<<Business Service>>` to indicate that the class represents a business service. A code generator can be written to recognize that a `<<Business Service>>` is not the same as a `<<Business Entity>>` and to treat it differently.

Another useful UML extension mechanism is the *tagged value*. Tagged values let you define modeling properties that UML does not support “out of the box.” For example, when you define an association in UML, it does not allow you to state whether or not the reference at each end should be implemented as an XML HREF or, alternately, by an IDREF.

The tagged values in Figure 7 use a profile that defines the tags `href` and `enforceMaximumMultiplicity`. UML tools provide ways to insert values of the defined tags—that is, tagged values—in model elements<sup>5</sup>.

The examples shown in this section illustrate the two fundamental uses of UML profiles. One is to model a particular domain, such as business information and business services. There are UML profiles being created for other domains such as real-time systems and telecommunications. The other purpose of UML profiles is to parameterize mappings to technologies, as we did in the example shown in Figure 7. In the context of Web services, we use one profile for modeling the technology-independent aspects of a service, and others for parameterizing the mappings to specific implementation technologies.

In our example we have combined two profiles. The profile that includes the definitions of the stereotypes `<<Business Entity>>` and `<<Business Service>>` supports specifying the abstract properties of the system. The other profile includes the definitions of the tags `href` and `enforceMaximumMultiplicity`, which support specifying the XML aspect of the system.

A good UML tool makes it easy to remove and restore all of the elements of a model that stem from one particular profile. This makes it easy to: 1) maintain a purely abstract version of the model, which would not include, for example, the XML aspects of the system, and 2) superimpose technology-dependent aspects that, for example, would include the XML aspects that an XML generator would need.

---

<sup>5</sup> Not all tools support this in the same fashion. Some require you to drill down into the dialogue that contains the detailed properties of the model element in order to insert the tagged value. Some require you to load the tag definitions before you can insert values of the tags in model elements.

It should be noted that UML's profiling mechanisms impose fairly sharp limits on defining new modeling constructs. A sister standard to UML, called the Meta Object Facility (MOF™), provides much more freedom to define new modeling constructs that extend or supplement UML [MOF]. MOF is beyond the scope of this paper, but it is an important element of MDA.

## 2.4 Relevant Standards

MDA is a strategic architecture for the OMG. The model-driven approach is also being adopted by other industry standards organizations, notably ebXML and RosettaNet.

ebXML is an electronic business standard developed by the United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT) and the Organization for the Advancement of Structured Information Standards (OASIS). It is an adjunct to the UN's EDIFACT EDI standard. The ebXML Architecture involves describing abstract information and service models in UML, and defining mappings that support automatic generation of XML-based artifacts from the model. A UML profile called UN/CEFACT Modeling Methodology [UMM] is driving this work.

RosettaNet is a consortium companies also defining standards for B2B integration. It is gradually moving to a UML-based approach with automatic mappings to generate XML-based artifacts.

## 3 Automating Business Processes and Choreographies

The automation of business processes and choreographies is at the frontier of B2B integration. A *choreography* is a coordinated set of interactions among two or more trading partners.<sup>6</sup> Choreographies are also sometimes called *collaborations*.

Defining and supporting business processes and choreographies is a more complex problem than defining and supporting Web services. Choreographies may in fact be compositions of a number of Web services supplied by the trading partners, where each Web service plays a particular role in the choreography. This section briefly surveys this topic.

In order to do real e-Commerce, we must be able to specify both business processes and choreographies. The specific messages that are exchanged as part of a choreography trigger business processes within

---

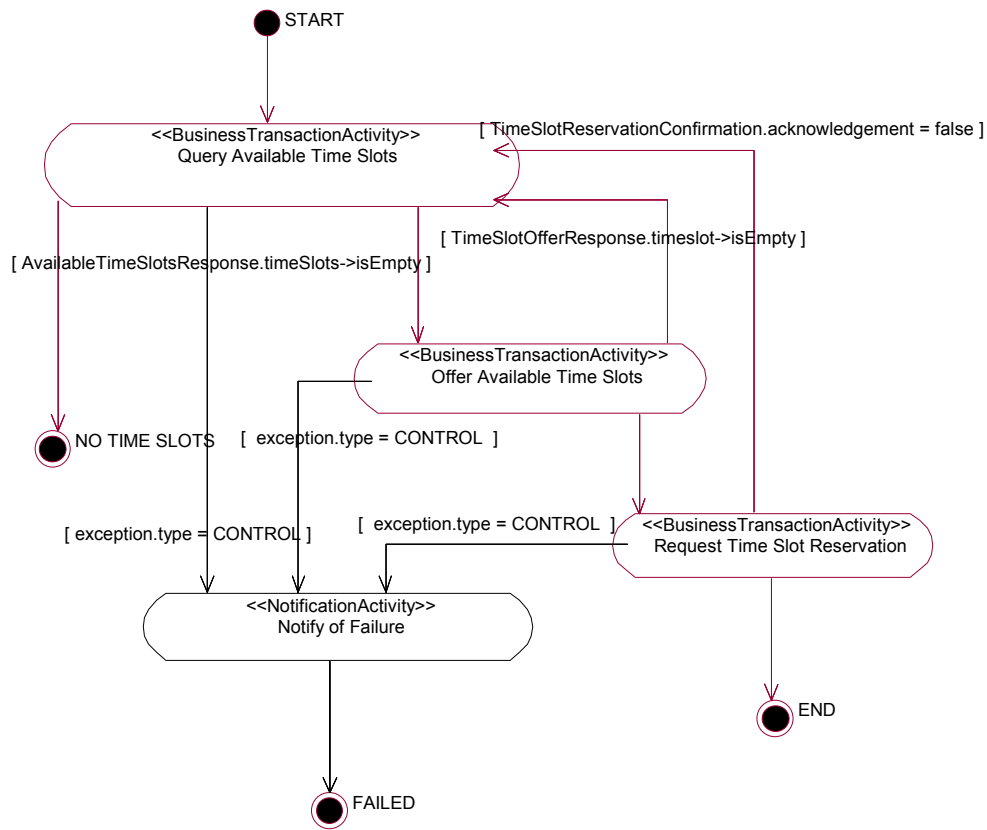
<sup>6</sup> The trading partners might be different companies or might be different administrative domains of the same company.

the trading partners' own systems. Thus both business process definition and interaction pattern definitions are important in this technology.

We have a mix of concrete technologies for describing these things, including ebXML's Business Process Specification Schema and RosettaNet's Partner Interface Processes (PIPs). What is central here are the processes and interactions, rather than these specific technologies.

We can use UML activity and interaction models to define processes and choreographies in a technology-independent fashion. We can also define mappings to the various implementation technologies.

Figure 9 is an example of a UML activity model, which is similar in some respects to a traditional flowchart. This activity model describes a business process in which a telecommunications company queries which field service time slots are available to a customer, examines the available time slots returned, and requests a reservation for a time slot. This type of activity model can drive a generator that produces code and XML for some implementation technology.



**Figure 9: UML Activity Model<sup>7</sup>**

Figure 10 is a UML interaction model. It specifies a choreography of messages among various parties in support of the business activity shown in Figure 9. This kind of interaction pattern is crucial to B2B commerce. Such interaction models can provide additional input to generators.

OMG, ebXML, UMM, and RosettaNet are laying a foundation of standards for business process and choreography definition. They are also developing standard mappings that specify how to map these definitions to various implementation technologies. Such a standards-based edifice is required if B2B commerce is to succeed on a large scale.

<sup>7</sup> Reused, with permission, from [CLARK].

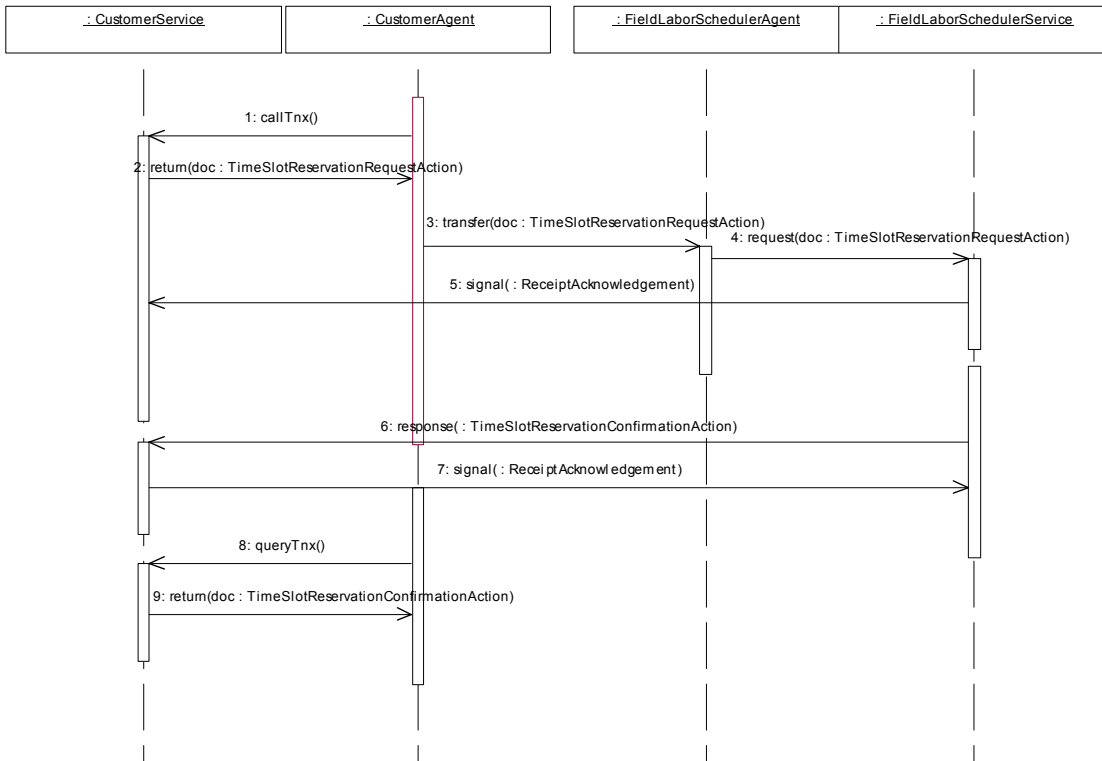


Figure 10: UML Interaction Model<sup>8</sup>

## 4 Model-Driven Architecture: The Big Picture

Model-Driven Architecture is part of a broad effort across the industry to raise the level of abstraction in how we develop systems. This is not the first time in the history of the computer industry that we have raised the level of abstraction to improve productivity. We did exactly this when we moved from assembly language to third-generation languages (3GLs).

The change to 3GLs did not happen overnight and the industry experienced some teething problems along the way. At first, 3GL compilers did not produce code as optimal as hand crafted machine code. Over time, however, the productivity increase justified the changeover, especially as computers speeded up and compiler technology improved. Today, programming in machine language for business applications is virtually unheard of because it would be less productive by an order of magnitude.

<sup>8</sup> Reused, with permission, from [CLARK].

Like Web services integration, MDA can be over-hyped. It is important to understand that the transition to MDA will unfold incrementally. The industry cannot and will not “turn on a dime” to adopt it, and there are some very tough problems that must be solved along the way. Although there are already some useful tools based on MDA, its full power will not be realized for a number of years. But the change, though gradual, will be profound.

## 5 Conclusions

MDA is about using modeling languages as programming languages. Modeling languages make it possible to program systems at a higher level of abstraction than is possible using languages such as Java and XML, thus improving development productivity. Using UML profiles or MOF, you define modeling constructs appropriate to a specific domain. You then create generators that can transform a system specification written using those constructs into a partial or full implementation of the system.

When we specify systems at a higher level of abstraction we also increase their longevity, because the specifications are less tied to underlying computing environments that are always in flux.

Using modeling languages as programming languages also puts rigorous design techniques such as Design by Contract (DBC) in a new light. Without MDA, DBC is an arduous quality improvement process that takes much more time up front but pays off in the long run. With MDA, code and testing harnesses can be generated from formal constraints, so that using DBC has the potential to improve short-term productivity as well.

Our simple UML profile for abstract business information and business service specification, along with XMI's profile for selecting XML production options, exemplify the MDA approach. Generators based on mappings of these languages to XML, WSDL, and other implementation technologies form the other crucial piece of an MDA framework for Web services.

MDA principles and MDA-based standards will contribute to the emergence of a second generation of Web services integration technology. Companies can prepare for the advent of this technology by becoming familiar with defining Web services in a technology-independent fashion.

## 6 References

- [ADAP] Adaptive Repository, Adative Ltd.,  
[http://www.adaptive.com/assets/downloads/adaptive\\_repository.pdf](http://www.adaptive.com/assets/downloads/adaptive_repository.pdf)
- [CLARK] Material circulated on ebXML email discussion lists by Jim Clark of I.C.O.T.
- [DIJ] E. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.
- [HGR] P. Harmon, M. Guttman, and M. Rosen, *Developing E-Business Systems and Architectures: A Manager's Guide*, Kaufmann, November, 2000.
- [HS] P. Herzum and O. Sims, *Business Component Factory: A Comprehensive Overview of Component Based Development for the Enterprise*, John Wiley and Sons, March, 2000.
- [ISO 10746-2] *ISO RM-ODP Part 2*, ISO/IEC 10746-2:1996
- [KR] H. Kilov, J. Ross, *Information Modeling: An Object-Oriented Approach*, Prentice-Hall, 1994.
- [MEY] Meyer, Bertrand, *Object-Oriented Software Construction*, Second Edition, Prentice-Hall, 1997.
- [MOF] ] *Meta Object Facility Specification Version 1.4*,  
<http://cgi.omg.org/cgi-bin/doc?ptc/01-08-22>
- [NB] NetBeans Metadata Repository, Sun Microsystems,  
<http://mdr.netbeans.org>
- [UML] *UML Specification Version 1.4*, <http://cgi.omg.org/cgi-bin/doc?ad/01-02-13>
- [UMM] *UN/CEFACT Modeling Methodology*,  
<http://www.ebtwg.org/projects/documentation/bioreference/>
- [XMI] *XML Metadata Interchange (XMI) Version 1.2*,  
<http://cgi.omg.org/cgi-bin/doc?ptc/01-08-27> and XMI Production for XML Schema, <http://cgi.omg.org/cgi-bin/doc?ptc/01-12-03>

## 7 Further Reading

1. D. Frankel, *UML Profiles and Model-Centric Architecture*, Java Report, June 2000, Volume 5, Number 6, p. 110.
2. D. Frankel, *The OMG Meta Object Facility*, Java Report, March, 1999, Volume 4, Number 3, p. 56.
3. D. Frankel, *XMI: The OMG's XML Metadata Interchange Standard*, XML Journal, Volume 1, Issue 4, p. 6.
4. IONA Technologies. *IONA E2A Web Services Integration Platform Product Brief*, January 2002.
5. IONA Technologies. *IONA E2A Application Server Platform*, January 2002.
6. IONA Technologies. *Preparing for Web Services*, December 2001.
7. IONA Technologies. *Architecting Web Services*, December 2001.
8. IONA Technologies. *Using Model-Driven Architecture™ To Define Web Services*, April 2002.
9. IONA Technologies. *XMLBus Edition Technology Overview*, April 2002.
10. IONA Technologies. *Enterprise Security in Web Services*, March 2002.
11. IONA Technologies. *Introduction to Orbix E2A J2EE Technology*, April 2002.
12. IONA Technologies. *A Detailed Look at IONA's J2EE Technology*, April 2002.
13. IONA Technologies. *Orbix E2A Application Server Clustering and Load Balancing*, April 2002.
14. IONA Technologies. *CORBA-EJB Interoperability*, April 2002.

IONA White Papers can be downloaded at [www.ionq.com](http://www.ionq.com).

## 8 Contact Details

IONA Technologies PLC  
The IONA Building  
Shelbourne Road  
Dublin 4  
Ireland  
Phone: ..... +353 1 637 2000  
Fax: ..... +353 1 637 2888

IONA Technologies Inc.  
200 West St  
Waltham, MA 02451  
USA  
Phone: ..... +1 781 902 8000  
Fax: ..... +1 781 902 8001

IONA Technologies Japan Ltd  
Akasaka Sanchome Bldg 7/F  
3-21-16 Akasaka  
Minato-ku, Tokyo  
Japan 107-0052  
Phone: ..... +813 3560 5611  
Fax: ..... +813 3560 5612

Support: ..... [support@iona.com](mailto:support@iona.com)  
Training: ..... [training@iona.com](mailto:training@iona.com)  
Orbix Sales: ..... [sales@iona.com](mailto:sales@iona.com)  
IONA's FTP site ..... [ftp.iona.com](ftp://ftp.iona.com)

**World Wide Web:**     [www.iona.com](http://www.iona.com)  
  
                                 [www.xmlbus.com](http://www.xmlbus.com)