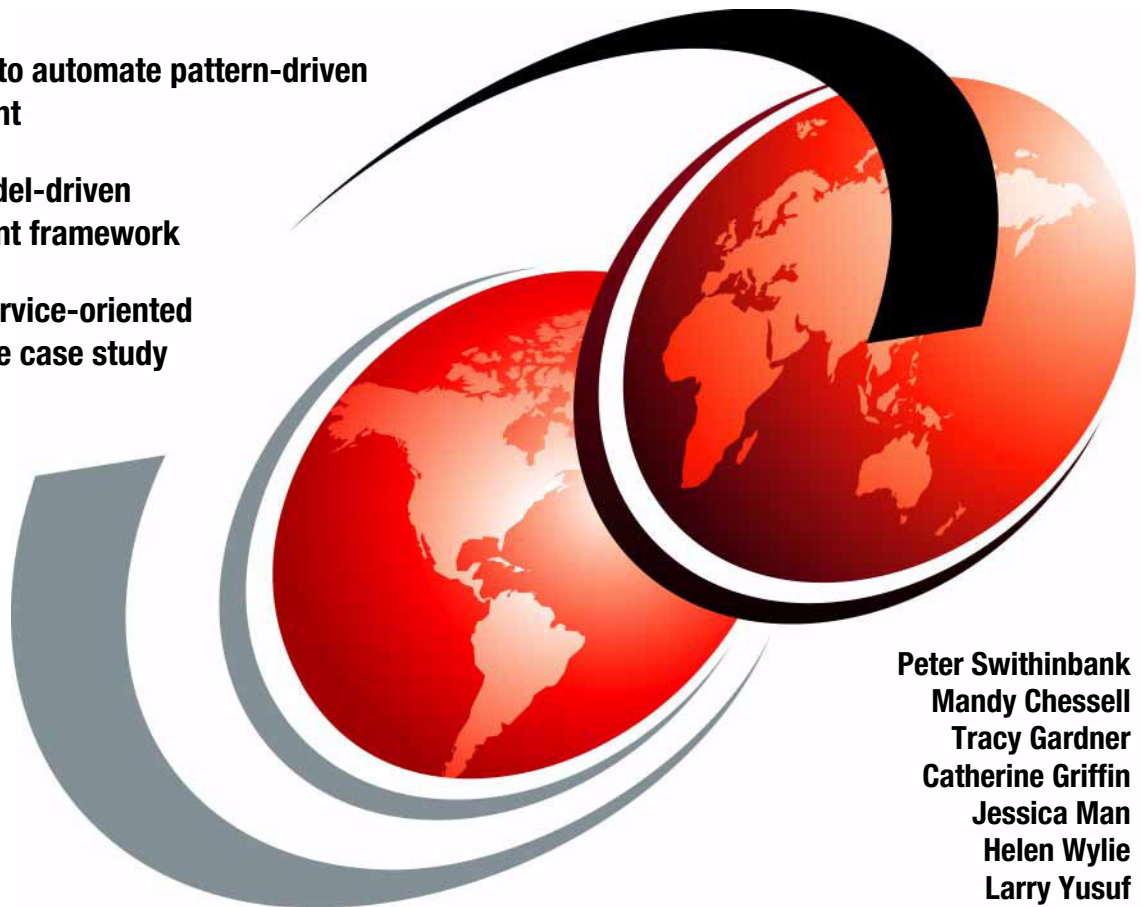IBM

Patterns: Model-Driven Development Using IBM Rational Software Architect

Learn how to automate pattern-driven development

Build a model-driven development framework

Follow a service-oriented architecture case study

Peter Swithinbank
Mandy Chessell
Tracy Gardner
Catherine Griffin
Jessica Man
Helen Wylie
Larry Yusuf

Redbooks

IBM

International Technical Support Organization

Patterns: Model-Driven Development Using IBM Rational Software Architect

December 2005

Note: Before using this information and the product it supports, read the information in
"Notices" on page ix.

First Edition (December 2005)

This edition applies to Version 6.0.0.1 of Rational Software Architect (product number 5724-I70).

Contents

iii

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

CICS®	IBM®	RUP®
developerWorks®	Rational Unified Process®	SoDA®
@server®	Rational®	WebSphere®
@server®	Redbooks (logo) ™	

The following terms are trademarks of other companies:

EJB, Java, Javadoc, JVM, J2EE, RSM, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Preface

"The convergence of patterns, models and tooling sets the scene for major increases in application development productivity by 2010. Now is a good time to get on board."

Jonathan Adams, Distinguished Engineer, IBM® Academy of Technology

You may read this IBM Redbook for a number of reasons. Perhaps you are already familiar with the ideas behind model-driven development (MDD), and you want to learn about how to put those ideas into practice and how to convince others in your organization to adopt the approach. Maybe you heard something about the benefits of MDD but want to learn more about it before you are convinced that it is suitable for your project. Or you recently joined an MDD project and need to understand what it is all about.

This IBM Redbook is written for technical practitioners and project managers who want to learn more about MDD in practice. It will help you understand how to put the ideas of MDD into practice using Unified Modeling Language (UML). You will learn how to articulate the advantages of MDD to both project managers and technical colleagues. You will see how the MDD software life cycle differs from other approaches and how you can effectively plan and manage an MDD project. If you are already working on an MDD project, you will learn how to use Rational® Software Architect to carry out your work.

For solution architects

Solution architects can expect to learn about the benefits of a model-driven approach to software development and how to determine whether it is a suitable approach for your project.

You will learn how to put the ideas of MDD into practice using Rational Software Architect (RSA). You will also learn how to articulate the advantages of MDD to both project managers and technical colleagues.

For project planners or project managers

If you are a project planner or a project manager, you will learn about the benefits of a model-driven approach to software development and how to determine whether it is a suitable approach for your project.

You will also learn how the MDD software life cycle differs from other approaches and how you can effectively plan and manage an MDD project.

For those working on a project that uses model-driven development

If you are reading this book because you are working on a project that has adopted MDD, then you will learn about the MDD approach: how an overall MDD project works and what your role is. You will also learn how to use RSA to carry out your work.

The following chapters are recommended based on particular roles:

► If you are a solution architect or developer responsible for developing an application using an existing MDD framework, read all chapters, but in particular read:

  – Chapter 1, "Overview and concepts of model-driven development" on page 3
  – Chapter 2, "Scenario overview" on page 17
  – Chapter 3, "Model-driven development approach" on page 29
  – Chapter 7, "Designing patterns for the scenario" on page 93
  – Chapter 8, "Applying model-driven development with Rational Software Architect" on page 129

► If you are a solution architect responsible for developing an MDD framework, read all chapters, but in particular read:

  – Chapter 1, "Overview and concepts of model-driven development" on page 3
  – Chapter 2, "Scenario overview" on page 17
  – Chapter 3, "Model-driven development approach" on page 29
  – Chapter 6, "Model-driven development in context" on page 73
  – Chapter 7, "Designing patterns for the scenario" on page 93
  – Chapter 9, "Extending Rational Software Architect" on page 161

► If you are a project manager, the following chapters may interest you the most:

– Chapter 1, "Overview and concepts of model-driven development" on page 3
– Chapter 3, "Model-driven development approach" on page 29
– Chapter 4, "Model-driven development project planning" on page 45
– Chapter 5, "Model-driven development solution life cycle" on page 59
– Chapter 10, "Conclusion" on page 219

► If you are a pattern/transformation developer, the following chapters may interest you the most:

– Chapter 1, "Overview and concepts of model-driven development" on page 3
– Chapter 3, "Model-driven development approach" on page 29
– Chapter 8, "Applying model-driven development with Rational Software Architect" on page 129
– Chapter 9, "Extending Rational Software Architect" on page 161

How this book is organized

In this book, we present MDD as an approach to improving on the mainstream software development practice. MDD treats UML models as primary software artifacts from which consistent implementation artifacts can be generated.

In Part 1, "Approach" on page 1, we describe the ideas behind MDD. We also explain how to apply MDD in practice using Rational Software Architect by introducing a scenario generalized from a number of real-world projects in which the authors participated. The scenario involves the development of integration services within a service-oriented architecture (SOA).

In Part 2, "Implementation" on page 91, we explain how to apply MDD to the scenario. The information you need to reproduce the examples is included in the additional materials provided with this redbook.

The team that wrote this redbook

A team of specialists from around the world working at the International Technical Support Organization (ITSO), Hursley Center, produced this redbook.

Peter Swithinbank is a project leader at the ITSO, Hursley Center. He writes IBM Redbooks and teaches IBM classes worldwide building business integration solutions. Peter has worked for IBM for 27 years and has been with the ITSO for one year. He has a diploma in software engineering from Oxford University and an MA in Geography from the University of Cambridge.

Mandy Chessell is a Senior Technical Staff Member (STSM) in the U.K. She has 18 years of experience in the middleware field. She holds a master degree in software engineering from the University of Brighton, U.K. Her areas of expertise include distributed transaction processing, object-oriented design, usability, and UML modeling.

Dr. Tracy Gardner is a solution architect in the IBM Software Group Services in the U.K. She has over five years of industrial experience in model-driven development. She has written and presented extensively on model-driven development. She holds a Ph.D. in software engineering from the University of Bath.

Catherine Griffin is a software engineer at IBM Hursley in the U.K. She has 10 years of experience in software development. Her areas of expertise include Eclipse and the Eclipse Modeling Framework. She holds a degree in mathematics from Nottingham University, U.K.

Jessica Man is an IT specialist in the U.K. She has five years of experience in software development. Her areas of expertise include the JVM™ and tooling for Java™ and J2EE™ application development, deployment, runtime problem determination, and performance measurement. She holds a master degree in advanced computer science from the University of Manchester.

Helen Wylie is a certified consultant IT architect in Hursley Architectural Services in the U.K. She has many years of experience in the IT industry and has focussed primarily on integration architectures in the last decade. Her areas of expertise are SOA and integration, and most recently the combination of patterns and models to support automated generation of services and deployment artifacts for deployment to an enterprise service bus (ESB). She holds a mathematics degree from the Open University and a post graduate diploma in computer science from Cambridge University.

Larry Yusuf is a solution designer with Software Group Strategy and Technology based at the Hursley Labs in the U.K. He has four years of experience in Business Integration and modeling, with a particular focus on Business Process Management, Event and Solution Management, and Integration patterns. He has written and presented extensively on these topics.

Credits and acknowledgements

Without the sponsorship of Jonathan Adams, this book would not exist. We are all indebted to Jonathan for championing the adoption of pattern-based techniques in software architecture. He also was an assiduous reviewer.

We also thank Ian Scott who worked with members of the team in applying these techniques to a large IT project. As well, we thank Charles Rivet, Lee Ackerman, Lisa Noon and David Kelsey for reading and commenting on the first draft for us.

We also want to acknowledge the help of our many IBM colleagues, many formerly of Rational, who are working with us to apply modeling techniques to SOA and integration.

Become a published author

Join us for a two-to-six week residency program! Help write an IBM redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will team with IBM technical professionals, Business Partners, or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at the following Web site:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Send us your comments about this or other redbooks in one of the following ways:

► Use the online **Contact us** review redbook form found at:

   ibm.com/redbooks

► Send your comments in an e-mail to:

   redbook@us.ibm.com

► Mail your comments to:

   IBM Corporation, International Technical Support Organization
   Dept. HZ8 Building 662
   P.O. Box 12195
   Research Triangle Park, NC 27709-2195

Part 1

Approach

This part describes model-driven development (MDD) as an approach to software development. We explain how to organize a software development project to use the MDD approach. Part 2, "Implementation" on page 91, takes you through the application of MDD to a hypothetical scenario that draws upon the experience of the team in real engagements.

► Chapter 1, "Overview and concepts of model-driven development" on page 3, provides a brief overview of what MDD is and describes some of its benefits.

► Chapter 2, "Scenario overview" on page 17, describes the scenario we selected to use throughout the book. MDD is not applicable to every scenario. We discuss the indications that suggest MDD is effective for the scenario we chose.

► In Chapter 3, "Model-driven development approach" on page 29, we step lightly into the waters of MDD and explain the key ideas behind model-driven development that we apply in the second part of the book.

► We wrote Chapter 4, "Model-driven development project planning" on page 45, with the project planner in mind, and we discuss how to introduce and manage an MDD project.

1

- ► Chapter 5, "Model-driven development solution life cycle" on page 59, continues in the planning vein and looks at the way MDD affects aspects of the solution life cycle that are not discussed elsewhere in the book.
- ► MDD is a rapidly advancing field, and in Chapter 6, "Model-driven development in context" on page 73, we discuss some of the currents in MDD and how they relate to some other aspects of software development.

1

Overview and concepts of model-driven development

In this chapter, we describe the issues associated with enterprise solution development. Then we present model-driven development (MDD) as an approach to improving on established mainstream practices. We also introduce IBM Rational Software Architect (RSA) as a tool for supporting MDD.

3

1.1 Current business environment and drivers

IT development does not take place in isolation. The purpose of IT is to facilitate the operations of a business. The needs of the business environment drive the way we develop IT.

Current business drivers include:

► *The On Demand Business*: As businesses are expected to be more adaptable and flexible, so too are the IT systems that enable them.

► *Business relevance*: Now more than ever, there is a strong focus on IT departments to deliver business value. Software must be business relevant. Miscommunication between business and IT people can lead to projects that, successful from an IT-delivery viewpoint, are deemed business failures.

► *Cost control*: The days of IT being invested in on the strength of its promises are long gone. IT departments now operate under strong budget constraints and are expected to demonstrate value for money.

► *Increasing complexity*: Software systems continue to increase in scale and complexity to meet business needs. Techniques that work well for small-scale development do not necessarily scale to enterprise-wide initiatives.

► *Skills availability*: The sophistication of today's IT platforms means that specialists' knowledge is required to deliver software. Many organizations struggle to find sufficient skilled professionals to support their development. In addition, projects often depend on key individuals and suffer significantly if those individuals leave a project or organization.

► *Changing middleware environment*: Today's applications are deployed to a huge variety of middleware platforms, and the rate of change in platform technology is showing no sign of slowing up. Businesses want to take advantage of advances in middleware but do not want to repeatedly rewrite their applications.

1.2 A model-driven approach to software development

Model-driven development is a style of software development where the primary software artifacts are models from which code and other artifacts are generated.

A *model* is a description of a system from a particular perspective, omitting irrelevant detail so that the characteristics of interest are seen more clearly. For example, a structural engineer creates a model of a building that is suitable for determining its load-bearing characteristics.

In MDD, we introduce the additional criteria that a model must be machine-readable. For example, we must be able to access the content of the model in an automated manner. Machine-readability of models is a prerequisite for being able to generate artifacts. A diagram on a whiteboard may meet the other criteria for being a model. However, until we capture it in a machine-readable manner, we cannot use it within a model-driven development tool chain.

Software models are typically expressed in the Unified Modeling Language (UML). UML is a language for specifying, visualizing, and documenting software systems. UML provides a visual notation and underlying semantics for software models. UML also has a standard machine-readable serialization format, thereby enabling automation.

Note: XML Metadata Interchange (XMI) is used to serialize UML models and other models that are defined using the Meta Object Facility (MOF).

Software models hide technical implementation details so that a system can be designed using concepts from the application domain. Application design is typically carried out using a UML modeling tool such as Rational Software Architect, using concepts relevant to the application domain.

Note: Modeling using a domain-specific language tailored to a particular subject area is also a possibility. This approach is further discussed in 6.2.1, "IBM and MDA" on page 76.

For example, when working in the enterprise integration domain we would start by modeling the application design using concepts such as message, proxy, and adapter. Later we can refine the software model and design the details of its components.

1.2.1 Models as sketches and blueprints

The use of models to design software is a well-established practice (though certainly not ubiquitous). Currently, models are used mostly as *sketches* that informally convey some aspect of a system or *blueprints* that describe a detailed design that you manually implement.

Using models as documentation and specification is valuable, but it requires strict discipline to ensure that models are kept up to date as implementation progresses. More often than not, time pressures mean that the implementation is updated without first changing the models. Inaccurate models can be more harmful than no models at all.

In this book, we use the term *model-driven development* to describe approaches where automation generates artifacts from models.

1.2.2 Precise models enable automation

In MDD, models are used not just as sketches or blueprints but as primary artifacts from which efficient implementations are generated by the application of transformations. In MDD, application domain-oriented models are the primary focus when developing new software components. Code and other target domain artifacts are generated using transformations designed with input from both modeling experts and target domain experts.

MDD has the potential to greatly reduce the cost of solution development and improve the consistency and quality of solutions. It does this by automating implementation patterns with transforms, which eliminates repetitive low-level development work. Rather than repeatedly applying technical expertise manually when building solution artifacts, the expertise is encoded directly in transformations. This has the advantages of both consistency and maintainability. A modified transformation is reapplied rapidly to generate solution artifacts that reflect a change to the implementation architecture.

MDD shifts the emphasis of application development away from the platform allowing developers with application expertise to design applications without being concerned with the platform-level concepts that are province of developers with platform expertise.

Platform expertise is captured directly in transformations rather than being documented as project guidelines or worse, being rediscovered many times during a project. Likewise, decisions about the implementation architecture are directly encoded in the transformations rather than documented as architectural decisions.

Depending on the situation, suitable off-the-shelf transformations are available for use directly or as a basis for extension. Alternatively, you may need to build custom transformations for a project.

1.2.3 The role of patterns in model-driven development

A pattern is a solution to a recurring problem within a given context. Patterns encapsulate a designer's time, skill, and knowledge to solve a software problem. And when used repeatedly in a number of different projects, a pattern becomes established as best practices.

Software patterns can apply within an abstraction layer (for example, design patterns and implementation patterns) and across abstraction layers (for example, patterns for relating design elements and code). You can compose patterns to produce pattern recipes for solving larger problems and pattern languages to cover best practices for a domain area.

The complementary nature of patterns and MDD is key to this book. MDD unlocks the potential of patterns to create well designed solutions, and patterns provide the content for MDD.

1.2.4 Not just code

The generation of code and other platform artifacts is an important part of MDD, but MDD-style automation can go much further than this. A software development project needs to produce many non-code artifacts and many of these are completely or partially derivable from models. The following list gives some common examples of artifacts that are generated from models, but you can probably think of others.

► *Documentation*: In organizations that follow a formal development approach, the production of documentation takes a significant amount of development effort. Keeping documentation in line with the implementation is notoriously difficult. When using MDD, documents are generated from models ensuring consistency and making information available within the models that developers are working with on a daily basis, rather than in documents that are difficult to navigate. Tools such as IBM Rational SoDA® and IBM Rational Software Architect Report Generator generate documentation, or documentation is generated by a transformation.

> **Note:** SoDA stands for IBM Rational Software Documentation Automation. To learn about SoDA, see:
>
> http://www.ibm.com/software/awdtools/soda/

► *Test artifacts:* It is possible to generate basic test harnesses (for example, using JUnit) from the information contained in software models.

> **Note:** Additionally, in RSA, Java code that has been generated from models can be used to drive the generation of Component Test code.

If additional test-specific modeling is carried out (for example, using the UML Profile for Testing) then complete test implementations are generated. Model-based testing is a discipline that is concerned with generating tests from models.

► *Build and deployment scripts:* Using their expertise, build and deployment architects can create transformations to generate build and deployment scripts.

► *Other models:* A system involves many interdependent models at different levels of abstraction (analysis, design, implementation), representing different parts of the system (UI, database, business logic, system administration), different concerns (security, performance, and resilience), or different tasks (testing, deployment modeling). In many cases, it is possible to partially generate one model from another, for example moving from an analysis model to a design model, or from an application model to a test model.

► *Pattern application:* Patterns capture best practice solutions to recurring problems. Patterns specify characteristics of model elements and relationships between those elements. You can automate patterns so that new elements are created and existing elements are modified to conform to the pattern when the pattern is applied to a model.

When we talk about MDD in this book, we include all of these techniques in addition to the generation of code.

1.3 Benefits of model-driven development

Model-driven development has the potential to greatly improve on current mainstream software development practices. The advantages of an MDD approach are as follows:

▶ *Increased productivity:* MDD reduces the cost of software development by generating code and artifacts from models, which increases developer productivity. Note that you must factor in the cost of developing (or buying) transformations, but careful planning will ensure that there is an overall cost reduction.

▶ *Maintainability*: Technological progress leads to solution components becoming stranded legacies of previous platform technologies. MDD helps to solve this problem by leading to a maintainable architecture where changes are made rapidly and consistently, enabling more efficient migration of components onto new technologies.

High-level models are kept free of irrelevant implementation detail. Keeping the models free of implementation detail makes it easier to handle changes in the underlying platform technology and its technical architecture.

A change in the technical architecture of the implementation is made by updating a transformation. The transformation is reapplied to the original models to produce implementation artifacts following the new approach.

This flexibility also means that it is possible to try out different ideas before making a final decision. It also means that bad decisions are easily changed. Software projects are often stuck with decisions that are a mistake in retrospect but are too costly to fix.

▶ *Reuse of legacy:* You can consistently model existing legacy platforms in UML. If there are many components implemented on the same legacy platform, you can develop reverse transformations from the components to UML. Then you have the option of migrating the components to a new platform or generating wrappers to enable the legacy component to be accessed via integration technologies such as Web services.

▶ *Adaptability*: Adaptability is a key requirement for businesses, and IT systems need to be able to support it. When using an MDD approach, adding or modifying a business function is quite straight forward since the investment in automation was already made. When adding new business function, you only develop the behavior specific to that capability. The remaining information needed to generate implementation artifacts was captured in transformations.

▶ *Consistency*: Manually applying coding practices and architectural decisions is an error prone activity. MDD ensures that artifacts are generated consistently.

- *Repeatability*: MDD is especially powerful when applied at a program or organization level. This is because the return on investment from developing the transformations increases each time they are reused. The use of tried and tested transformations also increases the predictability of developing new functions and reduces the risk since the architectural and technical issues were already resolved.

- *Improved stakeholder communication*: Models omit implementation detail that is not relevant to understanding the logical behavior of a system. Models are therefore much closer to the problem domain, reducing the semantic gap between the concepts that are understood by stakeholders and the language in which the solution is expressed. Improved stakeholder communication facilitates the delivery of solutions that are better aligned to business objectives.

- *Improved design communication*: Models facilitate understanding and reasoning about systems at the design level. This leads to improved discussion making and communication about a system. The fact that models are part of the system definition, rather than documentation, means that the models are never out of date and are reliable.

- *Expertise capture:* Projects or organizations often depend on key experts who repeatedly make best practice decisions. With their expertise captured in patterns and transformations, they do not need to be present for other members of a project to apply their expertise. An additional benefit, provided sufficient documentation accompanies the transformations, is that the knowledge of an organization is maintained in the patterns and transformations even when experts leave the organization.

- *Models as long-term assets*: In MDD, models are important assets that capture what the IT systems of an organization do. High-level models are resilient to changes at the state-of-the-art platform level. They change only when business requirements change.

- *Ability to delay technology decisions:* When using an MDD approach, early application development is focused on modeling activities. This means that it is possible to delay the choice of a specific technology platform or product version until a later point when further information is available. In domains with extremely long development cycles, such as air traffic control systems, this is crucial. The target platforms may not even exist when development begins.

Like any tool or technique, MDD can be used well or used badly. MDD has the potential to produce the benefits we just mentioned, but the approach must be applied effectively. This book is based on the collective experience of its authors in industrial applications of MDD. By following the practices introduced in this book, you will significantly improve the chances of success for your MDD project.

1.4 Model-driven development with IBM Rational Software Architect

RSA is an integrated design and development tool that leverages MDD with UML for creating well-architected applications and services. The description of RSA is taken directly from the product information, which you can find on the Web at:

http://www.ibm.com/software/awdtools/architect/swarchitect

RSA has the following features that are particularly relevant to MDD:

► UML 2.0 editor with refactoring support
► Support for UML 2.0 profiles
► Patterns infrastructure with pattern library
► Transformation infrastructure with sample transformations

Patterns, profiles, and transformations together provide the capabilities required to customize RSA to support the automation of a development process though an MDD approach.

RSA also includes development tooling for J2EE, Web, and Web services. Another product, IBM Rational Software Modeler (RSM™), includes MDD capabilities, but does not provide development tooling, nor does it include any prebuilt transformations. RSM is used in an MDD tool chain by architects or designers who are responsible for modeling only, or in scenarios where the chosen runtime platform is not supported by RSA.

For MDD scenarios where the target platform includes J2EE and Web services, RSA is the appropriate tool for architects, designers, and developers.

Throughout the book we refer to RSA, but note that Rational Software Modeler is an alternative for tasks that do not require development tooling.

The following sections provide an introduction to the MDD capabilities in RSA. We cover these features in greater detail in the remaining chapters of this book.

1.4.1 Unified Modeling Language 2.0 editor

RSA includes an editor that supports the major UML 2.0 diagrams.
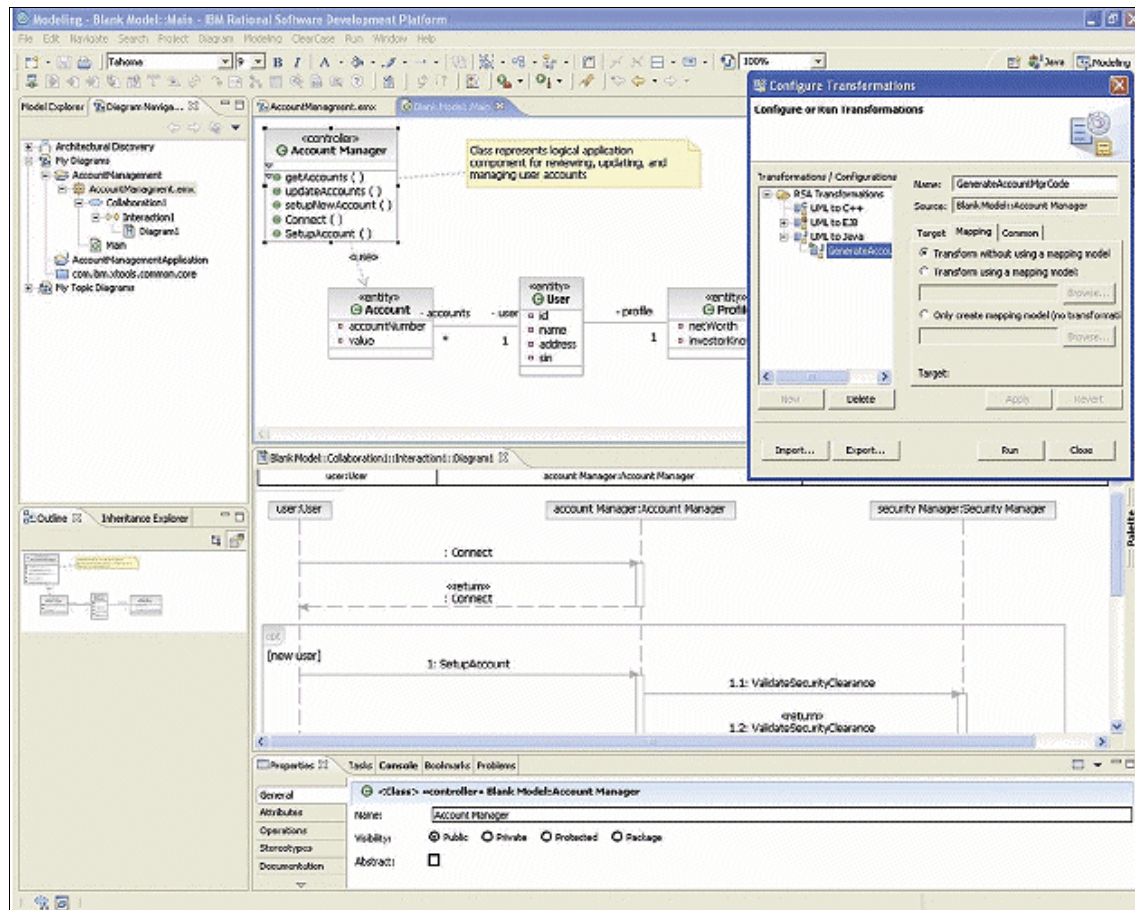
Figure 1-1   RSA UML 2.0 editor

1.4.2 UML profile support

UML profiles allow you to customize the language for a particular domain or method. UML profiles introduce a set of stereotypes that extend existing elements of UML for use in a particular context. This technique is used in MDD to allow designers to model using application domain concepts.

RSA ships with a set of UML profiles and also supports the creation of new profiles. One of the sample RSA profiles is the Rational Unified Process® (RUP®) Analysis profile that provides stereotypes for producing analysis models using the RUP approach. The RUP Analysis profile introduces stereotypes such as Boundary, Control, and Entity as shown in Figure 1-2.
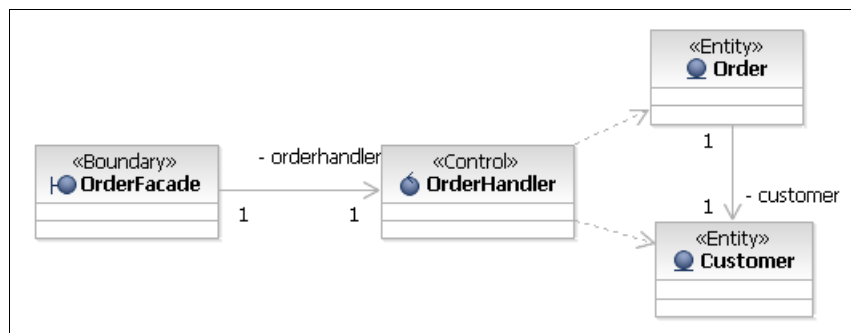
Figure 1-2   Example application of the RUP Analysis profile

1.4.3 RSA patterns

Patterns are solutions that solve a recurring problem in a particular context. RSA provides support for automating the expansion of patterns in UML models. RSA ships with a set of patterns including the Gang-of-Four patterns. RSA also provides a patterns infrastructure so that developers can build their own patterns.

One of the sample patterns in RSA is an Interface pattern that captures the relationship between an interface and a class that realizes the interface. Figure 1-3 shows the application of the Interface pattern to an ICurrentAccount interface and an Account interface that realizes it.

The pattern automates the creation of the realization relationship between the class and the interface, and adds operations to the class corresponding to those in the interface. The pattern can be reapplied if additional operations are introduced to the interface.

Patterns are commonly used in conjunction with a UML profile. The application of a pattern often introduces stereotypes to customize the model elements involved in the pattern.
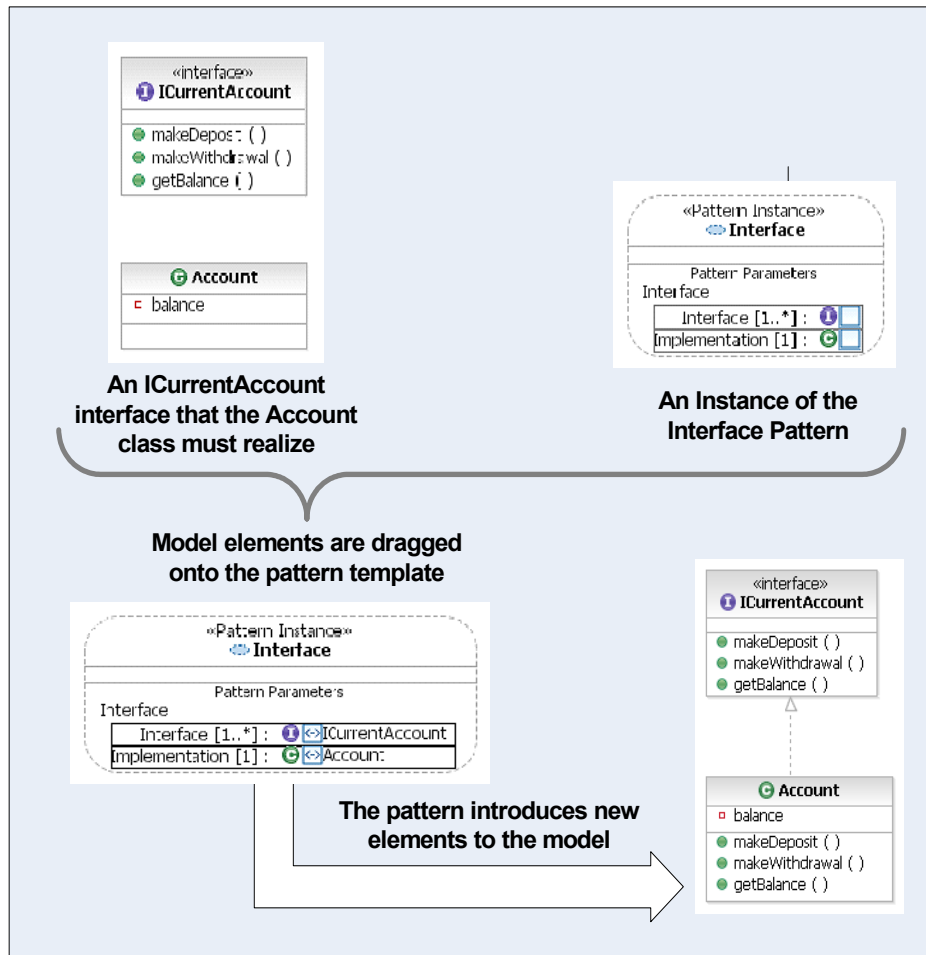
Figure 1-3   Application of the sample interface pattern

1.4.4 RSA transformations

Transformations support a layered, model-driven approach to development. They automate the transition between models at different levels of abstraction (for example, from analysis to design) and ultimately to code. RSA ships with a sample set of transformations, including a UML to EJB™ transformation. RSA also provides a transformation infrastructure so that developers can build their own transformations and extend the provided transformations.

> **Note:** For more details about extending the transformations that come with RSA, refer to the following articles:
>
> ► "UML to C++ Transformation Extensibility in Rational Software Architect"
>
>   http://www.ibm.com/developerworks/rational/library/05/412_uml_to/
>
> ► "Extending the UML to Java transformation with Rational Software Architect:
>
>   http://www.ibm.com/developerworks/rational/library/05/802_uml/index.html

Figure 1-4 demonstrates how the UML to EJB transformation works. A UML profile is used when developing the application model. This includes the Entity stereotype that is applied to the Account class in our example. Running the transformation (by right-clicking the model and selecting the UML to EJB transformation) generates a corresponding EJB project with a corresponding Entity Bean, deployment descriptors, and so on. The right side of Figure 1-4 shows a visualization of the resulting Account Entity Bean (RSA supports UML visualization of EJB artifacts; the transformation creates the actual Java code).
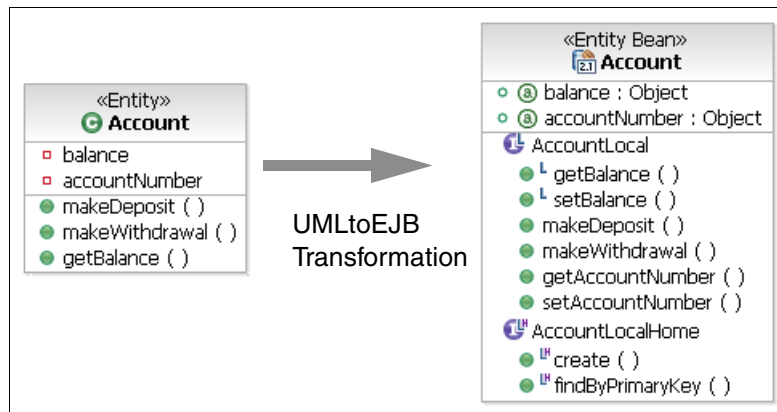
Figure 1-4   UML to EJB transformation

The UML to EJB transformation runs over a whole model and processes all Entities found. There are additional stereotypes for modeling sessions and other enterprise component concepts from which EJB artifacts can be generated.

1.5 Summary

This chapter introduced you to the concepts behind model-driven development, the relevance and benefits of MDD to business, and a simple example of applying a pattern and transformation.

In Chapter 2, "Scenario overview" on page 17, we introduce the scenario upon which the practical examples in Part 2, "Implementation" on page 91, are based. As we mentioned in this chapter, MDD is not appropriate to all scenarios. The next chapter goes beyond describing the scenario we are going to use. It discusses how to decide whether to use an MDD approach, how to apply an MDD approach, and why the selected scenario is appropriate for MDD.

2

Scenario overview

In this chapter, we describe the example scenario that we use to illustrate discussions throughout the book. We also identify the features that make it appropriate for a model-driven development (MDD) approach.

We chose this scenario to illustrate the application of MDD to an enterprise service bus (ESB) within an enterprise wide service-oriented architecture (SOA). This scenario is based on an amalgam of actual experiences but does not reflect any one specific engagement.

The project reflected most closely by the scenario began, not as a model-driven development, but simply as an ESB project which concentrated on integration patterns and reuse. It was not until after much of the formalization of the architecture and patterns occurred that the advantages of model-driven development were appreciated and Unified Modeling Language (UML) models were developed. For this reason, early architectural drawings were not UML models. This was reflected in the definition of the scenario in this chapter and in Chapter 7, "Designing patterns for the scenario" on page 93, in the description of the information on which the design model was based.

In a project that is model driven from the start, you use UML throughout the design process.

17

2.1 Enterprise architecture

The enterprise represented in this scenario provides products to many customers, who have ongoing service contracts of considerable complexity. The scenario looks at customer handling functions and how these are supported by back-end systems. The enterprise is going through a technology refresh in which a number of divisions, responsible for different product ranges, are introducing new front office systems. These require information from corporate and divisional back-end systems. The back-end systems are also to be replaced. However, the time scales of the two refresh programs cannot be fully aligned, so a "big bang" approach is considered too risky.

The business scenario on which the examples in this book are based assumes that an enterprise architecture has been established and that all future divisional development projects will be aligned with this architecture.

An SOA has been adopted as the enterprise architecture. The enterprise architecture has mandated several off-the-shelf packages to assist call handlers in their customer facing roles, to manage account information and services over their entire history and eventually to move towards self help applications. Currently the accounts and customer information are managed mainly by legacy systems. Some of these will be replaced over a period of time to meet requirements for new functionality.

To support the front-office packages with customer and account information from the back-end systems, and to manage updates to the back-end systems, the enterprise architecture mandates a service-oriented, enterprise-wide integration bus, or rather the *enterprise service bus*.

All divisional systems have to use the ESB for any integration between the customer facing systems and the back-end applications. It is the intention that all divisional systems will receive and maintain a consistent view from these systems.

A number of business processes will be developed under the control of divisional projects. The enterprise architecture defines business process management as being a layer outside the ESB, although processes may be initiated through the ESB and the business process may invoke services through the ESB. In the enterprise architecture, any long running process must be handled the Business Process Management layer rather than directly by the ESB. The enterprise architecture requires an enterprise System Management Framework, which monitors all enterprise applications including the ESB.

Initial product selections for the ESB include:

► WebSphere® Business Integration Server Foundation
► WebSphere Business Integration Message Broker
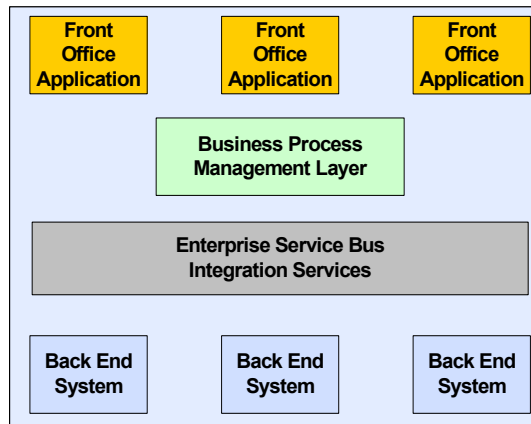
Figure 2-1   Enterprise architecture vision

2.1.1 Suitability for model-driven development

Even at this early, high-level definition of the architecture, there a number of pointers indicating that this program of development is a candidate for model-driven development.

There are development projects that require development of integration services to run on the ESB across all divisions. Any investment required to develop the necessary framework and tooling for a model-driven development approach to generating the integration services likely to be repaid manyfold.

Although the divisions of the enterprise have considerable autonomy with respect to procuring or developing their IT support, the definition of a well defined enterprise architecture, and the establishment of governance to ensure that divisions follow the enterprise architecture, are likely to increase repeatable patterns of behavior. This will increase the potential productivity and success of a model-driven development approach.

2.1.2 Contra-indications for model-driven development

Although there are factors that are expected to contribute to the potential success of a model-driven development approach, this enterprise does not currently have any significant investment in UML or related modeling skills. There might also be some misgivings from divisional IT managers about how to ensure that contracted IT suppliers are following a similar approach.

Further considerations as to suitability of the model-driven development approach are given after the following description of the integration architecture. Then the case for return on investment when the framework is used across the enterprise will be become more apparent.

2.2 Integration architecture

The enterprise architecture states that a service-oriented approach should be taken to all new developments. This includes the ESB, which adopted *service-oriented integration* (SOI) as the underlying principle.

Service-oriented integration supports a SOA in which applications do not call the services of other applications directly but call the services hosted by integration middleware. These integration services provide mediations such as validation and transformation. They may also provide integration orchestration before calling one or more services offered by other applications.

The integration services do not, in this case, support long-running, stateful business services because these are explicitly assigned to a separate business process management layer in the enterprise architecture.

A high-level architecture for the ESB defines the service-oriented principles to which services deployed onto the ESB must conform. In this scenario, it is particularly important that service requesters and providers are loosely coupled. It is also important that, although requesters must be aware of a service contract, they are unaware of service implementation. This allows provider implementations to be changed independently of any of the front-end requesters connecting to the ESB. Migration of the legacy to new OTS packages is a lengthy process and should happen without disruption to service requesters.

A further principle of the ESB architecture, and of the enterprise architecture itself, is that integration services deployed onto ESB should be reusable, and reused, by multiple applications both within and across divisions. The principle of loose coupling assists this goal. It is further enforced by the use of a searchable service portfolio to identify existing services and the mandatory use of a canonical data format for integration services. There already exists an embryonic data model and associated canonical data format. This is being developed under central governance to avoid the development of point to point services.

Neither off-the-shelf applications nor legacy providers conform to the Enterprise Canonical Data Format (ECDF). ESB integration services are therefore provided with facades for incoming requests and facades through which they call the services of providers outside the ESB. The facades apply transformations to and from the ECDF.

2.2.1 ESB structure

The ESB is structured into two layers. Services in the inner layer operate only with ECDF and provide all the orchestration of the integration. Services in the outer layer are facades that are responsible for transformation between external formats and ECDF and other boundary activities such as validation, audit, authorization and management of communications with external requesters and providers. The ESB also provides utility services, which support common functions, and are called from both facade services or integration services.

Figure 2-2 shows a simple end-to-end service (either synchronous or one-way) with a single integration service (IS) in the inner layer. This service is called from a front-office application via an integration service facade (ISF) and calls a back-end service provider via a provider facade. Both the facade service and the integration service can call utility services which reside in the inner layer.
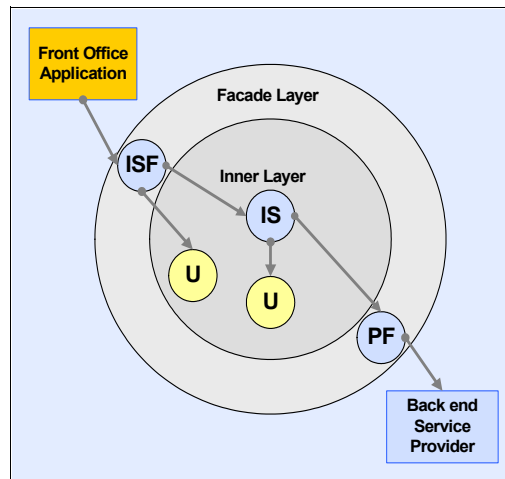
Figure 2-2   ESB structure

The utility services (U) are also part of the ESB architecture. They differ from the list of services indicated earlier in that they are implemented only once and are part of the ESB infrastructure itself.

Figure 2-3 shows an additional end-to-end service on the ESB. In this case, it is an asynchronous service (ISF, IS, and PF shown in smaller, light blue circles) and introduces three additional service types, the callback services. These services allow an asynchronous response to be returned to a front-office application which has made a request. The *provider facade call back* (PFCB) offers a service interface which a back-end service provider can call to return a response. This in turn calls the *integration service call back* (ISCB) and the *requester callback facade* (RCBF) ending up with a call to the callback service of

the original requester. These are effectively two one-way services which carry a transaction ID that allows a response to be matched to request at callback service of the requesting front office application.

In summary, six types of service make up the end-to-end integrations:

- ▶ Integration service facade
- ▶ Integration service
- ▶ Provider facade
- ▶ Provider facade call back
- ▶ Integration service call back
- ▶ Requester callback facade
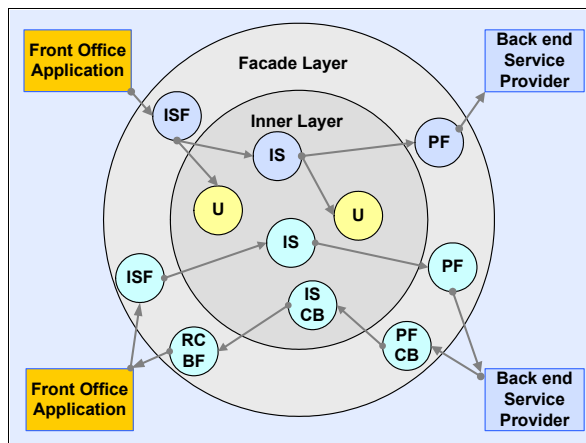
Figure 2-3   Additional Service Types

The two simple examples show basic use of service types. However these are the building blocks which can be composed to build more complex end-to-end services.

The ESB architecture requires any integration service facades that are deployed onto the ESB to present to external requesters a Web Services Description Language (WSDL) defined interface. This interface specifies a SOAP message over HTTP, JMS, or MQ.

ESB services in the inner layer are also required to have well defined WSDL interfaces. In this case, nonstandard lightweight binding may be used for efficiency. ESB services are implemented either in Web Sphere Application Server or in WebSphere Business Integration Message Broker.

In any real ESB architecture, consideration would be given to the non-functional aspects such as security, availability and failover. These aspects are be part of

the design of the overall infrastructure and are not included in our current scenario which this focuses primarily on the model-driven development of the services to be deployed on the ESB. This is not to say that such aspects are not relevant to model-driven development. They are considered again briefly in Chapter 5, "Model-driven development solution life cycle" on page 59.

2.3 Pattern definition

Our first pattern is the high-level ESB architecture, which wraps the integration service in facades. It defines the service types which can be deployed. It also imposes the first level of constraint on how these services can call each other and can interact with requesters and providers outside the ESB.
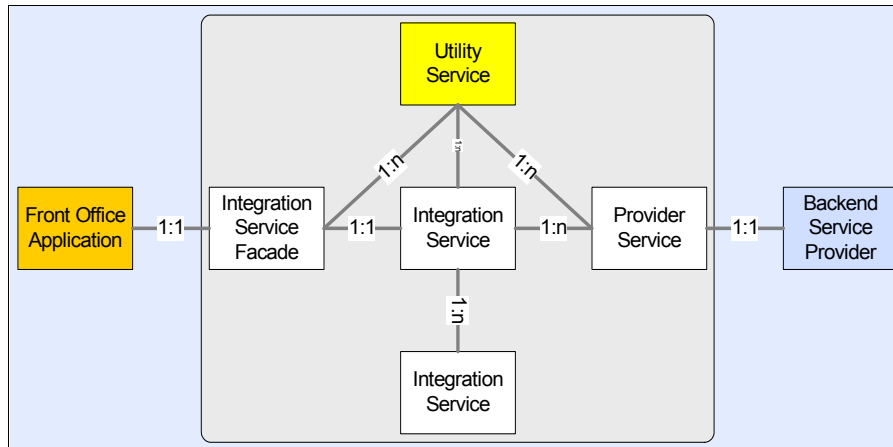
Figure 2-4   Service connection pattern in ESB

► An integration service facade provides an end-to-end service to an external requester. It provides a facade for a single integration service.

► An integration service may be called through one or more integration service facades. Each facade may present the service with a different request format. The facades must work as closely as possible in the format of the requesting application.

► Integration services may call one or more external service providers through a provider facade.

► Integration services may call other integration services.

Given this architecture, we now look in greater detail at the patterns for the integration services to be supported on the ESB.

2.3.1 Interaction behavior patterns

The first step is to look at some fundamental *contracts of behavior* between service requesters and service providers. In the context of this redbook, a contract of behavior is defined as a pattern of behavior that covers a number of aspects of the interactions.

A contract of behavior defines an agreed set of behaviors which can be applied to an interaction. The behaviors define the requirements that are placed on applications making service requests and on the middleware or application providers of services. These are technical, rather than business requirements, but are technology and protocol independent, for example:

► Synchronous, asynchronous, or one way
► Update or read only
► Responsibility for error handling
► Responsibility for error reporting
► Correlation between requests and responses
► Audit

Each contract of behavior carries with it a set of requirements on both the requester and the provider in any interaction. Any service deployed onto the ESB supports all the requirements of one of a specific contract of behavior when it acts as a provider and for each time it make a request to a provider.

The contracts of behavior used in our scenario are defined in Chapter 5, "Model-driven development solution life cycle" on page 59.

2.3.2 Individual service patterns

The next level of abstraction is to look at the patterns which apply to the individual service types. These are obtained by combining the ESB architectural principles applied to specific service types with a contract of behavior.

Facade patterns

The facades and their callback services include a sequence of standard boundary functions which process a request, or response, to ensure that it conforms to the architecture of the ESB and to execute the necessary transformations. The facade patterns also handle the service interactions and error handling in conformance with the selected contract of behavior.

Integration services

The internal integration services are more complex as they provide the orchestration of integrations. An example of the type of orchestration, which is supported by the integration service patterns, is a request for update to a single

target, optionally routed to alternate service providers based on message content. The integration service patterns is covered in greater detail in Chapter 5, "Model-driven development solution life cycle" on page 59.

It is less easy to provide a comprehensive set of patterns for the behavior of integration services. However considering 80/20 coverage, it is possible to define a set of patterns which can be added to as new requirements are refined. The patterns for integration services are closely aligned with the basic Patterns for e-business.

2.3.3 Suitability for model-driven development

As we add detail to the enterprise architecture to define the integration architecture, and the patterns that can be abstracted from that architecture, it becomes clear that the development of services on the ESB is a clear candidate for model-driven development.

We have shown in outline how the integration architecture can be defined as a hierarchical set of patterns. Given this set of patterns, it is possible to configure the patterns for specific instances. This means that two of the benefits defined in 1.3 can be realized by applying model-driven development to the ESB architecture, namely the expertise captured in the patterns and transformations and their repeated use.

The degree of abstraction achieved by the patterns means that a high degree of reuse can be obtained with the opportunities for cost savings being greatly increased.

2.3.4 Contra-indications for model-driven development

There are no obvious technical factors which would preclude the use of a model-driven development approach. The architectural framework if adhered to universally appears to be an eminently suitable.

There may be many organizational factors which could hamper adoption of the approach. Although there is an understanding of model-driven development among the enterprise architects who have responsibility for the enterprise architecture and the ESB architecture within it, there is little appreciation of this approach among the divisional IT leaders.

Divisional IT has been used with considerable autonomy in determining their IT strategy, The use of an enterprise-wide integration program (the ESB) is in itself a potential point of dispute. The additional requirement for the use of standard patterns and a new development method to support them will require organizational issues to be resolved.

2.4 Automation

We have described the ESB architecture in terms of the patterns which constrain the service implementations. These patterns can be used as input to a conventional detailed design phase for every service to be implemented on the ESB. As services are created under the control of different divisional IT management, with contracts let to multiple IT suppliers, this can result in the use of various development environments. Also each division and each supplier require the skills to implement service-oriented integration and to fully understand the ESB architectural constraints and ESB technologies in which it is implemented.

However, at enterprise level, the use of patterns also opens the door to automated generation of services and their associated artifacts.

The modeling of ESB service patterns, and the generation of service instances from these models, is included as a feature of the enterprise architecture. Rational Software Architect is included in the required product list of the enterprise architecture to model the ESB structure and ESB services and to create transformations of these models. These transformations generate both the code which implements the services and the associated artifacts. It ensures that a complete package is delivered which enables rapid deployment and testing of the services.

The organizational difficulties are recognized and addressed at three levels: technical, organizational and managerial.

2.4.1 Technical

The ESB implementation is an enterprise rather than divisional project and is managed as an enterprise level asset. A framework for model-driven development is in place. The initial set of patterns are included as models with transformations to generate complete, deployable services, and tooling to guide and assist service developers has been built. Also processes link the ESB to the enterprise system management.

The technical approach to compensating for lack of model-driven development skills in the divisions is to establish the basic framework. The approach also ensures that this framework eliminates much of the need for modeling skills outside the centrally run ESB project.

New patterns are created by the central ESB group against divisional requirements. Use of patterns is supported by more than one interface. For some users, a tailored user interface has been established to ensure that creation of

ESB services by divisional projects requires general integration skills, rather than a deep understanding of the modeling techniques.

2.4.2 Organizational

The necessary governance is in place. An enterprise-level organization owns processes, systems and controls, which allow divisional integration projects to generate their own services and to pass these to a central organization for certification and deployment. The central processes also ensure the distribution of information about existing and planned assets to encourage reuse of services and to prevent deployment of new services which duplicate existing functionality. The MDD approach assists this central organization. By imposing the architecture through patterns, the governance problems are reduced, and there are considerable benefits in quality and consistency.

2.4.3 Managerial

There is a program to disseminate information about the ESB and the model-driven development to all divisions. This is backed up by financial incentives to ensure that the cost benefits on which the ESB is predicated are achieved.

2.5 Summary

In this chapter, we described the scenario that we use throughout the book to demonstrate the use of MDD. Patterns were identified in the scenario, which indicated that use of an MDD approach would generate a positive economic return for the project. We also identified some non-functional requirements for the successful implementation of an MDD approach.

3

Model-driven development approach

This chapter describes the ideas behind model-driven development (MDD). The topics covered in this chapter include:

► The key concepts behind MDD
► The role of UML in MDD
► The role of patterns in MDD
► The MDD process

3.1 Abstraction

Abstraction is the process of ignoring irrelevant details in order to focus on relevant ones. In MDD, we use abstraction to enable us to work with a logical view of our application, focusing on the functionality of the system without being concerned with implementation details.

Abstraction can be used to model applications at different levels of abstraction (including analysis, design, and implementation) or from different perspectives (including security, management, user interface). Abstraction allows us to focus on the different aspects of a system without getting lost in detail that, while crucial to the system as a whole, is irrelevant to the current viewpoint.

When writing code we describe our applications using implementation concepts. Even when using full-featured middleware platforms we need to write a lot of code (and deployment descriptors, configuration files, and so on) to express application concepts. Much of the code that we write is similar, particularly when following the conventions adopted by a particular project or organization. In many cases, a large amount of code follows directly from a small number of design decisions and from the architectural principles of the project. MDD enables us to work at a level where we can directly capture those design decisions in models, and generate the appropriate code through transformations.

Modeling concepts are much richer than implementation artifacts. We can say more with less effort. It is quicker to create the models than to write the code manually. Additionally, we can focus on the logical design of our application, which frees us from distracting implementation details.

3.2 Precise modeling

Abstraction is not the same as imprecision. When we use abstraction we omit specific details while being precise about those details on which we do focus. When using an MDD approach we can be very precise about the architecture and high-level design of a system while saying nothing about implementation details.

The Unified Modeling Language (UML) is typically used to create models for MDD. UML is a software modeling language with a graphical notation and underlying semantics. It is often assumed that diagrams must be informal. While we can use UML in an informal way, it has semantics and precise models that we can create by using it in a consistent manner.

In MDD, we use UML models as part of the definition of a system, not just as sketches. These models have well-defined semantics and can be transformed

into implementation artifacts (in the same way that we compile Java code into byte code).

3.3 Automation

Modeling is a valuable technique in itself, but manually synchronizing models and code is error prone and time consuming. Automation is the main characteristic that distinguishes MDD from other approaches that use modeling.

> **Note:** The term *model-driven development* is sometimes used to describe non-automated but model-centric approaches. In this book, we reserve the term "model driven" for those approaches in which automation is practiced.

MDD is concerned with automating the development process so that any artifact, which can be derived from information in a model, is generated. This includes code as well as deployment descriptors, test cases, build scripts, other models, documentation, and any other kind of artifact that a software project needs to produce.

You can achieve automation by using two main techniques:

► *Transformations:* Transformations automate the generation of artifacts from models. This includes the generation of code and also the generation of more detailed models, for example generating a design model from an analysis model. Transformations are typically applied in batch mode to entire models as illustrated in Figure 1-4 on page 15.

► *Patterns:* Patterns automate the creation and the modification of model elements within a model to apply a given software pattern. Patterns can occur at all levels of abstraction so we can have, for example, architecture patterns, design patterns, and implementation patterns. Patterns are typically applied interactively with a designer selecting a pattern and providing parameters. See Figure 1-3 on page 14.

The transformations specify how higher-level concepts are mapped to lower-level concepts according to best practices. Lower-level patterns are often applied by a transformation rather than manually applied, for example, a transformation from a design model to code generates code according to implementation patterns.

3.4 Architectural style

MDD improves productivity and consistency by automating the generation of common aspects of software artifacts, which allows the application developer to focus on what is special for each piece of business function to be implemented. Software platforms, including programming languages and middleware, are intended to be broadly applicable and must cater to many different development styles. The more we can specialize and constrain how our software development domain makes use of its platforms, the more automation we can achieve.

MDD is not a one-size-fits-all approach. There is no single off-the-shelf tool that is suitable for automating the development of all kinds of software. There is not enough commonality to achieve the full benefit of MDD. MDD is most effective in automating the development process where there is a requirement to produce many components, services, applications, or solutions that are built according to a common set of conventions.

Richard Hubert, in his book *Convergent Architecture: Building Model-driven J2EE Systems with UML* (see "Related publications" on page 227 for the complete citation), introduces the term *architectural style* to refer to this common set of conventions: "An architectural style is a family of architectures related by common principles and attributes." He also states, "An architectural style conveys the principles and the means to most effectively achieve a design vision."

An architectural style can be influenced by the following items:

► Category of software under development (for example, enterprise integration, real-time embedded, end-user interface)

► Category of software platform that is to be used (for example, messaging middleware, rule-based system)

► Software development paradigms that are favored (for example, service-oriented, aspect-oriented)

► Subject matter of the software (for example, telecoms, finance, retail)

► Architectural principles for the system or systems under development

► Scope of the architectural style (from industry wide through to project specific)

► House style of the software development organization (for example, modeling conventions)

► Other factors influencing the concepts that are appropriate for software modeling in a particular context

We use the term MDD framework to refer to the method and tools that automate development for a particular architectural style. An MDD framework, as defined in this book, includes:

► UML profiles that tailor UML to the application domain

► Automated patterns that apply best practice solutions to repeated problems

► Transformations that generate implementation artifacts and other work products

► Guidance on how to build applications using the framework

► Sample models demonstrating how the framework is used

The following list provides the scope of an MDD framework that automates an architectural style:

► Commercial Off-The-Shelf MDD Framework

Commercial vendors can provide MDD Frameworks for use across a common domain.

► Enterprise or program-wide MDD Framework

A common MDD framework is developed by an enterprise or program that is responsible for delivering multiple projects that share common characteristics.

► Project-specific MDD Framework

An MDD framework is developed specifically for use on a particular project.

Enterprise-wide MDD frameworks are particularly important for capturing and enforcing a common architectural style across all the IT systems in an enterprise. The architectural style of an enterprise may consist of a set of interrelated architectural styles that address different kinds of software development carried out within the enterprise.

3.5 The role of UML

As we already learned, UML is typically used for defining models when using an MDD approach (see "UML and DSLs" on page 79). UML is an open standard and the de-facto standard for software modeling. UML is a general purpose software modeling language that we can apply in various ways. The modeling techniques appropriate for designing a real-time, embedded telephony application and those for developing a self-service retail application are quite different. We can use UML in both cases.

UML profiles

A UML profile is a set of extensions that customize UML for use in a particular domain or context. Examples of UML profiles include:

- ► The UML testing profile

  http://www.omg.org/cgi-bin/doc?ptc/2004-04-02

- ► The UML profile for software services

  http://www.ibm.com/developerworks/rational/library/05/419_soa/

- ► The UML profile for schedulability, performance, and time

  http://www.omg.org/technology/documents/formal/schedulability.htm

UML profiles are orthogonal extensions to UML so multiple profiles can be applied simultaneously. For example, we might apply both the UML profile for software services and a UML profile for security if we are modeling security considerations within an application that follows a service-oriented architecture.

> **Note:** For a proposed security modeling profile, see "Modeling Security Concerns in Service-Oriented Architectures" on the Web at:
>
> http://www.ibm.com/developerworks/rational/library/4994.html

A UML profile introduces a set of "stereotypes" that extend UML concepts. For example, the UML profile for software services introduces a Message stereotype that you can apply to a UML class to indicate that the class represents a message to be passed to a service. The use of a stereotype conveys additional semantic information both to human modeler and to automation tools. We are able to distinguish between UML classes that represent messages and UML classes that represent tables in a database. In this case, described using the Web Services Definition Language (WSDL), the Message stereotype enables a transformation to determine which UML classes are used to model messages and should therefore lead to the generation of message artifacts.

Figure 3-1 shows a simple example of MDD in which a Message concept in a model is transformed to a WSDL message definition in the context of a UML-to-WSDL transformation.
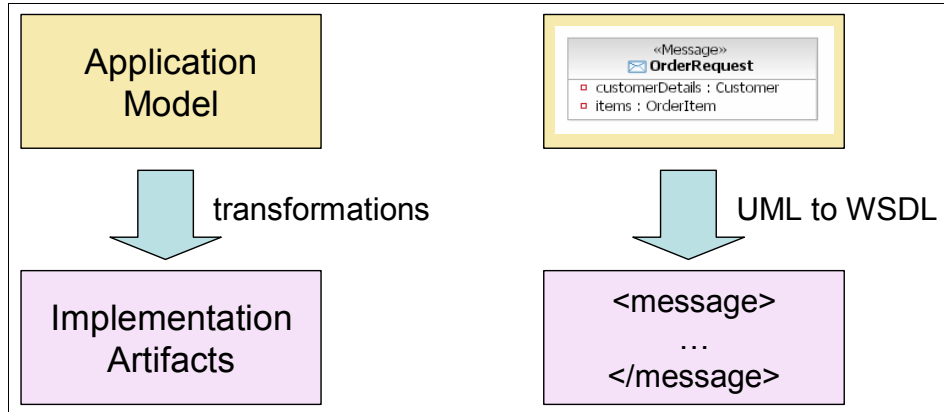
Figure 3-1   Simple MDD example

The modeler does not need to have any knowledge of the syntax of WSDL. In fact they do not even need to know that WSDL will be used in the implementation of the model. That decision might not have been made at modeling time. Later it may be necessary to support other standards in addition to WSDL. The model is independent of the details of any particular platform and it may be possible to develop transformations that generate artifacts for various platforms based on implementation requirements.

The value of a common modeling language

Using UML for MDD modeling has the following benefits:

► Using profiles of UML for MDD modeling allows us to leverage the considerable experience gone into the development of this language. This means we can provide a customized modeling environment (to the extent supported by the UML standard and UML tools) without the considerable cost of designing and implementing it from scratch.

► UML is an open standard and the de facto industry standard for software modeling. UML has proven to be durable. Its first version came out in 1995, making investment in UML skills worthwhile. The success of UML means that many excellent books and training courses are available. Many computer science and software engineering students are introduced to UML at university. Many UML tools are also available ensuring that UML expertise is a transferable skill.

► Most organizations need to develop multiple kinds of software. If we can use a common modeling approach, customized as appropriate, for describing each of these different software domains then it becomes much easier to deal with enterprise-wide development and integration.

3.6 Expertise capture

Many IT projects suffer from a lack of skilled experts. This often leads to over dependence on key individuals without whom the project would have severe difficulties.

A key aspect of MDD is expertise capture. Instead of needing experts to be on hand every time a best practice decision needs to be made, we capture their expertise in automated patterns and transformations so that we can reapply it. This approach enables developers without specialist knowledge to build sophisticated systems.

There are two main kinds of expertise to capture in order to build an MDD framework: logical architecture expertise and technical architecture expertise. Both logical and technical architecture expertise are part of the overall architectural style.

3.6.1 Logical architecture expertise

The logical architecture of a solution is concerned with the functionality that it must exhibit rather than the way in which that functionality is implemented using particular technologies. When using an MDD approach the goal is to model at the logical level and generate implementation artifacts.

In MDD, we define the logical architectural style and develop UML profiles, patterns, and modeling conventions in support of this style. This means that much of the expertise involved in defining a logical architecture is captured in the MDD framework and applied each time we develop a new application. The designer of a new application is able to focus more of the problem domain and less on the aspects of the software architecture that are common across applications.

3.6.2 Technical architecture expertise

Modern middleware platforms offer a rich array of features to support the wide variety of applications that will be built on top of them. Individual projects however often only use a subset of that functionality, and they use that subset in a specialized way.

The way in which a particular project (or program) makes use of its chosen middleware platforms defines the technical architecture. This technical architecture may be captured in best practices documents, or it may be passed from developer to developer. In practice, it is difficult to ensure that the technical architecture is applied consistently when this is a manual process.

MDD gives us the opportunity to implement the technical architecture directly. Rather than stating in a document that, for example, "entry and exit from all public methods will be recorded using log4j", we can implement a transformation that generates logging code according to this rule.

One side effect of the MDD approach is that it forces us to consider the technical architecture earlier in a project and more thoroughly. Provided this is understood at the outset and sufficient time is allocated to define the technical architecture, this is beneficial and saves a lot of time later in the project.

3.7 Patterns

We already introduced the notion of modeling using domain specific concepts, such as message, proxy, and adapter. Often the vocabulary of an application domain also includes patterns. For example, in the enterprise integration domain, we might have guaranteed delivery and publish-subscribe patterns. These are not individual elements but introduce relationships between elements and constraints on their behavior.

The (building) architect Christopher Alexander introduced the concept of a pattern language in his classic *The Timeless Way of Building* (see "Related publications" on page 227 for the complete citation). His notion of patterns as best practice approaches to common design problems is widely adopted by the software community. He introduces the concept of a pattern language that he defines as a related set of patterns that together provide a vocabulary for designing within a certain context or problem domain.

In a follow-up volume, *A Pattern Language* (see "Related publications" on page 227 for the complete citation), Alexander introduces a specific pattern language that has sub languages for designing towns (aimed at planners) and buildings (aimed at architects) and a sub-language for construction (aimed at builders).

Following are examples of Alexander's patterns:

- ► Town patterns: Ring roads, night life, and row houses
- ► Building patterns: Roof garden, indoor sunlight, and alcoves
- ► Construction patterns: Good materials, column connection, and half-inch trim

Each pattern includes information about the circumstances in which it is applicable and its relationship with other patterns. Alexander explains that when using a pattern language "we always use it as a *sequence,* going through the patterns, moving always from the larger patterns to the smaller, always from the ones that create structure, to the ones which then embellish those structures, and then to those that embellish the embellishments."

The idea of a pattern language is just as applicable to software development as it is to building architecture, and we adopt the term in this book. The nature of software development means that additional automation of the process of designing with a pattern language is possible.

Just as in building architecture, human expertise is required to select larger scale (Enterprise and Application) patterns within the context of a design. However, the application of a pattern can be automated so that when a pattern is selected the design is expanded with all the consequences associated with that pattern. A simple example of a software pattern is the Getter/Setter pattern in which attributes are always accessed through consistently named operations. We can automate this pattern so that applying the pattern to attribute `name` of class `Customer` adds operations named `getName` and `setName` (with appropriate parameters) to the `Customer` class.

In general it is, not possible to automate the physical construction of buildings. However, software systems are constructed from information artifacts that we can generate automatically according to implementation patterns. Where the builder must apply the construction patterns manually, we can describe software implementation patterns in such sufficient detail that they can be generated when given application-specific context. For example, if we have a model that uses the Getter/Setter pattern, we can generate implementation code for those methods for a specific platform (such as Java) according to implementation patterns for that platform. In some cases we may have an implementation pattern that implements an Application pattern directly. In other cases we might apply multiple lower-level implementation patterns to implement an Application pattern.

Model-driven development relies on well-defined models that we can use as input to automated transformations. A pattern language provides a structured way of designing such models.

3.8 Quality and consistency

Model-driven development is not just about producing software faster, although it can certainly do that. MDD is more so about producing better software.

Automation ensures that software components are consistently implemented, this leads to the following benefits:

► *Reduced service and maintenance costs:* The similarity across components generated using an MDD framework means less diversity needs to be coped with when dealing with service and maintenance of solutions. For example, we do not end up in the situation where different developers used different packages for logging for local optimization reasons (such as skills availability and performance) at the cost of overall consistency.

► *Improved user experience:* Consistency can have a positive impact on the user experience for the software being developed. Rather than each user trying to follow house style when designing user interaction components, the house style is applied automatically. This goes beyond common look and feel to the consistent treatment of the conceptual model presented to the user. This leads to a level of consistency for users that is very difficult to achieve manually.

► *Improved quality:* Because expertise is captured and encoded once and reused many times it is possible to invest more effort in quality. Any improvements made to patterns and transformations will benefit all software components that are generated using them.

3.9 Integration

It is a common misconception that MDD is most suited to new application development. In fact, some of the advantages of MDD are most pronounced when building integration solutions.

MDD allows us to ignore the implementation differences between existing components and applications and focus on the logical functionality offered by those applications and components. We can model integration logic or new composite application functionality independent of the technologies used to provide the components. Figure 3-2 shows the UML modeling of an integration solution built from heterogeneous components. This example uses the new composite structures modeling techniques from UML 2.0 (components-ports-connectors), which are well suited for modeling loosely-coupled integration solutions.
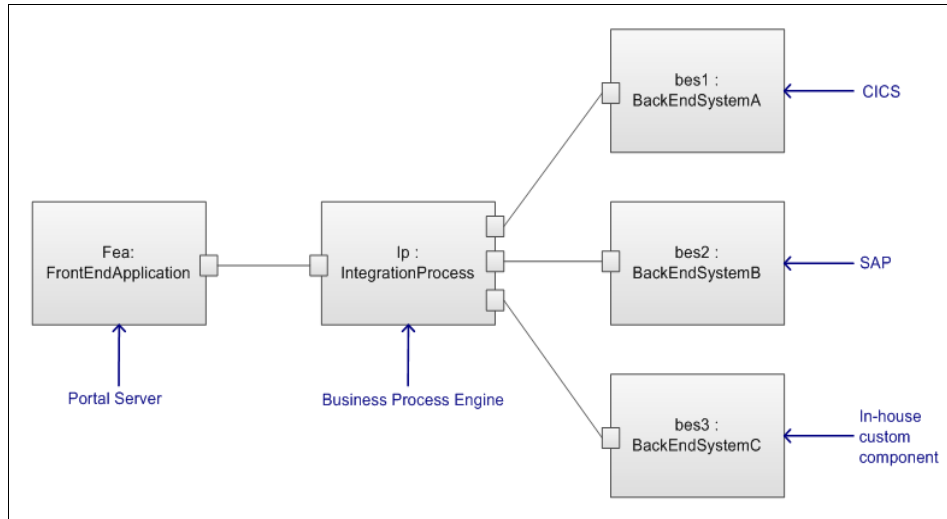
Figure 3-2   Consistent modeling of heterogeneous components

We model existing components and new components in a consistent manner. If we have a number of existing components built using the same technology (for example, CICS®), then it may be worthwhile to develop a *reverse transformation* that extracts a logical model of an existing component. Obviously this is only possible if the existing component has an appropriate interface. In other cases, we would need to build an adapter component with an appropriate interface. If we need to build several similar adapters, then this may be a candidate for MDD automation.

We can use transformations to generate platform-specific integration code, for example wrapping applications so that they can communicate using Web services.

MDD provides the basis for a consistent and well structured approach to enterprise integration. Using UML for modeling all components in a integration solution gives us a consistent view across the whole solution. We benefit from using a single-modeling language by working with logical representations of all of the components in a single tool.

3.10 Platform independence

The ability to develop platform-independent models is often quoted as a key advantage of MDD, particularly for the Object Modeling Group's (OMG's) formalization of MDD, the Model-Driven Architecture. The idea is that platform independent models capture the functionality of a system but do not depend on a specific implementation platform, so it is easier to switch between platforms in the same category (for example, enterprise application platforms such as J2EE and .NET). While this may be useful in some situations, the ability to switch platforms after a solution is deployed is not a common reason for choosing an MDD approach.

In practice, the role of platform independence in MDD can take several forms:

► *Platform-dependent MDD:* In cases where the platform is known and fixed in advance, the cost of being truly platform independent may be too high. Raising the level of abstraction does not necessarily mean becoming platform independent. We can provide abstractions over a runtime platform that take advantage of the way the platform is used in a particular architectural style. It is often more efficient to take advantage of platform knowledge when it is available.

► *Deferred choice of platform:* It is possible to set up an MDD framework so that a large amount of modeling work occurs before a choice is made on the platform. This can be useful if more time is needed to make a decision on the platform. In some cases, transformation to multiple platforms may be developed to compare non-functional characteristics before making a decision. Clearly the cost of developing the transformations must be taken into account when using this approach.

► *Multiple target platforms:* Many applications must be deployable into multiple target environments, for example on different operating systems for clients using multiple programming languages. In such cases, the common functionality is modeling and transformations are used to generate implementations for the different platforms, reducing the cost of supporting multiple platforms and ensuring consistency across them.

► *Moving between versions:* Most middleware platforms regularly release new versions with additional capabilities. To use the new features, it is often necessary to re-implement parts of an application. MDD allows us to update the transformations to take advantage of new features and regenerate the application.

Note: Platform independence does not mean that we can ignore the platform completely when modeling. We need to be able to model in a way that we can transform our models into effective implementations using our target platform.

When modeling we must understand the capabilities that are offered by the platform, or class of platforms, that we want to target. For example, a real-time platform would offer a timer service, and a rules execution platform would provide a rule triggering service. These capabilities can be assumed at modeling time and the transformations will generate the appropriate code to make use of the platform services.

Rather than being entirely platform independent, it is more accurate to say that MDD modeling is dependent on a class of platforms or a platform type.

3.11 Layered modeling

We already made a distinction between logical models and implementations of those models. These are different layers representing our solution with different amounts of detail. In some cases, it is appropriate to split our development into more layers, adding further detail at each layer as we make decisions regarding the architectural style of a solution.

Figure 3-3 shows three layered models.

► The *analysis model* focuses on capturing the use cases for the application and does not make any decisions regarding the architecture of the solution.

► In the *enterprise IT design model,* we make decisions about the IT architecture, for example, adopting a service-oriented architecture. However, we remain independent of specific technology platforms.

► In the *implementation model*, we adopt a specific runtime platform such as WebSphere.
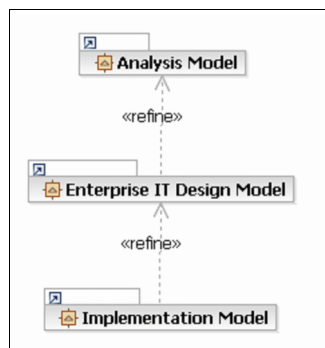

Figure 3-3   Layered models

The three layers shown in Figure 3-3 (analysis, design, and implementation) are commonly used, but this is by no means the only way of layering models. We can

also have business models, requirements models, high-level design, and detailed design models and so on. The key is that each successive layer adds further detail to the solution, answering questions that were left open in the layer above and constraining the implementation of the application.

In a layered modeling approach, we can apply patterns within any level (for example design patterns and implementation patterns) and transformations between any adjacent pair of models (for example, design to implementation).

3.12 Modeling of non-functional characteristics

When designing a solution, we must consider non-functional characteristics such as security and performance. In an MDD approach, it is often possible to capture many decisions related to non-functional characteristics in transformations. However, it is not always possible or desirable to completely automate these aspects of a solution.

Solution-specific design may be necessary. In such cases, we introduce modeling techniques relevant to specifying non-functional characteristics. For example, we might introduce stereotypes that indicate the kind of traffic that is expected over a connection (frequent/infrequent, high volume/low volume). The transformations then use this information to generate implementation artifacts that are optimized for these performance characteristics.

Modeling of security concerns is an example of a solution-specific design. You can learn about this in the article by Simon Johnston "Modeling Security Concerns in Service-Oriented Architectures", which is available on the IBM developerWorks® Web site at:

http://www.ibm.com/developerworks/rational/library/4994.html

3.13 Summary

In this chapter, we introduced the key ideas behind model-driven development and provided an outline of the activities that are performed in an MDD project. Chapter 4, "Model-driven development project planning" on page 45, builds on the understanding of the MDD development process and discusses what the MDD project planner needs to think about before embarking on an MDD project.

4

Model-driven development project planning

In this chapter, we describe how to plan and manage a model-driven development (MDD) project. We describe the new tasks involved in MDD:

► Who is responsible for the new tasks involved in MDD
► How much these new tasks cost
► How long these new tasks will take
► How to organize your development team
► How to modify the development process to get the best out of MDD

4.1 The value and cost of model-driven development

Model-driven development has a profound effect on the way we build business application software. It captures the expertise and decisions of your top technical people, making them available to the rest of the team through tooling customized for your project's needs. The cost of development and to test the business software is significantly reduced since much of the low-level coding work is automated. The number of errors are reduced and there is an increase in the consistency with which work is accomplished.

This sounds ideal, but what is the catch? Basically, as a project manager, you now control one project inside another. The inner project is developing MDD tools that the development team, building the business application in the outer project, can use.
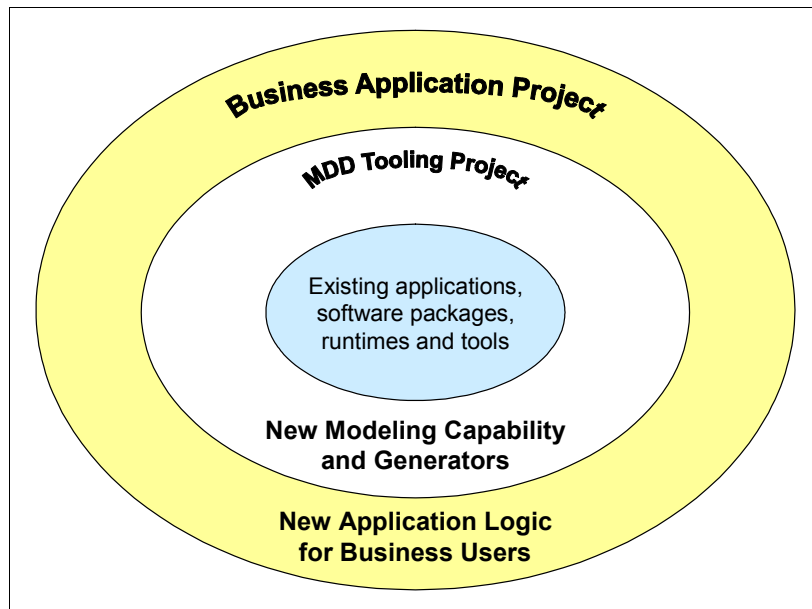
Figure 4-1   Managing the MDD tooling project inside the business application project

With two projects, you must organize and plan carefully, especially at the start of the project. This is because on top of the usual issues associated with development projects, you are now managing an additional set of internal dependencies. You must identify, develop, and work the right MDD tooling needs before the application developers need it. This means that the task flows for the two projects are inter-locked to ensure that the deliveries from the MDD tooling project are timely.

The upside is that since you control both projects, you can make the trade-off on how the effort is distributed between them. For example, you could temporarily move someone from the application project onto the MDD tooling project, so they can enhance the tooling when a new requirement is identified.

The following sections walk you through the additional steps necessary to plan and manage a MDD project.

4.2 Understanding the tasks for a model-driven development project

Figure 4-2 shows the flow of tasks in an MDD project. You would perform the shaded tasks in a traditional project. The tasks shown in white are the additional tasks that build the MDD tooling for your project.
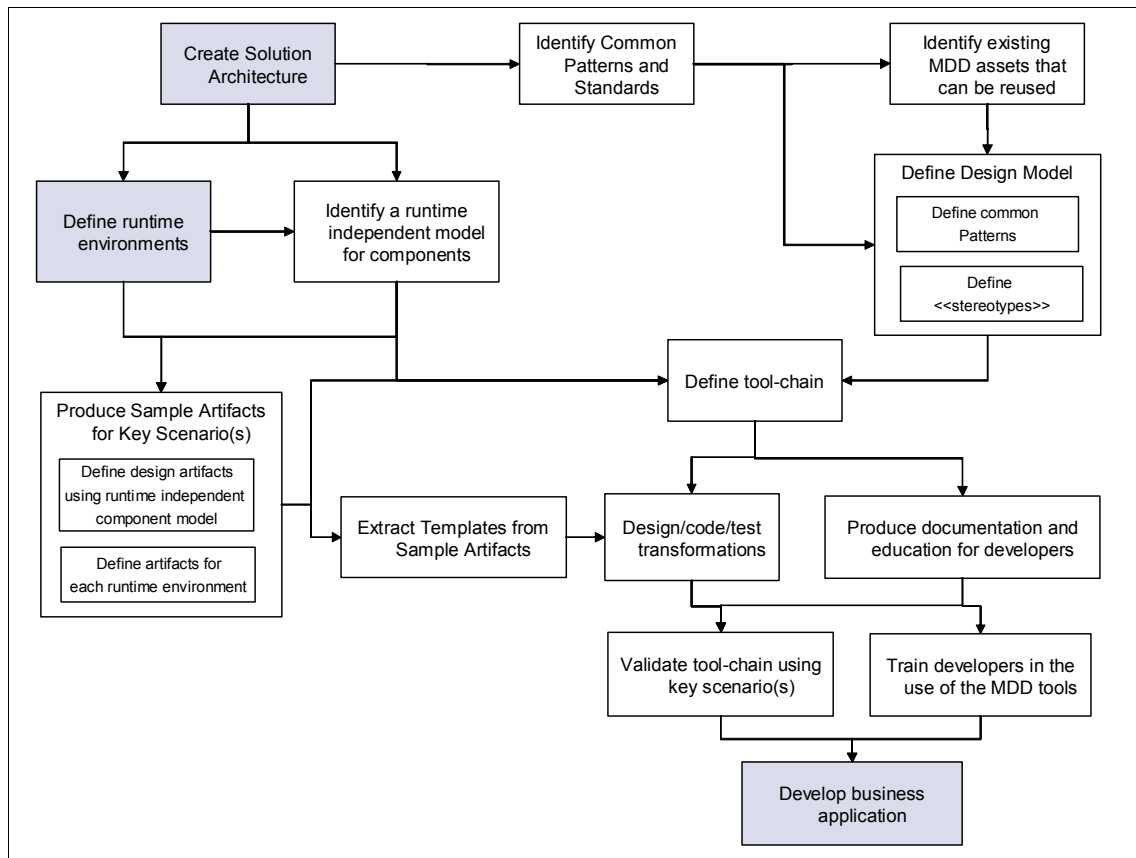
Figure 4-2   Steps used to develop the MDD tooling for the business application development team

You may choose to develop all of the tooling in advance of the business application development or take a more iterative "just-in-time" approach. In either case, be sure to allow additional time during the business application project to develop enhancements identified as the tooling is used for the first time.

4.2.1 Descriptions of tasks

The initial tasks for developing the MDD tooling occur in any traditional development approach. Your solution architect performs them and defines the high-level structure of the business application.

► *Create solution architecture:* During this task, the solution architect defines the conceptual structure of the business application. This is expressed as a number of architectural styles that the developers will adopt when building the business application.

► *Define runtime environments:* In this task, the solution architect defines the runtime environments in which the business application should run. This covers all test environments, including unit test and final production environment.

Once these first two stages are complete, the solution architect has a good understanding of what needs to be developed for the business application. At this point, the MDD-specific tasks can start.

1. *Identify common patterns and standards*: The solution architect identifies the repeating patterns within the business application. These patterns often occur either because of the consistent use of an architectural style, or due to the requirements of the runtime platforms. The common patterns can be described using the standard way for the organization's development process.

2. *Identify existing MDD assets that you can reuse*: In this task, the solution architect compares the common patterns they identified with existing MDD assets, making any necessary small adjustment to their architecture to exploit what is already available. Existing MDD assets could come from previous MDD projects or from standard tools and packages.

3. *Define design model*: During this task the solution architect chooses the type of UML model that is appropriate for the application developers to use when defining the specific details about a component that they are building. The solution architect also creates an initial list of stereotypes for the project's UML profile. To perform this task, the solution architect needs to understand the different types of UML diagrams (such as class diagrams, collaboration diagrams, activity diagrams) and when it is appropriate to use each one.

4. *Identify a runtime-independent model for components*: This task creates the definition of a UML model that specifies the components for the business application in a runtime-independent manner. It can be carried out either by the solution architect or by an experienced application developer who understands all of the runtime environments.

5. *Produce sample artifacts*: During this task, an application programmer hand-codes the resulting business application artifacts for a typical scenario. These sample artifacts act as the blueprint for the MDD templates and transformations. As a result, this task should be performed by your best application programmer.

6. *Define tool chain*: This task identifies the MDD tools that need to be developed for the project. It is performed by someone skilled in the development of MDD tooling. Once complete, you can create a detailed plan of the effort required to build the MDD tooling. There is more explanation of the importance of the tool chain in the next section.

The next four tasks build the MDD tooling:

1. *Extract templates from sample artifacts*: In this task, an MDD tooling developer reviews the sample artifacts and uses them as a basis for developing a template for each artifact you want generated. The template contains the code that is the same for every instance of the generated artifact. As such the MDD tooling developer needs skills in the language/format of the generated artifact. It also contains markers that the transformation uses to insert the specific parts of the artifact based on the content of a model.

2. *Design, code, and test transformations and patterns*: This task requires Java programming skills. For each transformation or pattern, the MDD tooling developer needs to write the Java code that reads a UML 2 model and either update the model or fill in the appropriate gaps in a template to generate a new artifact.

3. *Package MDD tooling*: The MDD tooling must be packaged into a form that can be installed into the workbench of each or your application developers. There are several options: simply put all of the files into a zip file with writing instructions on how to unpack it, use standard Eclipse plug-in management, use a RAS repository, or prove a full download Web site. The choice depends on the number of people that are likely to install the MDD tooling. A reasonable approach is to focus on supporting your initial set of application developers and upgrade the packaging mechanism as necessary when it a wider audience starts to use it.

4. *Produce documentation and education for application developers*: A solution architect or technical writer can perform this task. The result is a description of how an application developer builds a model and then selects the right transformation to generate the correct artifacts.

5. *Validate tool chain using key scenarios*: This final MDD tooling development task is a testing role. The MDD tooling models and generates all of the artifacts required for each runtime platform to support a few key scenarios.

Now the MDD tooling is ready for the application developers to start using it:

1. *Train application developers in the use of MDD tools*: Before using the MDD tools, educate the application developers on how the new development process works. They need to understand when and how to use the MDD tools, and also how they fit with their traditional tools such as configuration management.

2. *Develop business applications*: At this point, the application developers use the MDD tools to build the business application.

4.2.2 The model-driven development tool chain

The flow in Figure 4-3 shows how a developer can use the MDD tools to develop part of a business application. In this example, the developer reviews the business problem and selects a design pattern. This partially populates a design model and the developer fills in details of the specific business function they are building. After that the development process is fully automated. The developer selects an option to generate the artifacts. These are packaged up and placed in the build area. Then the developer can select further options to generate the additional artifacts for a particular runtime platform.
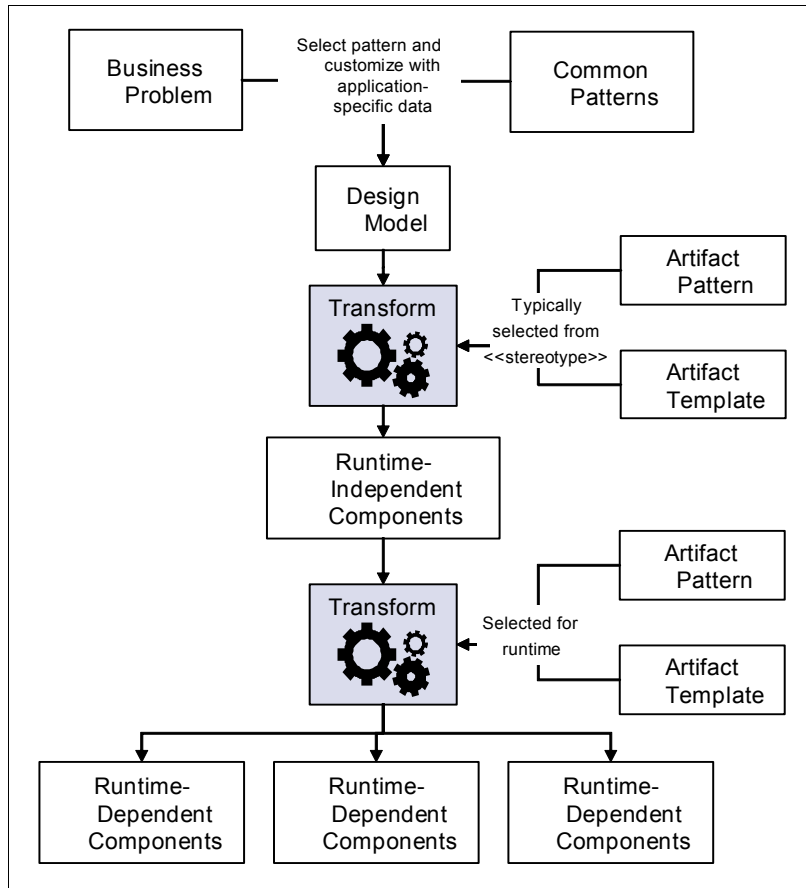
Figure 4-3   Template for an MDD tool chain

4.3 Planning a model-driven development project

When you understand the order of the tasks in an MDD project, the planning process is similar to planning for any software project. Define the tasks, size them, and allocate resources to them. The sections that follow provide some additional advice on the overall planning effort.

4.3.1 Using an iterative approach to model-driven development

In many organizations, a new approach is greeted with some skepticism until it is proven to work. If this is your first project that uses the MDD approach, it is advisable to split the development of the MDD tooling into a number of iterations.

The first iteration develops one example of each type of tooling (such as UML profile, pattern, template, and transformation), which gives you a complete tool chain for a narrow subset of the entire project. This approach has a number of benefits:

► You and your team can gain valuable experience and understanding of how much time and effort to allow to build the rest of the MDD tooling.

► It may also allow the application developers to start using the tooling earlier.

► It provides a valuable proof-point to show to skeptics in your organization.

Subsequent iterations build on the experience you gained to support a broader and broader set of scenarios for the business application.

4.3.2 Developing model-driven development skills

The following sections describe the skills you need in your MDD tooling team.

Skills for the solution architect

All solution architects have a good understanding of the business domain and the solution platform. When using MDD, the solution architect should also have a good working knowledge of UML 2 modeling. In particular, the solution architect needs to understand these items:

► The following basic diagram types and when to use them:
  – Class diagrams
  – Activity diagrams
  – Use case diagrams
  – Collaboration diagrams
  – Component diagrams

► How to extend UML 2 elements using stereotypes defined in a UML profile

There are numerous textbooks available on UML 2 that cover the diagrams and extending UML 2. In addition, there are specialized UML 2 courses. UML 2 may also be covered in an object-oriented design course.

Skills for the model-driven development tooling developer

Writing the MDD tooling is principally a Java coding exercise. The MDD Tooling Developer creates:

► *RSA pattern implementations:* These are RSA plug-ins that ensure your application developer's models follow the architectural style set out by the solution architect. This may pre-populate a UML model or run additional validation checks once the application developer completes their model.

- *RSA transformations:* These are RSA plug-ins that typically read a UML 2 model and generate new artifacts from it.
- *RSA profiles and constraints:* This is where new stereotypes and the rules that control their use are defined.
- *Model validation rules (constraints):* These constraints define the valid relationships between elements in a UML model.
- *RSA extensibility:* This is the framework for adding new functionality to RSA.

Each of the plug-ins are installed into the application developer's copy of RSA so that the plug-ins can be run from the RSA menu bars.

The MDD tooling developer therefore needs the following skills:

- Java design and programming skills
- Knowledge of the UML 2 programming interface from Eclipse
- Knowledge of how to build and deploy plug-ins into RSA
- Optionally, knowledge of RSA UML diagramming programming interface

Skills for the business application developer

The business application developer needs to understand how to use the MDD tools in RSA as part of their development process. The exact details of this work are dependent on the specific MDD tools that you developed. This is why one of the tasks in the MDD project plan is to provide education on your MDD tools to these developers.

4.3.3 Thinking about reuse

The MDD tooling built for your project will have been used many times by the application developers as they are generating artifacts for the business application. It is also possible that this tooling could be reused on subsequent projects.

Reuse is something we find difficult as an industry. Typically, software engineers prefer to design and write their own code, and most project managers prefer to avoid the cross-project dependencies that shared assets bring. However, reuse remains on the agenda because of its potential to save time and cost in software development.

Reuse is successful in organizations with the following characteristics:

- The software development process includes mandatory tasks that actively seek for assets to reuse.
- Teams are not given all of the resources they need to develop all of the code for the project.

- ▶ Proper care was taken to publish assets in a searchable way (using something like the Reusable Asset Specification).

- ▶ Shared assets have clear lines of ownership and funding for maintenance.

- ▶ All teams sharing assets need to develop to a common platform and standards.

Using MDD does not guarantee that your team builds tooling assets that can be reused in future projects. However, it has some interesting effects.

First, it splits the development effort into conceptual modeling and then generation of physical artifacts. This separation of concerns can provide follow-on projects with more places to customize for reuse, as seen in the following examples:

- ▶ When the follow-on project defines their conceptual model, it is easier for them to spot the overlaps between what they need to build and what was built before since both conceptual models are at a higher level of abstraction than the code, removing much of the detail that obscures common patterns.

- ▶ A small change to the UML profile and consuming transformation or transformations to provide one or two more stereotypes in the conceptual model could enable a follow-on project to reuse the entire downstream development environment.

- ▶ Where transformations make good use of configuration data files and templates, a follow-on project can make changes to the configuration data files, the templates to generate different artifacts, or do both.

Second, good MDD tooling increases the percentage of the time that software engineers are involved in the creative design of systems compared to the more mundane process of coding and debugging software. As a result, once they see the benefits of the MDD approach, they are motivated to exploit and extend the MDD tooling on subsequent project.

As such, it may be worthwhile to investigate whether the MDD tooling that your project produced has potential for reuse. This assessment could occur both during the planning stage at the start of the project, or during the review of the project towards the end.

4.4 Quality control for model-driven development tooling

You need to ask specific questions during the development of the MDD tooling to ensure that you are getting the maximum value from it. In broad terms, you are looking to the MDD tooling to support the following items:

► The enforcement of key design patterns and standards

These are specified by the solution architect as part of the business application architecture. In our integration example, the MDD tooling enforces the approved contract of behavior for every Web service developed. As a result, audit logging and performance data are gathered in a consistent and complete manner.

► Automating the development of any code, data files, test scripts, and documents is time consuming, mechanical, repetitive, error-prone, or likely to change frequently

For example, in this book we are automating the generation of the EJBs used in the Web services facades since this is difficult code for most application programmers to get right, and the implementation of each EJB follows some well understood patterns.

► The support of multiple target runtimes such as test and production environments

In our integration example, we have a unit test environment, system test environment, and a production environment. We are creating all of the business artifacts in a runtime environment independent method and then generating the deployment artifacts for each runtime environment

The solution architect should be able to explain to you which of these points is addressed in the MDD tooling.

The MDD tool chain is the flow of tasks and tools that the application developer performs to generate and manage a piece of the business application. This tool chain defines how much effort it will take to develop the business application once the MDD tooling is in place and is a key input into your estimating for the project plan of the business application.

Project managers should validate the tools-chain for the following characteristics:

► A business application developer should never modify a file that was generated. This can cause either a restriction to using the tooling only once and making all subsequent changes by hand, or it increases tooling expenses greatly because it would need to support round-tripping.

► The tooling should be fully integrated with your configuration management system. Configuration management covers the tools and processes you use

in your project to support the backup, versioning, and sharing of files generated by the project team. This should include specifications and design documents, data files, configuration, models, java code that implements the MDD tooling and, eventually the business application.

► The tooling should not require the same piece of information to be entered more than once as this slows your business application developers down. This would introduce inconsistencies and errors.

► It should be possible to regenerate all of the business application artifacts from a batch file so that if a transformation needs to be enhanced partway through the build of the business application, everything can be regenerated automatically.

You can use MDD to generate much more than just code. In fact, significant value can be realized when the artifacts that are generated include test programs, documentation, configuration, and status reports. When you are validating the tool chain check that all opportunities for automation are being considered.

4.5 Tracking a model-driven development project

Once you build a project plan that documents the effort, skills, and dependencies between the tasks, tracking an MDD project is no different than any other software development project. It takes standard project management skills to spot places where schedule slippage occurs, and make appropriate adjustments to the resource allocations whenever appropriate.

The one key advantage that MDD brings to the table, when it comes to tracking the use of the MDD tooling during application development, is that it generates status reports at the same time that it generates the code. In addition, generated test cases can be written so that they automatically record the test results each time they run. As a result you have progress data that accurately reflects the actual progress of the project (rather than programmer's estimates). This can give you advanced warning of potential slippage, giving you greater opportunity to make corrections.

4.6 At the end of the project

For your first MDD project, the end of the project is the time to assess the real value you derived from the approach since this influences the level of investment in MDD in the future.

It is useful to generate the following types of metrics.

► The cost of developing the MDD tooling

► The productivity of the application developers using the tooling

   For fair comparison with traditional projects, express this in terms of how much of their time is needed to develop all of the hand-written code and the code generated from the transforms that operated off of their models.

► The level of quality achieved by the MDD project team

► The effort required to reuse the MDD tooling on subsequent projects and the expected benefit

► The opinions of the development team in terms of whether they enjoyed working with a MDD approach, the skills they developed, and any suggestions they may have to improve the approach

With this information, you can engage in an assessment of how MDD should be exploited in future projects.

4.7 Summary

In this chapter, we covered the following additional steps required to exploit MDD within a project.

► Identify common patterns and standards.
► Look for opportunities to reuse existing assets.
► Define the design model.
► Identify a runtime-independent model for components.
► Produce sample artifacts.
► Define tool chain.
► Extract templates from sample artifacts.
► Design, code, and test transformations.
► Package tooling for application developers.
► Produce documentation and education for application developers.
► Validate tool chain using key scenarios.
► Train application developers in the use of MDD tools.
► Develop business applications.

We also presented guidance on how to plan and run your first MDD project.

► Plan early and make explicit investment in the MDD tooling.

► Use your top people to develop the MDD tooling since your aim is to capture and automate their best practice.

► Ensure the development process does not allow generated artifacts to be modified.

- ► Consider generating documents, configuration, status reports, and test cases as well as code.
- ► Ensure that your development process supports test environments as well as the production environment.
- ► Remember to define the configuration management strategy into your MDD tool chain.
- ► Allocate time to train the development team on the use of the MDD tooling.
- ► Take time to consider whether your MDD tooling is reusable on subsequent projects. If reuse is possible, ensure tasks to mine reusable assets are included in follow-on projects.

Chapter 5, "Model-driven development solution life cycle" on page 59, goes into more depth about the changes occasioned to the solution life cycle by applying MDD to a project.

5

Model-driven development solution life cycle

Model-driven development (MDD) requires the project planner to think about the solution life cycle afresh. In this chapter, we discuss the improvements to the performance of lifecycle activities, such as testing, deployment, and maintenance, that MDD can bring.

We discussed the solution life cycle in Chapter 4, "Model-driven development project planning" on page 45, in the context of planning an MDD project. In this chapter, we focus on the opportunities MDD creates for *improving* the solution life cycle. We also cover the aspects of the MDD solution life cycle that are not addressed in the other chapters.

MDD can change the solution life cycle in the following ways:

► Creates new artifacts at earlier stages of the life cycle, such as models. MDD artifacts need to be managed as primary artifacts, like code, rather than as project documentation.

► Stimulates changes in the way the project teams perform tasks. MDD changes what is developed and the way development is done. It also impacts later stages in the solution life cycle by potentially changing how test and production environments are configured.

► Creates opportunities for shifting responsibility for activities, such as writing, configuring, deploying, and monitoring scripts, from being performed

manually to being done automatically. Transformations can be written that will generate the scripts automatically using information captured into a model.

5.1 Introduction to the solution life cycle

The *software life cycle* refers to the phases a software product goes through from its conception to when it is no longer supported. The *solution life cycle* refers to the stages and environments a software solution goes through from a proof-of-concept to being a live production system and then going through maintenance until it is finally withdrawn from production. All software, including solution software, has a software life cycle that concerns the development and maintenance of software components. Solutions, additionally, have the stages they go through from being developed from components to being brought into production.

The stages of the solution life cycle differ from project to project and typically include the following items:

▶ **Design**: A development and unit test environment used for prototyping, and evaluation of technologies. This stage also involves requirement gathering and elaboration, and analysis.

▶ **Capability**: A system test environment used for multi-threaded testing across the components that make up the functions of the application/solution.

▶ **Operation**: A multi-platform integration environment that mirrors the requirements of the production environment. The operational environment is used for quality-of-service testing and proving that the solution meets the required service-level agreements.

▶ **Production**: This is the *live* environment. Before going live the solution and environment must be certified to meet the quality-of-service, fail-over, and high-availability requirements.

▶ **Life cycle**: This stage refers to the management of the production environment and solution artifacts with respect to the migration, versioning, and replacement of hardware, middleware, and application components

When we refer to life cycle in this chapter, we refer to the complete solution life cycle of a model-driven development project, including the development of software components and their configuration and deployment in a solution.

In what way does the introduction of a model-driven approach modify the activities that must be performed and managed throughout the life cycle? Consideration of lifecycle activities overlaps with the discussion about planning an MDD project in Chapter 4, "Model-driven development project planning" on page 45. The difference is that in Chapter 4, "Model-driven development project

planning" on page 45, we discussed how a project planner should think about modifying the development cycle to accommodate the changes wrought by an MDD approach. In this chapter, we focus on the following items:

► The changes to the solution life cycle that are necessary to benefit from the use MDD effectively

► The tools that are available to support the lifecycle stages of an MDD project

► The solutions to some of the project challenges around the consideration and implementation of MDD lifecycle activities

5.2 Model-driven development life cycle

Since the models are the primary artifacts during model-driven development, it becomes crucial that the lifecycle elements are considered a lot earlier in the project cycle and that we explore what aspects can be captured in the models. To be true to the promise of MDD, we must apply the same thinking of encoding and automating the expertise of software professionals so that it can be applied consistently and repeatedly to the lifecycle aspects. This means capturing and encoding the expertise for testing, deployment, packaging, security, and so on.

MDD adds another dimension to the life cycle. Traditionally, we develop the solution, test, and deploy into an environment. With an MDD project, the development is split into the following two phases:

► Creating the framework to generate the solution services
► Generating the solution services

5.2.1 Create the framework to generate the solution services

The framework covers the creation, testing, and deployment of the models, the patterns, and the transformations that generate the solution services. Ensure that the framework is properly tested to ensure that the generated solutions are accurate and consistent. A commonly used practice is to have the platform experts manually, or using the normal platform tools, create an example solution. This way, at every iteration of the framework, a generated test solution can be compared with that developed by the platform experts.

The transformations must be rolled out before commencing the generation and roll out of the solution. This ensures the integrity of the framework and significantly reduces situations where the transformations are being updated and corrected while the solution is being generated.

5.2.2 Generate, customize, and test the solution services

The generated services are executed in the Capability environment. Test and deployment artifacts are targeted for this environment, and in some cases are not automated, with deployment performed interactively using a GUI. This allows for the capture of the steps and information needed to deploy the application, which aids the creation of the deployment scripts for the more complex environments.

The generated services are validated against the sample services provided before the focus of the project moves to the next phase, where the generation of the real solution and the related artifacts (test, deployment, and so on) are prepared for the deployment environment.

Stated another way, the creation of the framework is synonymous with creating the model-driven solution factory, and the second phase is the use of the factory to create our products.

As with all software projects, there are lifecycle decisions to make towards the management and maintenance of the solution, assets, and tools. With MDD projects, there are certain lifecycle considerations that, need to be determined a lot earlier in the project, and have strong influences on the success of the project. For instance, how is versioning and problem determination managed in an MDD project where the models are the primary artifacts? We discuss these lifecycle considerations next.

5.3 Model-driven development and versioning

Versioning affects the generated solutions as well as the patterns, profiles, transformations, and other model artifacts that generate these solutions.

In our service integration scenario, for instance, imagine that the production environment is a server farm with clusters of middleware running at different versions, having different capabilities. Support for different versions of J2EE, or middleware products, may be a case in point. In such situations, we would version our transformations to tie in with each runtime, as well as versioning changes to the transformations that are specific to the generated applications and services on each runtime. For our framework, we must determine which version of the transformation to execute.

For the same scenario and analyzing the generated services, what rules can be applied for versioning an integration service? Let us assume a minor change to the updateCustomer integration service by supporting a previously unsupported optional field of *secondHomeAddress,* but without changing the interface for previously deployed applications? There is still the need for the integration

service to support all existing requesters and integration facades, and for all related schemas, xml transformation, and message validations not to fail or become invalid. In this case, there needs to be support built into the solution for determining and selecting the required versions into the services.

5.3.1 Versioning and replacement policies

Moving to the production environment, it is crucial to have a release plan that specifies the versioning and replacement policies. The policies provide rules and guidelines on when to version, roll in a new version, or replace a whole service. For instance, in our example with the new secondHomeAddress field, we would determine the best approach from our policy, which may include:

▶ If optional fields or structures are added to an existing data structure, the minor version number of the structure and any new services using this structure are augmented by one. Thereby moving the version number from M1m0 to M1n1. Existing implementations are supported by the modified service.

▶ If the new field or structure is compulsory, the major version number is increased and only new implementations are supported. Existing applications would continue to use the previous version until they are enhanced to support the new version.

▶ Minor version changes replace older versions since the support for older implementations is maintained.

▶ There is normally a release strategy that requires the enhancement to take place within a period of time so that old versions can be gracefully retired and no longer supported. Major version changes must co-exist with older (major) versions for the agreed support period during which migration needs to occur.

The versioning and replacement policies have an impact on the deployment. There has to be a mechanism for replacing or deploying new versions that must co-exist, and ensure they are accessible by the right service clients.

We must also determine the level of versioning (per file, per class, per service, per deployment unit, and so on) to apply. For our example, we version transformations, patterns, profiles, and all reusable artifacts. It is also important to understand any composite levels of versioning, for example, the combination of a specific transformation version with that of a specific profile version.

5.4 Model-driven development and artifact management

Models, transforms, and code are some of the artifacts that we generate and use in an MDD project. These assets may be as simple as text files containing configuration properties and documentation, or as complex as project archives containing executable code and deployment descriptors.

According to Rational Unified Process (RUP), an *artifact* is "a work product of the process: roles use artifacts to perform activities, and produce artifacts in the course of performing activities". The artifacts that we are considering are all examples of artifacts in the RUP sense. For this chapter, we limit ourselves to software assets that contribute to the solution as part of the model or the executable services.

5.4.1 Reuse model artifacts

The success of an MDD project depends on the successful reuse of model artifacts. With this in mind, it is worth analyzing the value and contribution of an artifact to determine whether it should be generated.

For instance, patterns and transformations provide value through reuse, while a configuration script may only be used once but contributes greatly. As such, while reuse is a major factor in determining whether it is worth going through the expense of generating an artifact, there are other considerations based on context and value. Reuse in MDD covers the reuse of patterns, models, transformations and to a lesser extent the code. The management of these artifacts, their related descriptions (probably captured in templates), and the maintenance of the repositories become increasingly important.

The management of artifacts includes, but is not limited to, the following issues:

► Successfully identifying and retrieving an artifact for reuse

► Ensuring that the appropriate artifact is retrieved for the version of the target runtime

► Checking the integrity of an artifact and verifying whether it is this the latest or appropriate version of the artifact

► Checking the certification of an artifact, and whether it is certified to run in this environment

  Perhaps it is only for test environments or restricted through license conditions, only to run in particular environments.

When determining the methods and tools to use for artifact management, there are a set of factors that must be considered, including integrity management services and deployment support.

5.4.2 Integrity management services

There is a need to determine the reliability of the model and solution artifacts in order to maintain the integrity of the system. As such, we recommend that a mechanism to certify that the service meets standards (set by governance), is adopted. This *Certification of Service* practice should be applied to all artifacts in a graded way. For instance, the artifact may be certified as ready for deployment to the test environment only, or to the production environment, or it may be certified as deprecated and not to be used by new applications.

You must analyze the certification practices early because it may influence the choice of repositories to use (for example, a repository may provide features that support some of the requirements).

The early consideration of artifact management helps determine how much, if any, information needs to be added to the generated artifacts to enable their certification, and the subsequent query/retrieval of the required artifacts for a particular context. For instance, in our service-oriented integration (SOI) scenario, the stored artifacts are tagged with enough information to be searched for reuse by the following items:

► Service type (IB, PF): See 2.2.1, "ESB structure" on page 21
► Stage (beta, deployment ready, deprecated)
► Protocol (SOAP/HTTP, SOAP/JMS, JMS)

Some of this information is captured in the RAS templates that describe the artifacts, and others are derived from the content of the artifact.

5.4.3 Deployment support

We are interested in the ease with which the assets are retrieved and deployed. The structure of the repository must take into consideration the versioning policies and the certification policies, to enable the effective maintenance of the service repository and assets.

The definition of the asset types is also important for supporting deployment (might be easier to deploy a pre-packaged EAR file than an EJB), a set of Java classes, and some data schemas. In most cases, the asset types to be stored are determined by the solution type and the deployment policy. Compare these two cases:

► A rapid deploy scenario where deployment may be automated with the newly created and certified services are hot-deployed

    In this case, the packaging of artifacts is normally part of the generation step to maximize the effectiveness of the rapid deploy.

- A scenario where there could be established packaging scripts and release to production practices.

    For this scenario, the repository may hold a simpler type of asset. A packaging application is applied to these artifacts, and the results are subsequently deployed.

The adoption of technologies and methods for artifact and configuration management, to a large extent, depend on the type and size of project and the type and size of artifacts. The certification policy, deployment policy, and so on, not only influence the technology and product to be adopted for artifact management but also inform the decision to divide the features provided by the tool and those to be built into the artifacts. The handover of artifacts through the stages are to a large extent dependent on these policies and must be analyzed as part of the lifecycle discussions early on.

5.5 Model-driven development and problem determination

The need to determine the problems in an MDD project also span both the framework and the generated solution. Mechanisms for debugging during the different stages, from design to production, are vital to the success of the project. With the models being the primary artifacts of the MDD approach, ensure that the mechanisms for debugging models are developed alongside the models themselves. In some cases, there might be a need to debug the generated code and other solution artifacts.

Here are two reasons why you should not debug the generated code:

- It is extremely hard to work back from the generated code to the underlying problem in the model.
- It is crucial that all changes are made in the models or transforms and not in the generated artifacts.

    This ensures the consistency of the models and solution and protects the integrity of the factory and generated solution. In essence, debug moves up the stack with the models and the transformations become the point of problem determination.

5.5.1 Tooling versus instrumentation

In practice, problem determination is achieved with a combination of tooling and instrumentation. Tooling covers the debugging features provided by the modeling and development tools to enable problem determination in artifacts ranging from the models to the generated artifacts. Instrumentation refers to the mechanisms that we build into our models and transforms to enable the detection of problems. Instrumentation ranges from building in trace messages to building in events and access to an event infrastructure for problem determination and solution management.

Remember that, after the software product or solution is shipped and is in production, the models still need to be treated as source and primary artifact. Any defects found must be corrected in the models and not in the generated code as this creates inconsistencies and invalidates the model for future releases, thus negating the benefits of MDD.

5.6 Information mining

The success of a project depends upon the quality of the information that is used and produced. As such, the models and transforms in a model-driven development, project using UML, must be populated with accurate and valid information. In practice, these sources for the information may be numerous and in different formats. RSA provides customizable features to enable the input of information from different sources and in different formats, as well as the generation of documents in different formats.

For example, in our SOI example using RSA, all models, packages, and model elements are described in the documentation tab. We use a predefined structure to enable the generation of reports and user documentation in the form of HTML for the Web and Microsoft® Word documents using SoDA for generation.

For those roles that may not work directly with the models, we provide a Web application that collects their information and populates the models. Depending on the level of abstraction, the input data for a model may be from an application in the form of data stored in a predefined structure to a file system. The information in the models may also generate input to other applications in the form of configuration and property files.

While information may be stored in different locations, you must understand your data inventory and to have a governance policy for that data. This understanding helps to limit the need to manage the same data in multiple locations and to ensure the integrity of your data. As much as possible, make the models your

main source of information and inventory with other tools simply providing views to the information in a customized format.

5.7 Testing

There are numerous on-going projects in the field of model-based and model driven testing.

► Model-based testing is an approach to testing that uses a model of a system to predict its behavior and uses the predictions to automate test cases.

► Model-driven testing focuses on providing a toolset and a test execution environment to automate the running of tests, and leaves the model developer free to focus on devising a good set of tests specific to the application.

We do not cover model-based testing or the automated generation of test cases here. However, we discuss the model-driven testing approach.

Model-driven testing can be described in two phases:

► Testing the model framework

Verify the framework to ensure that it generates the correct artifacts in the right format.

► Testing the generated solution artifacts

Validate solution artifacts against the solution requirement and the business logic of the services.

5.7.1 Modeling for testing

To model for testing we use the same approach of applying transforms to the models to generate the execution artifacts, but in this case, the focus is on generating the test artifacts.

A useful approach to modeling for test is to reuse the models that describe the solution, but mark these models with testing elements. Marking the models with test elements enable us to understand the boundaries, the system under test, and the test data. The generated test artifacts may be as simple as stub objects and methods, through to test services that simulate a live system, and may be test scripts to run a test harness. Another approach is to use the model to seed the code, and then use the stubbed out code to generate test cases.

Our focus here is to illustrate the methods for modeling for test and to discuss practices for testing in an MDD project. To this end, we use our service integration example to evaluate possible ways of modeling for test.

5.7.2 Applying test patterns

Our pattern language may also contain test patterns. In our SOI example, we captured the Application pattern for the updateCustomerDetails example in the format shown in Figure 5-1.
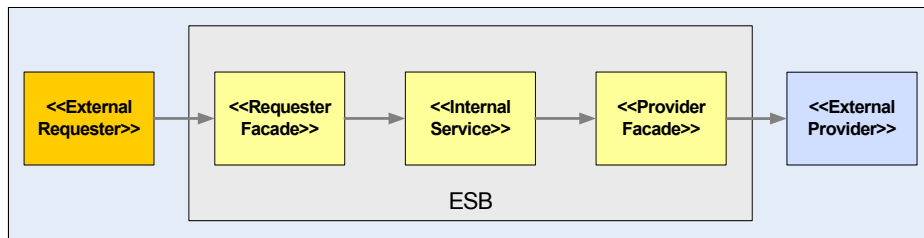
Figure 5-1   The ESB Application pattern

Our test patterns capture the expertise needed to test the solution in the different environments. For instance, during the end-to-end testing of the updateCustomerDetails service in the capability environment, there is a need to connect the provider facade to the required external provider. Since the external provider may be a live system that we cannot run tests against, there is a need to generate a test provider. The test provider is a simulator that provides all the public interfaces of the external provider and supports all required data and message types.

We provide an *external entities test pattern* in RSA that generates a test provider or requester based on the definitions of the external requester or provider. This test provider or requester is connected to the required facade service (as illustrated in Figure 5-2) to ensure the necessary operation calls are generated.
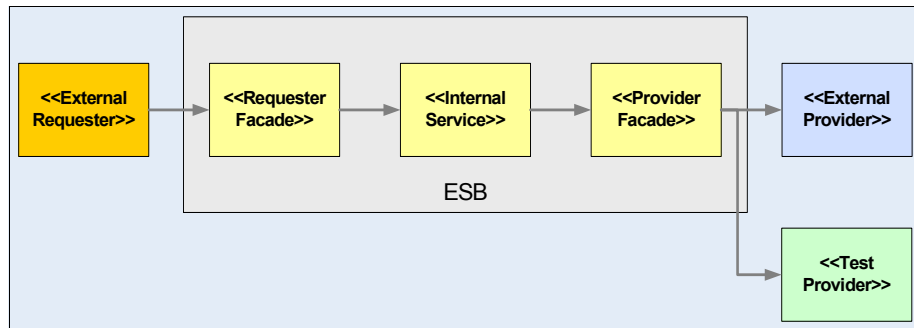
Figure 5-2   ESB Application pattern with a testing pattern applied

Using the patterns in this example enable us to capture and apply domain knowledge based on the environments and testing practices used in the project. The application of the patterns could have occurred with the manual augmentation of the models with test clients, test services, and stub classes ensuring that they reflect the required definitions. In either case, the generated artifacts, in the form of methods, classes, projects and so on, may be packaged separately or may be inline with the real solution artifact, depending on complexity, and the packaging and deployment policies.

5.7.3 Modeling using the UML testing profile

The UML testing profile provides a formal extension to UML that defines tests for our system captured using UML models. The Object Modeling Group (OMG) UML Testing Profile specification describes the UML testing profile as a language for designing, visualizing, specifying, analyzing, constructing, and documenting test artifacts. It is a language you can apply to testing systems in a wide variety of application domains, whatever object or component technologies were used to build the applications. The UML testing profile can simply describe test systems, or it can be used with UML to handle system and test artifacts together.

The UML testing profile has the following characteristics:

▶ It is based upon the UML meta-model and extends UML with test specific concepts.

▶ It enables the specification of tests for static and dynamic aspects of UML models.

► It integrates with UML and existing testing technologies and standards (for example, JUNIT and TTCN3).

The UML testing profile is organized in four logical groups (descriptions come from the OMG UML testing profile specification) of test concepts:

► Architecture defining concepts related to test structure and test configuration
► Data defining concepts for test data used in test procedures
► Behavior defining concepts related to the dynamic aspects of test procedures
► Time defining concepts for a time quantified definition of test procedures

The UML testing profile addresses black-box conformance testing and as such the system under test (SUT) is not specified as part of the test model. It is only accessible via its publicly available interfaces and operations. No information about the internal structure of the SUT is available for use, because it is a black-box. The test architecture package imports the model of the SUT in order to get the necessary access rights.

In our example, we would treat the ESB and contained elements as the SUT, and use the UML testing profile to define the test cases, test elements, test behavior, test data, verdicts, and so on, that enable the generation of our testing artifacts and the subsequent test exercise in JUNIT. The creation of a transformation to handle the profile and models that had the UML testing profile applied is not covered here. We left that as an exercise for you.

For more information about model-driven testing, see:

► http://www.agedis.de/
► http://www.haifa.il.ibm.com/projects/verification/mdt/tools.html
► http://heim.ifi.uio.no/~janoa/wmdd2004/presentations/alan.pdf

5.8 Summary

In this chapter, we introduced key ideas behind lifecycle management in a model-driven development project. We also provided an outline of certain lifecycle activities that model-driven development can improve, such as testing and deployment.

Chapter 6, "Model-driven development in context" on page 73, considers some complementary approaches to the MDD methodology we described so far. It also discusses how MDD relates to the concept of Model-Driven Architecture as defined by the OMG.

6

Model-driven development in context

Model-driven development (MDD) is not an approach that exists in isolation. We already saw the strong connection between MDD and pattern-based development.

This chapter sets model-driven development in the context of other initiatives that are taking place in the industry. We cover the role of the Object Modeling Group (OMG) industry standards body in MDD. We also review how other approaches to software development compare with MDD, to include the following items:

► OMG Model-Driven Architecture
► Asset-based development
► Business-driven development
► Software Factories and domain-specific languages (DSL)

It is not necessary to read this chapter before moving on to Part 2, "Implementation" on page 91. You can skip this chapter on your first reading of this book.

6.1 OMG and Model-Driven Architecture

The Object Management Group is an open consortium that produces standards for interoperability in the enterprise application space. The OMG is responsible for the Unified Modeling Language (UML) that is central to MDD. The OMG is also driving the Model-Driven Architecture (MDA) initiative. MDA is a formalization of an MDD approach such as the one that IBM Rational promoted for years. As defined by the OMG, MDA is a way to organize and manage enterprise architectures supported by automated tools and services for both defining the models and facilitating transformations between different model types.

You will find that the terms MDA and MDD are often used interchangeably. In this book, we use the MDD to refer to the activity that is carried out by software developers. The term MDA is reserved for its formal OMG definition that is more focused on creating a formal framework in which MDD can operate.

The Object Management Group's MDA guide describes MDA as having the following three primary objectives:

► Portability
► Interoperability
► Reusability

It aims to achieve these objectives by separating the specification of the operation of a system from the details of the way the system is realized on a particular platform.

MDA enables tools to be provided to help meet these objectives by doing the following tasks:

► Specifying a system independently of a platform
► Specifying platforms
► Choosing a platform for the system
► Transforming the system specification into a specification for a particular platform

The notion of platform is central to OMG MDA. A platform is a realization of interfaces and usage patterns by a set of subsystems and technologies that an application can use without concern for the details of how the platform is implemented.

6.2 MDA models

MDA defines three kinds of models.

- ► **Computation independent model (CIM)**: A CIM describes the requirements for a system and the business context in which the system will be used. The model customarily describes what a system will be used for, and not how it is implemented. The model is often expressed in business or domain-specific language and makes only limited reference to the use of IT systems when they are part of the business context.

- ► **Platform independent model (PIM)**: A PIM describes how the system will be constructed, but without reference to the technologies used to implement the model. The model does not describe the mechanisms used to build the solution for a specific platform. The PIM may be more suited to be implemented by one platform rather than another, or it may be suitable for implementation on many platforms.

- ► **Platform-specific model (PSM)**: A PSM is a model of a solution from a particular platform perspective. It includes both the details from the PSM that describe how the CIM can be implemented, and the details describing how the implementation is realized on a specific platform.

At the heart of MDA is the concept of transformation from a PIM to a PSM. The process of transforming one model to another in the same system is called *model transformation*. Figure 6-1 illustrates the transformation of a PIM, possibly with additional information, into a PSM.
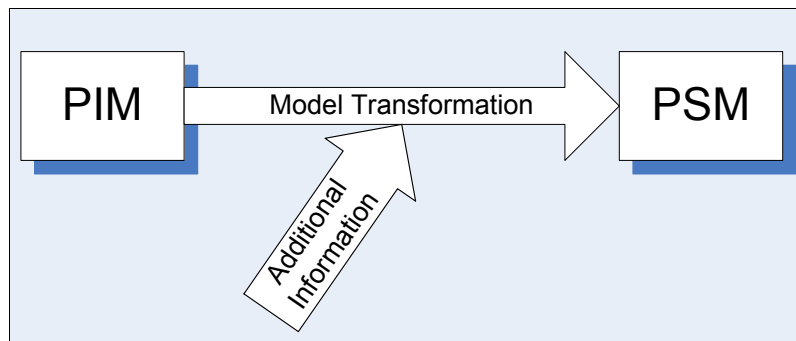
Figure 6-1   Transformation of a PIM to a PSM

MDA does not treat these models as fixed layers but explains that PIMs and PSMs can be layered with each model being a PSM with respect to a more abstract model and a PIM with respect to a more concrete model. For example, we could have a high-level design model, a detailed design model, and an implementation model with two occurrences of the MDA pattern. At each level,

we introduce further assumptions regarding the implementation platform. The detailed design model is a PSM with respect to the high-level design model and a PIM with respect to the implementation model.

There has been much discussion in the MDA community about whether a PIM should first be transformed into a non-code PSM and then to code, or whether it is permissible to generate code directly from a PIM (meaning that your PSM is the code). When working with the J2EE and Java platform, Rational Software Architect (RSA) offers visualization of code artifacts as UML diagrams. This provides the advantages of being able to visualize platform artifacts while avoiding the need for an extra level of transformation.

In OMG terminology, the main focus of this book is modeling applications as PIMs and then transforming those models into PSMs, captured directly as implementation artifacts.

Much of what is presented in this book is also appropriate for transitions between other modeling layers. For example, when following the Rational Unified Process (RUP), we can begin with an Analysis model that is then transformed into an outline of a Design model. We could also transform a WebSphere Business Integration (WBI) Modeler model into a PIM that acts as a specification for software development. RSA includes such a transformation and can import WBI Modeler models.

6.2.1 IBM and MDA

The white paper *An MDA Manifesto* articulates the IBM vision for MDA. You can find this manifesto on the Web at:

http://www.ibm.com/software/rational/mda/papers.html

This introduces the three basic tenets of MDA as shown in Figure 6-2 and Figure 6-3.
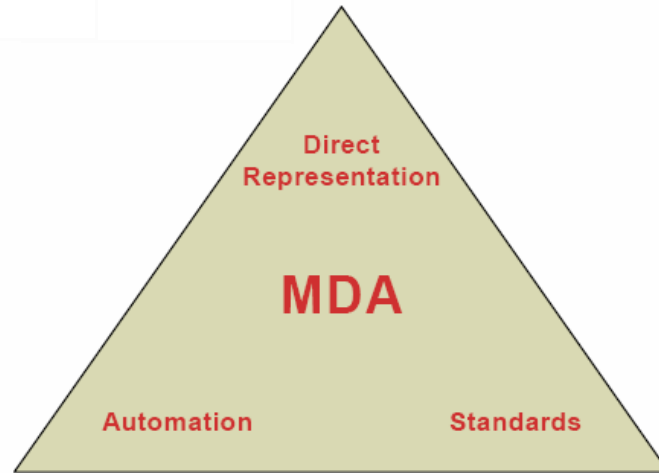
Figure 6-2   Three basic tenets of MDA

Basic tenets of the MDA manifesto

In essence, the foundation of MDA consists of three complementary ideas:

1. *Direct representation*. Shift the focus of software development away from the technology domain toward the ideas and concepts of the problem domain. Reducing the semantic distance between problem domain and representation allows a more direct coupling of solutions to problems, leading to more accurate designs and increased productivity.

2. *Automation*. Use computer-based tools to mechanize those facets of software development that do not depend on human ingenuity. One of the primary purposes of automation in MDA is to bridge the semantic gap between domain concepts and implementation technology by explicitly modeling both domain and technology choices in frameworks and then exploiting the knowledge built into a particular application framework.

3. *Open standards*. Standards have been one of the most effective boosters of progress throughout the history of technology. Industry standards not only help eliminate gratuitous diversity but they also encourage an ecosystem of vendors producing tools for general purposes as well as all kinds of specialized niches, greatly increasing the attractiveness of the whole endeavor to users. Open source development ensures that standards are implemented consistently and encourages the adoption of standards by vendors.

Figure 6-3   Basic tenets of the MDA manifesto (IBM)

These concepts of direct representation, automation, and open standards are at the core of the model-driven approach.

6.3 Software Factories and domain-specific languages

You may have come across the ideas of domain-specific languages (DSL) and Software Factories recently, in particular in the book *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools* (see "Related publications" on page 227 for the full citation), put forward these ideas. For those of you who come across, or are interested in, both approaches it is useful to compare them.

The Software Factories Web site defines the term *Software Factory* in the following way:

> "A Software Factory is a software product line that configures extensible development tools like Visual Studio Team System with packaged content like DSLs, patterns, frameworks and guidance, based on recipes for building specific kinds of applications. For example, we might set up a Software Factory for thin client Customer Relationship Management (CRM) applications using the .NET framework, C#, the Microsoft Business Framework, Microsoft SQL Server, and the Microsoft Host Integration Server. Equipped with this factory, we could rapidly punch out an endless variety of CRM applications, each containing unique features based on the unique requirements of specific customers. Better yet, we could use this factory to create an ecosystem, by making it available to third parties, who could extend it to rapidly build CRM applications incorporating their value added extensions."

There is an overwhelming similarity between MDD and the Software Factories approach. Both approaches advocate working with application domain concepts and introducing automation into the software life cycle. Both approaches emphasize the importance of visual modeling and capturing expertise through patterns.

The main difference between the approaches is the emphasis that is put on open standards, in particular UML.

To learn more about Software Factories, see:

http://www.softwarefactories.com

6.3.1 UML and DSLs

The issue of the role of UML is often stated in overly simplistic terms: MDD advocates the use of UML for all domain modeling while the Software Factories approach advocates that UML never used. This is an incorrect statement of the positions of both camps.

While the MDD approach treats UML, with customization, as the modeling language of choice for most application modeling, it also acknowledges the value of custom languages in certain specialized circumstances. This is the purpose of the OMG Meta-Object Facility (MOF) standard that plays an important role in MDD (see page 80 for more details about MOF). UML itself is defined using MOF and there are MOF definitions of many other languages. The MDD approach acknowledges the value of non-UML DSLs as a technique to be applied judiciously.

Further, the Software Factories approach does not reject UML entirely. It suggests that you use UML for developing sketches and documentation, where DSLs should be used for developing models from which code is generated. To learn more, go to:

http://msdn.microsoft.com/vstudio/default.aspx?pull=/library/en-us/dnvs05/html/vstsmodel.asp#vstsmodel_uml

Advantages of using UML profiles as DSLs

► UML is an open standard modeling language for which there are many available books and training courses. UML is a recognized and transferable skill for software developers.

► UML profiles provide a lightweight approach that is easily implemented using readily available UML tooling. In the future, it may be possible to generate tooling for DSLs, but some customization is still likely to be necessary.

► Models with UML profiles applied can be read by all UML tools even if they do not have any knowledge of the profile. In some cases, the extensions introduced by the profile will be ignored.

► Basing all DSLs on UML creates a set of related languages that share common concepts. This makes new profiles more readily understandable and enables models expressed using different DSLs to be integrated easily. Having a set of models expressed using different DSLs replicates the middleware integration problem at the modeling level, which is clearly not desirable.

► UML can be used for high-level architectural models as well as detailed models from which code can be generated. This gives consistency throughout the software life cycle, enabling users to move seamlessly from

modeling-in-the-large to modeling-in-the-small. The approach of using UML only for sketches and documentation misses this opportunity for consistency.

Disadvantages of UML profiles as DSLs

► UML profiles only permit a limited amount of customization. It is not possible to introduce new modeling concepts that cannot be expressed by extending existing UML elements. For example, UML is not a suitable basis for a DSL for designing electrical circuit diagrams.

► The use of UML does require familiarity with modeling concepts. In some cases domain experts may have knowledge that can be utilized for the purposes of code generation, but they may not have the expertise to express these concepts using UML.

Alternatives to UML profiles for DSLs in MDD

While UML is the appropriate basis for many DSLs, there are cases where we advocate an alternative approach for some parts of the MDD process.

The following techniques can be used instead of UML:

► **MOF-based language**: In cases where a custom language is appropriate, use MOF to define a new language. The Eclipse Modeling Framework, an open source component that is included with RSA, generates a Java implementation for working with a MOF-defined language and basic Eclipse tooling to create instance of models in the language. In the future it is likely that the generation of full graphical editors will be possible.

This is the technique used to implement many of the languages that RSA supports including UML and XSD. Take care when using this approach to ensure that the language is integrated with UML and other non-UML DSLs that are used in the same solution context.

► **Custom user interface**: For some users, a visual modeling approach may not be appropriate for capturing their expertise. It may be that a custom tool with step-by-step guides and custom graphical elements is more suitable. Clearly this approach has further cost associated with it, but it is a useful technique to apply in the right circumstances.

► **Transformation from an existing format**: Information that is needed to drive an MDD tool chain may already be captured in an existing tool. This might be another software modeling tool, a business modeling tool, or a desktop tool such as Microsoft Excel. Any information that can be accessed can be transformed into UML models. Particularly for pre-existing assets, use this approach to generate models rather than manually modeling the same information.

6.4 Asset-based development

Asset-based development (ABD) promotes the reuse of assets at all points in the software development life cycle. Unlike previous reuse initiatives, ABD does not focus only on code-based components that can be directly invoked. In particular, ABD complements MDD through supporting the reuse of models, patterns and transformations. For more information about the IBM approach to asset-based development, see:

http://www.ibm.com/developerworks/rational/products/patternsolutions/

RUP for asset-based development is a RUP plug-in that describes the tasks, roles, and deliverables associated with asset-based development. You can learn more about RUP on the Web at:

http://www.ibm.com/developerworks/rational/library/4145.html

The OMG Reusable Asset Specification (RAS) standard is an important enabler of ABD. RAS provides a standard way to package, document, discover, and retrieve software assets. For more information about the RAS standard, see:

http://www.omg.org/cgi-bin/doc?ptc/2004-06-06

Rational Software Architect provides a reusable assets perspective that includes RAS client capabilities to search for and import RAS assets from repositories.

RSA supports remote RAS repositories such as the RAS repository for workgroups, and also local lightweight file-store based repositories. The RAS repository for workgroups is a WebSphere Application Server-based repository that is available from IBM alphaWorks:

http://www.alphaworks.ibm.com/tech/rasr4w

RSA also supports the packaging of artifacts into RAS assets and the publishing of those assets to local or remote repositories. RSA models, patterns, transformations, and other extensions can be packaged as RAS assets.

6.5 Pattern-driven development and IBM Patterns for e-business

Pattern-driven development is concerned with developing software by applying best practice solutions to problems.

There are two key ways in which patterns and MDD are related:

- ► MDD can automate the application of patterns

  Traditionally, patterns were written down as documents, often with the aid of UML models to explain the pattern. Patterns were then applied manually. MDD can automate the application of patterns.

- ► Patterns provide content for MDD

  MDD allows us to move from well-designed models to well-designed implementations. Patterns capture the best practices at both the modeling and implementation levels. MDD is not possible without knowledge of the patterns of the application domain and the patterns of the implementation domain.

6.5.1 IBM Patterns for e-business

IBM Patterns for e-business help facilitate the reuse of assets that capture the experience of IT architects in such a way that future engagements are simpler and faster. The reuse of these assets saves time, money, and effort, and helps to ensure the delivery of a solid, properly architected solution. The purpose of IBM Patterns for e-business is to capture and publish e-business artifacts that were used, tested, and proven to be successful. The captured information is assumed to fit the majority, or 80/20, situation. IBM Patterns for e-business are further augmented with guidelines and related links for their better use.

The Patterns for e-business layered asset model

The Patterns for e-business approach enables architects to implement successful e-business solutions through the reuse of components and solution elements from proven successful experiences. The Patterns approach is based on a set of layered assets that any existing development methodology can exploit. These layered assets are structured in a way that each level of detail builds on the last. These assets include the following items:

- ► Business patterns that identify the interaction between users, businesses, and data

- ► Integration patterns that tie multiple Business patterns together when a solution cannot be provided based on a single Business pattern

- ► Composite patterns that represent commonly occurring combinations of Business patterns and Integration patterns
- ► Application patterns that provide a conceptual layout describing how the application components and data within a Business pattern or Integration pattern interact
- ► Runtime patterns that define the logical middleware structure that supports an Application pattern

  Runtime patterns depict the major middleware nodes, their roles, and the interfaces between these nodes.
- ► Product mappings that identify proven and tested software implementations for each Runtime pattern
- ► Best-practice guidelines for design, development, deployment, and management of e-business applications
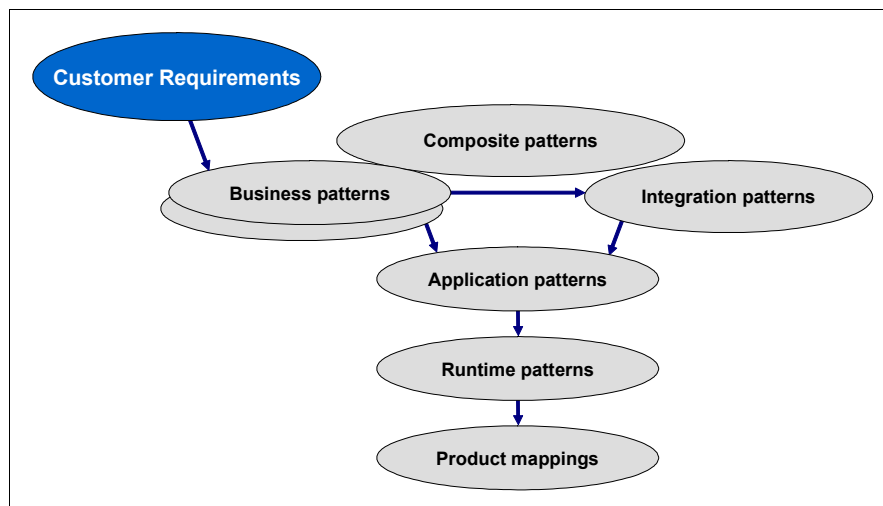
Figure 6-4   The Patterns for e-business layered asset model

Patterns and MDD are key to closing the business and IT gap, and ensuring the delivery of business value. The adoption of Patterns and MDD reduces time-to-react, enables on demand design and development, and reduces complexity. Patterns and MDD are coming of age and are delivering tangible results.

Thomas Murphy of META Group (now part of Gartner Group) is widely quoted as saying that "Organizations using model-driven, pattern-based development frameworks and tools can attain dramatic productivity and quality improvements across the development team."

MDD promotes improvements in business agility, which is a real key to success in an on demand world. The successes of pattern-driven development, IBM Patterns for e-business, and MDD are a result of the complementary nature of these techniques.

MDD works by automating the creation of common aspects of software artifacts and a key aspect of MDD is expertise capture. Instead of needing experts to be on hand every time a best practice decision needs to be made, we capture their expertise in automated patterns and transformations so that it can be reapplied.

IBM Patterns for e-business are layered, reusable, integrated and proven patterns, providing quality input to the MDD method. MDD augments the reusability of the Patterns for e-business by automating them so that they can be reapplied easily. Patterns for e-business provide key content when creating enterprise-wide MDD frameworks that are particularly important for capturing and enforcing a common architectural style across all the IT systems in an enterprise.

In our scenario and example, we use the Patterns for e-business with focus on the Application and Runtime patterns. The Patterns for e-business Application and Runtime patterns were extended for SOA and the ESB patterns are derived from these extensions. Patterns for e-business greatly influence and mold the architectural style adopted.

6.6 Business-driven development

Business-driven development (BDD) is concerned with bridging the business-IT gap to enable business to drive IT more directly. Figure 6-5 shows the business-driven development life cycle, which includes the business, development, and operations parts of the business.

When using BDD, a key part of the life cycle is concerned with business modeling. This is carried out using a tool that is appropriate for business users such as WebSphere Business Integration Modeler (WBI Modeler). WBI Modeler allows business users to capture and simulate their business processes at a business level.

True business modeling of business processes is not concerned with how those processes are implemented in software. The models produced by business modeling activities are business models rather than software models. However, they contain information that provides a valuable specification for those parts of the business that are implemented with or supported by software. An automated transformation can be used to generate a partial software model from a business process model.
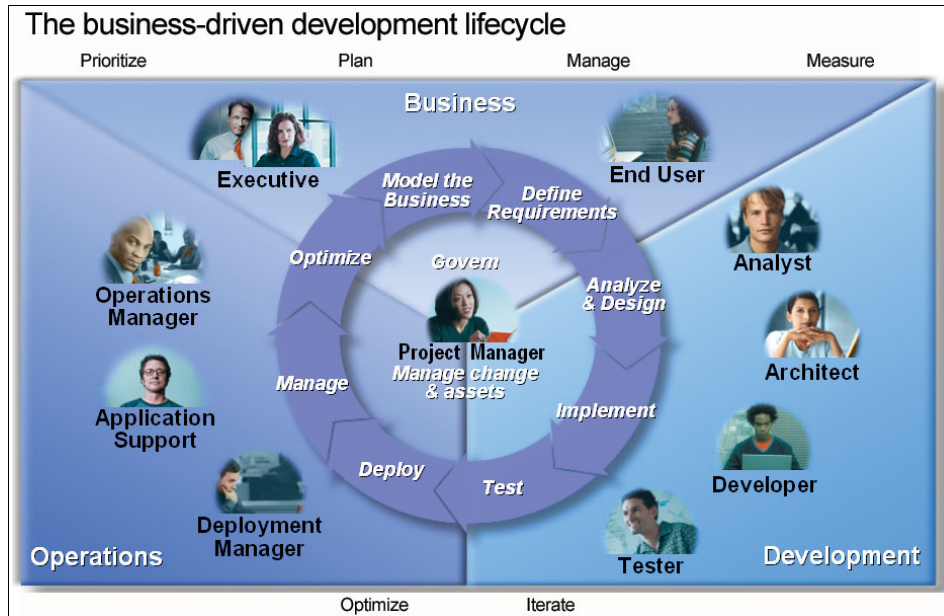
Figure 6-5   The business-driven development life cycle

Rational Software Architect can import WBI Modeler business process models to provide a specification of the software to be implemented. The UML models that RSA creates from WBI Modeler models contain use cases and collaborations that specify the behavior of the system. The models do not make implementation decisions about how the business processes are realized, for example, whether a centralized choreography approach or a distributed object-oriented approach are taken.

In some cases, it is possible to go a step further than generating specifications because the architectural style for application development is known. In such cases, we can implement transformations that generate partial application models from business process models. In this book, we focus on using automated transformations to move from application models to implementations. However, the same techniques can be used to move from business models to application models.

In some cases, if the target implementation platform is known early then it may even be possible to move from business process models directly to skeleton implementation artifacts. For example, WBI Modeler can export processes using the Business Process Execution Language for Web Services, an executable language supported by WebSphere Process Choreographer.
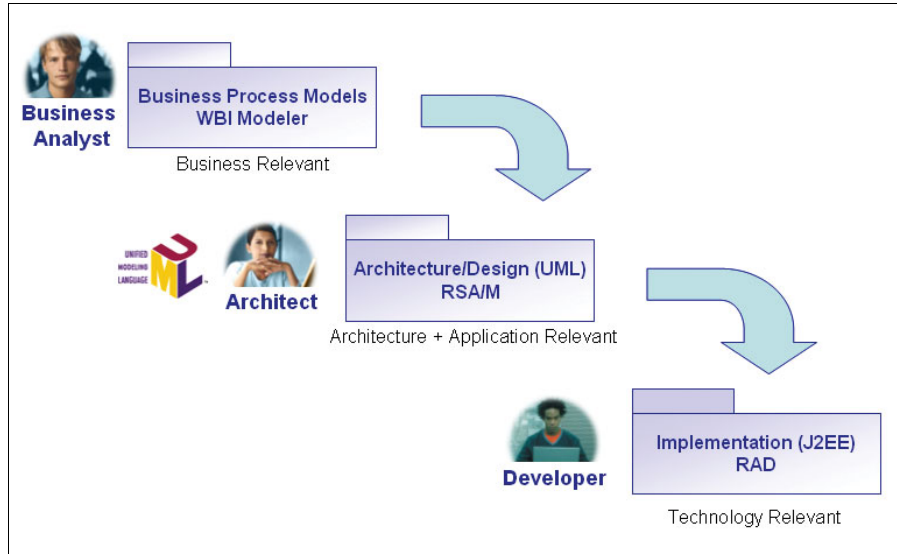
Figure 6-6   Business-driven development with MDD

In summary, business-driven development relies on the techniques of model-driven development to connect business models to software models. MDD can be used within a full BDD life cycle or it can be applied to just the software part of the life cycle. Adopting an MDD approach to software development puts an organization on the right track for a full BDD life cycle in the future.

6.7 Model-driven development and On Demand Business

IBM put a name to a new breed of business: on demand. IBM CEO Sam Palmisano defines On Demand Business as, "An enterprise whose business processes, integrated end-to-end across the company and with key partners, suppliers, and clients, can respond with flexibility and speed to any client demand, market opportunity or threat."

An On Demand Business understands what it does today and is able to rapidly turn business decisions into operational reality.

Model-driven development, especially when taken to the level of business-driven development is a strong facilitator of On Demand Business.

- ► When using an MDD approach, the time taken to add new business function is reduced since much of the implementation work is automated through existing transformations.

- ► Design-level models are always available for MDD systems so current system behavior is understood. Understanding what a system currently does is crucial to being able to make rapid changes. This is often a problem for non-MDD systems where the current function of a system is often poorly understood or dependent on the knowledge of key individuals.

- ► An MDD project can readily take advantage of new technology innovations. Transformations are updated and reapplied with much less effort than that required to reimplement a system on a new platform. An MDD project can rapidly take advantage of new technology platforms.

- ► When linking business models to software models with BDD, the impact of changes at a business level are rapidly understood. In some cases a change to a business model can lead to an automated change to a software model.

In summary, MDD enables an On Demand Business by increasing the availability of *knowledge* about the business, by *integrating* that knowledge, by connecting business and software models, and by reducing the cost of change through *automation*.

6.8 Model-driven development and middleware

Many of the goals of model-driven development and middleware coincide. They both aim to do the following things:

- ► Raise the level of abstraction
- ► Reduce the amount of low-level coding required by application development
- ► Improve consistency across applications

However, middleware, by its nature, is broadly applicable. It is not possible for a general-purpose middleware platform to offer specialized constructs to all of its user communities. MDD provides a less costly alternative to heavily specialized middleware.

6.9 Visualization

Visualization is the technique of viewing implementation artifacts as UML models. Such models are platform-specific, but they hide some of the implementation detail that can detract from the overall architecture of an application.

RSA support visualization of Java and J2EE artifacts. This mode of working is a way of getting some of the benefits of modeling while using a code-centric approach. When using visualization, code is the primary artifact and models are generated from code. Models are not persisted (although their layout information is), they are simply visualizations of the code. Figure 6-7 shows a simple J2EE project and its corresponding UML visualization.
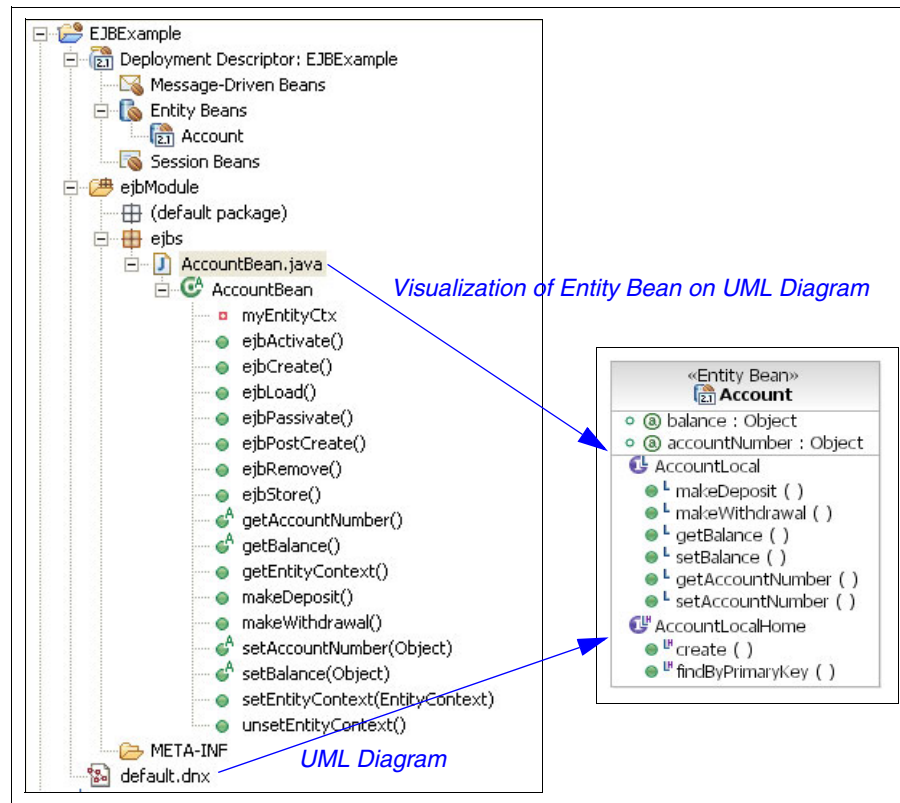
Figure 6-7   Visualization of an Entity Bean

The model is generated from the code, not the other way around. Visualization made easier when a model (often in-memory) of the visualized artifacts is

available so that there is no need to parse a textual representation of the artifacts. This is an example of using non-UML models within an MDD tool chain.

Visualization provides some of the benefits of a model-driven approach:

▶ Some implementation detail can be hidden to allow a design-level view of a system.

▶ UML models and code are kept synchronized avoiding mismatches between documentation and implementation.

Although visualization can be valuable, it can only raise the level of abstraction in a very limited way. When using visualization, we are restricted to hiding details of the concepts supported by the platform.

There are two main ways in which visualization is used in model-driven development.

First, visualization is a step towards model-driven development. In addition to having direct benefits of its own, visualization improves the familiarity of developers with UML modeling. This familiarity can ease the transition to an MDD approach.

Second, visualization is a useful technique for working with platform-specific UML models. Instead of maintaining an intermediate implementation model between the application models and the implementation artifacts, we can generate code directly but still have the benefit of visualizing the implementation model.

6.10 Executable UML

The term *Executable UML* does not refer to the idea of UML having execution semantics. UML has execution semantics, although they could certainly be strengthened and formalized in some areas, and UML 2.0 improved these semantics significantly.

Instead, Executable UML is normally associated with treating UML as a complete programming language, not just for the expression of high-level semantics, but for expressing complete executable models. Such models are executed either on a UML virtual machine or through compilation to a lower-level programming language.

The alternative to Executable UML, and the current common practice in MDD, is to switch to a lower-level programming language to fill in the details.

UML does provide support for expressing detailed semantics through its Actions and Activities. However, the only notation provided for these constructs is a visual notation. It is generally agreed that a textual language is more efficient for capturing detailed semantics, such as mathematical algorithms and text processing.

A number of Executable UML languages, including textual notations for actions, were defined and there is currently an Executable UML standard under development at the OMG. See the book *Executable UML: A Foundation for Model-Driven Architecture* by Stephen J. Mellor and Marc J. Balcer. Also see the book *Model-Driven Architecture with Executable UML* by Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. You can find the complete citations in "Related publications" on page 227.

6.11 Summary

The Object Management Group's Model-Driven Architecture initiative is working to formalize the ideas of model-driven development, and to provide standards in support of the approach. We use the term MDD to refer to a development approach that is broadly in line with the formal MDA that is under development at the OMG.

MDD also has much in common with other initiatives such as Microsoft's Software Factories and domain-specific languages. Although there are significant differences between these approaches, such as the importance of standards to MDD, there is also much in common between the approaches.

MDD is not an end in itself. It is only of value if it can help an organization to meet its business objectives. In particular, you can apply the techniques of MDD to business-driven development to support the goals of an on demand enterprise.

Finally, MDD does not exist in isolation it has much common ground and complementarity with other software development paradigms that are gaining acceptance, including pattern driven development and asset-based development.

This concludes Part 1, "Approach" of the book. In Part 2, "Implementation" on page 91, we apply the ideas of MDD to a hypothetical scenario that draws upon the experiences of the team.

Part 2

Implementation

In this part, we apply the model-driven development approach to the scenario described in Chapter 2, "Scenario overview" on page 17.

► The next chapter, "Designing patterns for the scenario", shows the steps taken to analyze the scenario and design the patterns and transformations to implement with Rational Software Architect.

► Chapter 8, "Applying model-driven development with Rational Software Architect" on page 129, describes how to apply patterns and transformations using Rational Software Architect, and then how to create new stereotypes using profiles.

► Chapter 9, "Extending Rational Software Architect" on page 161, shows how to add new patterns and transformations by extending Rational Software Architect with plug-ins.

► Chapter 10, "Conclusion" on page 219, includes a list of factors that the authors believe are critical to the success of introducing model-driven development into an enterprise.

7

Designing patterns for the scenario

This chapter is based on the scenario described in Chapter 2, "Scenario overview" on page 17. It presents an overview of the process used to model enterprise service bus (ESB) services and generate artifacts from these services. The process considers the following items:

► ESB architecture
► Contracts of behavior
► Integration patterns

Following the overview, we describe the steps we take to create the pattern we use to derive a model of the solution, and then flesh out the detail of the model to the point that services are generated.

We review some practical aspects of the implementation and the final section examines some options for presenting the model information to users.

7.1 Relationship to the project plan

If we relate this chapter to the project structure defined in Figure 4-1 on page 46 then it is mainly addressing the model-driven development (MDD) Tooling Project rather than the underlying middleware or the surrounding business application project. In 7.9, "Use of the framework" on page 125, we cover some aspects of the use of the framework, but we do not cover the business application project in any depth.

Sections 7.2, "Overview of pattern design" on page 95, through 7.7, "Detailing the initial model with service patterns" on page 110, are concerned with the first three steps in the MDD tool chain shown in 4.3, "Planning a model-driven development project" on page 51. For this example, they show how the business problems are analyzed, the solution architecture is defined, and the patterns are selected to define the design model.

Section 7.8, "RSA transformation" on page 118, addresses transformations. In this section, the example takes a simpler approach than the one shown in Figure 4-3 on page 51. The tool chain shown in Figure 4-3 on page 51 assumes that there are two transforms: one from the design model to run-time independent components and one from the run-time independent components to run-time dependent components. This is a more general case to consider, but the example used in this redbook assumes that a single run-time environment is defined and moves straight from the design model to run-time components.

7.2 Overview of pattern design

This chapter looks first at the patterns and models which form the framework for the ESB program, which we are using as an example. Figure 7-1 shows the process we use to model the ESB services and generate artifacts.
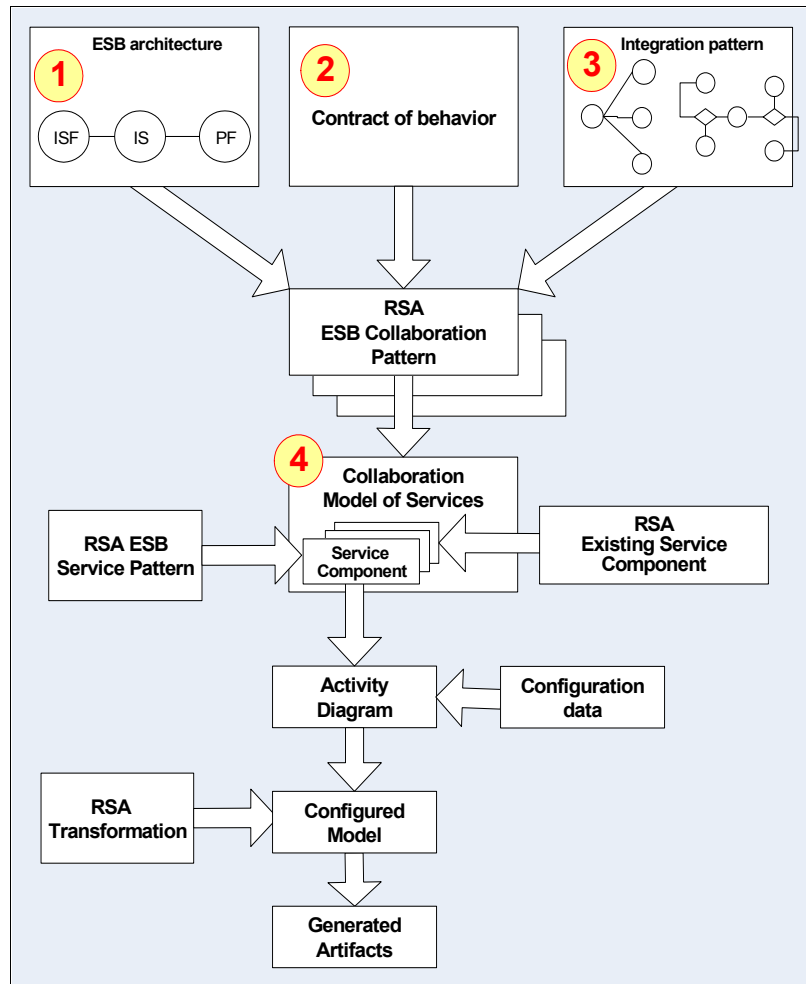
Figure 7-1   Pattern hierarchy

At the highest level in the process, we could create categories of patterns in RSA that define:

► The architecture of the ESB (1) describing its structure: The architecture is described in 7.3, "Architecture patterns" on page 97.

► The contracts of behavior (2) describing how the services interact (for example a query or an update): These are described in 7.4, "Contracts of behavior" on page 99.

► The integration patterns describing how the interactions are performed for each contract of behavior (3): See 7.5, "Integration patterns" on page 104.

In this example, we create Rational Software Architect (RSA) ESB collaboration patterns, which when applied to a collaboration generate an ESB collaboration model (4), see 7.6, "Applying a pattern to create a high-level model" on page 106.

The RSA ESB collaboration patterns are composed of one pattern from each category (1), (2), and (3) in Figure 7-1. Not all combinations of these patterns are valid; therefore, our approach is to implement a pattern in RSA for each valid combination of patterns drawn from the three categories.

Reuse

When an ESB collaboration is modelled, the individual services in the collaboration, which are the facades and the integration services, must be selected or generated. It is a goal of the ESB project that reuse at the service level be pursued as well as reuse at the pattern level. Any supporting tooling should therefore allow a collaboration pattern to be partially (or fully) met by existing services defined in a repository of reusable assets, a RAS repository.

In the case of service reuse, a package containing service components, service interface specifications, or data definitions may be marked with a stereotype to indicate that the implementation artifacts already exist and do not need to be generated by the transformation. A typical example of service reuse is a provider facade created to access a particular external provider function, where this provider facade is then reused in a number of collaborations.

In the case where a new service is to be generated, an ESB service pattern is applied to the service component. The ESB service patterns are selected from a set that reflects the service type and the contract of behavior to be supported. These ESB service patterns attach an activity diagram to the service component that models the behavior of the service at a greater level of detail.

7.3 Architecture patterns

One of the abstract high-level patterns is defined by the architecture of the ESB. This ESB implementation provides all integration functionality as services. We use the term *service-oriented integration* (SOI) to refer to this architectural style.
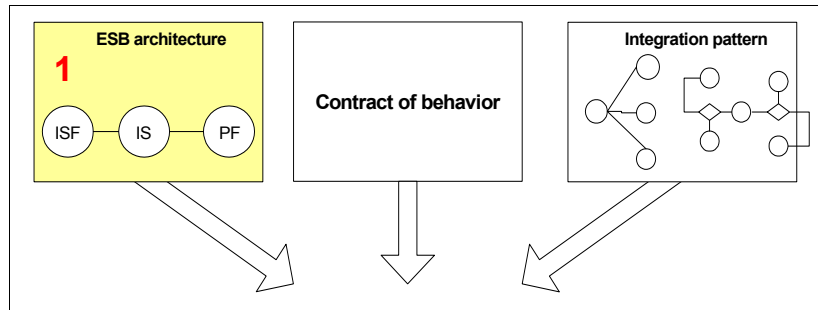
Figure 7-2   Pattern hierarchy: ESB architecture

At a high level, the architecture for this ESB defines the following items:

► Types of service
► Interface restrictions
► Permitted calling patterns
► Architectural constraints

These are imposed by having a set of utilities and rules for when these must be called. These constraints include the following items:

– Validation requirements
– Authorization requirements
– Transformation requirements
– Error reporting
– Gathering of metrics

Service types

The service types permitted in the ESB are:

► Integration service facade
► Requester callback facade
► Integration service
► Integration service call back
► Provider facade
► Provider facade call back
► Internal utility service

Interface restrictions

- ► Integration service facades support a single operation.
- ► Document Web Services interfaces have a literal style.
- ► All external interactions must be fully Web Services Description Language (WSDL) defined.

Permitted calling patterns

This pattern enforces the architectural constraints that all requests and responses between the ESB and external applications are allowed only through the facade services and their call backs. No direct access is allowed to the integration services or the utility services. This ensures that the architectural principles are enforced at the boundary and also that internal integration and utility services can trust their input and do not need to validate. The facades also ensure that all input data are transformed into the Enterprise Canonical Data Format (ECDF) before internal services are called.

Additional constraints of permitted calling patterns are:

- ► Any ESB service may call a utility service.

- ► Integration service facades are requesters of only one integration service.

- ► Asynchronous pattern responses are returned to a callback service rather than to the requester.

- ► Integration services may call provider facades directly or call other integration services, which in turn may call one or more provider facades.

- ► Provider facades may call only one external provider. Multiple calls are made when it is necessary to orchestrate a dialog to retrieve data, but only one function should be addressed with a provider facade.
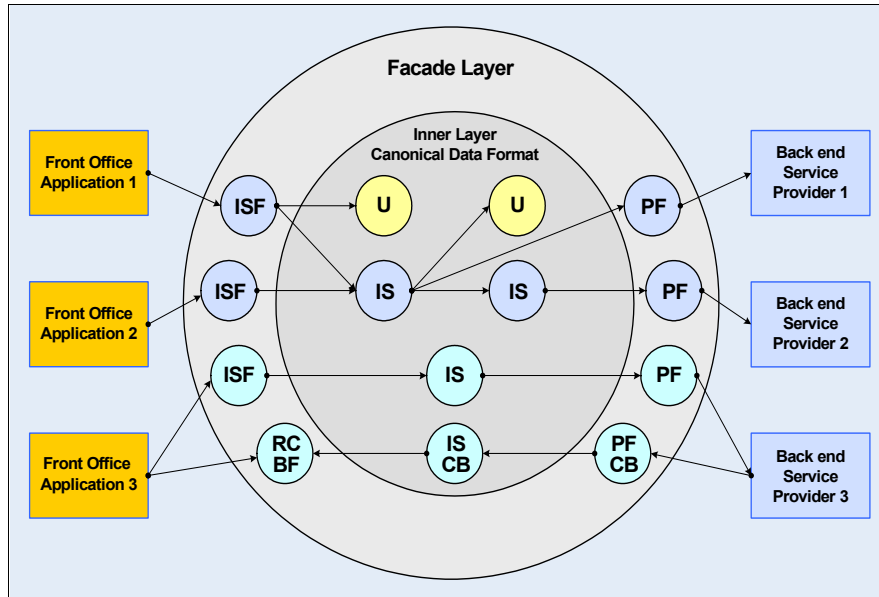
Figure 7-3   ESB architecture pattern

The Architecture pattern illustrates the permitted connectivity and how services can be reused. However, it does not illustrate any specific integration pattern as this is determined by the details of each of the integration patterns.

7.4 Contracts of behavior

We briefly discus the contracts of behavior in Chapter 2, "Scenario overview" on page 17. In this chapter, we look at the contracts of behavior defined for the ESB scenario.
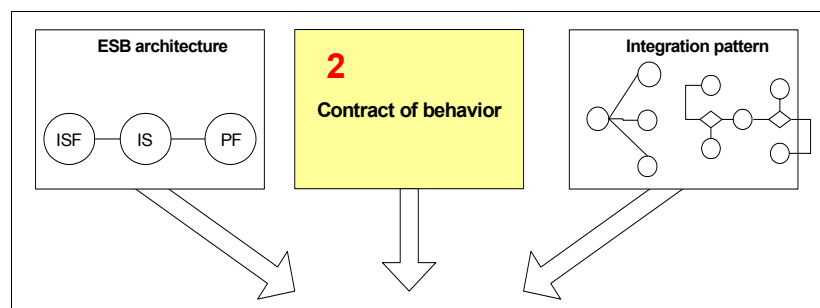
Figure 7-4   Pattern hierarchy: Contract of behavior

Contractual Behavior patterns define interaction styles and qualities of service. We adopted five basic patterns for the ESB in our example.

► Synchronous request for information with requester time out
► Synchronous update with confirmation/error notification and time out
► Asynchronous request for information
► Updates with asynchronous response/error notification
► Asynchronous one way update with managed failure

These patterns are abstract. They define the way the interactions behave in terms of synchronous/asynchronous interfaces, level of assurance of delivery, handling of time outs and late responses, and error handling. They do not define the format of an interface concretely.

These contracts of behavior are not implemented directly as RSA patterns but are combined with the overall pattern derived from the ESB architecture, modified for particular integration functions, to create the first level of patterns implemented in RSA.

The example used in this redbook is based on a synchronous request for update and this contract of behavior is therefore described in detail in 7.4.1, "Contract of behavior for synchronous updates" on page 100.

7.4.1 Contract of behavior for synchronous updates

With a synchronous request for update contract of behavior agreed between a requester and provider, the requester expects to receive a response when the update completes. The requester times out if the response is not received within a given time.

This pattern is included because the most common transport for Web services is synchronous HTTP. Although the unreliable nature of this protocol makes it less than ideal for updates, possibly resulting in indeterminate states, it is widely used. The responsibility for recovery from an indeterminate state is assigned in the stated requirements and is assisted by relevant logging.

Usage
This pattern is suitable for any updates where failure is detected and remedied by the requesting application. The pattern assumes that the requesting client takes control in the case of all types of business related failure by invoking the necessary business processes to ensure that business data is maintained in a consistent state. In this context, business related failure means any failure that has a business impact. The fact that an update fails has a business impact even if the failure is due to a technical error. The business processes may not be totally automated and may involve business and IT support calls.

The requesting process only waits for a specified interval for its response, and on timing out the request must generate a call (possibly asynchronous) to a process that determines the result and takes appropriate action. This support process is assisted by the logging of events as the request is processed.

7.4.2 General requirements for synchronous update

This synchronous update pattern is particularly appropriate for requests from applications that are supporting interactive users.

If we look at the end-to-end pattern, we can see that it consists of the following three pieces:

► Initial requester: This is the requesting application.
► Intermediaries: These are the ESB services.
► Final provider: This is the providing application or applications.

The requirements placed on each of these are defined in the following sections.

Requirements on requester

If you are a requester in this end-to-end pattern, then the following requirements apply:

► Submit a request.

► Receive a response, or failure response, if returned within a given time.

– Retrieve by correlation ID if request uses a messaging transport.

► Abort the retrieval of response (that is, time out) if no response is received in a predetermined interval.

– Invoke business recovery process if response indicates errors.
– Invoke business recovery process if time out occurs before response received.
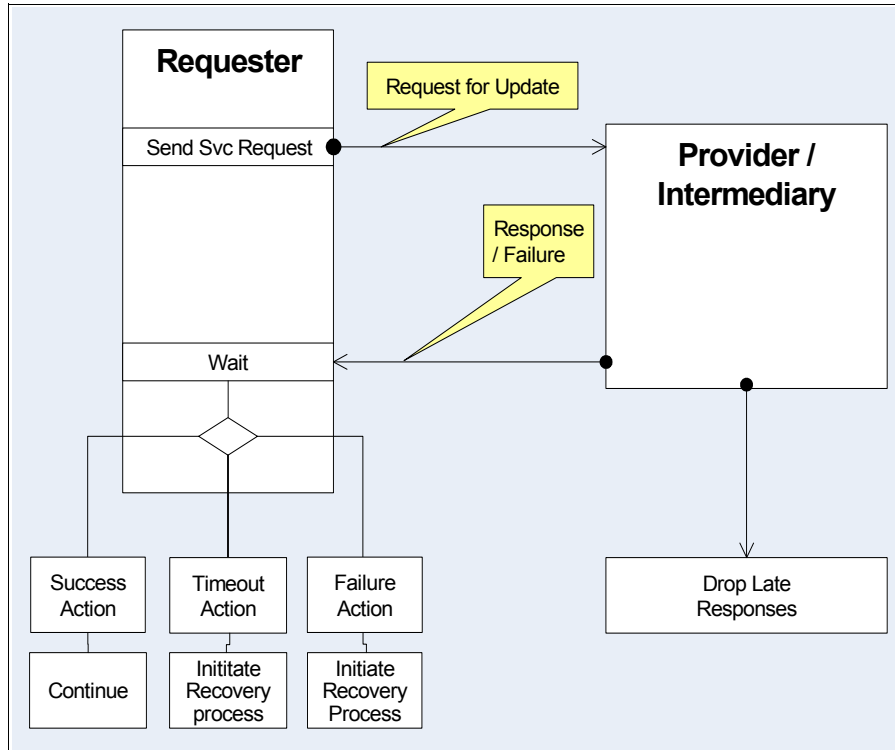
Figure 7-5   Requester behavior for synchronous requests for update

The only true requesters in the scenario are the front office applications. The contract of behavior makes these applications, rather than the ESB, responsible for recovery after failed or timed-out updates.

Requirements on intermediary

► Return immediate error if request is not validated or authorized.
► Return an error for any error detected in processing the request.
► Log errors to ESB utility service in accordance with configuration.
► Submit Request to next Intermediary/Provider.
► Receive response, or failure response, if returned within a given time.

– Retrieve by correlation ID if request uses a messaging transport.
– Log result ESB utility service in accordance with configuration.

► Return response, success or failure to requester (if possible).
► Abort retrieval of response (for example, time out) if no response is received in a predetermined interval.

– Log result ESB utility service in accordance with configuration.

▶ Drop any late responses. Messages must not accumulate because requester times out.

   – Maintain correlation information if request uses messaging transport.
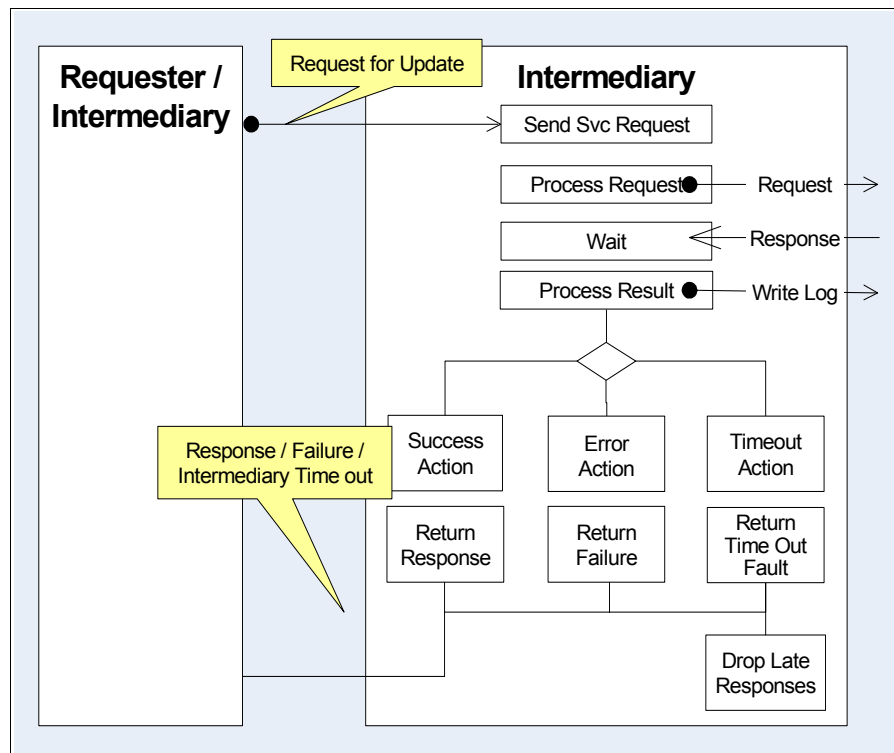
Figure 7-6 Synchronous update behavior of intermediary

All ESB services act as intermediaries. They are neither an initial requester nor a final provider.

Requirements on provider

▶ Return immediate error if request is not validated or authorized.
▶ Process request.
▶ Return result.
▶ Optionally log result according to configuration.

Figure 7-7 shows the behavior between an ESB provider facade service and the back-end provider service.
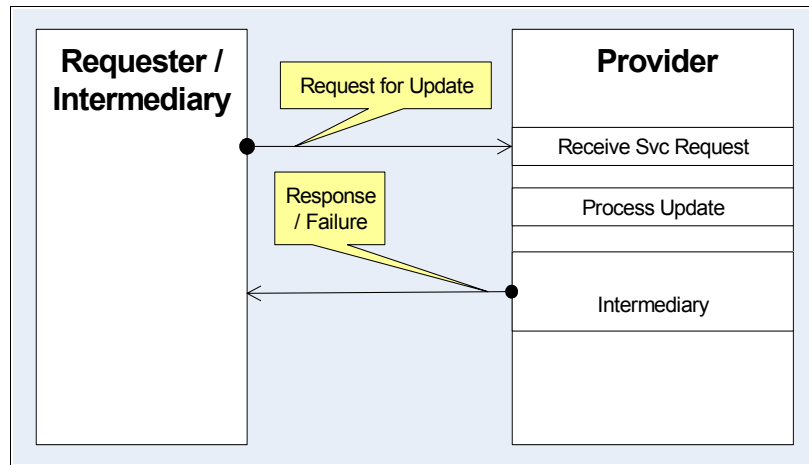
Figure 7-7   Synchronous update: Behavior of provider

7.5 Integration patterns

The basic ESB pattern is modified by the type of orchestration carried out by the integration service. We refer to these as the *integration patterns*.
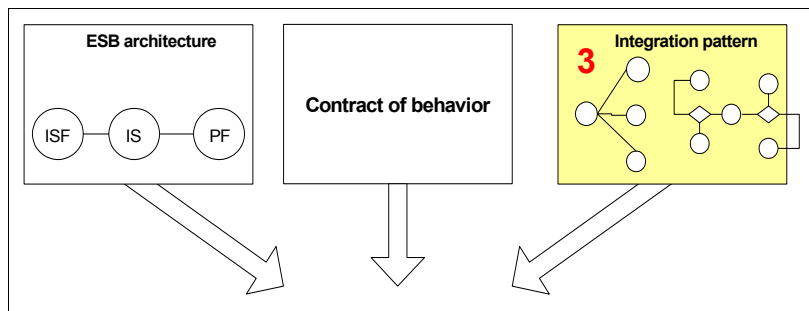
Figure 7-8   Pattern hierarchy: Integration patterns

It is not necessarily possible to provide, initially, a comprehensive set of patterns that cover all the behavior of integration services. But following the familiar 80/20 practice, it is desirable to define a set of patterns to which additions can be made as new requirements emerge and not spend too great an effort aiming for completeness at a particular point in time.

The patterns at this level are aligned with the application integration Patterns for e-business. The first set of integration patterns for the ESB in this scenario is:

► Request for information from a single target

  Optionally routed to alternate service providers

► Update to a single target

  Optionally routed to alternate service providers or to specified files

► Requests for information from multiple targets

  Aggregate responses to give a single result

► Distribution of updates to multiple targets

  Based on message content

► Sequence of calls to multiple integration services or providers (via provider facade)

  – Each step determined by the result of the previous call
  – The final response accumulated from any responses

► Publication and management of subscriptions

► Processing of files

  – Processing of each record
  – Dispatch of validated and transformed records to targets based on content based routing

► Merging and splitting of records from various sources

7.6 Applying a pattern to create a high-level model

The fourth step in the process of applying patterns, shown in Figure 7-1, is to combine the collaboration patterns to produce RSA collaboration models of the services required. See Figure 7-9.
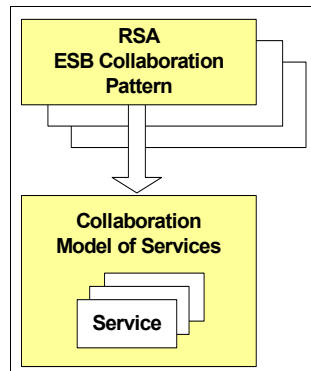
Figure 7-9   Applying a pattern to create a high-level model

The integration scenario is of an external requester updating a customer record hosted by an external provider. The external requester is a front office package supporting call handlers. The front office package makes its request to the external provider indirectly, by requesting an ESB service. The ESB service is responsible for passing the request to update a customer record to the external provider. The external provider is responsible for updating the current address on the master record for a specified customer.

Master records for customers are distributed over multiple back-end systems, and the integration service must route the request to the correct system depending on the content of the message. In a real situation, the distribution of master records might be the result of regional systems or mergers and takeovers and require complex mediation to resolve. To simplify the example, we assume that the split between master systems is alphabetical, based on the customer's surname.

7.6.1 The pattern

In selecting the RSA ESB high-level patterns, we can make the following assumptions:

► ESB architecture pattern is a given
► Contract of behavior is a synchronous update

  The front office call handlers require a response before they can continue their business process.

► The integration pattern is that of request for information from a single target, optionally routed to alternate service providers

The RSA ESB pattern selected for our example reflects the pattern shown in Figure 7-10 (CB3 = contract of behavior, 3). This pattern shows a combination of a basic pattern of services used within the ESB and the contract of behavior, Synchronous Update.
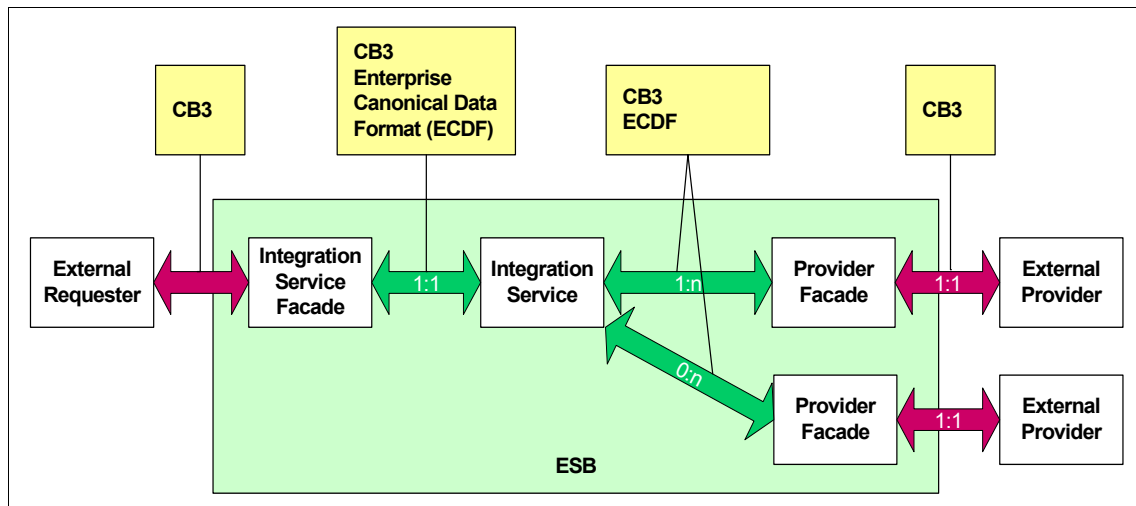
Figure 7-10   Pattern combining the ESB architecture, integration behavior, and contract of behavior

7.6.2 The model

When a selected RSA ESB pattern is applied to a collaboration, it creates a collaboration model of services. The model created uses class diagrams and stereotypes from the UML 2.0 profile for software services. For details, see:

http://www.ibm.com/developerworks/rational/library/05/419_soa/

The pattern selected determines the shape of the model and the way in which it is configured, as shown in Figure 7-11. In this case, the pattern creates components for one integration service facade, one integration service, and a number of alternate provider facades. It captures the items presented in the following sections.
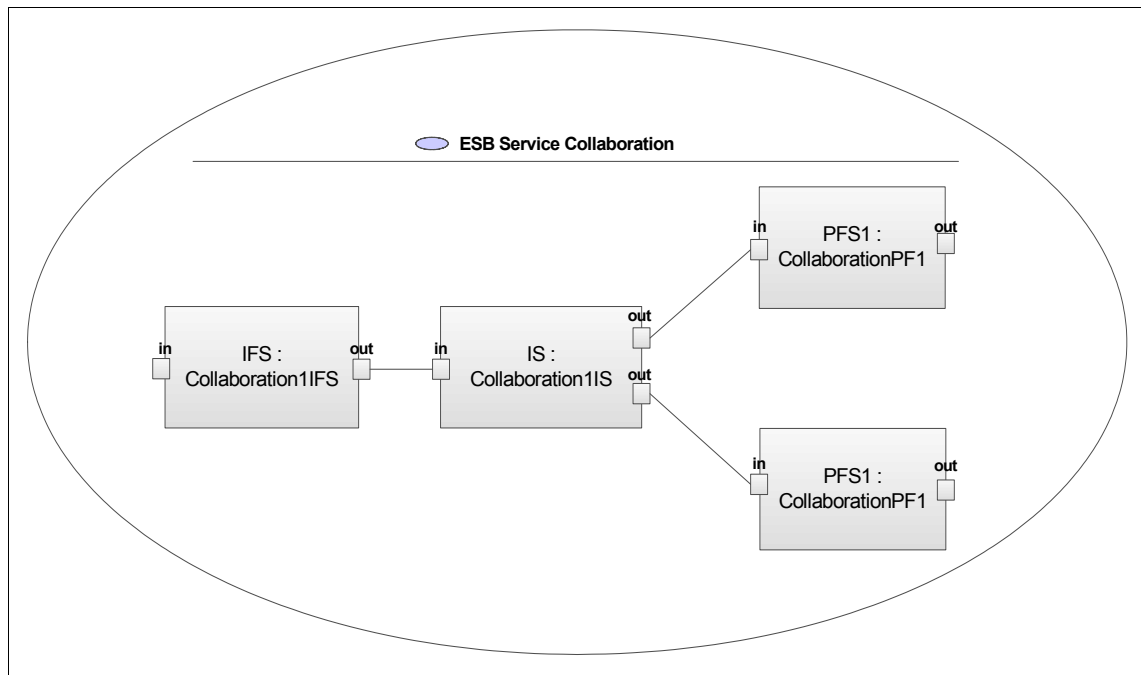
Figure 7-11   Model after application of the ESB pattern

Integration service facade

- ► Service
- ► Service consumer (makes request of one integration service)
- ► Service provider (processes requests from external requester)
- ► One to one (1:1) link to integration service
- ► Constrained to support CB3 as a consumer
- ► Constrained to support CB3 as a provider
- ► Constrained to connect only to requesters or providers supporting CB3

Integration service

► Service
► Service consumer
► Service provider
► Multiple alternative links to provider facades (with same definition of input message schema)

   In the general case, the alternatives might include an integration service.

► Constrained to support CB3 as a consumer
► Constrained to support CB3 as a provider
► Constrained to connect only to requesters or providers supporting CB3

Provider facade

► Service
► Service consumer (makes request of one integration Service)
► Service provider (processes requests from external requester)
► One to one link to external provider
► Constrained to support Contract of behavior 3 as a consumer
► Constrained to support Contract of behavior 3 as a provider
► Constrained to connect only to requesters or providers supporting CB3

External requester

The external requester is included in the model to generate test stubs:

► Service consumer
► One to one link to integration service facade
► No constraints

External provider

The external provider is included in the model to generate test stubs and to define the interface of the external provider:

► Service provider
► One to one link with provider facade
► No constraints

Options for creating the model

The model shown in Figure 7-11 is the starting point for further configuration. The model is part of the tooling, and it could be created in one of two ways:

► Select from a set of models created to match the selected high-level patterns, and copy the selected model.

► Implement an RSA pattern that can be applied to a collaboration to create a model that links the service components in the collaboration.

Creation of a *set of models*, from which an instance is selected and copied, requires less effort where the number of high-level patterns is small and the pattern parameters and their inter-dependencies are low. It is certainly a starting point to establish model requirements before investing in implementation of a pattern.

Creation of *RSA patterns* to generate models matching the high-level collaboration patterns has benefits where there is significant commonality between a set of patterns and where the application and re-application of the pattern can be used to ensure that the resulting model conforms to quite a complex set of conditions.

As defined, the ESB example does not show fully the benefits of using an RSA pattern. However it is a simplified example with a limited set of high-level collaborations to be modeled. In reality, you could use a richer set of options for the ESB, which makes the creation of RSA collaboration patterns more appropriate. For this reason, and to illustrate how this approach works, the example uses an RSA pattern that generates a model of the synchronous update collaboration used in our example.

The pattern, when first applied, creates the components within the model. Subsequent re-application does not recreate components that already exist, but does add new components.

In this example, the patterns ensure that only one high-level pattern is applied, or reapplied to the model.

7.7 Detailing the initial model with service patterns

When a high-level collaboration model is created for an integration, the next step is to look at each of the ESB services in that model and provide the necessary detail to generate or select all the components to complete the end-to-end integration path through the ESB.
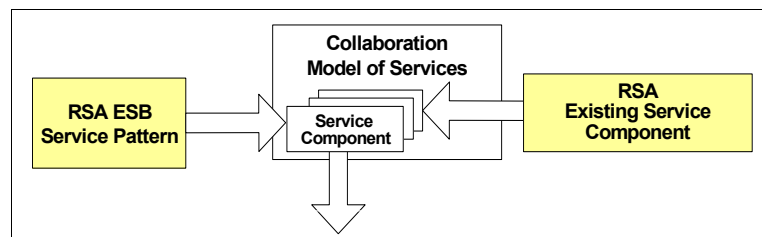
Figure 7-12   Pattern hierarchy: Service patterns

At this point, the model must hold information that is specific to the selected architectural style, that is SOI, and to the detailed architecture of the ESB. The standard UML 2.0 profile for software services does not contain the required stereotypes and a new profile (the SOI Example Profile) is therefore defined.

The SOI Example Profile contains stereotypes to:

► Identify all the possible actions within the activity diagrams for ESB services. Each activity type has its own stereotype.

► Hold attributes defining the behavior of the service, for example, the transform activity has an attribute `transformationID` that defines the transformation to be applied by the transformation service.

► Identify parameters guiding the generation of a service, for example, the `<<Optional>>` stereotype has a boolean "generate" attribute that determines whether this activity is included when a service is generated. The stereotype `<<Generate>>` has a boolean attribute that defines whether this is a service to be generated or whether the service is a reference to an existing service.

► Identify parameters identifying and guiding behavior of the service as a whole, for example, the `<<Integration Service Facade>>` stereotype defines the service version and what level of logging is to be recorded.

Note: The most important information about the service, the definition of its interfaces, is held in the UML 2.0 profile for software services.

Given an ESB collaboration model, any service in the high-level model can be embellished in one of two ways.

If the service component already exists then an `<<external>>` stereotype can be applied to it. This indicates that the service implementation is not to be generated from this set of models. The name of the service component and other information from the model is used if necessary to determine how other services connect to the existing service.

If a service is to be generated from the model then a different approach is taken. The model is further detailed by attaching to the service component an activity diagram. In the example, the pattern to attach activity diagrams is not created, but can be copied from a sample model supplied.

As in the case of the high-level models, the addition of activity diagrams to services in the model can either be carried out by applying a pattern or by manually selecting an activity diagram from a predefined set and attaching a copy to the current model. In this example, we use a simple pattern to select and apply an activity diagram from a predefined set.

7.7.1 Service patterns: Activity diagrams

This section looks at the activity diagrams that model the behavior of the individual services in the collaboration model.
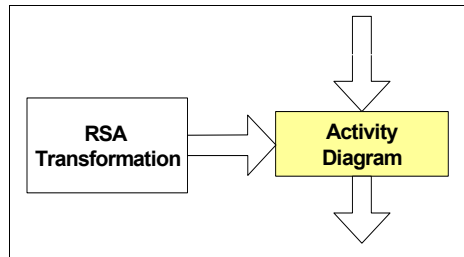
Figure 7-13   Pattern hierarchy: Activity diagram for services

The functions of the facade services (and their call backs where used in the asynchronous patterns) are to:

► Ensure that the architectural constraints are enforced at the boundaries of the ESB

► Handle the communications between the ESB and external applications and between ESB services

There is one basic pattern for each valid combination of service type and contract of behavior to give an overall pattern. These patterns can then have small variations to cope with differences between bindings and outgoing protocols. These patterns incorporate the constraints that limit some combinations of contracts of behavior and service bindings.

There should be a high degree of commonality across all pattern variations within each service type. Take this into account as the patterns and models are developed.

The integration services are each unique and each need independent modeling. Although there may be some commonality, it is the functionality required of the pattern rather than commonality that drives the design.

In many cases, the integration pattern takes a variable number of connections between services and a variable number of steps and creates an RSA pattern that is applied, or reapplied, to create or update the activity diagram. It is a viable alternative to the creation of a predefined set of activity diagrams. Integration services within our example were listed in 7.5, "Integration patterns" on page 104. This chapter looks in detail at only one of these, the synchronous update to a single target, which we use as our example.

Integration service facade

An integration service facade sits between an external requester and an integration service. There are two basic patterns for these facades, the synchronous that handles both request and response and the asynchronous that passes on a request and leaves the processing of responses to a callback service if required.

Our selected example supports a synchronous update and the pattern for the integration service facade is shown in Figure 7-14. The associated activity diagram for this facade is shown in Figure 7-15.
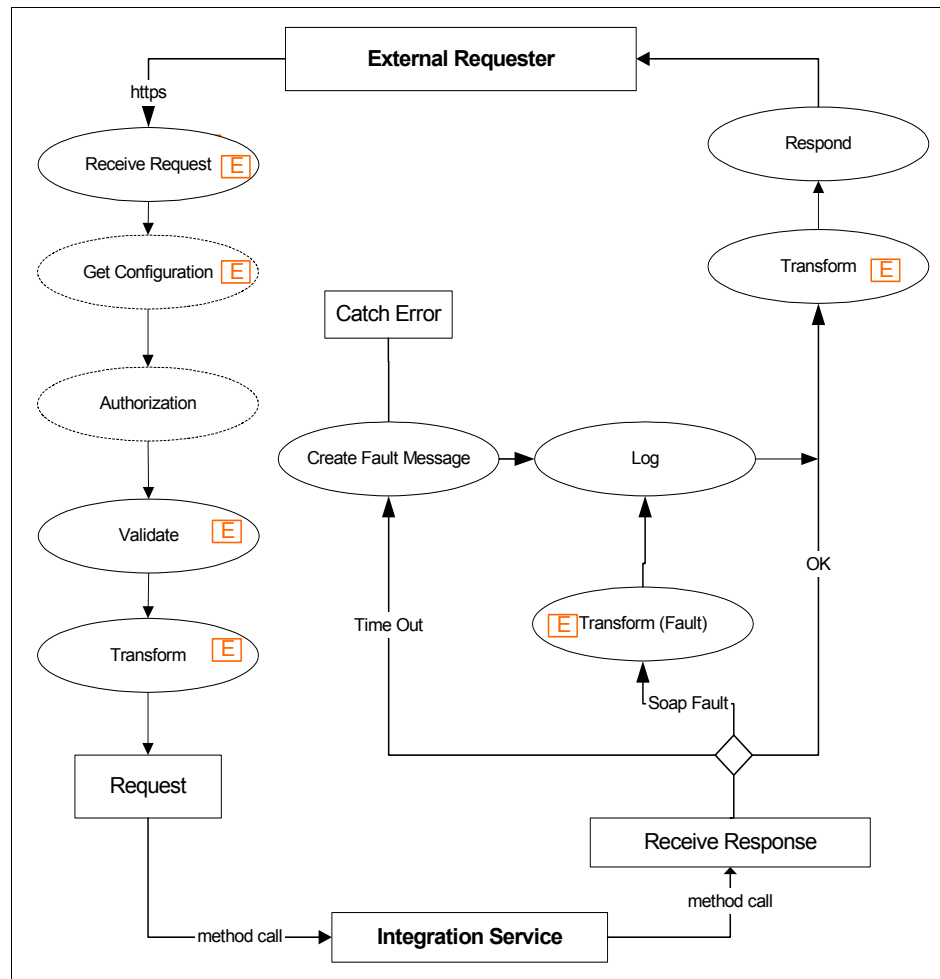
Figure 7-14   Integration service facade pattern
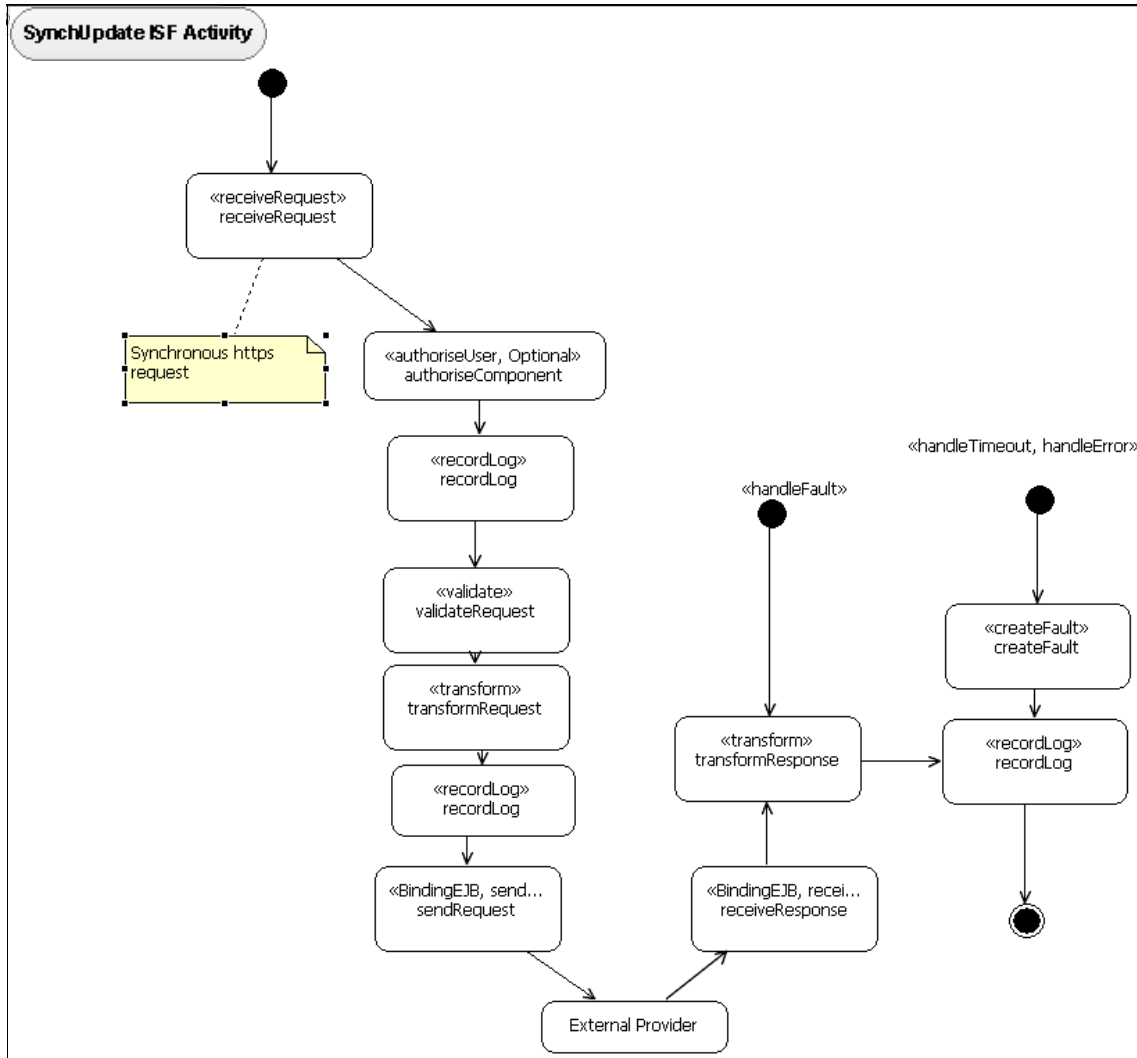
Figure 7-15   Activity diagram model of integration service facade

Provider facade

The provider facade sits between an ESB integration service and an external provider. If the provider service is generated from the model, as is the case for our scenario, then an activity diagram must be attached to the service component in the initial model. The pattern for the provider facade used in our example is shown in Figure 7-16 along with the corresponding activity diagram shown in Figure 7-17.
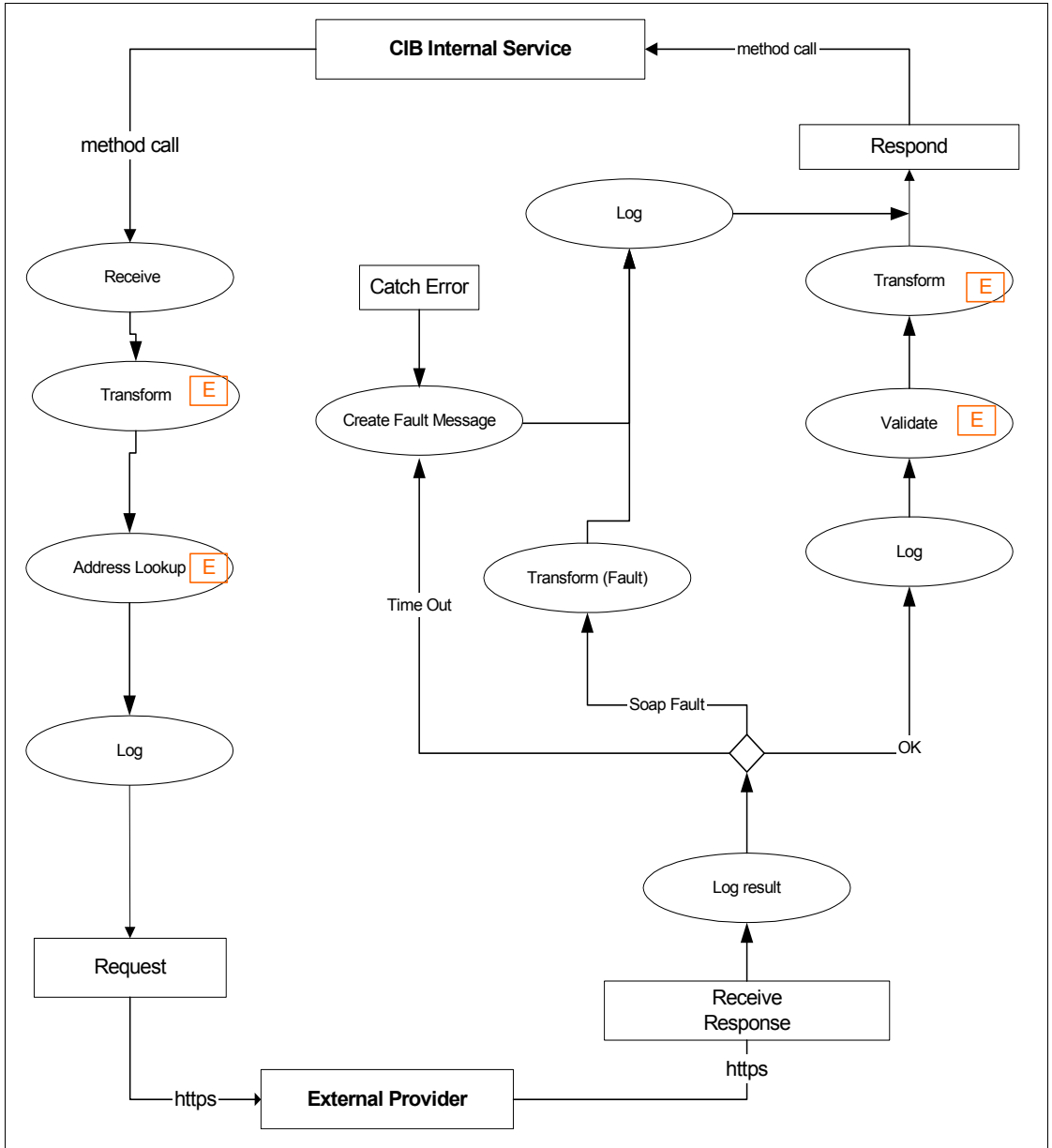
Figure 7-16   Provider facade pattern
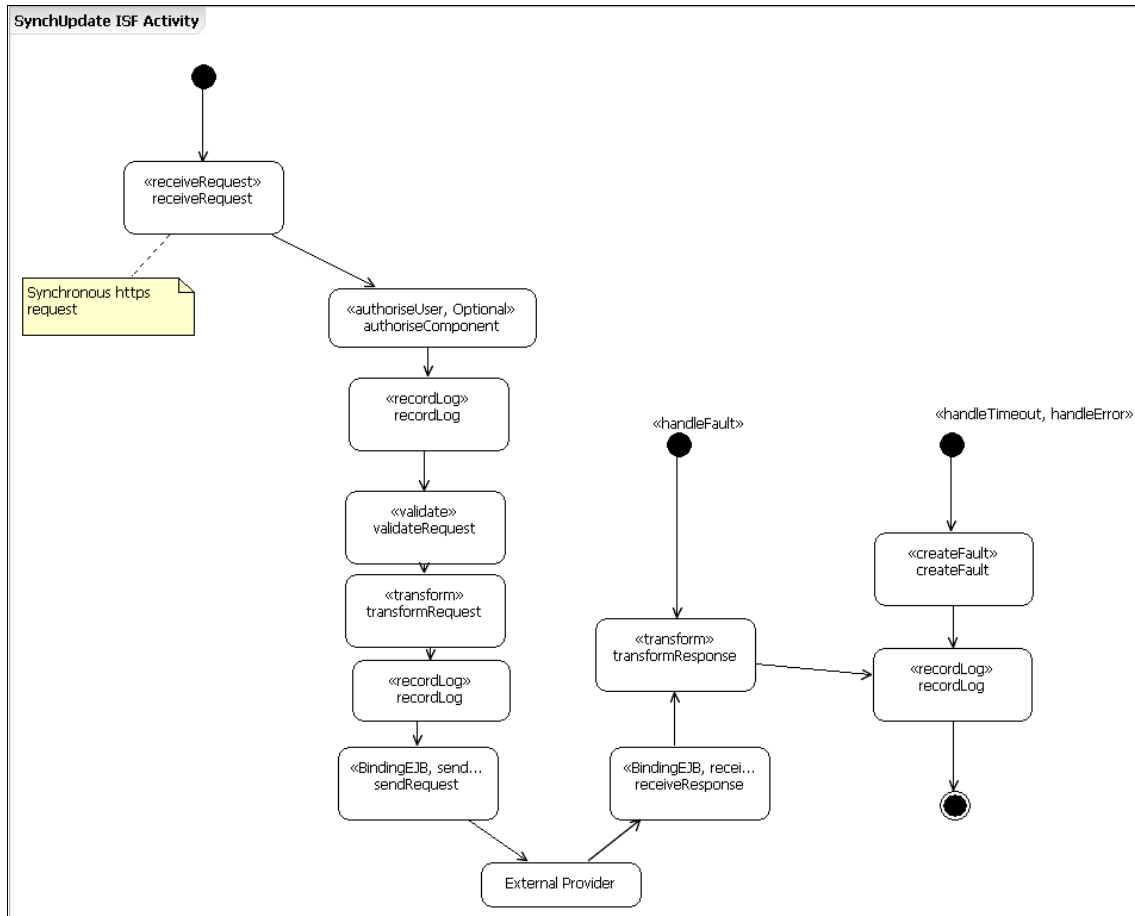
Figure 7-17   Model of provider facade

Extension points

As described so far the patterns and models take account of the contracts of behavior, the ESB architecture, and the integration function to be executed. However, there is one other very significant factor that has to be taken into account for an ESB, and that is the handling of any special requirements in the interfaces of particular external applications.

Some characteristics, for example standard security functions such as encryption or application of digital signature, may be sufficiently general that they are included as optional activities in the standard patterns.

Other features may be specific to a single service, and it is not a good practice to overload the "patterns" with optional one-off items, particularly where this would mean a new release of the tooling to do so.

The solution is to allow for the inclusion of activities without a recognized stereotype, and to provide documentation for the activity to determine the code to be included. This is not included in the example, but it demonstrates how you can use the extension points.

Classes generated through these extensions are constrained to conform to the overall architecture and contract of behavior but can still provide great flexibility to add to the patterns.

Extension points are included in recognition of the fact that, while there is great benefit to be obtained where reusable patterns are abstracted, there remains the need for one-off functions for which conventional bespoke development approaches are far more cost-effective.

7.7.2 Integration services

The integration service used for this example supports an update to a single target, which is determined by a call to a routing service based on the initial letter of the customer's surname.
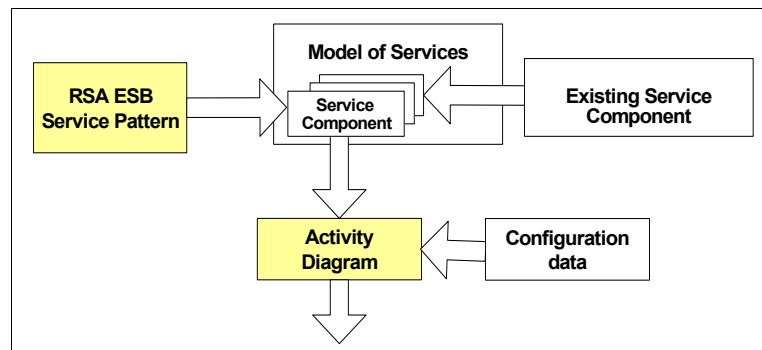
Figure 7-18   Pattern hierarchy: Integration service pattern
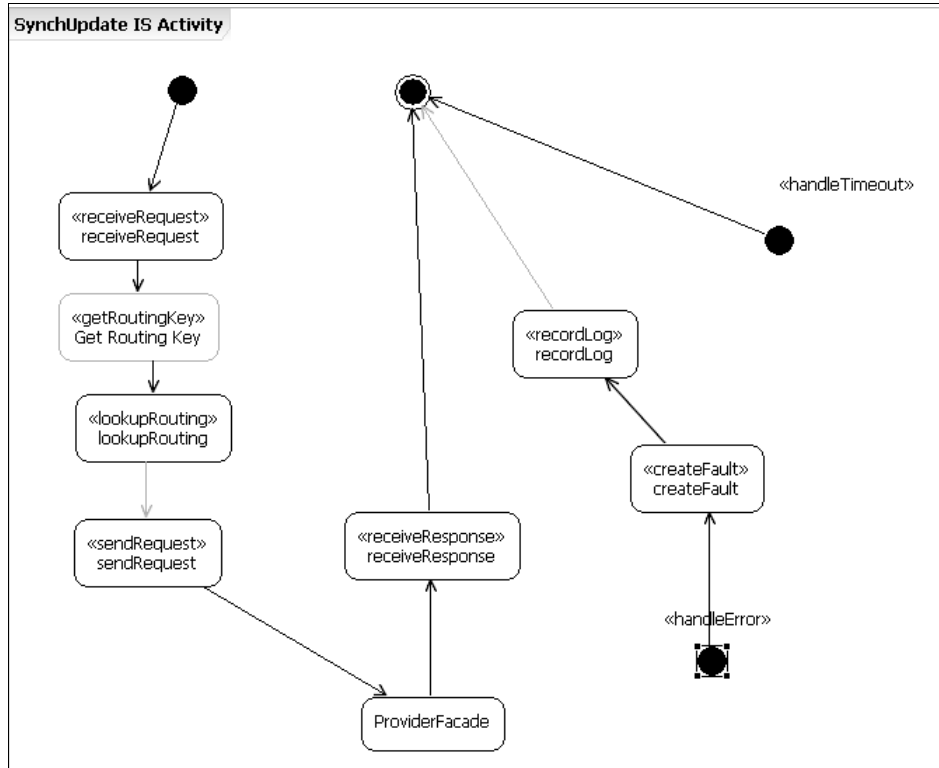
Figure 7-19   Integration service model

7.8 RSA transformation

The final step in creating the MDD framework for this ESB example is to create the transforms that use the information in the service models to generate the code and other artifacts.
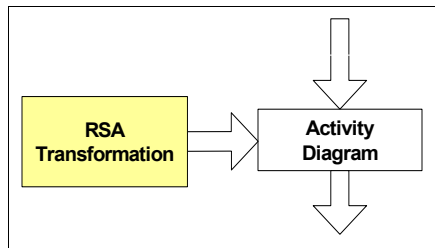
Figure 7-20   Pattern hierarchy: RSA transformation

Before implementing a transform, have a detailed design for the resulting services created from the model. At this stage, before any design is committed to the model, you consult best practices to ensure that the code, which was created from the model, meets the following requirements:

► Well structured
► Standards conform where necessary
► Well instrumented
► Performs well
► Makes best use of the middleware supporting the Web services framework (in this example)

In addition to the detailed design, develop and test a sample implementation to remove problems before a transform is created.

The earlier steps capture and encapsulate domain expertise. In this case, knowledge of suitable integration patterns for an ESB is required.

Creation of the transforms captures and encapsulates the expert knowledge about service implementation. In this integration project, this includes expertise on the middleware capabilities, performance implications, metadata structure, and general skill in Java/J2EE development. Capture this expertise in the design and implementation of the sample, which you test and optimize before the RSA patterns, models, and transformations are created.

In the limited example, which can be developed for this redbook, it is not possible to illustrate all of these aspects. However, the following notes illustrate some points of interest, and in the case of the Provider facade show the design in sufficient detail for the transformation to be developed.

7.8.1 Implementing the integration facade

The integration service facade is itself a Web service. It has a WSDL defined interface and supports a single operation using document style literal and HTTP binding. All clients are expected to send requests conforming to this interface.

Note: The example is limited to an HTTP binding, where additional bindings are supported in practice.

Implementation class
The implementation class is the service name, which reflects both the function of the service and the type of service (in this case ISF for integration service facade).

> **Note:** In a more general context, governance is necessary to ensure that service names are unique.

The method invoked through the standard WebSphere Web services infrastructure is the operation name. No handlers are used in this example but in practice the handlers are used to provide authentication and authorization before the required method on the implementation class.

The method takes a single argument, which is the body of the request, as an encoded string (an alternative would be an XMLDocument type argument). It is a significant performance hit to deserialize the body of the incoming SOAP request into Java objects unless this is needed for the implementation class. However the functions of the facade are those of an intermediary. They only require the body as an XML message to validate and transform before making the request to the next service.

This approach requires a specific external WSDL for the facade that defines the interface fully and is used by clients to generate requirements. It also requires a standard, implicit, internal WSDL with a string (XMLDocument) body for generating the facade. It is in fact the only schema from this WSDL that is required.

> **Note:** The implementation class is not dependent on any classes shared with other services. Each service must exist as an independent entity and be changed without impacting any other services.

However this is a SOI architectural style, and the ESB services use a set of utility services that can be called by all services. These utility services are deployed on all nodes supporting the ESB, thus ensuring that local interfaces can be used and network traffic reduced.

Any calls to EJBs or utility classes where the message body is required are passed as strings (again XMLDocument would be an option).

Validation

Validation is achieved by a call to a utility service with EJB binding. A simple service is implemented for the example. The interface takes the body as an XMLdocument and a relative URL defining the schema against which to validate. The utility service calls Xerces.

> **Note:** We assume for this example that the requests are validated against a single schema and that the schema contains all the information necessary to validate the message.

In practice, a facade must cope with varying styles of WSDL and schema and has to validate the message regardless of WSDL style. The starting point is always the WSDL defining the service interface. However, this may need processing to extract message definitions and to follow nested schema to provide all information to the parser.

Transformation

Transformation is achieved by a call to a transformation service that uses the XALAN transformation engine with an EJB binding. The interface takes two parameters: the body as an XMLDocument and the transform to be applied as a relative URL. The deploy time configuration of the service needs the run time location of all transforms.

For this example, we assume that the transformations are simple and do not require any additional Java code or lookup functions.

Address lookup

The integration service is generated against a WSDL that contains the service address. However, as these services move through their life cycle, they need to be deployed to different environments, for example development for unit test, system test environments, performance test environments. A look up service is therefore used to obtain the address for the current environment. In practice, this is a lookup utility service capable of handling multiple address types. In this example, it is handled as JNDI lookup.

Integration service invocation

All integration services on the ESB are implemented with EJB bindings and take a single parameter, XMLDocument.

Event logging

For some contracts of behavior, it is crucial that particular events in the processing of a request are logged, so that in the event of client time out, before receiving a response, the events can be viewed by application support.

It is also necessary to catch some technical errors and to log these so that they can be monitored by enterprise system management. For example, failure to contact a provider system should be logged so that corrective action is taken.

Logging normally records both business events and technical events. In practice, there are a number of log files active. Technical errors are certainly separated from business errors, and business events are logged according to the service invoked and requester.

In the example, logging is simplified and outputs to a console.

7.8.2 Implementing the integration service

The architectural constraints are enforced by the facades. The implementation of the integration service is constrained only by the calling patterns of the integration facade and the bindings of the provider facades.

7.8.3 Implementing the provider facade

The provider service for our example is developed in full in accordance with the high-level design shown in Figure 7-21.
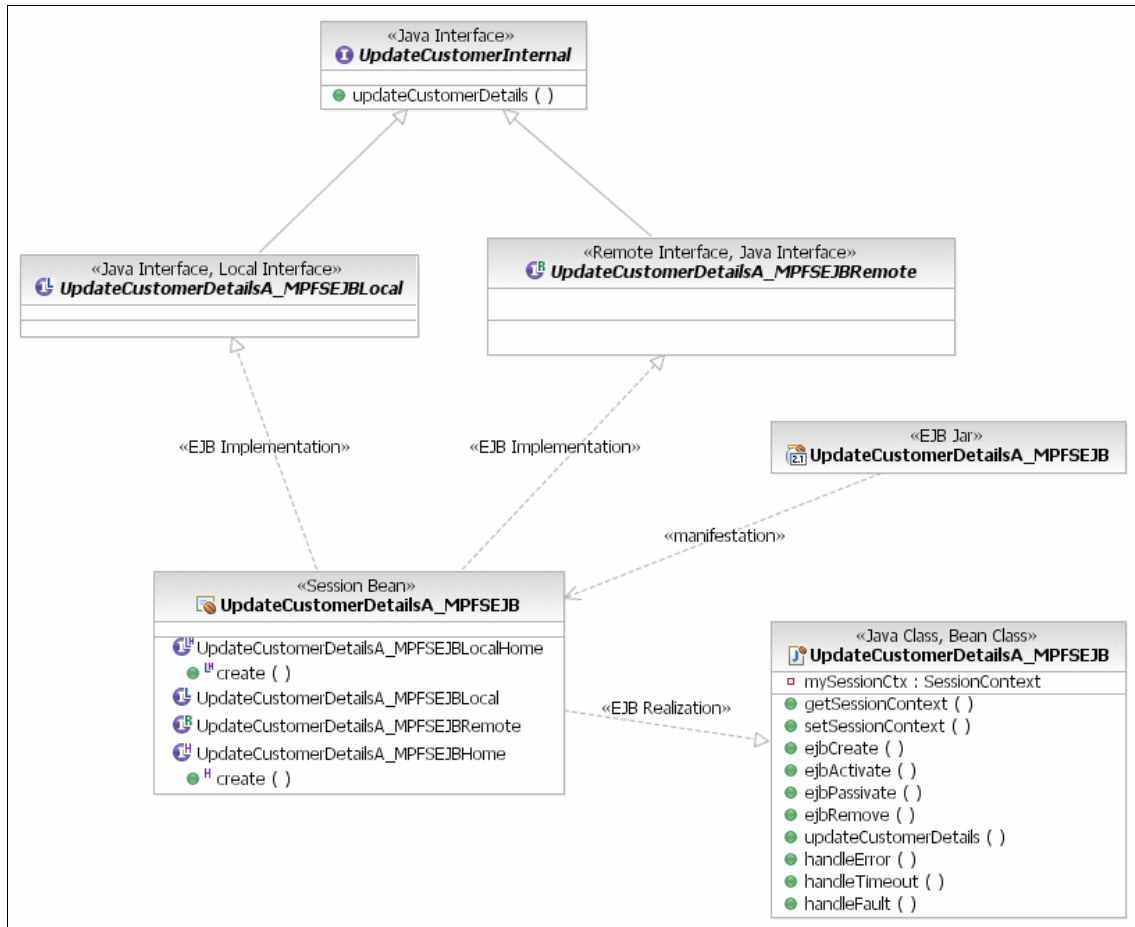
Figure 7-21   High-level design

Web services interface

The Integration Service is either a Web service call over JMS (for asynchronous or one-way requests) or an EJB call. It has a WSDL defined interface and supports a single operation.

Implementation class

The implementation class is the service name, which reflects both the function of the service and the type of service (in this case ISF for integration service facade).

The method takes a single argument, which is the body of the request, as an encoded string (an alternative would be an XMLDocument). It is a significant performance hit to deserialize the body of the incoming SOAP request into Java objects unless this is needed for the implementation class. However, the functions of the facade services are those of an intermediary and only require the body as an XML message to validate and transform before making the request to the next service.

This is a SOI architectural style and the ESB services are dependent on a set of utility services that can be called by all services. These utility services are deployed on all nodes supporting the ESB, thus ensuring that local interfaces are used and network traffic reduced.

Any calls to EJBs or utility classes where the message body is required are passed as strings (again XMLDocument would be an option).

Transformation

Transformation of responses from external providers' formats into ECDF is achieved by a call to a transformation service with an EJB binding. The interface takes two parameters: the body as an XMLDocument and the transform to be applied as a relative URL. As with the integration service facades, these transformations may include Java classes called from XSLT.

Address lookup

The integration service generates against a WSDL that contains the service address. However as these services move through their life cycle they need to be deployed to different environments, for example development for unit test, system test environments, performance test environments. A look up service is therefore used to obtain the address for the current environment. In practice, this is a lookup utility service capable of handling multiple address types. In this example, it is handled as JNDI lookup.

External provider invocation

All external providers are accessed as Web services with a fully defined WSDL interface. XML documents and strings are not accepted when calling outside the CIB.

Event logging

Logging for a provider facade is the same as for the integration service facade. In the example, logging is simplified and writes to a console.

7.9 Use of the framework

This chapter has so far concentrated on the issues facing the developers of the MDD framework for ESB services. These tasks are carried out at the start of a project and require the following significant skills:

► Platform and middleware skills
► Software design
► Domain knowledge
► UML modeling and Rational skills
► Modeling standards

Now we move on in the project life cycle to the point where the framework is in place and service developers are ready to use the framework to support the integration tasks of a project which uses the ESB.
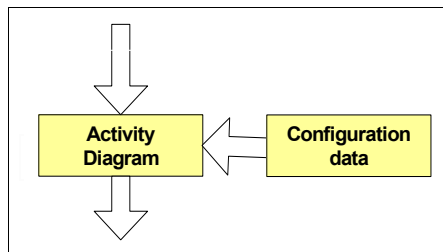
Figure 7-22   Pattern hierarchy: Use of the framework

These service developers understand the following concepts:

► Requirements for application integration
► Functionality of the ESB integration they are trying to create
► Overall system context
► Where data is retrieved from or updated

Then they need to define the following items:

- ► Service interfaces
- ► Transformations between external data formats and ECDF

In short, they need to understand what actions to take across the enterprise space when a business event occurs in the application or applications they are integrating and then define the interfaces to the services that perform the actions.

However it is not necessary for all these users to have the skills required in setting up the framework. There are a number of ways in which the framework can encapsulate models so that service developers need to be aware only of the functional requirements of the service.

7.9.1 Presentation of model information to users

If we return to the scenario defined in Chapter 2, "Scenario overview" on page 17, we have a central group making a decision to adopt a model-driven development approach that raises an immediate question of skills in the divisional IT group. This is a potentially large problem. The issue is not the skills required for the ESB development, as this is seen as a necessary part of the initial investment that is repaid by cost savings over a number of projects.

The real issue is if there is an ongoing requirement, in the divisional projects using the model-driven development framework, for skills in modeling and ESB architecture. These skills are difficult to resource over many simultaneous projects.

The following options are considered:

- ► Introduce modeling skills to all divisional IT groups and make them mandatory for all IT suppliers carrying out integration projects. Interaction is then directly with the modeling tools; although, this is limited to the application of patterns and transformations in strict accordance with ESB guidelines.

- ► Limit the requirement for modeling skills to an understanding of models by allowing users to see the structure of the ESB and representations of the ESB patterns. Interaction in this case is through a controlled user interface, but the models are visible to the users generating services.

- ► Remove all requirements for users generating services to understand the underlying modeling techniques, and control the interactions through a tailored user interface that allows users generating services to operate mainly in the domain of understanding data content of service requests and responses and the business operations to be achieved.

- ► Define the parameters required to generate particular services types and allow the generation of services to be driven without direct user interaction.

This sounds like a retrograde step, editing files to define configuration data, but would in fact only be done in conjunction with other options. You can combine this option with the previous options, which would capture all the configuration data and generate the file to drive the non-interactive generation for repeatable attempts with variations. This step could also be used with existing sources of data to which a transformation is applied to generate the required input.

The current recommendation for the example scenario, included in the model-driven development framework and tooling, is that the developers of integration services are protected from the need to understand the modeling techniques and the ESB implementation but that there is value in allowing them to view the models used. This introduces a degree of understanding of the ESB architecture.

The customized model-driven development tooling for the ESB also provides for a non-interactive method of operation, effectively an API that can be invoked by scripts or by other automated processes that allow for easy repetition of service generation, avoiding the need for "multiple clicks" when a user has no need for the implied guidance of an interactive interface. This leaves divisional projects to apply additional automation, for example automated extraction of service interfaces and required transformations from application data models.

7.9.2 Service creation

If we assume that one of the predefined patterns fits with a particular service to be developed, the steps are as follows:

1. Understand the event triggering a service call:

    – Define the use case

2. Define the data involved in the service:

    – Message schema of requester and providers
    – WSDL definition of service
    – Transformations

3. Select the integration pattern and create the collaboration model by selecting the appropriate RSA pattern.

4. Configure the individual services through the user interface.

5. Generate and test each service.

The model-driven development approach is about automation and it is not only the automation of the service code that you can automate. The model holds all the data necessary to generate a package of artifacts required for service deployment. This includes any deployment descriptors and configuration files

required. Ideally parameterized deployment scripts are created allowing deploy time adjustments of environment specific data. This package must also include any further information needed to deploy the service without manual intervention.

There are times when the predefined patterns and models do not meet the requirements. Business developers then have to work with the framework developers to add or enhance patterns, models, and transformations. A well structured framework facilitates this process.

7.10 Summary

In this chapter, we stepped through how our methodology applies to the synchronous update behavior associated with updating a single target integration pattern within the overall ESB architecture. Our focus is on analyzing the scenario and understanding how the methodology is applied. It was a pencil and paper simulation of what we now want to discuss, which is how to build the automation extensions into Rational Software Architect to implement the patterns that we identified.

But there is a penultimate step that you must take on this journey before finally building the RSA extensions, and that is to understand how to apply model-driven development with RSA.

8

Applying model-driven development with Rational Software Architect

This chapter explains how to carry out model-driven development (MDD) application development and framework development using Rational Software Architect (RSA). We cover the activities that the architects and designers carry out. The coding activities that the programmers perform are covered in Chapter 9, "Extending Rational Software Architect" on page 161.

We cover the following topics:

► The MDD process applied with RSA
► Applying Unified Modeling Language (UML) profiles, patterns, and transformations
► Creating UML profiles
► Designing patterns and transformations

The previous chapter sets out an architectural style for service-oriented integration (SOI). It describes the architectural principles, the patterns the UML profiles, and the technical architecture.

In this chapter, we look at how this architectural style is supported with an RSA-based MDD framework. The MDD framework developed in this book is a

129

sample rather than a complete framework. It does include samples of all the key components that make up an MDD framework and is extendable to provide a complete MDD framework.

8.1 An overview of the Model-driven development process in RSA

There the two distinct types of activity in the MDD process:

► **Expertise Capture and Automation:** This is where we build the MDD framework that partially automates the development of software that follows a particular architectural style.

► **Application Development**: This is where we apply our chosen MDD framework to build software components, applications, and solutions.

These activities are typically performed by different groups of people and require different skills. RSA supports both sets of activities. We use RSA to build UML profiles, patterns, and transformations that are then used to customize RSA to provide an MDD framework.

The activities in the framework development partition of Figure 8-1 are concerned with building the framework while the activities in the application development partition are concerned with applying the framework to build an application.
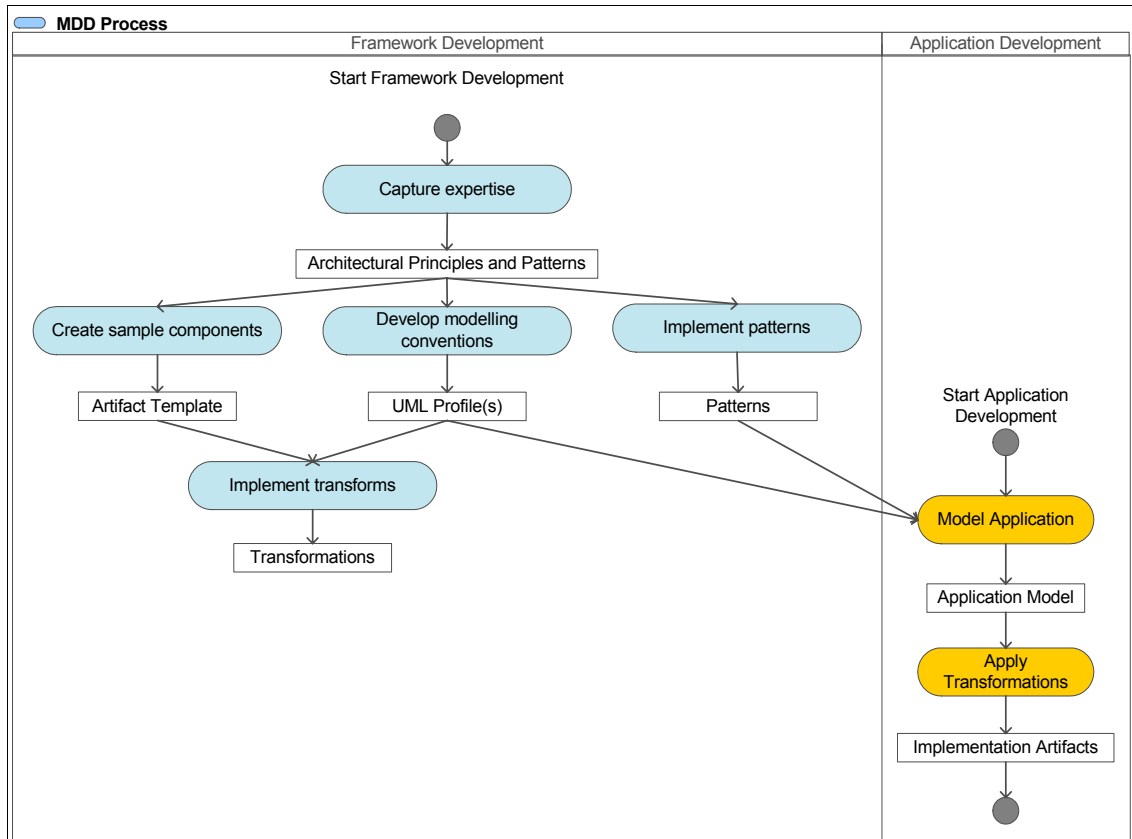
Figure 8-1   The MDD process

There is no magic to MDD. Someone must come up with a set of modeling conventions that are suitable for the software under development. Someone must also develop transformations that can automate the generation of code from models that follow these conventions.

The key dependencies between the two streams of activity are as follows:

► UML profiles and patterns must be available when application modeling begins. In some cases, this dependency is managed in an iterative manner with profiles and patterns that address some aspects of design being made available before others.

► Transformations must be available in order to generate implementation artifacts. In some projects the target platform and the transformations are selected at the start of the project. In others, this decision is deferred.

8.1.1 Framework development

MDD framework development is concerned with designing and automating an architectural style. Framework development is an architecture-led activity that requires input from domain experts, which includes the following actions:

► Capturing expertise in the form of architectural principles and patterns

► Implementing sample components and defining the technical architecture

  Chapter 7, "Designing patterns for the scenario" on page 93, develops an SOI architectural style for our sample scenario. This chapter also shows how to use RSA to apply the MDD approach to the sample scenario.

► Designing and implementing UML profiles

► Designing and implementing RSA patterns and transformations

Chapter 9, "Extending Rational Software Architect" on page 161, shows how to implement new RSA patterns and transformations.

8.1.2 Application development

MDD application development is concerned with using an MDD framework to rapidly build well architected applications and components.

This includes the following actions:

► Modeling the application using the UML profiles and patterns provided by the framework

► Applying transformations to generate implementation artifacts and other work products

8.2 RSA model-driven development framework for SOI

In this book, we build a sample RSA MDD Framework for SOI. This framework automates key aspects of the approach that we set out in Chapter 7, "Designing patterns for the scenario" on page 93.

The RSA MDD framework for SOI includes the following RSA extensions:

► (Sample) SOI Example Profile
► (Sample) Pattern Library for SOI
► (Sample) SOI Transformation

The MDD framework has the prerequisite of IBM Rational UML profile for software services.

In the following sections, we first explain how to carry out application development using the framework. We then cover the architecture and design activities associated with building the framework.

8.3 Application development

In this section we describe how to implement the update customer details example, which was introduced in Chapter 7, "Designing patterns for the scenario" on page 93, using the RSA MDD framework for SOI.

The description of the example is repeated here for convenience:

> The integration is that of an external requester, a front office package supporting call handlers, requesting an ESB service which is responsible for passing a request to update a customer record to an external provider. The external provider is responsible for updating the current address on the master record for a specified customer. Master records for customers are distributed over multiple back-end systems, and the integration service must route the request to the correct system depending on the content of the message. In a real situation, the such distribution of master records might be the result of regional systems or mergers and take overs. To simplify the example, we assume that the split between master systems is alphabetical, based on the customers surname.

The example is implemented using the MDD framework and following the architectural style that we set out in Chapter 7, "Designing patterns for the scenario" on page 93, using the RSA MDD framework for SOI.

8.3.1 Installing the framework

Before we can begin developing our application, we need to install the MDD framework that we are using.

Installing the UML profile for software services

1. Visit the following Web site to get the *UML profile for software services*, which is available as an RSA update:

   http://www.ibm.com/developerworks/rational/library/05/510_svc/

2. Download the plug-in to your machine and install it using **RSA Help** → **Software Updates** → **Find and Install**.

You are now able to use the software services profile.

Installing the MDD framework for SOI and example

Follow the instructions in Appendix A, "Additional material" on page 221, to install the MDD redbook samples (transformation and pattern library), updateCustomer.zip, and sampleSource.zip. You are now able to use the SOI example profile, the SOI pattern library and the SOI transformation.

8.3.2 Creating a model and apply the profiles

We are now ready to start developing the update customer details example.

Create a new UML Project named `Update Customer Details` using **File → New → Project → Modeling → UML Project**. The default options in the wizard are fine, although you can choose a new name for the model if you want.

> **Note:** The spaces are important, to avoid a conflict with a project with the same name in the sample materials.

Next we apply the UML profile for software services and the SOI example profile to the model.
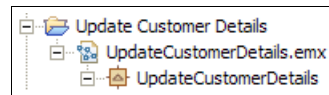
1. Identify the model in the Model Explorer , and select it. This updates the Properties view to show the properties of the model.

2. In the Properties view, choose the **Profile** tab.

3. Click **Add Profile**.

4. The Select Profile dialog box appears (Figure 8-2). Select **UML Profile for Software Services**.
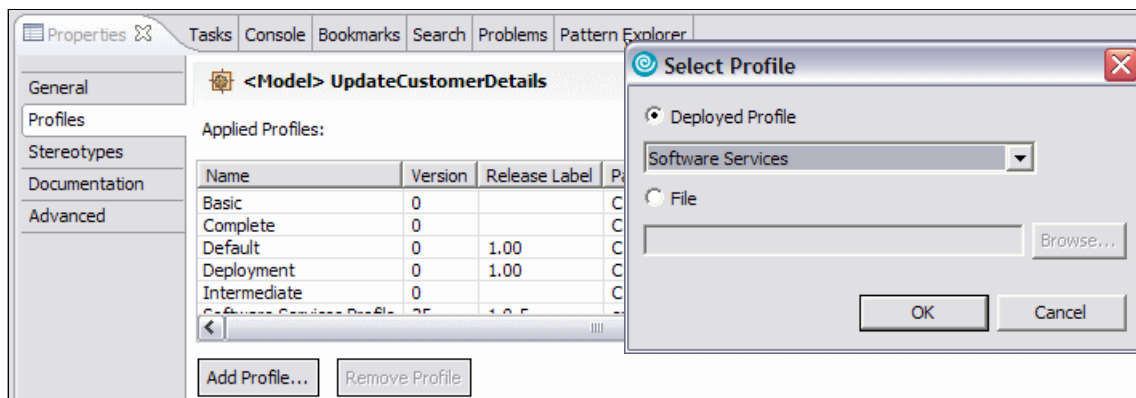
Figure 8-2   Applying the UML Profile for Software Services

The software services profile now appears in the list of profiles applied to this model.

As an alternative to sharing deployed profiles, such as the software services profile, we can also refer to profiles as .epx files.

Tip: When sharing profiles within a team we recommend that you set up a pathmap in RSA. Refer to the article "Comparing and merging UML models in IBM Rational Software Architect, Part 6: Parallel model development with custom profiles" for further information:

http://www.ibm.com/developerworks/rational/library/05/0823_Letkeman/

To add the SOI example profile to your model, find it in the workspace of the UpdateCustomerDetails project you loaded from the sample materials.

1. Return to the Select Profile window (Figure 8-3), and select **File** rather than **Deployed Profile**.
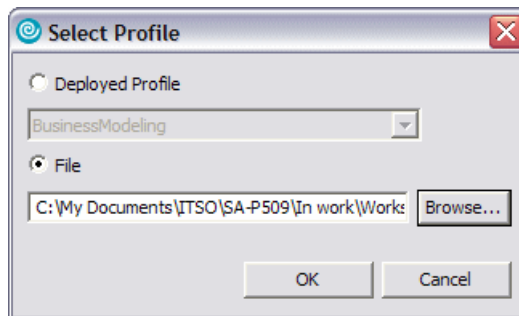
Figure 8-3   Applying the SOI example profile

2. Navigate to the SOI example profile, and select it as in Figure 8-4, to add it to the applied profiles.
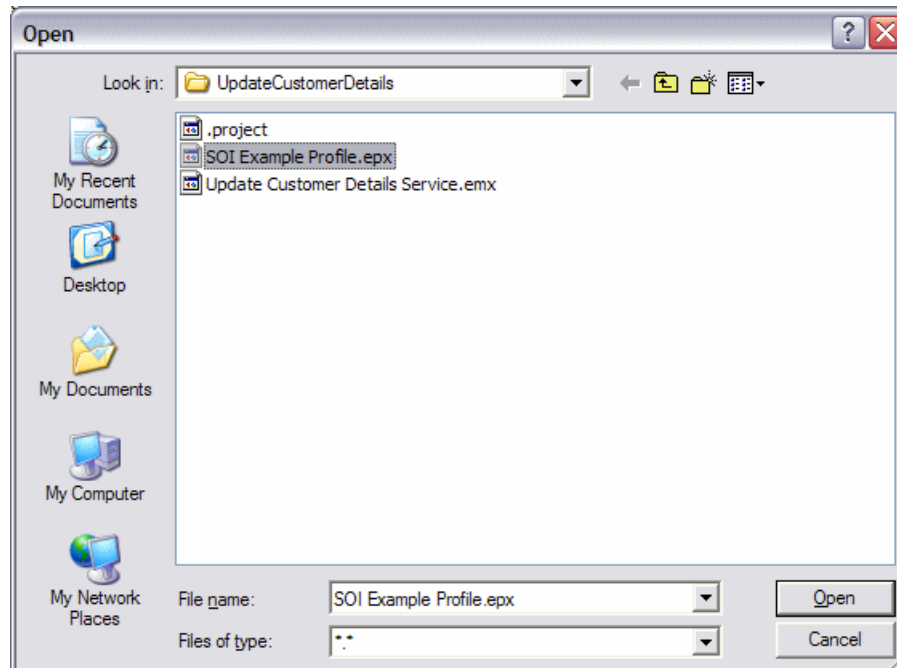
Figure 8-4   Locating the SOI Example Profile

8.3.3 Applying patterns

Now we can model the application using the stereotypes defined in the profiles we applied and the patterns in the SOI pattern library.

Recall, from Chapter 3, "Model-driven development approach" on page 29, Christopher Alexander's description of how a pattern language is used:

> Alexander describes how a pattern language is always used as a sequence going through the patterns, moving always from the larger patterns to the smaller. He describes how a pattern language always moves from the patterns which create structure, to the ones which then embellish those structures, and then to those which embellish the embellishments.

This is the approach we take here, working from larger scale patterns to smaller scale patterns as we make design decisions.

The MDD framework for SOI, which we installed earlier, includes a sample SOI pattern library.

1. To view the SOI Pattern Library choose **Window** → **Show View** → **Pattern Explorer**. The pattern explorer includes the SOI Pattern Library as well as any other pattern libraries you installed (including those included with RSA).

2. Expand the SOI Pattern Library to see the available list of patterns as shown in Figure 8-5.
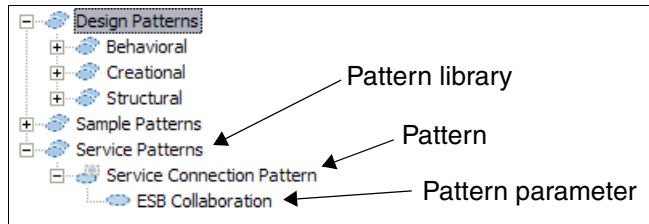
Figure 8-5   SOI pattern library

In our sample pattern library, we only have one high-level pattern: the service connection pattern, which is the appropriate pattern for the UpdateCustomerDetails example. In a more complete pattern library, we would have a selection of patterns suitable for other scenarios.

The service connection pattern takes a UML collaboration as a parameter and creates the elements required to realize that collaboration to provide an end-to-end integration service such as update customer details.

1. To use the pattern, create a UML collaboration.

2. Create a freeform UML diagram using **Add Diagram** → **Freeform Diagram.**

3. Add a UML collaboration to the diagram from the Composite Structure Diagram drawer in the RSA palette.

4. Name the collaboration `UpdateCustomerDetails` (see Figure 8-6).
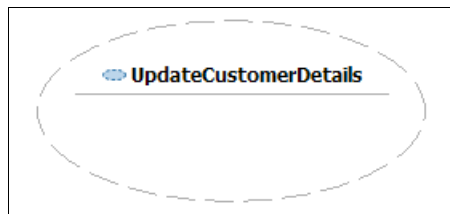
Figure 8-6   Diagram showing the UpdateCustomerDetails collaboration

5. To create an instance of the Service Connection pattern, drag from the pattern explorer to the diagram containing the collaboration (see Figure 8-7).
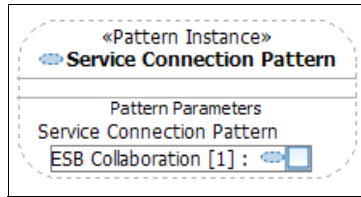
Figure 8-7   Instance of the service connection pattern

We now have an instance of the Synchronous Update pattern. This pattern takes a collaboration as a parameter and adds detail to the collaboration.

6. Drag the collaboration (from the Model Explorer or the class diagram) on to the collaboration parameter of the pattern instance. The pattern is applied and adds details to the collaboration as shown in Figure 8-8. You must manually arrange the diagram to achieve the ideal layout.
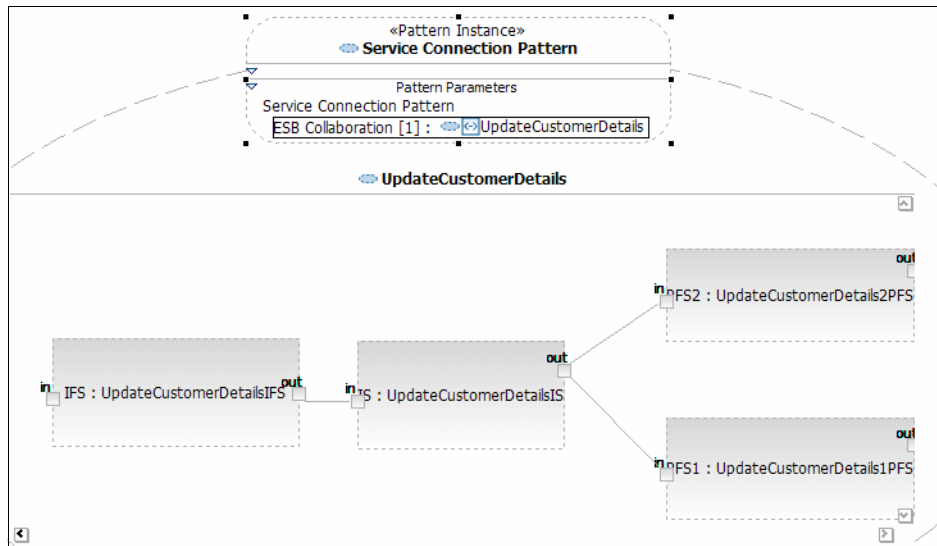
Figure 8-8   Collaboration after applying the service connection pattern

Notice that the structure inside the collaboration follows the pattern introduced in 2.2.1, "ESB structure" on page 21. The pattern created reusable service components that could also be used in other collaborations (see Figure 8-9). The names of the components are based on the name that we gave the collaboration. For the purposes of the sample, the pattern always creates new service

components. A more sophisticated pattern would reuse existing service components where available.
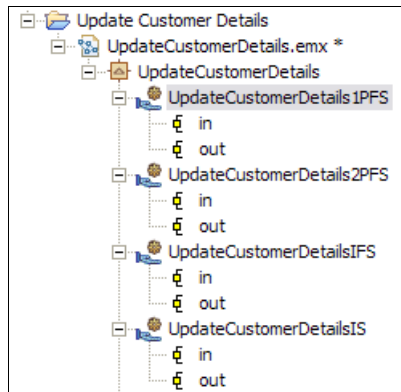
Figure 8-9   Service provider components created by the service connection pattern

The pattern also applied appropriate stereotypes from the UML profile for software services. In particular, the service components have the stereotype <<serviceProvider>>.

The resulting model could have been created manually, without using the pattern, but this would be considerably more time consuming and error prone.

Now that we have the high-level structure of our service the next step is to apply lower-level patterns to the individual components. We did not implement these patterns in our sample framework, so we need to do this step manually.

This work was done in the sample modeling project that we imported earlier. Locate the UpdateCustomerDetails project in your workspace. This project contains an UpdateCustomerDetails model that contains a completed version of the example we are working on.

The following steps were carried out to complete the model:

1. The external components that are being integrated were modeled in the CustomerRecords package. This package has the stereotype <<external>> applied to it. This is used to indicate that the modeled components exist already and do not need the transformation to generate new artifacts. We modeled an external client component in the same way.

2. The service specifications were modeled in the Service Specifications package, and the messages that are exchanged were modeled in the Data package.

3. The provided and required service specifications were defined for the components in the Service Components package. This additional information is also visible in the composite structure of the UpdateCustomerDetails collaboration within the End-to-End Service Collaborations package.

4. The behavior of each component was modeled using an activity diagram that follows the appropriate behavioral pattern. The stereotype properties of the actions within the activity diagrams were configured appropriately.

8.3.4 Applying transformations

We are ready to apply the SOI transformation to generate the EJBs that implement our services. Our sample transformation generates the complete implementation of the provider service facade components and a partial implementation of the IS and ISF components.

To understand the design of the transformations, refer to 7.8, "RSA transformation" on page 118.

In this case, the transformation can generate complete executable code based on the activity diagrams that define the behavior of each component and the defined technical architecture. In other scenarios, only structural code is generated and detailed code must be added to complete the implementation. In such cases, use care to define a development approach that ensures that models and code remain synchronized. In our example, complete code is generated and does not need to be modified manually. While this approach avoids the problems of synchronization, it is not always possible to achieve this level of code generation.

As this is an integration scenario, we already have implementations of the external service providers available to us. These were included in the sample that we imported earlier.

To apply the SOI transformation, right-click the model, and choose **Transform** → **Services Transformation** → **Run**. The transformation generates a set of EJB projects that implement the service components in the application.

If the source model is missing some information that is required for code generation, the transformation generates some of the output but terminates with an error. When the transformation is applied to the UpdateCustomerDetails sample model, the workspace ends up as shown in Figure 8-10.
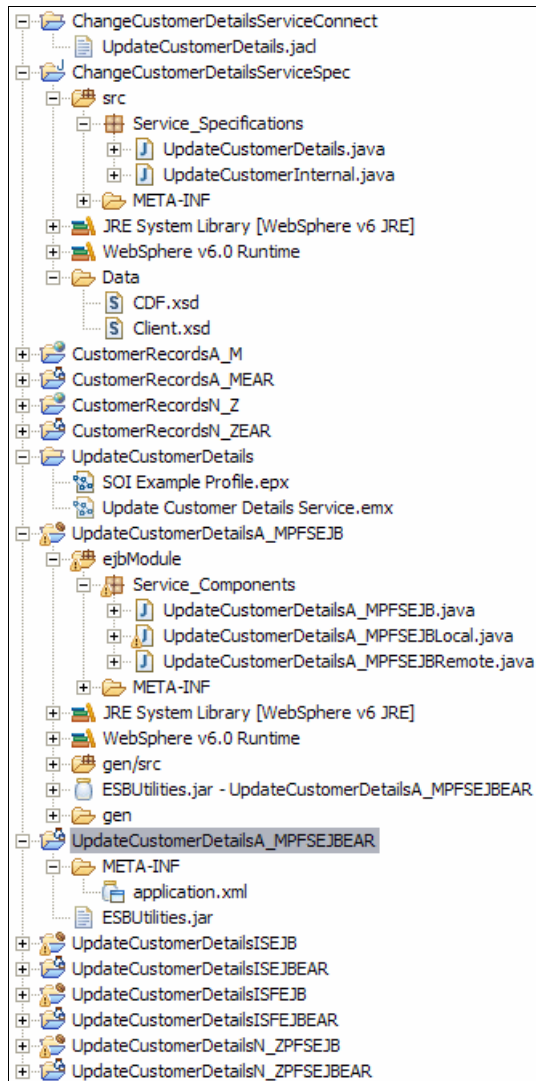
Figure 8-10   Workspace with generated projects

The generated code follows the technical architecture introduced in Chapter 2, "Scenario overview" on page 17.

8.3.5 Testing the generated code

Use the following steps to deploy and test the generated provider service facade EJBs using the IBM WebSphere Application Server V6.0 Integrated Test Environment (on Microsoft Windows®).

1. Switch to the J2EE perspective.

2. Create an application client project for the EJB you want to test.

3. In the Project Explorer, select **UpdateCustomerDetailsA_MPFSEJBEAR**.

4. Right-click, and in the pop-up menu, select **New** → **Application Client Project**.

5. Type a name for the new project in the wizard, and then click **Next**.

6. On the Module Dependencies page, ensure that you select the modules shown in Figure 8-11.
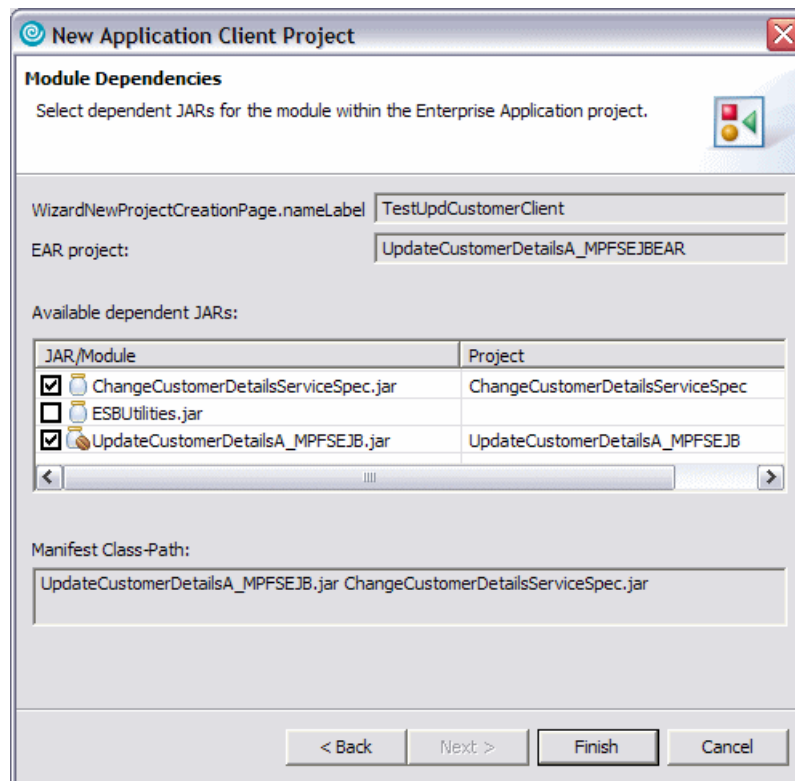
Figure 8-11   Application client module dependencies

7. Edit the generated client application (Main.java)

8. Open the Main.java file in the new application client project, and create a new method called test.

9. Call it from the main method, as shown in Example 8-1.

Example 8-1   Main.java

```
public class Main {
    public static void main(String[] args){
        new Main().test();
    }
    public Main(){
        super();
    }
    public void test(){
    }
}
```

10. Use the Snippets view, shown in Figure 8-12, to insert the Call a Session Bean service method snippet in the test method.
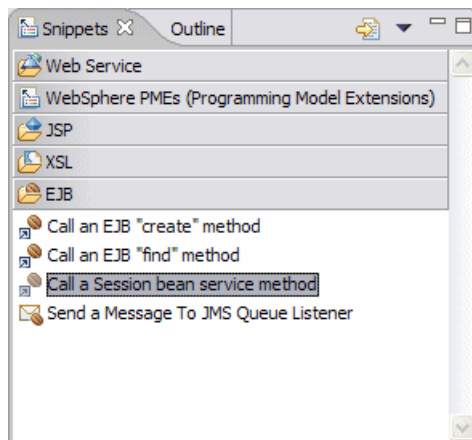
Figure 8-12   Snippets view

The EJB to be referenced is UpdateCustomerDetailsA_MPFSEJB, and the method to call is updateCustomerDetails.

Setting up the string to be passed to the EJB method

Once the snippet is inserted, the next step is to set up the string to be passed to the EJB method.

> **Note:** The architecture allows for the provider facade EJB to perform an XSLT transform to create the message it passes to the provider service. However, the sample model does not specify any such transform. Also the EJB base class supplied with the sample (in the ESBUtilities project) does not support performing message transformations.

The argument we want to pass to the EJB updateCustomerDetails method is the SOAP message that is sent to the provider service. This is quite long so we will read it from a file rather than hard-coding it in the client application.

A suitable SOAP message is shown in Example 8-2.

Example 8-2   Test message data (test.xml file)

```
<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:tns="http://customer.soi.example.itso"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
    <tns:updateCustomerRecords>
    <tns:customerInformation>
        <tns:FName>John</tns:FName>
        <tns:SName>Smith</tns:SName>
    </tns:customerInformation>
</tns:updateCustomerRecords>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

1. Put the test.xml file in the root of the application client project.
   One more line of code is needed, to print out the return message from the
   EJB so we can see if it was successful. The test method should now look like
   Example 8-3.

Example 8-3   Application client test method

```
// read in test message
StringBuffer buffer  = new StringBuffer();
BufferedReader file = null;
try {
    file = new BufferedReader(new FileReader("test.xml"));
    String line = file.readLine();
    while( line != null ){
        buffer.append(line);
        line = file.readLine();
    }
} catch (FileNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} finally {
    try {
        file.close();
    } catch (IOException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
}
String in = buffer.toString();

UpdateCustomerDetailsA_MPFSEJBRemote anUpdateCustomerDetailsA_MPFSEJBRemote
= createUpdateCustomerDetailsA_MPFSEJBRemote();
try {
    String aString = anUpdateCustomerDetailsA_MPFSEJBRemote
            .updateCustomerDetails(in);

    System.out.println(aString);

} catch (RemoteException ex) {
    // TODO Auto-generated catch block
    ex.printStackTrace();
}
```

Tip: If you have compile errors in the client application that cannot be
resolved, perform a clean build of the EJB.

2. Deploy the Web service and EJB projects.

3. In the Servers view, right-click the server you want to use, and in the menu select **Add and remove projects...** Add the CustomerRecordsA_MEAR and UpdateCustomerDetailsA_MPFSEJBEAR projects. You can test the N_Z projects too, but they aren't any different.

> **Tip:** If you cannot successfully deploy the EJB because the server says that a URI is too long, then follow the instructions in the RSA help to create a new server profile (with a shorter path). Create a new server based on the new profile and use it for testing.

4. Check the port number for the provider Web service.

5. In the CustomerRecordsA_M project open the WebContent\WEB-INF\wsdl\CustomerRecords.wsdl file. At the end of the file, the service is defined with a service location URL, as shown in Example 8-4.

Example 8-4   CustomerRecords Web service definition

```
<wsdl:service name="CustomerRecordsServiceA-M">
<wsdl:port binding="intf:CustomerRecordsSoapBinding" name="CustomerRecords">
<wsdlsoap:address
location="http://localhost:9080/CustomerRecordsA_M/services/CustomerRecords"/>
</wsdl:port>
</wsdl:service>
```

The default port number is 9080, but depending on how you set up your WebSphere server you may need to use a different port number.

6. If you do not know what it should be, use the following steps to run the administrative console for your server.

    a. Navigate to the **Servers** → **Application servers** page.
    b. Click your server name.
    c. Click **Ports** (under Communications). At the end of the table of ports, you need the port **WC_defaulthost**, unless you are not using the default host.

7. Change the service location URL in the wsdl file if necessary so that the host and port number are correct.

Next, you test the provider service.

Testing the provider Web service

Before trying to test the provider facade EJB, it is a good idea to test the provider Web service (CustomerRecordsA_M) to make sure that it is correctly set up.

1. Right-click the **CustomerRecords.wsdl** file, and select from the menu **Web Services** → **Test with Web Services Explorer**.

2. In the Web services explorer, select the **updateCustomerRecords** operation.

3. Click **Go** to invoke the service. You should see the OK response (see Figure 8-13).
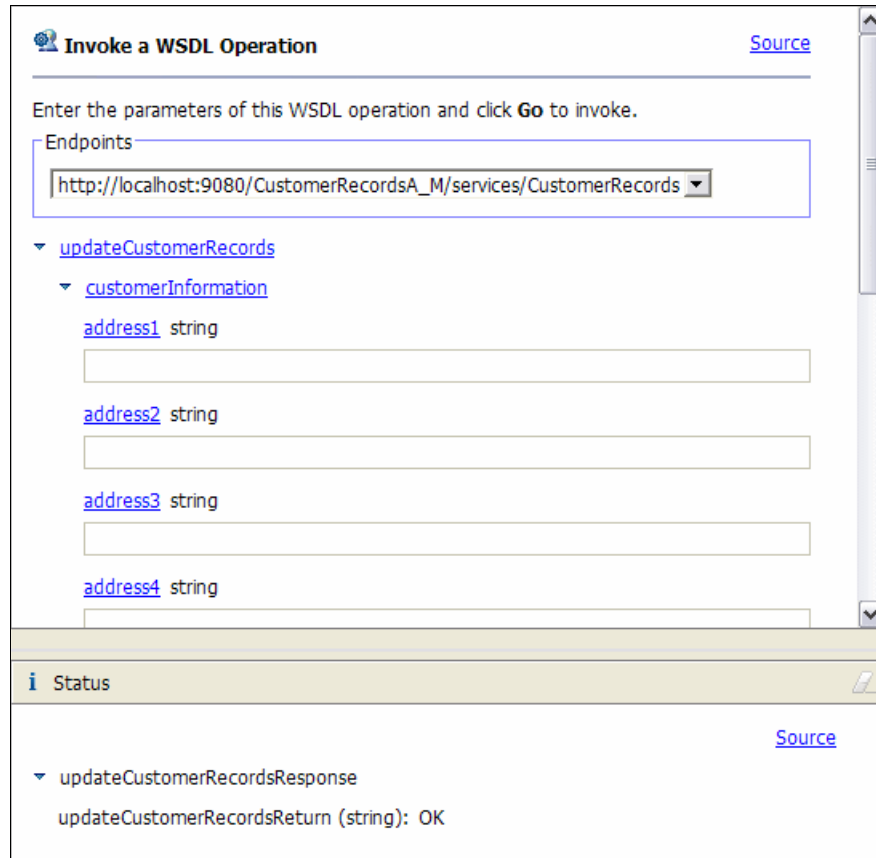
Figure 8-13   Testing the provider service

4. Run the connection script.

   The provider service facade EJB looks up in JNDI the URL of the provider service it should call. The transformation generates a script that you can run to set up the JNDI namespace bindings required by the generated EJBs. The script is in the project called ChangeCustomerDetailsServiceConnect.

5. Open the script (**UpdateCustomerDetails.jacl**), and check that the URL, which is bound as the location of the provider service, matches the service location URL that you set.

6. In the Servers view, right-click *your server* and select from the menu **Run external admin script...**

7. Select the **UpdateCustomerDetails.jacl** script file, and run it.

   The script takes a few moments to run. Make sure it has terminated and then use the administrative console to check that the namespace bindings have been created.

8. Run the client application.

   a. Right-click the **Main.java file** in the application client project, and select from the menu **Run** → **Run...**

   b. In the Launch Configuration dialog, create a new WebSphere 6.0 Application Client configuration.

   c. Ensure that the **Enable application client to connect to server** option is selected, as shown in Figure 8-14.
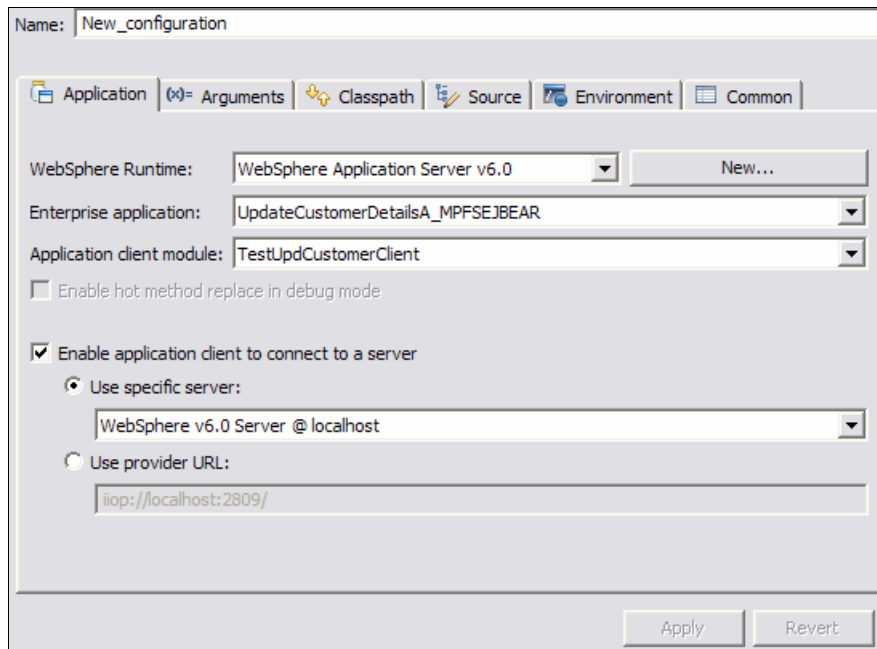
Figure 8-14   Application client launch configuration

Setting the working directory

You also need to set the working directory so that the client application can find the test.xml file.

1. Switch to the arguments tab, and change the working directory to point to the application client project, as shown in Figure 8-15.
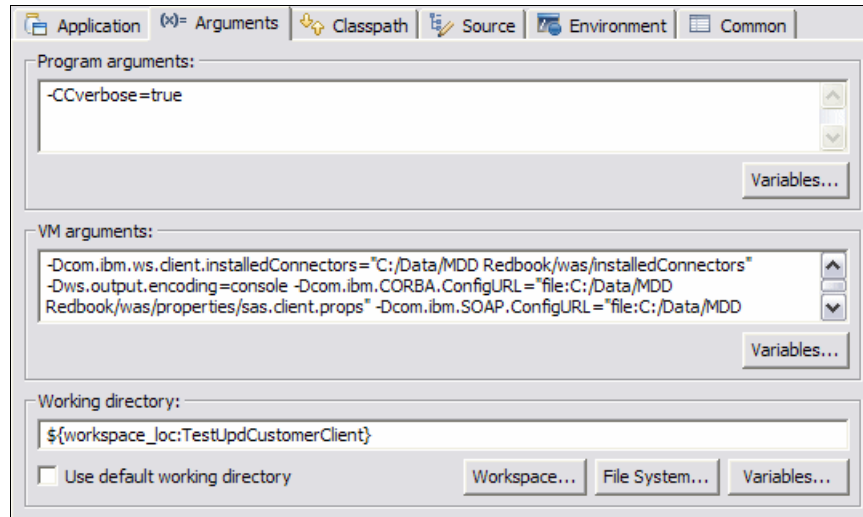
Figure 8-15 Application client working directory

2. Click **Apply** to save your changes.

3. Click **Run** to execute the application. If it runs successfully, you see the response message from the EJB written to the console, similar to Figure 8-16.

Figure 8-16 Application client console

8.3.6 Application development summary

In this section, with relatively little effort, we developed service implementations that conform to the architectural style introduced in Chapter 2, "Scenario overview" on page 17. The service implementations follow best practices in terms of high-level architecture and detailed technical architecture.

8.4 Framework development

In the previous section, we used the MDD framework for SOI to build a new application. In this section, we look at how to develop the MDD framework for SOI as a customisation of RSA.

8.4.1 Developing the architectural style

When developing a new MDD framework we must first develop the architectural style that is to be automated.

This includes the following tasks:

► Develop high-level architectural principles and patterns
► Design UML profile or profiles
► Develop the technical architecture and sample components

The SOI architectural style is documented in Chapter 2, "Scenario overview" on page 17. Sample components that demonstrate the technical architecture were also developed.

Building an MDD framework is about expertise capture. It is crucial that appropriate experts are available to define the architectural style.

> **Tip:** The activity of developing sample components that demonstrate and validate your technical architecture must be allocated sufficient time and expertise. Failing to adequately define the technical architecture leads to costly time delays in implementing the transformations.

8.4.2 Creating a UML profile

A UML profile must be carefully designed to provide an appropriate set of concepts for modeling in a given context. The design of the SOI Example profile is presented in 7.7, "Detailing the initial model with service patterns" on page 110. The design results in the need to support stereotypes in order to identify the following items:

▶ All the possible actions within the activity diagrams for ESB services: Each activity type has its own stereotype.

▶ Attributes defining the behavior of the service: For example, the transform activity has an attribute "transformationID" that defines the transformation to be applied by the transformation service.

▶ Parameters guiding the generation of a service: For example, the <<optional>> stereotype has a boolean "generate" attribute that determines whether this activity gets included when a service is generated. Also, the stereotype <<generate>> has a boolean attribute that defines whether this is a service to be generated or whether the service is a reference to an existing service.

▶ Parameters identifying and guiding behavior of the service as a whole: For example, the integration service facade stereotype defines the service version, what level of logging is to be recorded.

The steps in the following sections demonstrate how to create a subset of the stereotypes that are included in the SOI Example Profile.

Creating a profile project

Create a new UML Profile project called `SOI Example Profile` using the following steps:

1. Click **File → New → Project → Modeling → UML Extensibility → UML Profile Project**.

2. Call the profile `Test`. The default options in the wizard are fine.

Adding stereotypes

A profile includes a set of stereotypes that are applied to model elements. As an example, we create the <<external>> stereotype that is used in the SOI Example Profile to identify elements that are being reused and therefore do not need to be generated by the transformation.

Use the following steps to add the <<external>> stereotype:

1. Right-click the **Profile model → Add UML → stereotype**.

2. Name the stereotype external.

3. We now need to specify which UML elements the stereotype can be applied to. Select the <<external>> stereotype in the Model Explorer → **Properties view** → **Extensions** → **Add Extension** → **Class.**
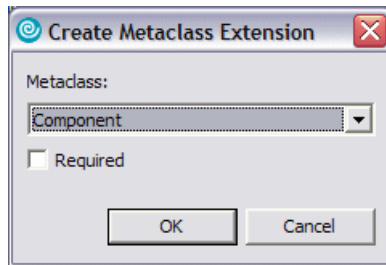
Figure 8-17   Adding component extension to a stereotype

4. Repeat these steps for **Component, Package** and **Interface**.

We now have an <<external>> stereotype that can be applied to UML Classes, Components, Packages, and Interfaces.

Tip: The usual naming convention for stereotypes is to use initial lower case.

Validating and testing the profile
Although our profile does not have much in it yet, we can still test it at this stage, and it is useful to do so.

1. Save the profile project.

2. Select **Run Validation** from the right-click menu. Any validation errors or warning for your profile appear in the **Problems** view.

3. Fix any problems before continuing.

4. Create a new UML modeling project and select the model.

5. Go to **Properties** → **Profiles** → **Add Profile**. We have not deployed the profile yet, so select **File** in the profiles pop-up and navigate to the profile on your file system (it appears as an .epx file). Figure 8-18 shows the result.
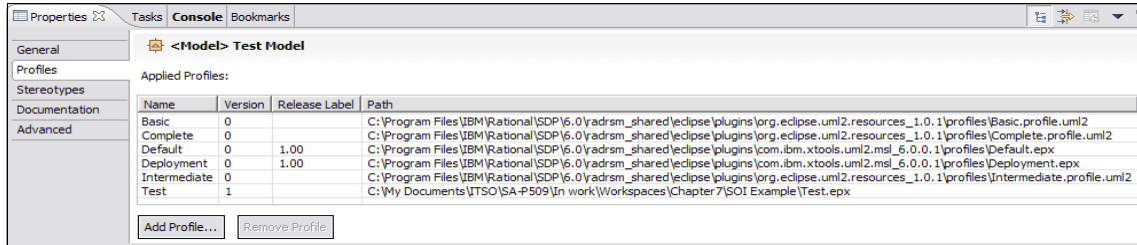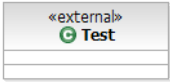
Figure 8-18   Result of adding the test profile

6. Create one of the UML elements to which you can apply the <<external>> stereotype (Class, Component, Package or Interface). We chose to stereotype the Test class.

7. Select the element, and view the stereotypes tab in the Properties view. Click **Add**.

The <<external>> stereotype appears in the pull-down list, as in Figure 8-19.
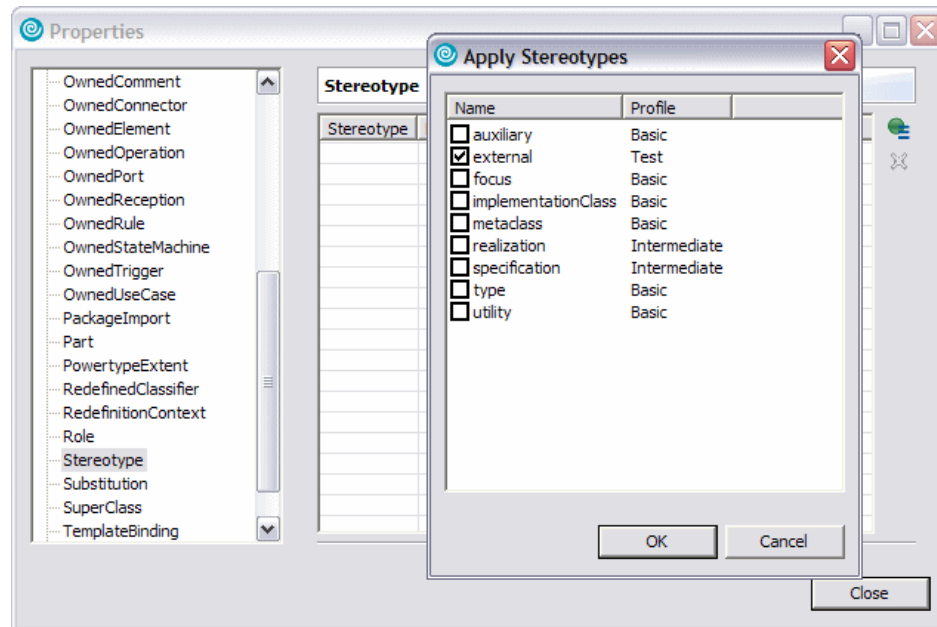
Figure 8-19   Applying a new stereotype

When you make a change to the profile you need to synchronize any models that have it applied:

1. Return to the Profile project, Test.

2. Add a new stereotype called externalService. This marks external service components that are invoked from provider service facades.

3. Save the profile.

4. Add the extension component to the new stereotype, and save the Test.epx profile.

5. Return to your example model, and look at the Profiles tab. The SOI Example Profile is marked as (OUT OF SYNC) as shown in Figure 8-20.

6. Select **Test profile** → **Migrate Profile** to load the new version of the profile. You can now use the stereotype that you added.

Name	Version	Release Label	Path
Basic	0		C:\Program Files\IBM\Rational\SDP\6.0\radrsm_shared\eclipse\plugins\org.eclipse.uml2.resources_1.0.1\prof
Complete	0		C:\Program Files\IBM\Rational\SDP\6.0\radrsm_shared\eclipse\plugins\org.eclipse.uml2.resources_1.0.1\prof
Default	0	1.00	C:\Program Files\IBM\Rational\SDP\6.0\radrsm_shared\eclipse\plugins\com.ibm.xtools.uml2.msl_6.0.0.1\prof
Deployment	0	1.00	C:\Program Files\IBM\Rational\SDP\6.0\radrsm_shared\eclipse\plugins\com.ibm.xtools.uml2.msl_6.0.0.1\prof
Intermediate	0		C:\Program Files\IBM\Rational\SDP\6.0\radrsm_shared\eclipse\plugins\org.eclipse.uml2.resources_1.0.1\prof
Test (OUT OF SYNC)	3		C:\My Documents\ITSO\SA-P509\In work\Workspaces\Chapter7\SOI Example\Test.epx

Figure 8-20   Model with out of sync profile

Tip: During profile development you can make incompatible changes to the profile that may cause problems with your test model. Make sure your profile is stable before developing models that depend on it.

Using the Inheritance Explorer

It is often useful to create hierarchies of stereotypes to capture commonalities. A common pattern is to have an Abstract stereotype at a root of the hierarchy that specifies the UML element (metaclass) to be extended and a set of concrete sub-stereotypes.

The actions that the SOI Example Profile provides for applying to actions in activity diagrams follow this pattern.

1. Create a <<ServiceAction>> stereotype, and add an extension for the Action metaclass.

2. Under **Properties** → **General**, mark the stereotype **Abstract**. This means that users cannot apply this stereotype directly in their model. The SOI Example Profile uses the convention that abstract stereotypes begin with an initial uppercase letter, where concrete stereotypes begin with an initial lowercase letter.

3. Drag the ServiceAction stereotype to the Inheritance Explorer.
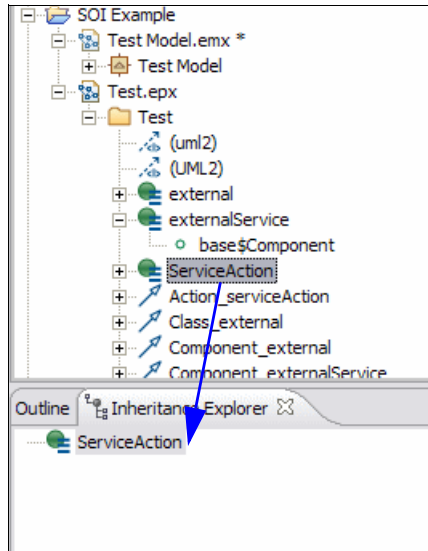
Figure 8-21   Dragging the ServiceAction stereotype to the Inheritance Explorer

4. Right-click the stereotype, and select **Create New Subtype** to introduce sub-stereotypes.

5. Add sub-stereotypes named pushCallbackAddress, popCallbackAddress, and createAcknowledgement.

6. Save the profile, and test it using your example model.

7. Create an Activity diagram, and then add actions to it. Apply the three concrete sub-stereotypes to the actions.
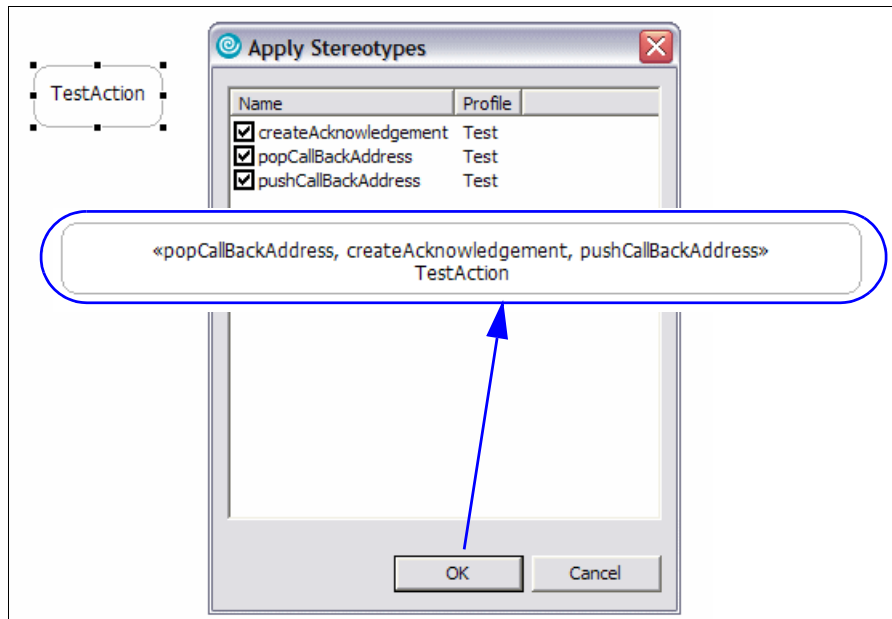
Figure 8-22   Applying the subtypes to the testAction

Tip: Do not forget to migrate any models that you have open when you make a change to a profile. When you open a model after a change to a profile, RSA asks if you want to migrate to the latest version of the profile.

Adding attributes to stereotypes

We can also add attributes to stereotypes. When a developer applies the stereotype they can provide values for the attributes.

1. Create the TechnicalServiceAction and recordPerformance stereotypes in the hierarchy.
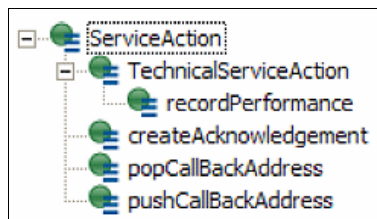
Figure 8-23   TechnicalServiceAction and recordPerformance stereotypes in hierarchy

2. In the Model Explorer (not the Inheritance Explorer), right-click **recordPerformance** and select **Add UML** → **Attribute**.

3. Name the attribute `subidentifier`.

4. Under **Properties** → **General** for the subidentifier attribute, set the type of the attribute to String. The UML primitive types are available for use as types for attributes. You can set a default value for the attribute in cases where this makes sense.

Setting enumeration types

It is often useful to have attributes with an Enumeration type. The recordPerformance stereotype has two such attributes: class and level.

1. Right-click the **Test** model and select **Add UML** → **Enumeration** to add enumerations called `PerformanceEventClass` and `PerformanceEventLevel`.
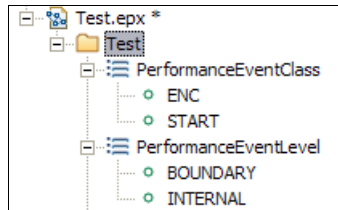
Figure 8-24   Literals added to the enumerations

2. Right-click each literal and select **Add UML** → **Enumeration Literal** with the enumeration literals shown in Figure 8-24.

3. Apply these enumerations as the types for the class and event attributes on the recordPerformance stereotype as in Figure 8-25.

Figure 8-25   Adding types as enumerations to recordPerformance

4. Save the profile, and test it again at this point.

5. Apply the recordPerformance stereotype to an action, and access
   Properties → **Stereotypes** to see the attributes and set values for them. See
   Figure 8-26.
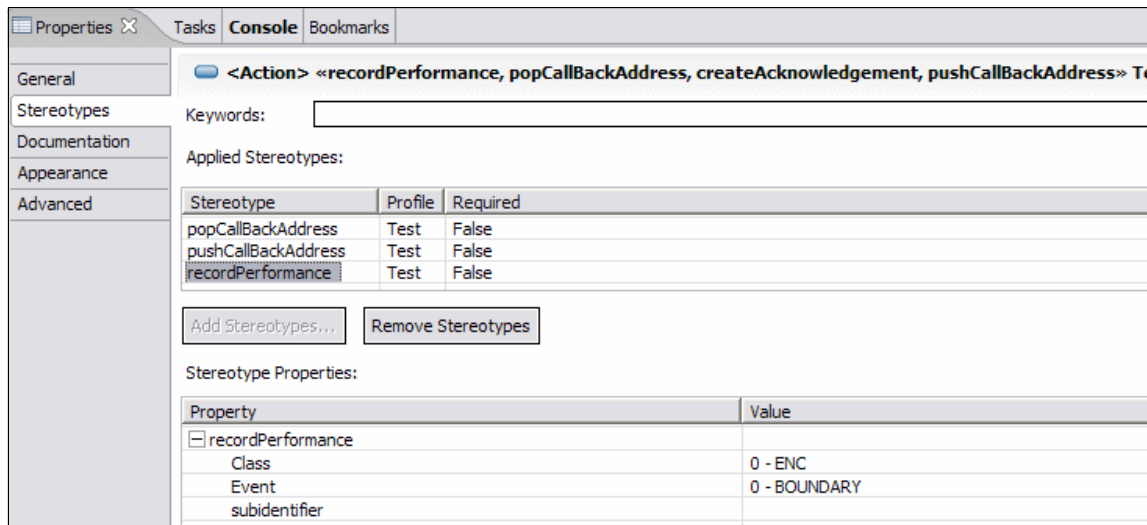
Figure 8-26   Stereotype attributes/properties

Adding icons

Stereotypes can also introduce icons that will appear in RSA. Icons are added in
the Properties General tab for a stereotype. See Figure 8-27.
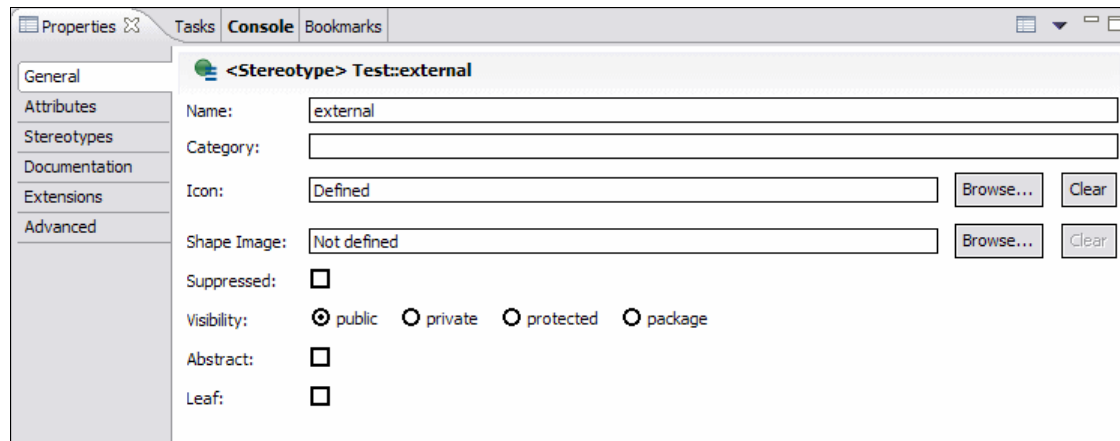
Figure 8-27   Icon and Shape Image can be set for stereotypes

Icons are used in the Model Explorer and as decorations on diagrams. Icons can be GIF or BMP files, and the recommended size is 16 x 16.

Shape Images are used on diagrams when stereotype display for an element is set to Shape Images. Shape Images are SVG files, and the recommended size is 50 x 50.

The SOI Example Profile does not introduce any new icons, but it does make use of those introduced in the UML profile for software services. For example, the icon associated with the <<service provider>> stereotype is .

> **Tip:** Stereotype icons are very useful. They not only make your profile look more professional, they can also help to make models more understandable, especially if you choose good icons. They also save space in the Model Explorer and on diagrams as the icon takes up less space than the stereotype name.

Releasing the profile

Once a profile reaches stability, you can chose to release the profile. After a profile is released, you can only make changes that are compatible with the released profile. This ensures that models that use a released profile can always migrate successfully to new versions of the profile.

You can add to a released profile but you cannot delete or rename elements (stereotypes, stereotype attributes, etc.).

To release a profile, right-click the profile in the Model Explorer and choose **Release ...**. Enter a release label in the dialog box that appears.

> **Tip:** Carefully consider the right time to release a profile. Once it is released, you cannot make any incompatible changes, such as renaming a stereotype.

Packaging the profile for deployment

When you are ready to share a UML profile with others, one convenient way to deploy it is as a plug-in. When a profile is deployed it appears in the pull-down list of deployed plug-ins that can be applied to a model. Deploying a profile as a plug-in is described in 9.3, "Deploying UML profiles" on page 163.

8.4.3 Implementing sample components

Before transformation development begins, you must create sample components that follow the technical architecture. The creation of these sample components has two main purposes:

- ► To validate the technical architecture
- ► To provide sample artifacts as input to transformation development

8.4.4 Developing patterns and transformations

The patterns and transformations needed for an MDD framework are identified as part of the development of the architectural style. The patterns and transformations needed for the MDD framework for SOI are introduced in Chapter 2, "Scenario overview" on page 17.

RSA patterns and transformations are implemented as RSA plug-ins written in Java. Chapter 9, "Extending Rational Software Architect" on page 161, describes how to implement RSA patterns and transformations.

8.5 Summary

In this chapter, we looked at how RSA supports the MDD process. We covered the activities that are carried out by architects and designers in both the application development and the framework development phases of the MDD process.

In the following chapter, we cover the implementation of RSA patterns and transformations in RSA.

9

Extending Rational Software Architect

Chapter 8, "Applying model-driven development with Rational Software Architect" on page 129, explained how to apply Rational Software Architect (RSA) to application and framework development. This chapter describes how you can extend RSA with your own transformations and patterns.

In this chapter, we provide step-by-step instructions for deploying profiles, and implementing patterns and transformations. The examples we use here are based on the ESB scenario described in Chapter 2, "Scenario overview" on page 17, and in Chapter 7, "Designing patterns for the scenario" on page 93.

9.1 Introduction to implementing patterns and transformations to RSA

Chapter 3, "Model-driven development approach" on page 29, introduced the concepts of UML profiles, patterns and transformations. In this chapter, you learn how to implement patterns and transformations as extensions to RSA.

Profiles, patterns, and transformations are all packaged for deployment as Eclipse plug-ins. A plug-in is a package of extensions to RSA. One plug-in can contain many extensions which might contribute to different aspects of RSA. A plug-in can contribute menu items, toolbar buttons, views and perspectives as well as patterns, transformations and UML profiles.

Any RSA plug-in can declare extension points that other plug-ins can contribute to. This is the basic extension mechanism for the Eclipse platform which RSA is based on. The extension points declared by the RSA UML model editor plug-ins include those that allow you to contribute UML profiles, patterns, and transformations.

RSA includes all the tools you need to create new plug-ins for RSA. For general information about plug-ins and the Plug-in Development Environment, see the RSA help topic "Extending Rational Software Architect functionality", starting with the section "Extending the workbench - Platform Plug-in Developer Guide".

This chapter contains instructions for performing specific tasks. Read 9.2, "Setup: Enabling Eclipse Developer" on page 163, first. You can read the subsequent sections in any order.

To learn how to package a UML profile for deployment as an RSA plug-in, read 9.3, "Deploying UML profiles" on page 163.

To learn how to extend the RSA UML model editor by implementing patterns, read 9.4, "Implementing patterns" on page 173.

To learn how to implement a transformation that generates code from a UML model, read 9.5, "Implementing a transformation" on page 194.

9.2 Setup: Enabling Eclipse Developer

Before trying any of the examples from this chapter, enable the Eclipse Developer capability in order to use the Plug-in Development Environment.

1. From the main menu, select **Window** → **Preferences**:

2. Select the **Eclipse Developer** capability, as shown in Figure 9-1.
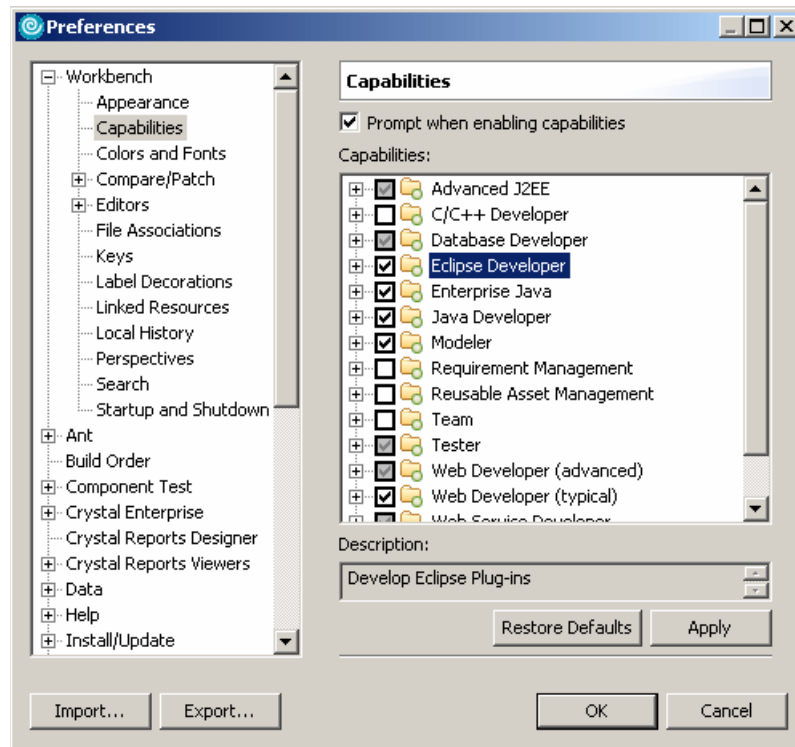
Figure 9-1   Preferences dialog

9.3 Deploying UML profiles

Defining a UML profile is one way to extend the RSA UML model editor. You create a new profile as an .epx file in your RSA workspace. The profile in this form can be applied to models, and you can continue to modify it.

If you want to share your models with other people, you need to also give them the profile. Once a profile is finalized, it is convenient to provide it in the form of an RSA plug-in, which anyone who needs to use the profile can install. Models can then be exchanged freely without having to also copy the profile or set up path maps.

To deploy a profile, you need to complete the following steps:

1. Define a path map to the profile's location.
2. Release the profile.
3. Add the profile to a plug-in.
4. Deploy the plug-in.

Complete the first step, define a path map, when you create a new profile. Complete the subsequent steps after you develop the profile.

9.3.1 Defining a path map

If the model file and profile were originally in the same project, then the model file has references to the profile using a relative file path. Therefore, you must keep the model and profile files in the same folders relative to each other. This allows those references to be resolved.

You can make it easier to share models by having everyone in your team define a path map to where they put the profile in their workspace. The path map associates a symbolic name with a file path. Whenever a model has a reference to a file that is on that path, the symbolic name is used instead of the actual file path. If everyone in your team has path maps defined with the same symbolic names, you can exchange models without having to worry about where the profile is kept.

It is a good idea to set up a path map whenever you create a new profile. If when you set up the path map you already have models that reference the profile, you need to make sure the profile and models are in separate projects; otherwise, the models will not use the path map.

In the test example from the last chapter, the model and profile were in the same project, and did not use a path map. The first step is to put the profile into a separate project and then use a path map to refer to it.

1. Access **File → New → Project → UML Profile Project**, and call the project TestProfile.
2. Drag and drop the Test.epx profile into the new project, and delete the default profile.epx from the new project.

   The models automatically refactor to point to the new location of the profile.

If they do not refactor correctly, clean up the test model by removing the test profile and adding it again from the new location. Then reapply the stereotypes.

3. Create the path map pointing to the new TestProfile project. Path maps are defined in the workbench Preferences.

   a. From the menu, select **Window** → **Preferences** under the category **Modeling**.

   b. Using the test profile from the previous chapter as an example, set up a path map to the folder containing the test profile.
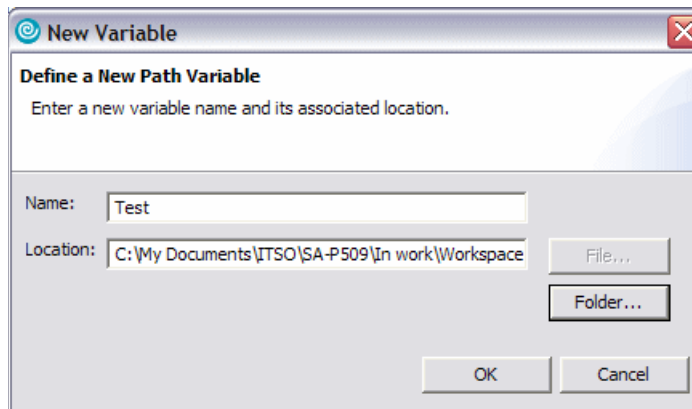
Figure 9-2   Setting up a path map variable

> **Tip:** It is worth shutting down the RSA workspace and starting it up again at this point to check that there are no path not found errors.

Make a note of the path map name that you use because you will need this information later.

9.3.2 Releasing the profile

If you have not already done so, create a released version of the profile before you deploy it.

1. Open the profile, and in the Model Explorer select the profile package. Right-click, and select **Release**. Enter a label for the release.

2. After you create a new version of the profile, either by releasing it by changing something, you need to open any models to which that profile was applied so that they can be migrated to the new version.
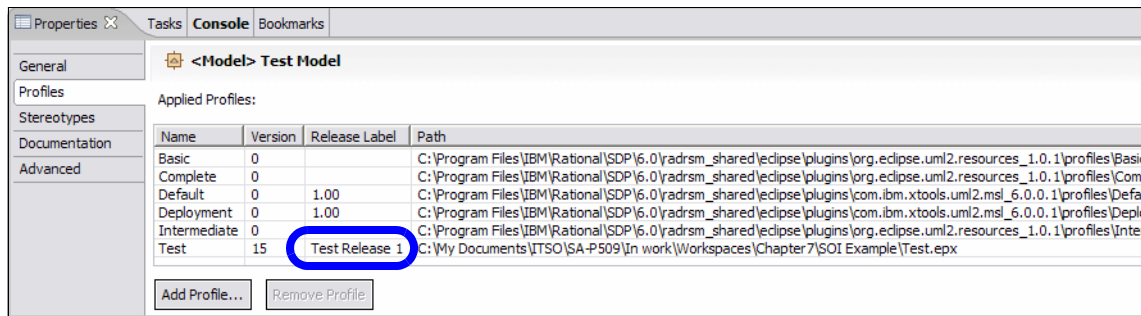
Figure 9-3   Released Test profile migrated into the test model

9.3.3 Adding the profile to a plug-in

You can add the profile to an existing plug-in project, or create a new plug-in project.

Creating a new plug-in project

1. From the main menu, select **File** → **New** → **Project...** → **Plug-in Development** → **Plug-in project**.

2. A plug-in must have a unique identifier, and the name of a plug-in project is usually the same as the plug-in's identifier. Because it must be unique, the identifier usually takes the form of a Java package name. This may be a Java package name if the plug-in happens to include Java source code, but it does not matter. The identifier must be unique.

   A plug-in usually includes Java code, so a plug-in project is often also a Java project. In this case we are creating a plug-in to contain a UML profile and no Java classes is needed, so it does not need to be a Java project.
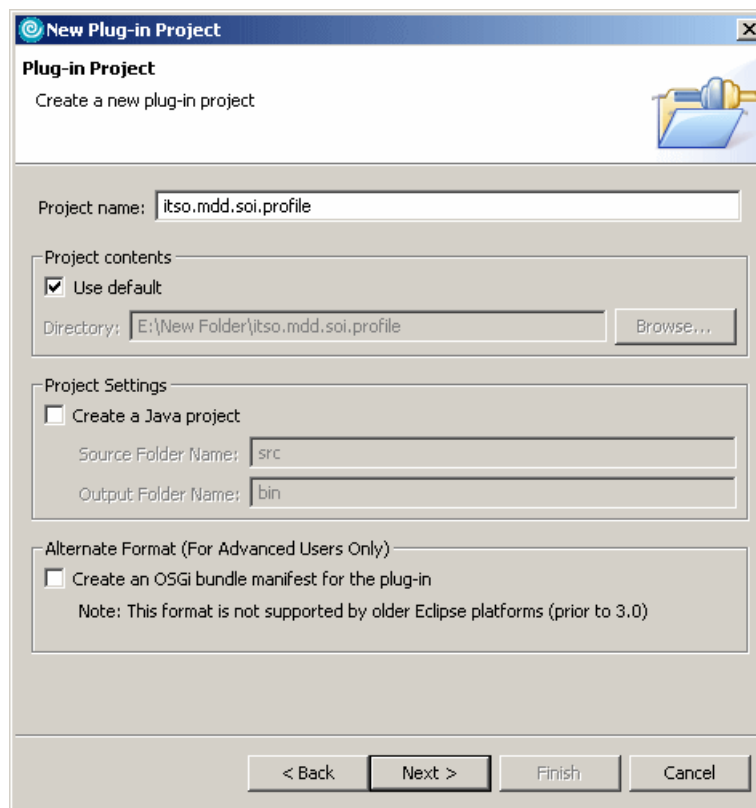
Figure 9-4   New Plug-in project wizard

3. On the next page of the wizard, the plug-in ID is filled in for you on the assumption that the project name is the same as the plug-in ID. The Plug-in Name is a human-readable name for the plug-in. The Plug-in Provider is a human-readable name identifying your company or organization.

    Click **Finish** to create the new project.

Figure 9-5   New Plug-in properties

4. The plugin.xml file is the plug-in manifest, which identifies the plug-in and declares what extensions it contributes to RSA. This file is open for editing, as shown in Figure 9-6.

   Click the **plugin.xml** tab to see full contents of the file as XML.
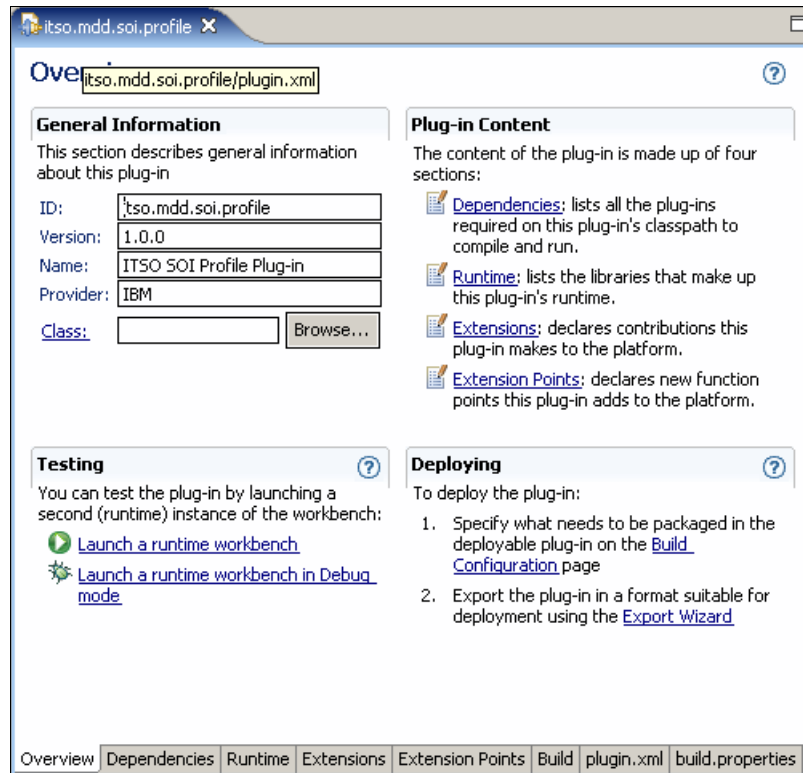
Figure 9-6   Plug-in manifest editor

Example 9-1 shows the contents of the plug-in manifest file.

Example 9-1   Plug-in manifest file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="itso.mdd.soi.profile"
   name="ITSO SOI Profile Plug-in"
   version="1.0.0"
   provider-name="IBM">
</plugin>
```

Adding the profile to a plug-in project

1. Edit the plug-in manifest by adding (before </plugin>) the text shown in Example 9-2.

Example 9-2   Pathmap and UML profile extensions

```
<extension
        point="com.ibm.xtools.emf.msl.Pathmaps">
    <pathmap
        name="SOI_PROFILE_PATH"
        plugin="itso.mdd.soi.profile"
        path="profiles">
    </pathmap>
</extension>
<extension
        point="com.ibm.xtools.uml2.msl.UMLProfiles">
<UMLProfile
id="itso.mdd.soi.profile"
name="SOI Example Profile"
path="pathmap://SOI_PROFILE_PATH/SOIExampleProfile.epx"
required="false"
visible="true">
</UMLProfile>
</extension>
```

This declares two extensions:

– The first is a path map. Use the same path map name that you previously used (see Step 1) to locate the profile. The path map name is resolved to the 'profiles' folder within this plug-in—create this simple folder now. After you build the deployed profile you can remove the path map from the workspace preferences.

– The second extension declares that the profile SOIExampleProfile.epx resides at the location identified by the path map, and associates it with human-readable name.

2. Drag the Test.epx profile from the TestProfile project into the **Profiles** folder in itso.mdd.soi.test.

3. In the plug-in manifest editor, click the **Build** tab.

4. In the Binary Build box, select the files and folders that you want to deploy in the plug-in. You must always include the plugin.xml file. Here, we also want to include the *profiles* folder.
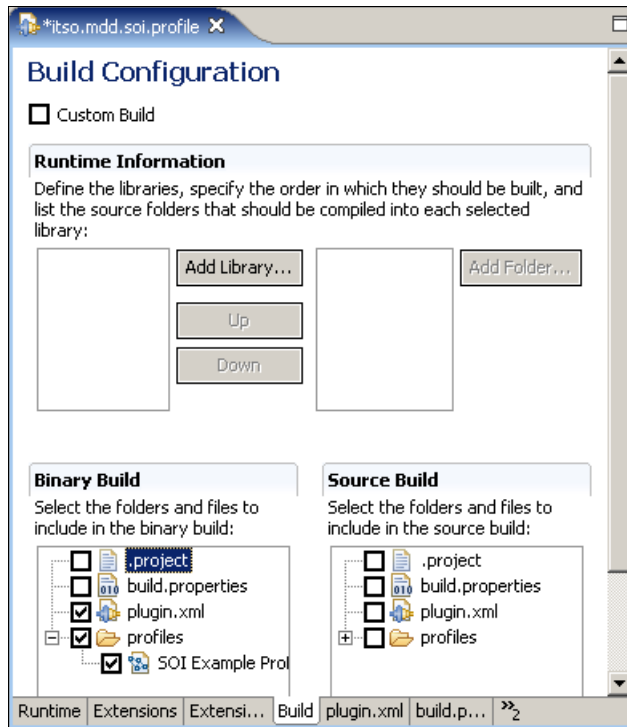
Figure 9-7   Plug-in manifest build page

5. Save and close the plug-in manifest.

9.3.4 Deploying the plug-in

1. From the main menu, select **File** → **Export - Deployable plug-ins and fragments**.

2. To deploy the plug-in into your own workbench, use the options shown in Figure 9-8, checking that the directory is correct for your RSA install.

   The effect is to create a directory called *plugin-id*_1.0.0 (the plug-in ID combined with its version) in the plug-ins folder in the location specified, containing the files you selected to be included in the plug-in. You can do this manually or in an ANT build script instead of using the wizard.
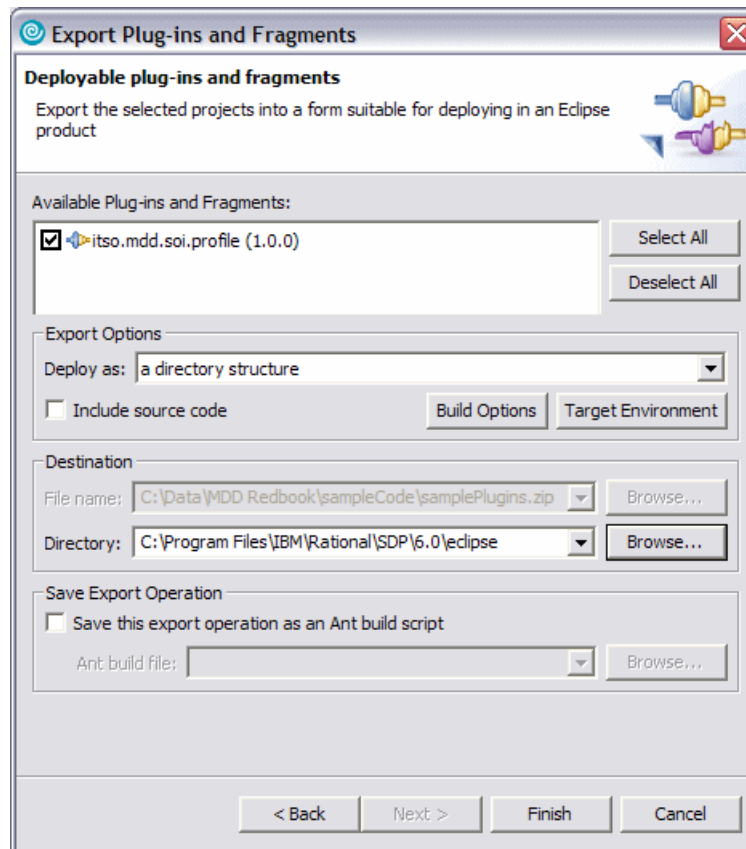
Figure 9-8   Export deployable plug-ins wizard

3. Restart RSA. The profile is available for use as a "deployed profile", identified by the name you gave it in the plug-in manifest.

   To package the plug-in in a zip file to give it to someone else, then you can export it as a zip file. To install this, unzip the file into the appropriate directory and restart RSA.

9.4 Implementing patterns

There are tutorial materials in the Rational Software Architect Help that are relevant to what is described in this chapter. Look for the topic **Creating modeling artifacts for reuse** → **Authoring patterns**. There are also sample pattern plug-in projects that you can import into the RSA workspace. See the Samples Gallery under Help, **Technology samples** → **Patterns**.

9.4.1 Getting started

Patterns are packaged as plug-in projects. Therefore, the first thing we need to do for pattern authoring is to create a new plug-in project.

Creating a new plug-in project

1. Click **File** → **New** → **Project...** and select **Plug-in Development** → **Plug-in Project**.

2. Specify a project name on the next page of the wizard. As mentioned in the previous section, this project name identifies the plug-in and must be unique. For our example, we call it `ESB Service Pattern`. You can leave the remaining settings on this page as the defaults.
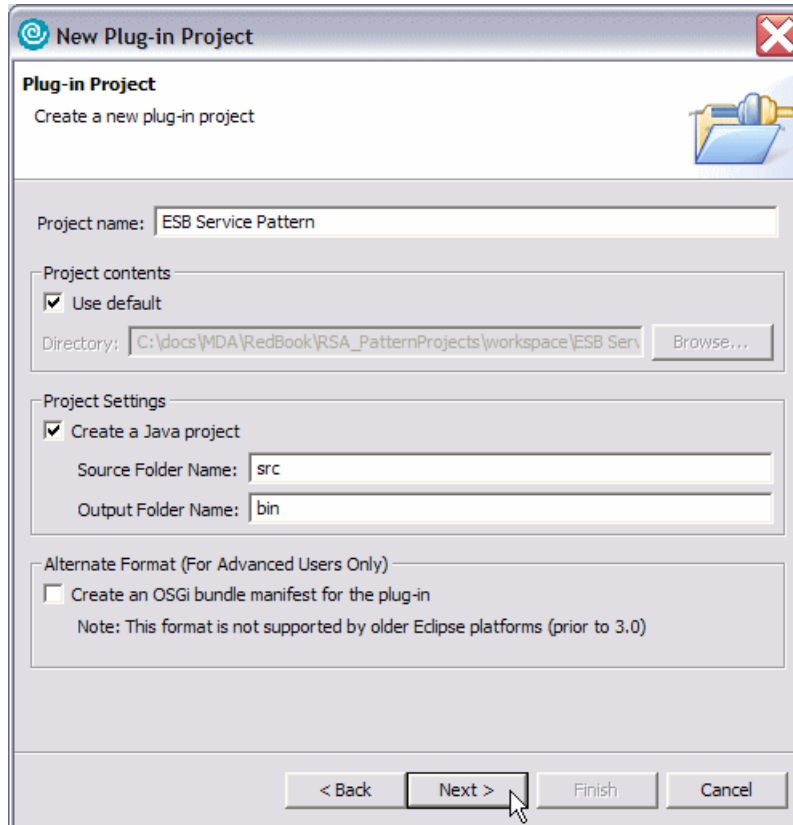
Figure 9-9   Creating a pattern plug-in project: Specifying the project name

3. As illustrated in Figure 9-10, the Plug-in ID is the unique identifier for the plug-in, it is filled in based on the assumption that the ID is the same as the project name. The Plug-in Name is a human-readable name for the plug-in. The Plug-in Provider is a human-readable name identifying your company or organization.

Accept the defaults and click **Next**.
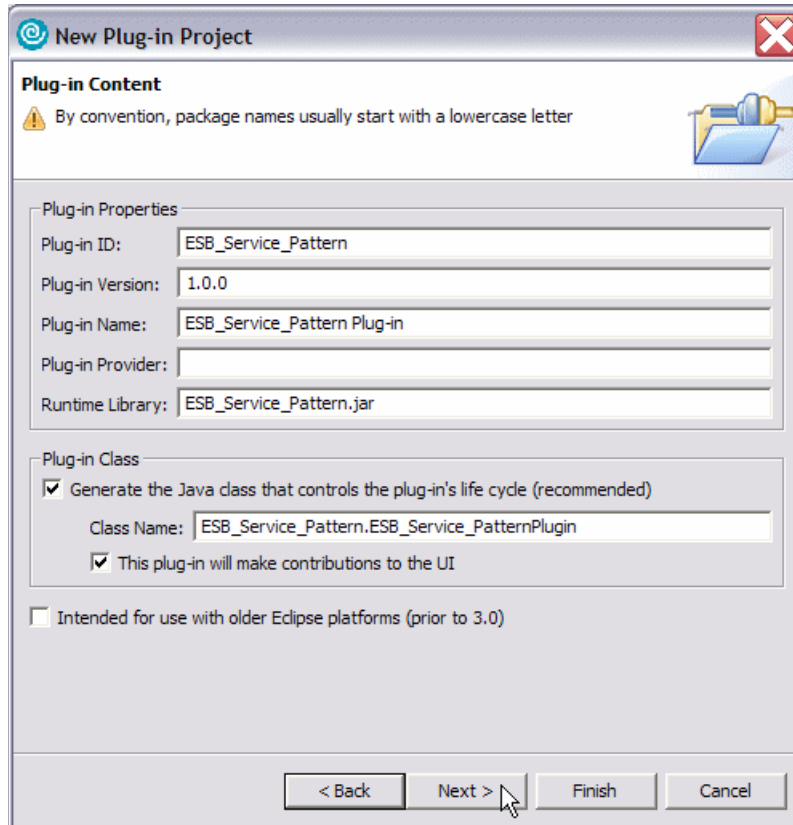
Figure 9-10   Creating a pattern plug-in project: Plug-in properties

4. On the Templates page, select the **Create a plug-in using one of the templates** box, and select **Plug-in with Patterns**. The plug-in template creates a new project with a plug-in manifest and a skeleton pattern library implementation class.

Click **Finish**.
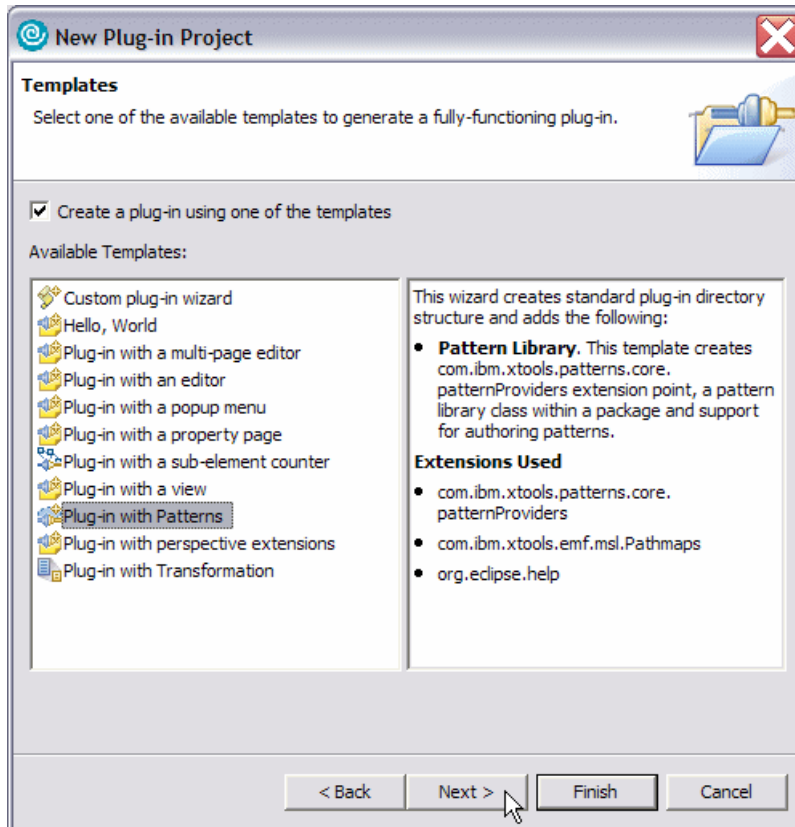
Figure 9-11   Creating a pattern plug-in project: Choosing the template

5. In the Confirm Perspective Switch and Confirm Enablement dialog boxes, select **Yes**.

Figure 9-12 illustrates how the pattern plug-in project looks after the project has successfully created.
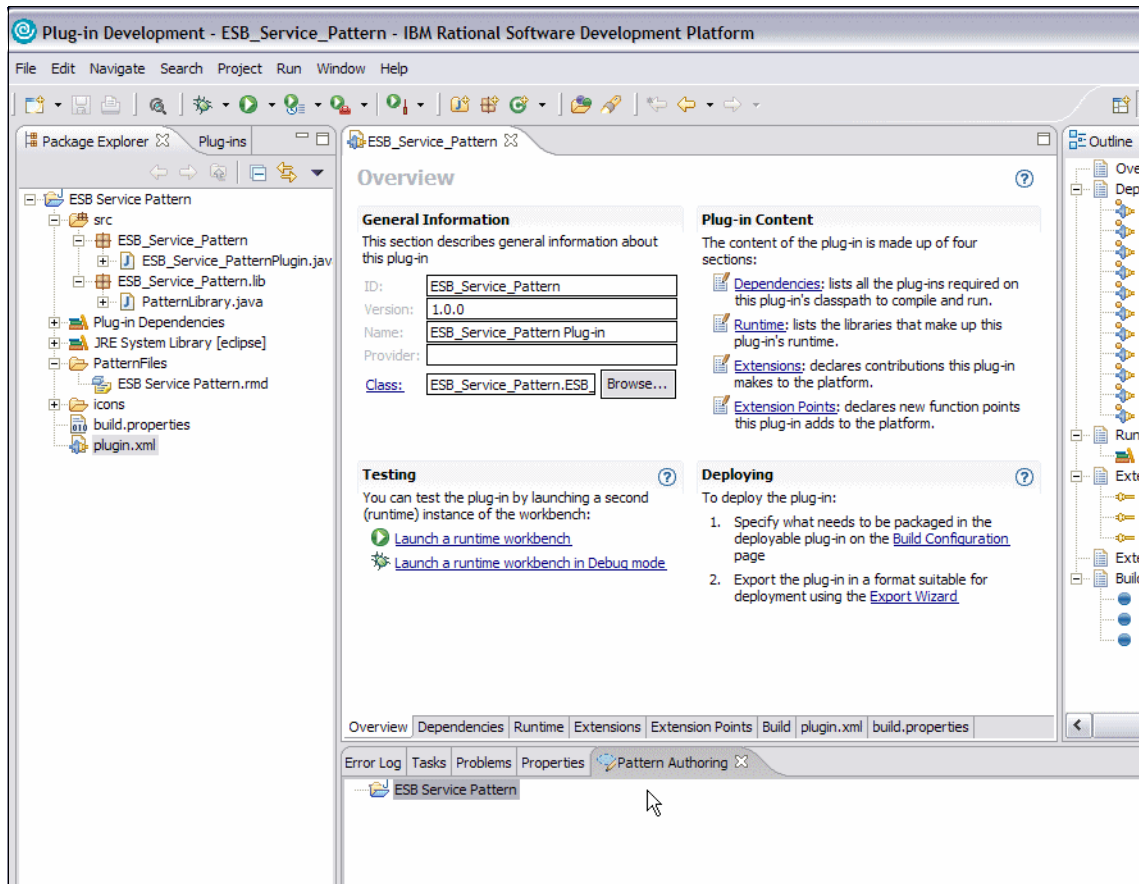
Figure 9-12   Creating a pattern plug-in project

In this new plug-in project, a Java-based pattern framework is formed. The framework provides functions for the pattern author to use in defining the pattern's behavior.

The basic code created in this framework, known as the implementation model, includes:

► Pattern library that contains pattern bodies and their parameters
► Pattern bodies, each represented by a Java class
► Java class with nested classes for each parameter
► Parameter classes, each with an empty expansion and update methods to address the addition, removal, or changes of an argument

These expansion and update methods are known as hot spots. The implementation of the pattern's behavior goes into these hot spots. The implementation does not get generated automatically; therefore, authors must code the behavior manually.

In RSA, a pattern authoring view is provided as a GUI-based tool for pattern library design. By default after the plug-in project is formed, the view is shown at the bottom of the workbench as illustrated in Figure 9-13. Right-click the listed **ESB Service Pattern**.
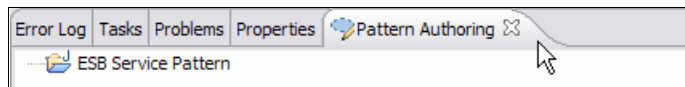
Figure 9-13   Pattern authoring view

The following menu appears.
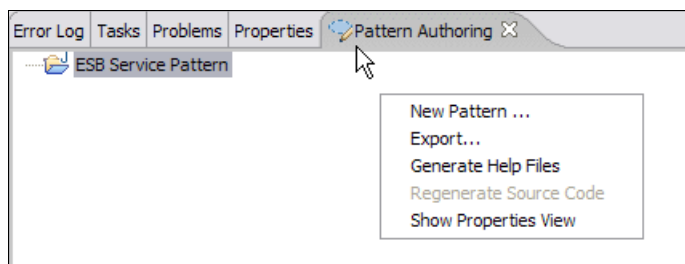
Figure 9-14   Pattern authoring view menu

There are five items on this menu:

► New Pattern...

  Brings up a wizard to create a new pattern under this library

► Export...

  Brings up a wizard for exporting the pattern library as a reusable asset

► Generate Help Files

  This generates a standard RSA help documentation for the pattern. When selected, the following items are created:

  – HTML file using information in the pattern library Reusable Asset Specification (RAS) descriptor
  – Table of contents for the pattern library
  – HTML files for each pattern in the library using information in the pattern RAS descriptor

- Table of contents file for each pattern
- References to the table of contents files to the plugin.xml for the library

► Regenerate Source Code

Regenerates the basic code for the pattern library; useful when the code gets out of sync with the static data in the library or manifest files

► Show Properties View

Switches to the Properties View showing the property settings in the pattern library

Next, we create a pattern in the ESB Service pattern library, and then implement the expansion method for the created pattern.

9.4.2 Defining a pattern

A pattern has the following variables:

► Name

► Implementation class: This does not need to exist; a skeleton file will be created if it is not there.

► One or more parameters: The pattern parameters define what the pattern is applied to.

Use the following steps to create a pattern, and add it to the newly created pattern library (that is ESB service pattern in our example).

1. Go to the pattern authoring view, and right-click the new pattern library.

2. Select **New Pattern...**

3. The New Pattern page opens.

   a. Change the field for the Pattern Name to the name of the pattern to be created. In our example it is `Service Connection Pattern`. Notice that the Class Name and Package are updated automatically to reflect the new name.

   b. Three Pattern Types are available to be associated with the new pattern: Collaboration, Package, and Class, which define the UML element type of an instance of this pattern. These types match the UML2 superstructure specification for templates:

      • Collaboration, as its name suggests, is a collaboration between various elements. It does not only apply to collaboration, but can have parameters for almost any type of UML element. This is the most commonly used type and it is what we are going to use in the example.

- Package is typically used to represent architectural structures in a software system, such as layering or a common packaging structure.

- Class is the equivalent of a parameterized class and is seldom used.

c. For all the parameters specified, skeleton code is generated in the pattern implementation class, which can then be extended to include operations on the values supplied for the parameters. Parameters can be added and removed either through this wizard or from the pattern authoring view.

We use the latter option, leaving the parameters text field blank for now. In the next section, we describe how parameters are added and implemented.

By default, new patterns are under the Miscellaneous Patterns group, which means, when the pattern plug-in is published, it will be shown as a member of the Miscellaneous group in the pattern explorer view. For our example, we put the pattern in a new group called "Service Patterns".

Next to the Groups text box, click the **Add** button.

Enter `Service Patterns` as the group name, and then click **OK**.

d. The version field can be left unfilled for this example, but note that this field does become important when the pattern is packaged as a RAS asset, where it becomes the version of the asset. Similarly for the Details tab, the information provided on this page populates the RAS manifest for the pattern as well as for its documentation, but it can be left empty for this example.

Figure 9-15 illustrates how the page now looks.

The Detail tab allows you to add a description of the new pattern.

Click **OK**. Now we have a pattern which we will implement shortly.
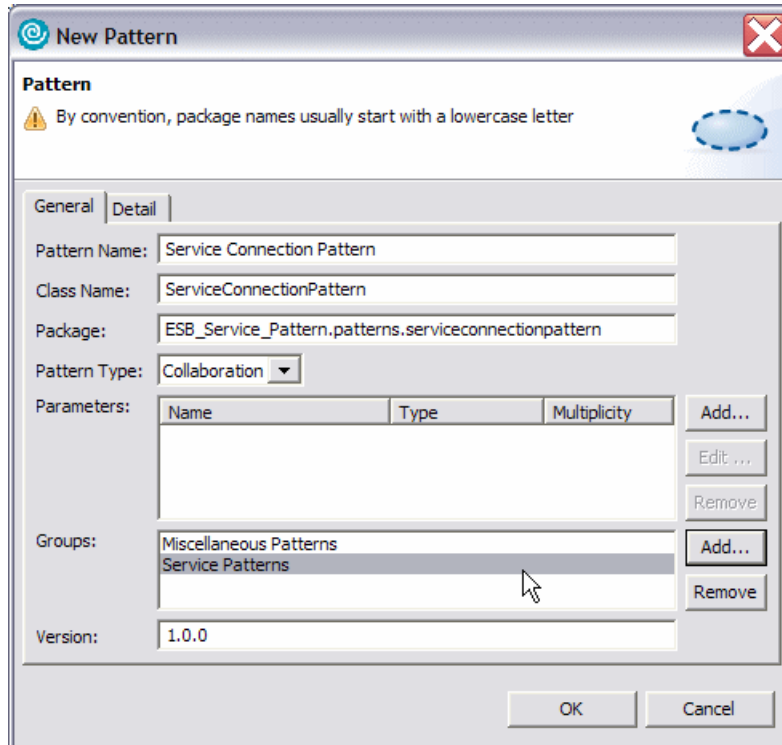
Figure 9-15   Defining a new pattern

The pattern project contains the following directories and files:

► Java code classes

 Stored under the *src* directory. Classes are created automatically for the plug-in project and pattern library. In our example they are under the ESB_Service_Pattern and ESB_Service_Pattern.lib packages respectively. Individual classes are also generated for each pattern added in the library. They will appear under the following: ESB_Service_Pattern.patterns.<Pattern name> package

► PatternFiles directory

 This default directory contains the following: a RAS manifest documentation (.rmd) file for the library and each pattern, a pattern model file that contains the pattern definition, and if generated, Pattern HTML help files with supporting XML file.

► Icons directory

 By default it includes one sample icon, the sample.gif

► Plug-in files

A build file (build.properties) and plug-in XML file (plugin.xml) are used when the pattern is exported to create the pattern plug-in on RSA. By default, the pattern source code is not included when the plug-in is created by using the RAS export.

► Overview diagram graphic

A pattern author can include an optional GIF file that shows a high-level representation of pattern components for the pattern applier.

9.4.3 Pattern implementation

For details about the pattern we are implementing and the full example, refer to Chapter 5, "Model-driven development solution life cycle" on page 59. To quickly see the relationships, here is the diagram we used in Chapter 5 to describe the pattern.
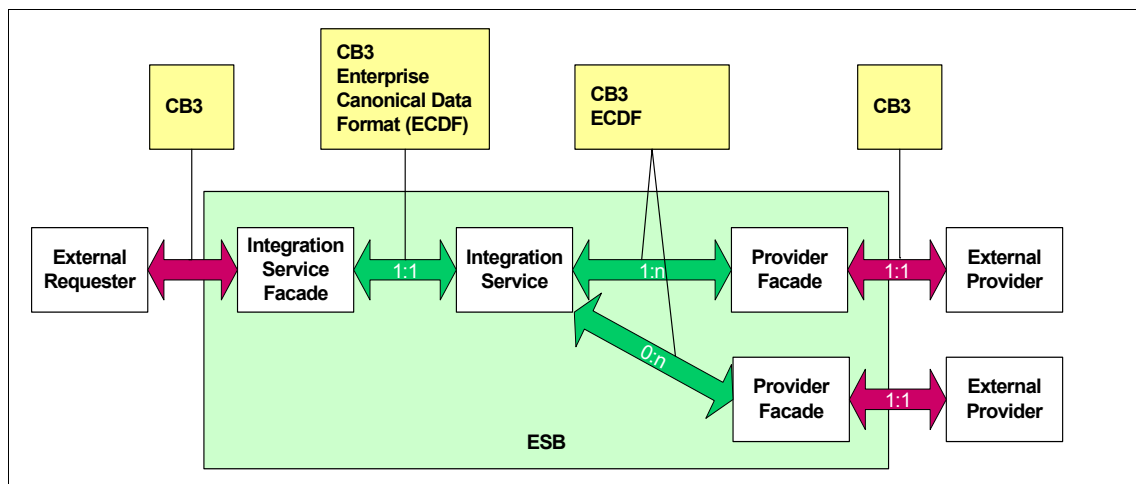
Figure 9-16   Pattern Combining the ESB architecture, integration behavior and contract of behavior

We implement what is inside the box labelled as "ESB". The ESB is a collaboration in the UML model. When the pattern is applied to the collaboration, it creates these four components:

► Integration service facade (ISF)
► Integration service (IS)
► Two provider facades (PF1 and PF2)

There are connectors between these components inside the collaboration, which use the ports defined in each component.

Implementing the expansion methods

The basic function of a pattern is to expand the model elements that are bound to its parameters. This means to add additional information, model elements, and relationships between elements. To implement the expansion methods, we need to define a parameter and add it to the pattern first.

Our service connection pattern has only one parameter, which is of type collaboration (that is, the ESB). Use the following instructions to define a parameter.

1. From the pattern authoring view, select **Windows** → **Show view** → **Other...** → **Modeling** → **Pattern Authoring**.

2. Select the pattern of interest. In this example, we select **Service Connection Pattern**. Right-click, and select **New Parameter...**.

3. In the Parameter panel (Figure 9-17), name the parameter `ESB Collaboration`. Select **Collaboration** as the type. The multiplicity is 1.
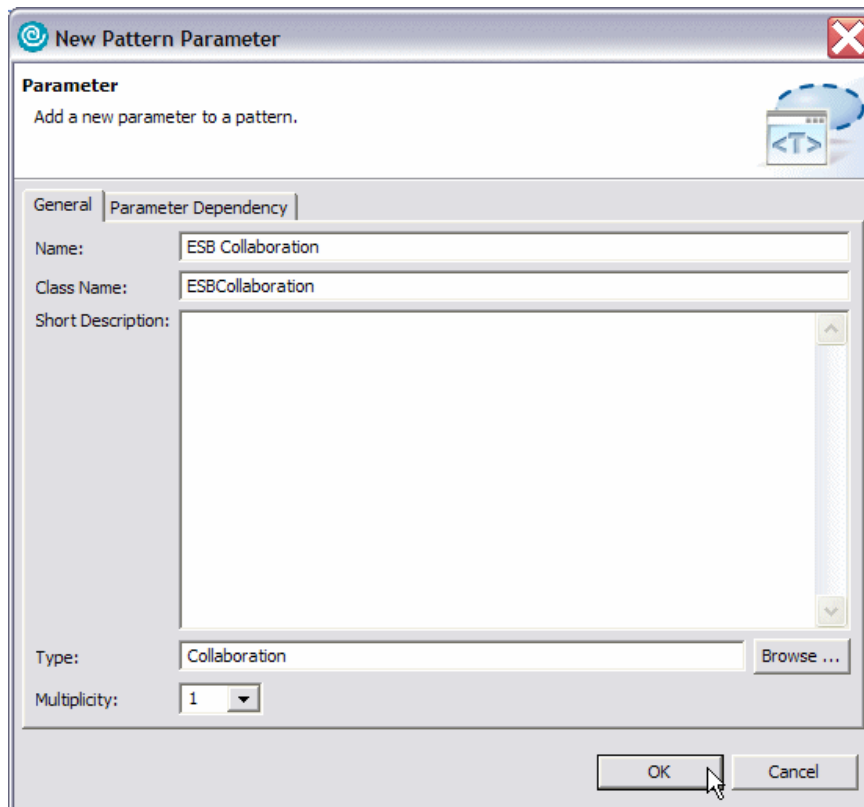
Figure 9-17   Defining a pattern parameter

Let's look at the changes made in the pattern implementation class (ESB_Service_Pattern.patterns.serviceconnectionpattern.ServiceConnectionPattern.java). You can find the class under **ESB Service Pattern** → **src** in the Package Explorer view.
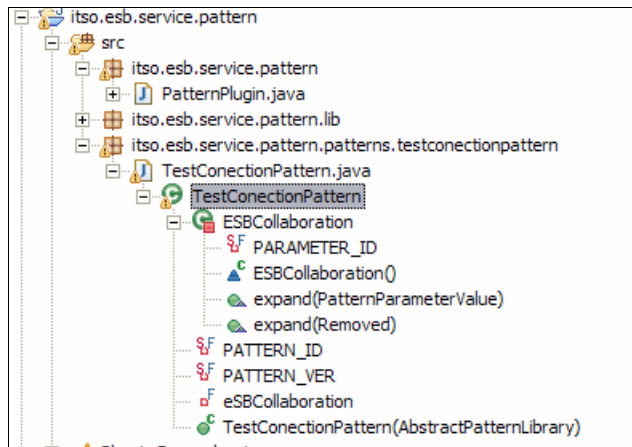
Figure 9-18   Exploring a newly created pattern

An inner class is generated for the specified parameter, ESBCollaboration, which contains the expand methods required for the implementation. Example 9-3 shows their signatures.

Example 9-3   Expansion methods signatures

```
public boolean expand(PatternParameterValue value);
public boolean expand(PatternParameterValue.Removed value);
```

The first of these methods must be implemented. The object passed as an argument to this method is a wrapper for the actual parameter value, which is a Collaboration in our example.

The second method is called when the association between the created model and pattern is removed. It is optional to implement it. However, in our example, we are not going to because it makes more sense to let the generated components remain in the model for reusability.

There might be warnings on some imported packages not being used, but you can safely ignore them.

To implement the pattern, you need a good understanding of the UML2 API. Information about UML2 can be found at the following Web site:

http://www.eclipse.org/uml2

Also refer to the Patterns framework API reference in the RSA help for details of how to use these classes. Look under **Help → Help Contents → Extending Rational Software Architect functionality → Extensibility reference → API Reference**.

In the following descriptions of the sample code, we assume that you already have some knowledge of UML2 API.

Sample code for ESB service pattern

First, we look at how to create the four components within the ESB collaboration model. A component is a type of attribute (property in UML terms). Therefore we need to create four property objects first and then set their types to Component.

Creating the property objects

Inside the generated *expand*() method, replace the entire contents of Example 9-4 with the contents in Example 9-5.

Example 9-4   Generated basic expand() body

```
/**
 * This is the default general expand method that is typically implemented
 * by concrete pattern parameters for handling the added and maintained
 * expand variants which are usually similar.
 *
 * The default behavior is to return true.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public boolean expand(PatternParameterValue value) {
    //TODO : implement the parameter's expand method
    return true;
}
```

Example 9-5   Creating properties and components in expand()

```
/**
 * This is the default general expand method that is typically implemented
 * by concrete pattern parameters for handling the added and maintained
 * expand variants which are usually similar.
 *
 * The default behavior is to return true.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 *
 */
```

```java
public boolean expand(PatternParameterValue value) {
    Collaboration collValue = (Collaboration) value.getValue();

    // Creating the Integration Facade Service (IFS) component
    Property collIFSProperty = createProperty(collValue, "IFS");
    Component ifsComp = createComponent(collValue, collIFSProperty,
                                        collValue.getName() + "IFS");

    // Creating the Integration Service (IS) component
    Property collISProperty = createProperty(collValue, "IS");
    Component isComp = createComponent(collValue, collISProperty,
                                        collValue.getName() + "IS");

    // Creating the first Provider Facade Service (PFS1) component
    Property collPFS1Property = createProperty(collValue, "PFS1");
    Component pfs1Comp = createComponent(collValue, collPFS1Property,
                                        collValue.getName() + "1PFS");

    // Creating the second Provider Facade Service (PFS2) component
    Property collPFS2Property = createProperty(collValue, "PFS2");
    Component pfs2Comp = createComponent(collValue, collPFS2Property,
                                        collValue.getName() + "2PFS");

    return true;
}
```

Tip: Many of the types, such as Property, are unresolved because the necessary uml packages are not imported. To fix the problems, point at the red cross in the left margin of the source editor, and select the uml package to import for each type.

Notice that the Javadoc™ @generated tag was removed because the implementation model is regenerated each time the pattern library is modified. If the tag is not removed, the modifications are overwritten when the code is regenerated.

Implementing the utility methods

Next we implement the utility methods used in the *expand*(), namely *createProperty*() and *createComponent*(). Place these immediately before the `public boolean expand(PatternParameterValue value)` method.

Example 9-6   createProperty() utility method

```
private Property createProperty(Collaboration coll, String propertyName) {
    Property collProperty =
            coll.createOwnedAttribute(UML2Package.eINSTANCE.getProperty());
    collProperty.setName(propertyName);
    return collProperty;
}
```

Example 9-7   createComponent() utility method

```
private Component createComponent
        (Collaboration coll,Property collProperty, String compName) {
    Component aComp = UML2Factory.eINSTANCE.createComponent();
    aComp.setName(compName);

    collProperty.setType(aComp);

    //Adding the created components as separate entities to the model package
    coll.getPackage().getOwnedMembers().add(aComp);

    return aComp;
}
```

Now we are ready to create the connectors for the following component pairs:

- ► IFS → IS
- ► IS → PFS1
- ► IS → PFS2

Associating the components to the in and out ports

To connect the components, they need to have the *in* and *out* ports. A port object represents the UML model object *port*, it is the interaction point between a classifier and its environment. An *out* port represents the side where the request is originated and the *in* port represents the receiving end.

Add the code in Example 9-8 to the *createComponent*() utility method, before the call to *collProperty.setType(aComp)*.

Example 9-8   Create Ports for each components

```
Port inPort = aComp.createOwnedPort(UML2Package.eINSTANCE.getPort());
Port outPort = aComp.createOwnedPort(UML2Package.eINSTANCE.getPort());
inPort.setName("in");
outPort.setName("out");
```

Each component that is created now has an *in* and *out* port associated. Next we create the connectors to connect the *out* ports to the *in* ports.

Utility method for creating connectors
Add the code in Example 9-9 at the end of the *expand*() method, before the *return true* call.

Example 9-9   Create Connectors for the component pairs

```
//Connector for IFS -> IS
Connector ifsOutPortToIsInPort = createConnector(collValue,
                        collIFSProperty, ifsComp, collISProperty, isComp);
//Connector for IS -> PFS1
Connector isOutPortToPfs1InPort = createConnector(collValue,
                        collISProperty, isComp, collPFS1Property, pfs1Comp);
//Connector for IS -> PFS2
Connector isOutPortToPfs2InPort = createConnector(collValue,
                        collISProperty, isComp, collPFS2Property, pfs2Comp);
```

Implement the utility method *createConnector*().

Example 9-10   createConnector() utility method

```
private Connector createConnector
        (Collaboration coll, Property outPortProperty,
         Component outComp, Property inPortProperty, Component inComp) {
    Connector aConnector =
        coll.createOwnedConnector(UML2Package.eINSTANCE.getConnector());
    ConnectorEnd outPort =
        aConnector.createEnd(UML2Package.eINSTANCE.getConnectorEnd());
    ConnectorEnd inPort =
        aConnector.createEnd(UML2Package.eINSTANCE.getConnectorEnd());
    outPort.setPartWithPort(outPortProperty);
    outPort.setRole(outComp.getOwnedPort("out"));
    inPort.setPartWithPort(inPortProperty);
    inPort.setRole(inComp.getOwnedPort("in"));
    return aConnector;
}
```

Applying the service provider stereotype

Finally we need to apply the service provider stereotype to the four components. The service provider stereotype is defined in *UML 2.0 Profile for Software Services*, which you must install as a plug-in at the runtime environment. For information about this profile, go to:

http://www.ibm.com/developerworks/rational/library/05/419_soa/

We also have to make sure that this profile was applied to the model before we can apply the stereotype. In our *expand*() method, we perform a check whether the Software Services profile was applied. If it is not, we apply it.

Near the beginning of the *expand*() method, locate the following line:

```
Collaboration collValue = (Collaboration) value.getValue();
```

After this line, add the code in Example 9-11.

Example 9-11   Apply software services profile

```
//Applying the UML 2.0 Profile for Software Services
//Get the installed profile
      Resource resource = collValue.eResource().getResourceSet().getResource(
URI.createURI("pathmap://SOFTWARE_SERVICES/profiles/SoftwareServices.epx"),
true);
Profile softSrvProfile = (Profile) EcoreUtil.getObjectByType(
               resource.getContents(), UML2Package.eINSTANCE.getProfile());
//Check if the profile has been applied, if not, apply it
if (false == collValue.getModel().isApplied(softSrvProfile)) {
   collValue.getModel().apply(softSrvProfile);
}
```

The full qualified URL name used in *createURI*() was obtained from the plugin.xml of the installed profile.

1. To open the plugin.xml file, go to **Windows** → **Show View** → **Other...**.

2. From the Show View list, select **PDE** → **Plug-ins**.

3. After the Plug-in view is opened, go to the package where the profile is installed, for the Software Services profile the package name is **com.ibm.rational.softsvc**. The plugin.xml is listed under the package name.

4. Open the plugin.xml and look at the source, the URL name is defined in:

   ```
   <extension><UMLProfile path="......"></UMLProfile></extension>
   ```

   Here "......" is the qualified URL name. In our case, it is defined as "pathmap://SOFTWARE_SERVICES/profiles/SoftwareServicesSecurity.epx".

Next comes the final part of the implementation for our service connection pattern, applying the service provider stereotype.

Add the code shown in Example 9-12 to the end of *expand*() method, just before
the *return true;* line.

Example 9-12   Applying the service provider stereotype

```
//Get the service provider stereotype and apply it to the four components
Stereotype sTypeObj = ifsComp.getApplicableStereotype(
                                   "Software Services Profile::ServiceProvider");
if (null != sTypeObj) {
    ifsComp.apply(sTypeObj);
    isComp.apply(sTypeObj);
    pfs1Comp.apply(sTypeObj);
    pfs2Comp.apply(sTypeObj);
} else {
    System.out.println("The Stereotype ServiceProvider cannot be found from the
Software Services Profile");
}
```

Example 9-13 contains the list of imported classes required.

Example 9-13   Imported classes list

```
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.util.EcoreUtil;
import org.eclipse.emf.common.util.URI;
import org.eclipse.uml2.Collaboration;
import org.eclipse.uml2.Component;
import org.eclipse.uml2.Connector;
import org.eclipse.uml2.ConnectorEnd;
import org.eclipse.uml2.Port;
import org.eclipse.uml2.Profile;
import org.eclipse.uml2.Property;
import org.eclipse.uml2.Stereotype;
import org.eclipse.uml2.UML2Factory;
import org.eclipse.uml2.UML2Package;
```

9.4.4 Testing the pattern

Use the following steps to test the pattern.

1. Launch a runtime workbench. See 9.6, "Launching a Run-time Workbench"
   on page 215, if you do not know how to do this.

2. Once the runtime workbench starts, switch to the Modeling perspective.

3. You need a UML model to apply the pattern to. Create a new model project by
   going to **New** → **Project** → **Modeling.**

4. Select **UML Project** and click **Next**.

5. Fill in the project name `ESB Service Pattern Test` and click **Next**.

6. Change Blank Model to `My Simple Model`, as shown in Figure 9-19, and click **Finish**.
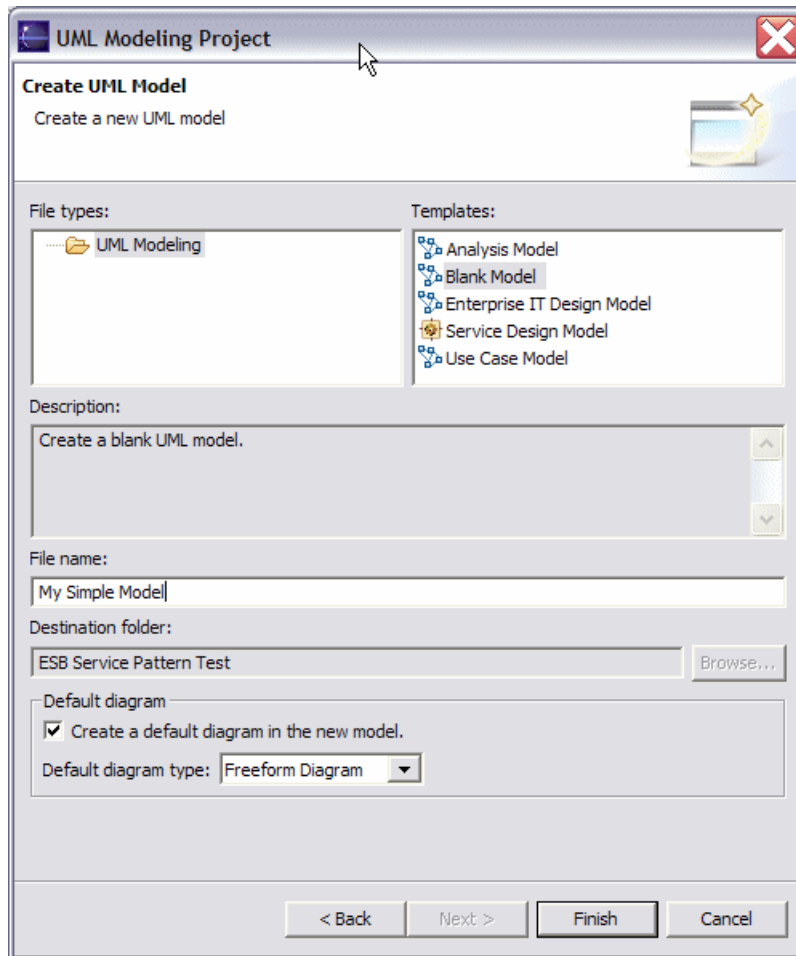
Figure 9-19   Create a new UML model project

7. From the Model Explorer, expand **ESB Service Pattern Test** → **My Simple Model.emx** → **My Simple Model**.

8. Select **My Simple Model** and double-click the freeform diagram **Main**.

9. After the My Simple Model Main freeform diagram opens, open the Pattern Explorer view again, and
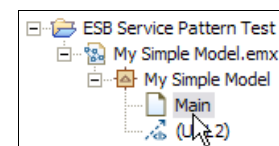
drag the **Service Connection Pattern** onto the diagram, as shown in Figure 9-20.
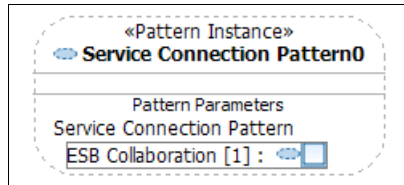
Figure 9-20   Service Connection Pattern added to the model

Applying the pattern

Use the following steps to apply the pattern.

1. Open the **Composite Structure Diagram** → **Collaboration**.

2. Click in the white space on the freeform diagram. A collaboration shape is now shown on the diagram. Notice the newly created collaboration is listed under **My Simple Model** on the Model Explorer. It is also on the properties view.
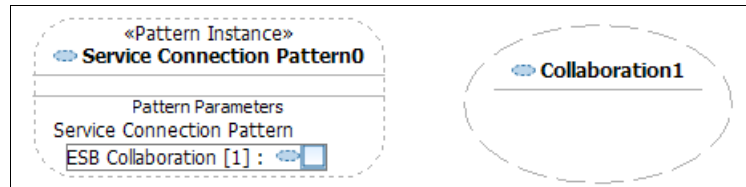
Figure 9-21   Model with a collaboration added

3. On the Properties view, rename **Collaboration1** to `ESB Service Collaboration`.

4. To apply the pattern, drag the **ESB Service Collaboration diagram** to the box labeled as ESB Collaboration[1] to the Service Connection Pattern instance.
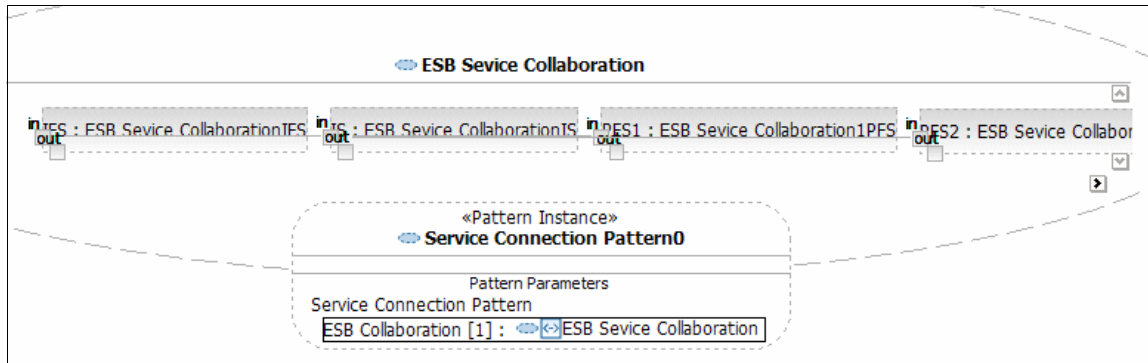
Figure 9-22   Service connection pattern applied

The created components are now listed in Model Explorer.
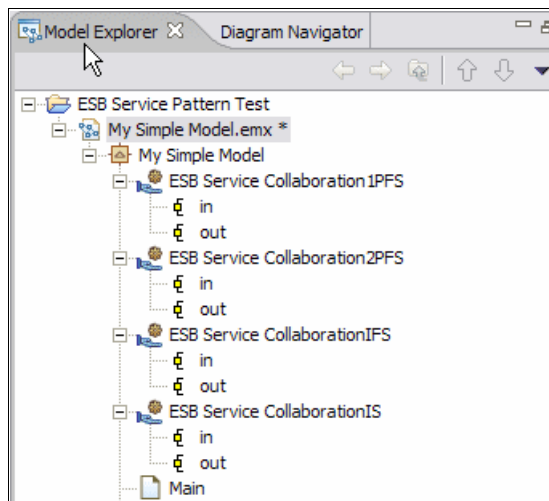
Figure 9-23   Model structure after the service connection pattern is applied

After the pattern is applied, ensure that the profile was applied to the model:

5. Double-click **My Simple Model.emx**, which opens up the model on a UML Model Editor.

6. On the editor, look at the Applied Profiles section, and you see that Software Services Profile is listed there.

7. Also, check to see if the stereotypes from the profile were applied to the model elements. You can do it by looking into the Properties of each created component.

   a. Right-click the component, and select **Show Properties View**.

   b. On the Properties view, click **Stereotypes** on the left side panel, and look under Applied Stereotypes. The ServiceProvider stereotype from the Software Services Profile is listed there.

9.4.5 Publishing patterns

You can publish a pattern by exporting it as a deployable plug-in. See 9.7, "Deploying plug-ins" on page 217, for instructions on doing this.

9.5 Implementing a transformation

As discussed in Chapter 8, "Applying model-driven development with Rational Software Architect" on page 129, you can apply a transformation to a model using the Modeling perspective Modeling → Transform menu item. To add your own transformation to this menu you need to create an Eclipse plug-in that contributes to the transformation provider extension point. A plug-in project template is available to set up the manifest and basic classes for you.

This section outlines how we implemented the sample transformation for the scenario discussed in previous chapters. It includes an introduction to the RSA transformations API together with some general techniques for code generation. Detailed instructions for implementing the sample transformation are not given here. However, you might refer to the source code for the sample transformation, included in the additional materials.

9.5.1 Creating a new plug-in with a transformation

1. From the main menu, select **File** → **New** → **Project...** → **Plug-in Development** → **Plug-in Project**.

2. The Plug-in Project panel opens. A plug-in must have a unique identifier; the name of a plug-in project defaults to the plug-in identifier. Because it must be unique, the identifier usually takes the form of a Java package name. This may be a Java package name if the plug-in happens to include Java source code, but it does not matter. The identifier must be unique.

   The transformation plug-in includes Java code, so select the **Create a Java project** option and click **Next**.
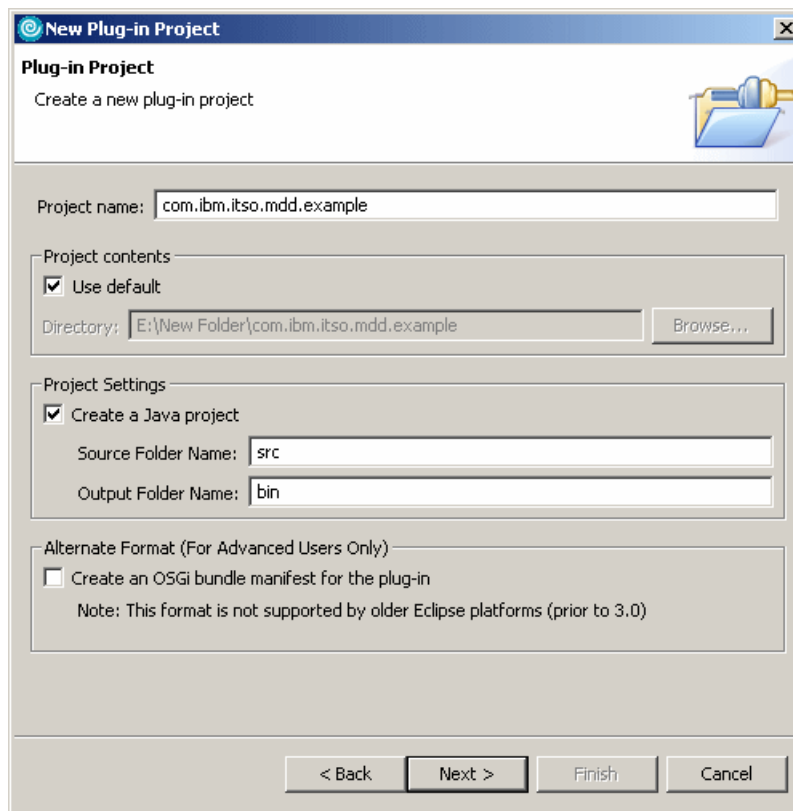
Figure 9-24   New plug-in project wizard

3. The Plug-in Content panel (Figure 9-25) opens. On the plug-in properties, the plug-in ID defaults to the project name. The Plug-in Name is a human-readable name for the plug-in, and the Plug-in Provider is a human-readable name identifying your company or organization.

Accept the defaults and click **Next**.

Figure 9-25   New plug-in properties

4. In the Templates panel (Figure 9-26), select a template that generates some of the basic classes you need for the plug-in. Select the **Plug-in with Transformation** template and click **Next**.
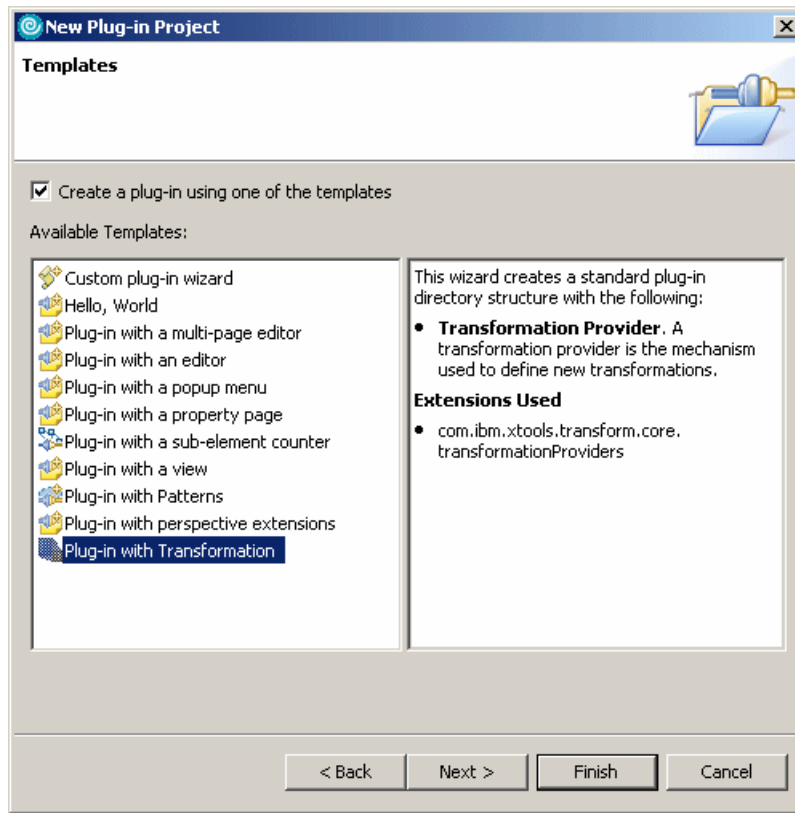
Figure 9-26   New plug-in templates

5. In the New Transformation Provider window (Figure 9-27), you can choose the Java package and class name to be used for one of the main generated classes, the transformation provider. This is essentially the entry point for your transformation. The transformation provider class is declared in the plug-in manifest, so if you decide to rename it later, remember to also change it there.

Click **Next**.
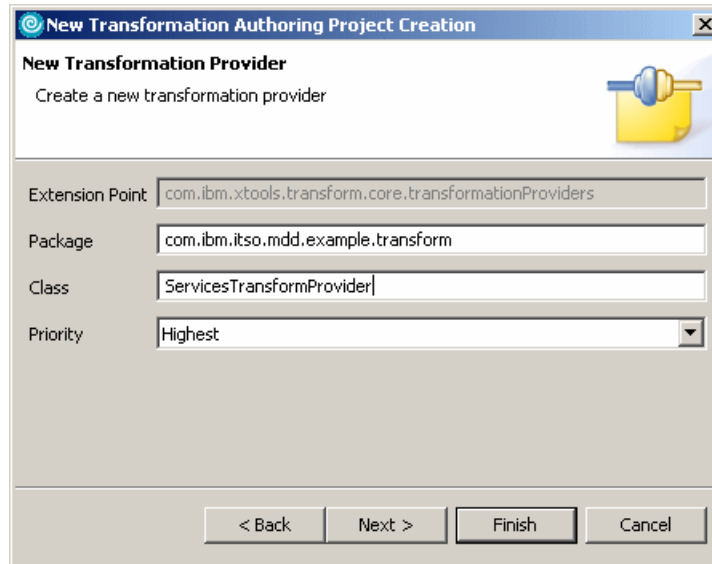
Figure 9-27   Transformation provider properties

6. The New Transformation panel (Figure 9-28) provides information you can use to generate a skeleton transformation class. A transformation provider can provide one or more transformations, each implemented by a Java class.

7. Each transformation declared in the plug-in manifest appears in the Configure Transformations wizard, which you find under **Modelling** → **Transform** → **Configure Transformations...**. The name, group path and description options determine how the transformation appears in the wizard.

The source and target model type options influence how the user selects the source and target for the transformation. The type *Resource* means anything other than a UML model.

Notice that at the bottom of this page, the **Use default UML2 Transformation framework** option is selected. This setting determines what style of skeleton transformation code is generated. For the sample transformation, we are going to use this default style.

Click **Next**.

Figure 9-28   Transformation properties

8. Because the default UML2 transformation framework is being used, the New Rule Definitions panel (Figure 9-29) allows you to define specific rule classes to be associated with particular UML element types. This information is used to generate skeleton code for the transformation. We define the rules later.

Click **Finish** to generate the plug-in project.
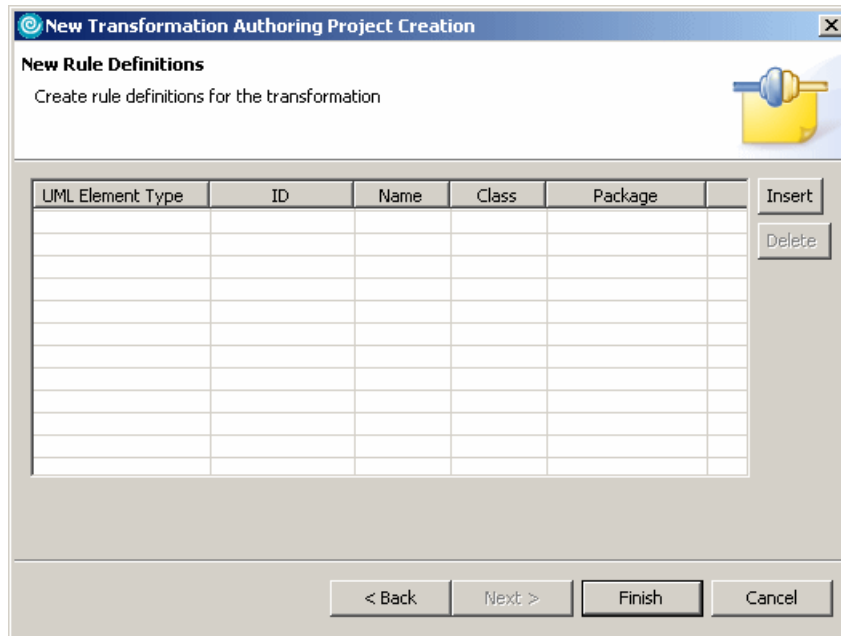
Figure 9-29   Transformation rules

Figure 9-30 shows the files that are generated in the new project. The classes that are generated are the plug-in class *ExamplePlugin.java*, the transformation provider *ServicesTransformProvider.java* and the root transformation *ServicesTransformation.java*.

Any plug-in can have a plug-in class. This is a singleton that is instantiated when the plug-in is activated. Commonly it is used as a convenient place to put utility methods as well as status or configuration properties that apply to the plug-in as a whole.
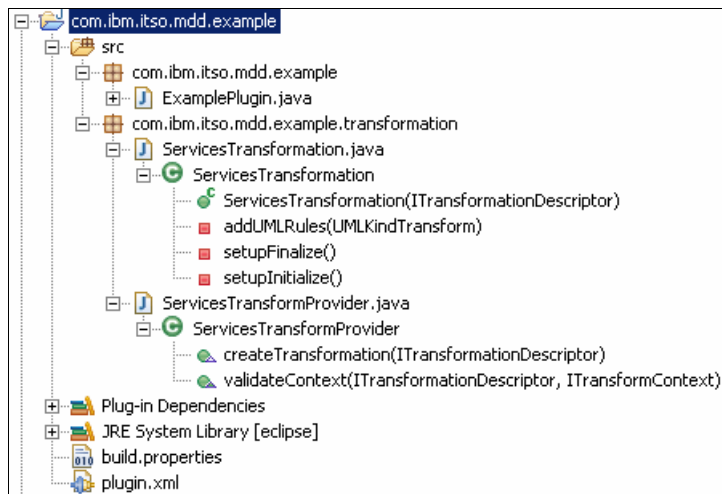
Figure 9-30   New transformation plug-in project contents

Transformation provider

The plug-in manifest, plugin.xml, declares the transformation provider class. It extends the abstract base class, AbstractTransformationProvider, and lists the transformations that it can perform. The transformations listed here appear in the Configure Transformations wizard, which you find under **Modelling** → **Transform** → **Configure Transformations...Modeling** → **Transform**, when the plug-in is deployed. When a user selects one of those transformations, it is the transformation provider that is responsible for validating the transformation context (the source and target objects and any properties) and constructing a root transformation that performs the transformation.

The generated transformation provider is set up to handle a single transformation and does no special validation of the transformation context.

Root transform

The transformation provider *createTransform* method is required to return an instance of the class AbstractTransform. This is an abstract class, so in practice you must return an instance of one of the sub-classes of AbstractTransform.

The transformation plug-in template creates a class that extends RootTransform. It is an instance of this that is constructed by the generated transformation provider.

To understand what the RootTransform class does, we must quickly review the main classes from the RSA transformation API.

9.5.2 Transformation API

JavaDoc for the transformation API can be found in the RSA help. The transformation API consists of four packages, but only two of these are of immediate interest: com.ibm.xtools.transform.core and com.ibm.xtools.transform.uml2.

Package: com.ibm.xtools.transform.core

The important classes and interfaces in this package are shown in Figure 9-31.
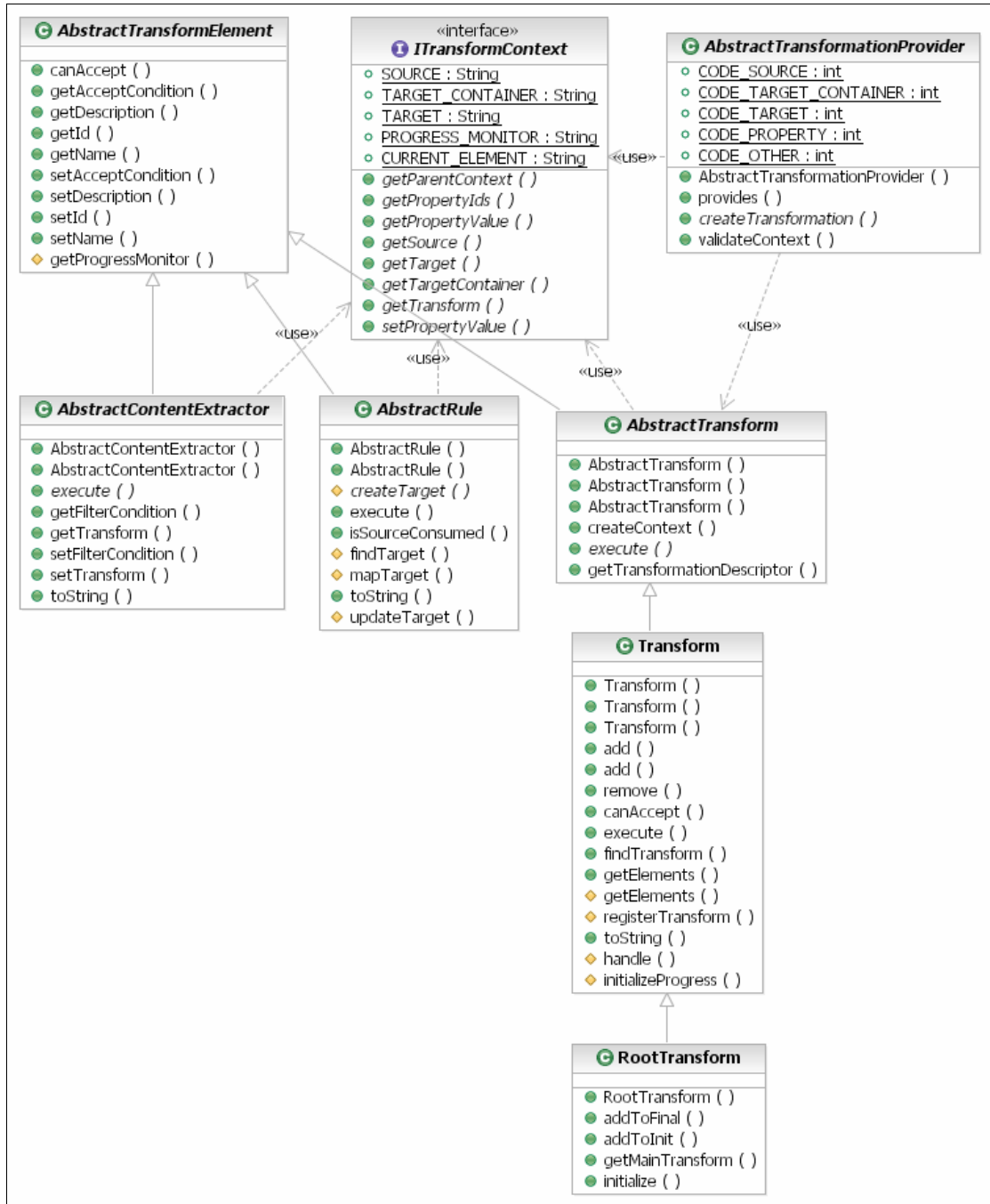
Figure 9-31   Principal classes in com.ibm.xtools.transform.core

AbstractTransform

AbstractTransform is the abstract base class for all transformation implementations. It has one abstract method:

```
public void execute(ITransfomContext context);
```

It is not usually necessary to directly extend AbstractTransform. The concrete sub-classes Transform and RootTransform, together with UMLTransform and UMLKindTransform from the package com.ibm.xtools.transform.uml2, are sufficient for most purposes.

ITransformContext

ITransformContext is the interface to the transform context. The transform context allows access to the transformation source object and its target container (where the target object or objects are to be created) and target, if these values are set. It can also contain other properties as name-value pairs passed into the transformation. You can use this facility to pass information between different parts of the transformation code.

You do not need to implement this interface or write code that constructs transform contexts. Transform contexts are created by the framework as required. A root transform context is passed into your root transform when the transformation is run. The framework classes create additional transform contexts, forming a stack. Each context has a parent context ultimately linking back to the root transform context.

Transform

Instances of the class Transform implement the execute method by iterating over a collection of transforms, rules, and extractors and calling their execute methods.

A transform is an instance of a sub-class of AbstractTransform. Any collection of transforms, rules and extractors are executed by a Transform. Different Transform instances can execute the same rules, and a Transform is allowed to execute itself.

AbstractRule

A rule is an instance of a sub-class of AbstractRule. Rules know how to create or modify a target object given a source object (or can do any other required processing). Rules perform most of the real work of the transformation. It is in rules that you write the code that generates the transformation target artifacts.

AbstractContextExtractor

An extractor is an instance of a sub-class of AbstractContextExtractor. An extractor knows how to extract a set of source model elements for further processing. It references an associated transform that is applied to the extracted source elements.

RootTransform

RootTransform is a sub-class of Transform. It allows you to define three phases of execution—an initial, main, and final phase. One transform is executed in the main phase, and you can specify any set of rules, extractors, and transforms for the initial and final phases.

Package: com.ibm.xtools.transform.uml2

This package contains some classes intended specifically for working with UML2 models. The most interesting class here is UMLKindTransform. Figure 9-32 shows how UMLKindTransform extends Transform via UMLTransform.
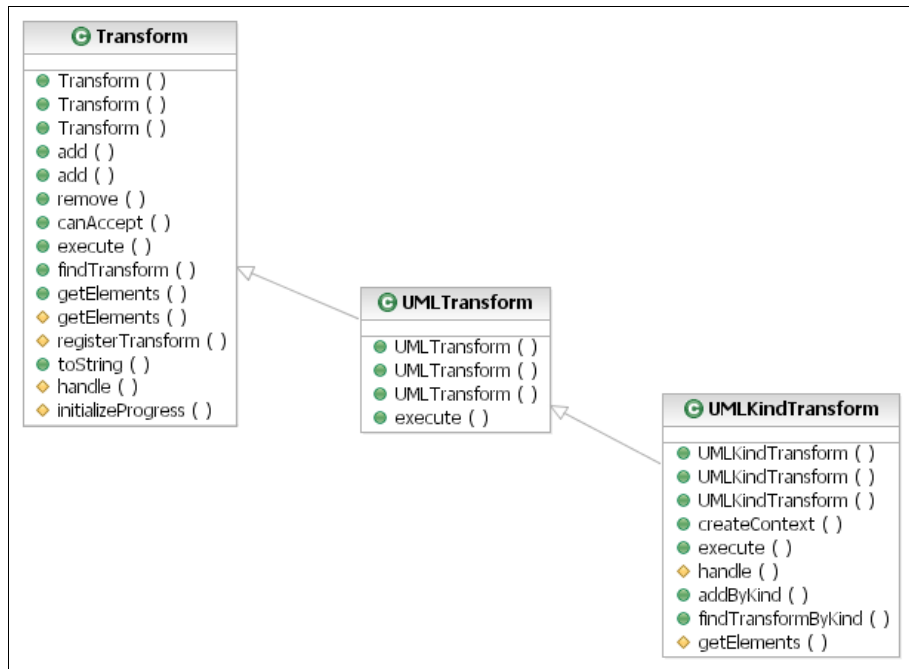
Figure 9-32   UMLKindTransform

UMLKindTransform

UMLKindTransform is also a sub-class of Transform (indirectly) but executes quite differently. It has a set of rules, extractors, and transforms each of which is associated with a type of UML model element (Class, Property, Activity, etc.). It walks over a source UML model, or part of a model, and for each source element it executes all the rules, extractors, and transforms registered for that type of model element.

Using a UMLKindTransform makes it simple to implement UML transformations, as you just need to implement rules for each kind of model element to be processed, but will not give the best performance. Bear in mind that the order in which rules are executed is partly determined by the order of elements in the source model (it is also influenced by the order in which the rules were added to the UMLKindTransform).

When you use UMLKindTransform, you can't use the transformation extensibility mechanism to allow other plug-ins to extend your transform with new rules.

UMLTransform

UMLTransform is a base class for transforms that read or modify UML models. It behaves exactly like Transform. If you do not want to use UMLKindTransform but want a transform that read or updated a UML model, you use this class directly.

9.5.3 Implementing the root transformation

The default generated root transformation class is a sub-class of RootTransform, which sets up a UMLKindTransform to be executed in its main phase.

Example 9-14   Generated root transformation constructor

```
/**
   * Constructor.
   * @param descriptor  A transformation descriptor.
   */
  public ServiceTransformation(ITransformationDescriptor descriptor) {
    super(descriptor);

    // Initialize root transform with main UML transform.
    // The false argument indicates the main transform will only
    // accept individual source objects (and not the list of
    // selected objects).
    UMLKindTransform umlkindTransform = new UMLKindTransform(descriptor);
    initialize(umlkindTransform, false);

    // Define the rules to be executed before and after the main
    // UML transform.
    setupInitialize();
```

```
        setupFinalize();

        // Add the rules to the main UML transform.
        addUMLRules(umlkindTransform);
}
```

The generated methods setupInitialize, setupFinalize, and addUMLRules are empty placeholders to add your own code. setupInitialize and setupFinalize are, as the names suggest, where any processing to be performed before or after the main transformation is defined. No initial or final phase processing is needed for the example.

In addUMLRules, define the rules to be used in the main transformation and associate them with types of UML element. The rules are called each time a UML element of the specified kind is found in the model.

In the example, we want to generate the artifacts listed in Table 9-1.

Table 9-1   Artifacts required for example transformation

UML element	Generate...
Component (Service Provider)	EJB (EAR and EJB project, EJB bean class)
Interface (Service Specification)	Java interface for EJB
Class (Data classes)	XML Schema
Collaboration	Script to set up JNDI namespace bindings

To create the artifacts we set up rules associated with UML Components, Interfaces, and Collaborations. To process the UML Classes which represent data, we associate the rule with UML Packages rather than Classes. That is because we generate one XML Schema file for all the Classes in one Package, not one XML Schema file for each Class.

Table 9-2 shows the rule classes associated with each UML element. Each of the rule classes extend AbstractRule.

Table 9-2   Rule classes associated with UML elements

Class	UML element
ServiceProviderRule	Component
ServiceSpecificationRule	Interface
DataRule	Package
CollaborationRule	Collaboration

Example 9-15 shows the implementation of the addUMLRules method in the root transformation.

Example 9-15   addUMLRules implementation

```
/**
 * Add rules to the main UML transform where each rule is
 * associated with a UML language element kind.
 *
 * @param transform  The main UML transform.
 */
private void addUMLRules(UMLKindTransform transform)
{
    DataRule dataRule =
        new DataRule("data", "Data Rule");
    transform.addByKind(UML2Package.eINSTANCE.getPackage(), dataRule);

    ServiceSpecificationRule specRule =
        new ServiceSpecificationRule("spec", "Service Specification Rule");
    transform.addByKind(UML2Package.eINSTANCE.getInterface(),specRule);

    ServiceProviderRule componentRule =
        new ServiceProviderRule("comp", "Service Provider Rule");
    transform.addByKind(UML2Package.eINSTANCE.getComponent(),
                        componentRule);

    CollaborationRule collaborationRule =
        new CollaborationRule("collab", "Collaboration Rule");
    transform.addByKind(UML2Package.eINSTANCE.getCollaboration(),
                        collaborationRule);

}
```

9.5.4 Implementing the transformation rules

The UMLKindTransform in the sample transformation, when executed, walks over the content of the UML model, or part of the model, and invokes the appropriate rule whenever it encounters a UML Component, Interface, Package, or Collaboration.

By specifying a condition for the rule, the elements that a rule is invoked for are restricted further. For example, the class ServiceProviderCondition in Example 9-16 restricts the ServiceProviderRule to execute only for components that have the stereotype ServiceProvider. The condition is also coded to ignore elements that are in a package with the stereotype "external".

Example 9-16   Condition implementation

```
private static class ServiceProviderCondition extends Condition {
    public boolean isSatisfied(Object arg0) {
        if (arg0 instanceof Component) {
            if (!Utilities.isExternal((Component) arg0)) {
                if (UML2Helper.hasStereotype((Component) arg0,
                        "ServiceProvider")) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

The "external" stereotype marks parts of the model for which no code is generated.

The inner class ServiceProviderCondition is added to the class ServiceProviderRule. In the example Utilities and UML2Helper are helper classes that we created.

To associate the condition with the rule, the constructor in Example 9-17 is changed to invoke the ServiceProviderCondition constructor.

Example 9-17   Rule constructor

```
public ServiceProviderRule(String id, String name) {
    super(id, name);
    setAcceptCondition(new ServiceProviderCondition());
}
```

Now when the UMLKindTransform finds a Component, it checks the condition before invoking the rule. Only if the condition accepts the Component does the rule's createTarget method get called. The input to this method is a transformation context that has the Component set as the source object.

The createTarget method is coded to extract the source element from the transformation context and then creates, or updates, whatever target files are required. This is where the real work of the transformation is done. Example 9-18 shows an outline of the createTarget method implementation for the ServiceProviderRule.

Example 9-18   ServiceProviderRule createTarget method

```
public Object createTarget(ITransformContext ruleContext) {
    // get the source model element from the context
    // we know this will be a Component
    org.eclipse.uml2.Component src =
            (org.eclipse.uml2.Component) ruleContext.getSource();
    // get the progress monitor from the context so we can report
    // progress
    IProgressMonitor monitor = (IProgressMonitor) ruleContext
        .getPropertyValue(ITransformContext.PROGRESS_MONITOR);
    monitor.beginTask("Generating EJB...", 20);
        try {
        // create the target projects, folders, and files
        // based on the source model element
        ....
    } catch (Exception e) {
        throw new RuntimeException("Error transforming component "
                + src.getName(), e);
    }
    monitor.done();
// it is not necessary to return anything from createTarget
    return null;
}
```

In Example 9-18, the source model element is retrieved from the transform context passed to the rule. We know that this element is a UML Component because we set up the UMLKindTransform to associate this rule with Components and the type of the source element is also checked in the rule condition.

The transform context passed to the rule also allows a progress monitor to be accessed by the rule. While the transformation is running, a progress monitor dialog is shown. Rules can use the progress monitor from the transform context to provide some indication of their progress, which is reported to the user via the progress monitor dialog.

The createTarget method does not throw any checked exceptions. If the rule throws a runtime exception the transformation terminates with an error message to the user.

In the next section, we look at how to implement the code indicated by four dotes (....) in Example 9-18. You see how to create projects, folders, and files in the RSA workspace.

9.5.5 Creating and modifying files in the RSA workspace

In this section, we look at various techniques for creating and modifying files in the RSA workspace.

Eclipse Resource API

Eclipse provides APIs for navigating and modifying the workspace in the org.eclipse.core.resources package. JavaDoc and additional documentation with code samples is in the RSA help.

Example 9-19 shows how to create a file in the workspace using the Eclipse workspace resources API.

Example 9-19   Creating a MANIFEST.MF file in an EJB project

```
IProject requiredProject = ...;
// set dependent JAR list in manifest
StringBuffer manifest = new StringBuffer(
        "Manifest-Version: 1.0\nClass-Path: ");
manifest.append(requiredProject.getName());
manifest.append(".jar\n");
ByteArrayInputStream ain =
    new ByteArrayInputStream(manifest.toString().getBytes());

IFile file = project.getFile("ejbModule/META-INF/MANIFEST.MF");
file.create(ain, true, null);
```

When you are writing code to create files and folders remember to check whether they already exist and take some appropriate action if they do. This approach is suitable for generating simple, small text files, but as the output becomes more complex, this style of code becomes difficult to maintain.

The natural progression is to encapsulate the string building in a class. The static text can be separated from the code to make it easier to maintain.

Templates

Another approach for string building is to use a template processing tool such as Java Emitter Templates (JET). JET works on template files that contain all the static text needed in the output together with snippets of Java code that control how dynamic content is generated. JET compiles template files into Java classes that are used in transformation rules to generate file content as in Example 9-20.

Example 9-20   - JET template for service EJB interface

```
<%@ jet package="com.ibm.itso.mdd.example.templates" imports="java.util.*
org.eclipse.uml2.*"
class="EJBRemoteIntfTemplate"%>
<%Interface intf= (Interface)argument;%>
package ejbs.remote;

public interface <%=intf.getName()%>
{
<%for(int i=0; i<intf.getOwnedOperations().size(); i++){
Operation op=(Operation)intf.getOwnedOperations().get(i);
%>
     String <%=op.getName()%> (String in)
      throws java.rmi.RemoteException;
<%}%>
}
```

This is compiled to a class that generates Java interfaces of this form given a
UML Interface model element as input, for example.

Example 9-21   Using a JET template to create a file

```
org.eclipse.uml2.Interface intf = ...;
String content = new EJBRemoteIntfTemplate.generate(intf);
ByteArrayInputStream ain = new ByteArrayInputStream(content.getBytes());
IFile file = folder.getFile(intf.getName()+".java");
file.create(ain, true, null);
```

The best way to get started with JET is to follow the JET tutorials in the RSA help.
Search for JET in the Help.

The template file can become difficult to maintain if it includes a lot of Java code.
A better practice is to separate the code into helper classes, or perform some
pre-processing and pass in more information with the argument object instead.

Parsing and updating files

The approaches described in the previous section are fine for generating a
complete file but not so useful when loading an existing file and making changes
to it.

The Java Development Tooling includes APIs to access and modify Java project
information (like class paths), create Java packages and classes, and parse and
update Java source code. There are examples and JavaDoc in the RSA help.
The JDT API also has methods to build complete Java source files from scratch.

XML files can be parsed and updated using the org.w3c.dom API.

9.5.6 Testing the transformation

1. Launch a run-time workbench. See 9.6, "Launching a Run-time Workbench" on page 215, if you do not know how to do this.

2. In the launched workbench workspace, add the example Web service projects and UML model.

3. Switch to the Modeling perspective, and open the example model. In the Model Explorer, expand the **example model file**, and select the **model element**.
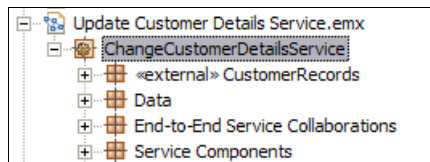
Figure 9-33   Selecting the model element in the Model Explorer

4. Right-click the **model**, and from the menu, select **Transform** → **Run Transformation** → **Services Transformation**.
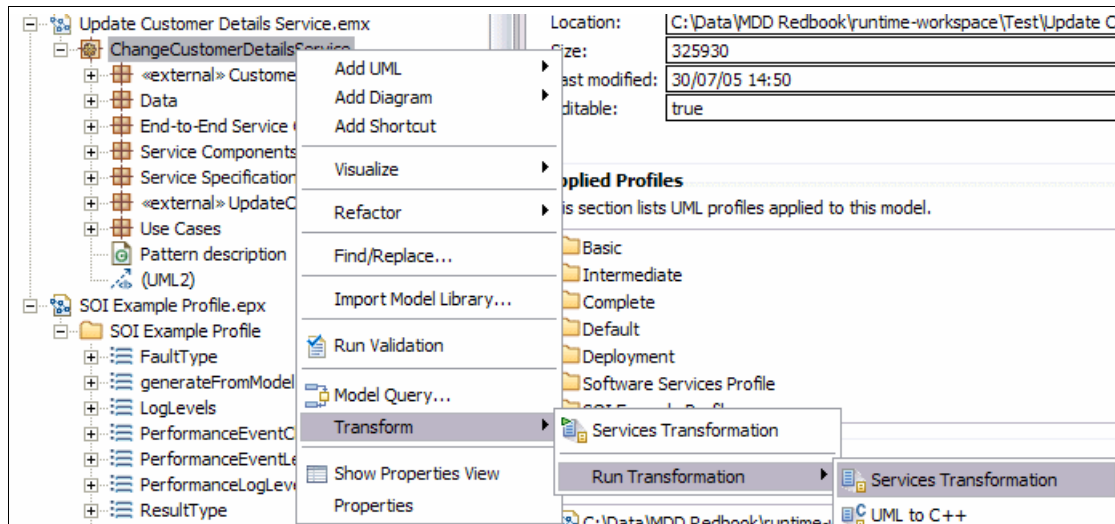
Figure 9-34   Selecting the transformation to run

5. To run the transformation, in the Run Transformation window (Figure 9-35), click **Run**.
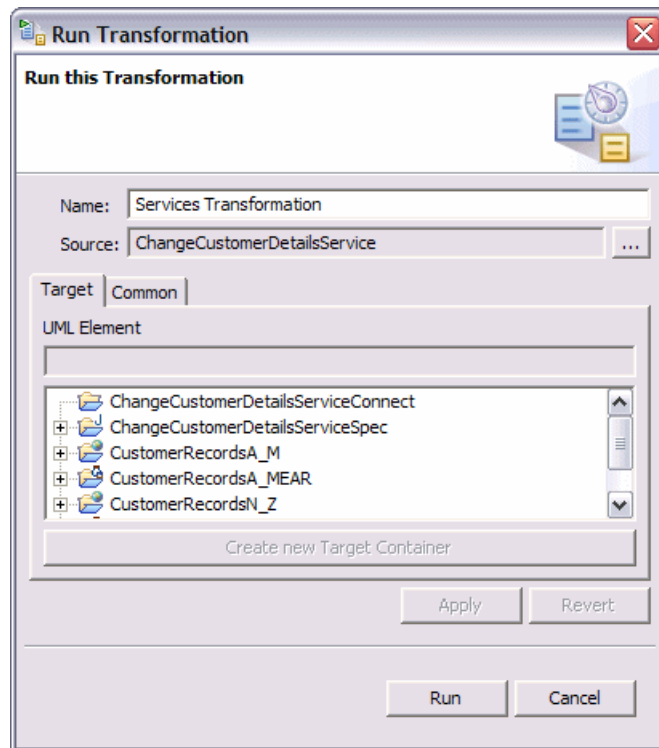
Figure 9-35   Run transformation dialog

9.6 Launching a Run-time Workbench

To test a plug-in, you can launch a new RSA workbench from RSA. This allows you to debug your plug-in code using the Java debugger. The first time you do this, you must create a new launch configuration.

1. In the RSA menu bar, select **Run** → **Run...** (see Figure 9-36).
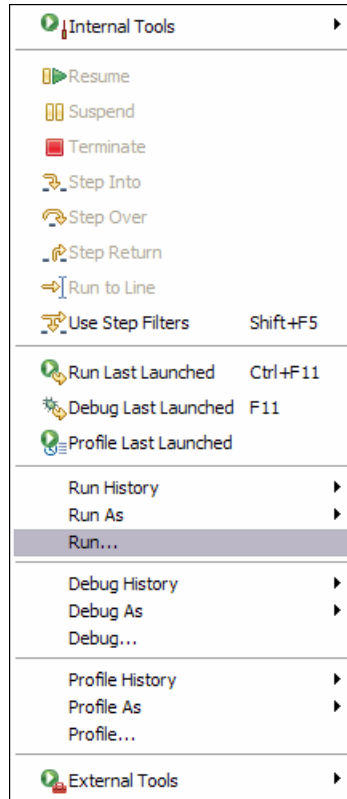
Figure 9-36   Starting the Run-time Workbench

2. The Run window opens where you create, manage, and run configurations (see Figure 9-37). Under Configurations, select **Run-time Workbench** → **New_configuration** to create a new workbench. In the right pane, for Name, type the name of your configuration. The key here is that you create a new workbench to test the transformation; you can't test it in the workbench it is developed in.

Depending on the hardware resources and the processes that are running at the time, the start up time could be slow, especially the first time it is run.
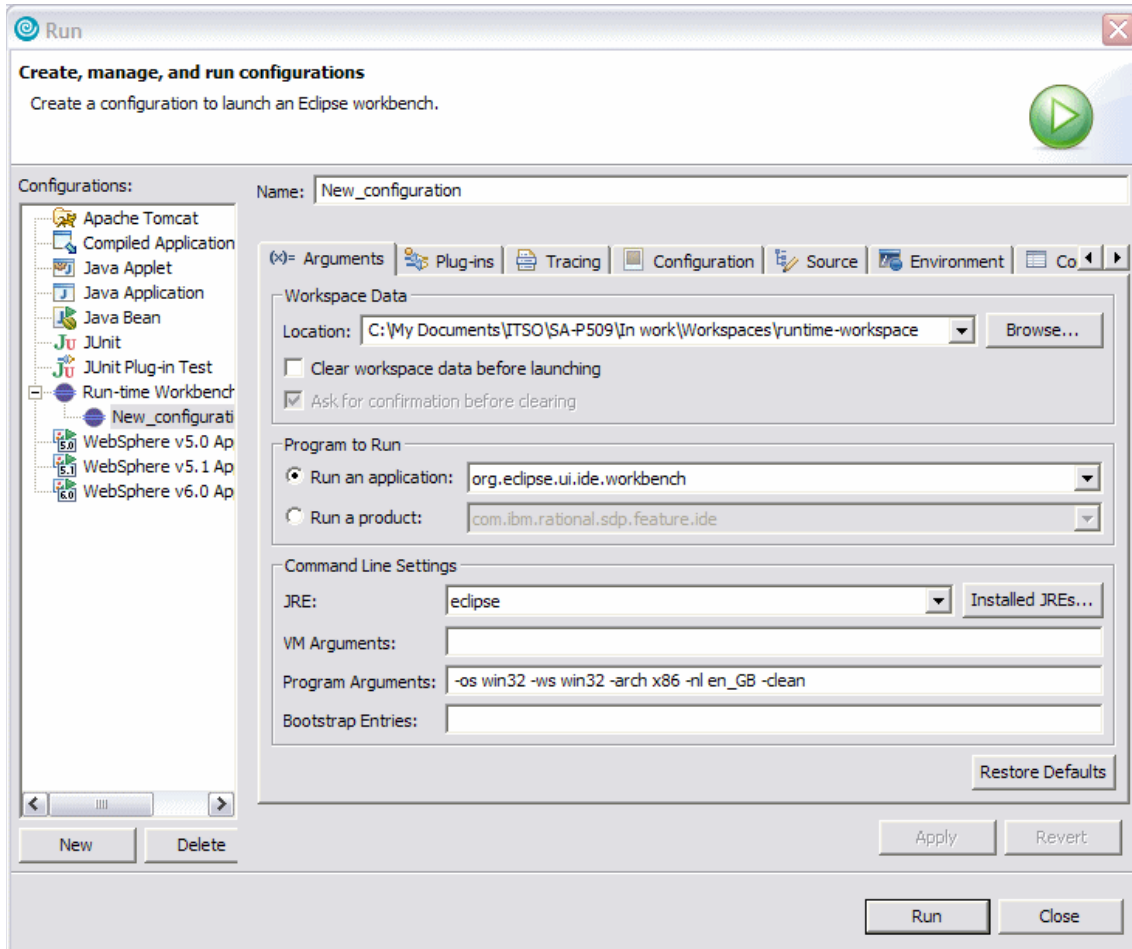
Figure 9-37   Naming the new Workbench configuration

The next time you want to test your plug-in, you can select the name of this launch configuration from the Run shortcut menu. To debug your plug-in code, you need to start the runtime workbench from the Debug (instead of Run) menu.

> **Tip:** Refer to the IBM developerWorks article "Configuring IBM Rational Software Architect or IBM Rational Software Modeler for Transformation Development", which explains how to change the launch configuration so that the runtime workbench can start quickly:
>
> http://www.ibm.com/developerworks/rational/library/05/426_funk/

9.7 Deploying plug-ins

1. From the main menu, select **File** → **Export** → **Deployable plug-ins and fragments**.

2. The Export Plug-ins and Fragments window opens. To deploy the plug-in into your own workbench, use the options shown in Figure 9-38, checking that the directory is suitable for your RSA install.

   The effect is to create a directory called *plugin-id*_1.0.0 (the plug-in ID combined with its version) in the plug-ins folder in the location specified, containing the files you selected to be included in the plug-in. You can do this manually or in an ANT build script instead of using the wizard.
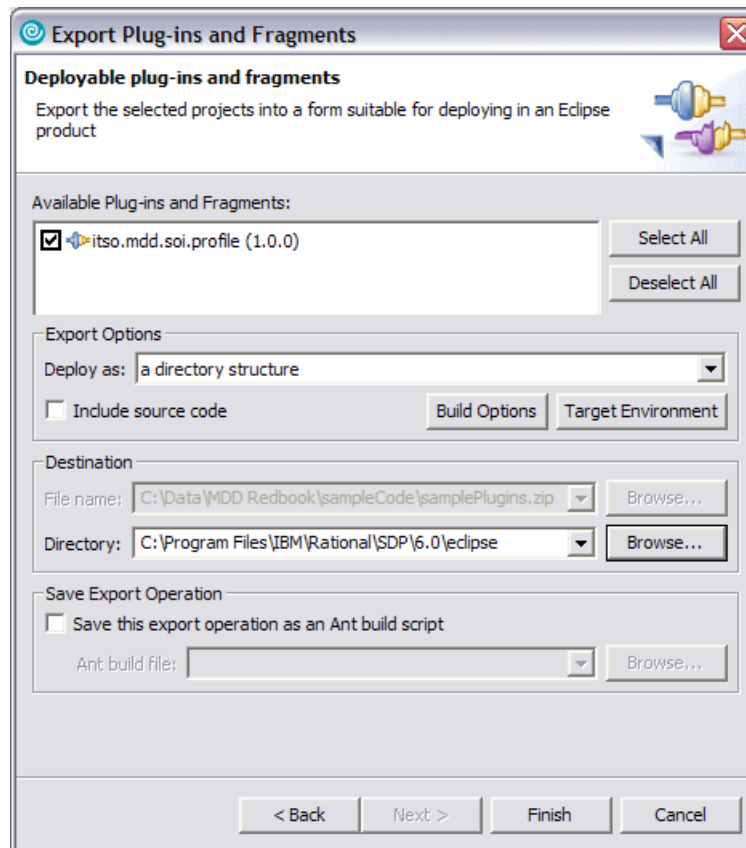
Figure 9-38   Export deployable plug-ins wizard

3. Restart RSA and you should be able to use the plug-in without launching a runtime workbench.

If you want to package the plug-in in a zip file to give to someone else, then you can instead export it as a zip file. To install this, unzip it into the appropriate directory and RSA must be restarted.

4. If you make changes to the plug-in code after it is deployed, repeat this process and restart RSA with the command-line option `-clean`. This is because RSA caches plug-in JAR files to improve performance. If you do not use the `-clean` option when RSA is started, it may not be running with the new version of your code.

9.8 Using a RAS repository

Another way to distribute your plug-in is to use a RAS repository. Plug-in projects can be packaged as RAS assets and exported to a repository. Users can then import the assets from the repository, and any plug-ins are automatically installed into their workbench. For more information, see the RSA help.

An IBM developerWorks RAS repository contains more sample patterns and transformations which you can connect to. For details, go to:

http://www.ibm.com/support/docview.wss?uid=swg24009749

9.9 Summary

In this chapter, we explained:

► How to deploy new UML profiles
► How to create plug-ins implementing new patterns and transformations

We also described methods to create or modify files from within a plug-in that could be used in a transformation to store new implementation artifacts.

10

Conclusion

The objective of this book is to draw upon our experiences with model-driven development, and inform technical practitioners and project managers about what it is and how to apply it to real projects. Perhaps you are one of those people and you wanted to know more about the practicalities of model-driven development. We hope you found this book valuable.

For the solution architect, we identified the benefits of model-driven development, and discussed where the method may be applied. Our experience is with large enterprises and organizations that see:

► The value in using MDD to apply architectural strategies consistently and reliably

► The benefits of automating the generation of much of the boilerplate infrastructure code that accompanies large IT projects

You may encounter situations where the obstacles to applying MDD outweigh the benefits. It is important to establish MDD as appropriate for your organization and then identify where in your organization you can apply it.

For the project planner or project manager, we discussed the changes that adopting MDD will make to existing solution development practices, and given guidance from our experience about how to plan an MDD project. Particular attention must given not only to the increased initial investment to develop software models, but also to realizing the potential gains in testing, deployment,

and maintenance that are easily lost if MDD is not applied throughout the whole MDD life cycle.

For the technical practitioner, we described how to identify, analyze, and apply patterns and transformations using our example of synchronous update, where we used one service provider within an ESB architecture. We also explained how to customize Rational Software Architect by developing and deploying new UML profiles and extending Rational Software Architect with Eclipse plug-ins to add new patterns and transformations.

These roles need to team up within their organization to evaluate the potential of MDD and to build a road map for MDD within their business. We think the critical success factors for the team are as follows:

► Get executive sponsorship for the MDD strategy and engage an experienced project manager who is going to be responsible for delivering the benefits of MDD to the business.

► Get a core team with modeling skills, domain knowledge and platform expertise. Seed the team with people who used the approach before.

► Build a wider team representing the whole solution life cycle. Train the wider team in MDD to deliver on its promise. They will champion MDD to their colleagues.

► Find a scenario where the application of MDD yields significant benefits and where a business case can be constructed and measured.

► Gather the domain experts in the scenario together and have them identify the parts of the scenario that can be automated. Have them develop patterns and leverage existing patterns that capture their expertise.

► Work out a plan that identifies the scope of the solution that will be tackled by MDD and what implementation artifacts to generate.

Go Modeling!

A
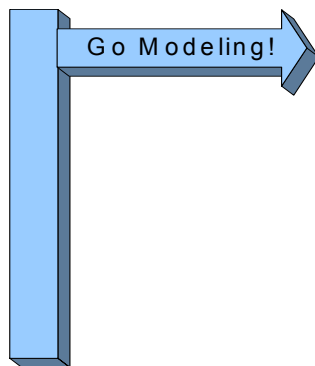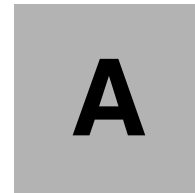
Additional material

This redbook refers to additional material that you can download from the Internet.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG247105`

Alternatively, you can go to the IBM redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG24-7105.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name	Description
sampleCode.zip	Zipped Code Samples

221

System requirements for downloading the Web material

To use the code samples, you must have IBM Rational Software Architect Version 6.0.0.1 with the IBM WebSphere Application Server v6.0 Integrated Test Environment.

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder. The sampleCode.zip file contains the following files:

File name	Description
sampleSrc.zip	Source code for the sample pattern and transformation described in the redbook
updateCustomer.zip	Sample UML model and Web services

It also contains a folder named **MDD_RedbookSamples**, which contains a version of the sample pattern and transformation that you can install into IBM Rational Software Architect.

To install the sample plug-ins into IBM Rational Software Architect, follow these steps:

1. Start RSA.

2. Select **Help** → **Software Updates** → **Find and Install...**

3. In the Install/Update wizard, select **Search for new features to install** → **Next**. → **New Local Site**, and select the **MDD_RedbookSamples** folder. This folder is added to the Sites to include in the search tree view. Ensure that this site is checked and any other sites are not selected.

   Click **Next**.

4. Select the **MDD Redbook Samples** feature, and continue with the installation.

> **Attention:** If you need to re-import plug-ins with the same name and version, you may need to clean out the Rational Software Architect cache to find the new version of the plug-in. Start RSA using the following command in the C:\Program Files\IBM\Rational\SDP\6.0 installation directory:
>
> ```
> rationalsdp -clean
> ```

The sampleSrc.zip and updateCustomer.zip files are Project Interchange files; these contain IBM Rational Software Architect projects.

5. To import into your workspace, select **File** → **Import...** → **Project Interchange**.

6. From the zip file, select the projects to import. sampleSrc.zip contains the source code for the sample pattern and transformation described in the redbook. The updateCustomer.zip file contains a sample UML model, the UML Profile for the service-oriented integration scenario, and sample Web services. Import these projects in order to try running the sample transformation.

Abbreviations and acronyms

ABD	Asset-Based Development		**OMG**	Object Management Group
ANT	Another Neat Tool		**PFCB**	provider facade call back
BDD	business-driven development		**PFS**	Provider Facade Service
C#	C-Sharp		**PIM**	Platform Independent Model
CIM	Computer Independent Model		**PSM**	Platform Specific Model
CRM	Customer Relationship Management		**RAS**	Reusable Asset Specification
DSL	domain-specific language		**RCBF**	request callback facade
ECDF	Enterprise Canonical Data Format		**RSA**	Rational Software Architect
EJB	Enterprise JavaBean		**RSM**	Rational Software Modeler
EMF	Eclipse Modeling Framework		**RUP**	Rational Unified Process
ESB	enterprise service bus		**SOA**	service-oriented architecture
HTTP	Hypertext Transfer Protocol		**SOAP**	Simple Object Access Protocol
IBM	International Business Machines Corporation		**SOI**	service-oriented integration
IFS	Integration Facade Service		**SQL**	Structured Query Language
IS	integration service		**STSM**	Senior Technical Staff Member
ISCB	integration service call back		**UML**	Unified Modeling Language
IT	information technology		**WBI**	WebSphere Business Integration
ITSO	International Technical Support Organization		**WSDL**	Web Services Description Language
J2EE	Java 2 Platform, Enterprise Edition		**XALAN**	Extensible Stylesheet Language Transformation (XSLT) processor said to be named after a rare musical instrument, but the only one that comes close is the xalam, a precursor of the banjo. See http://en.wikipedia.org/wiki/Xalam
JET	Java Emitter Templates			
JMS	Java Messaging Service			
JNDI	Java Native Directory Interface			
JUNIT	Java Unit Test			
MDA	Model-Driven Architecture			
MDD	model-driven development			
MOF	Meta-Object Facility			
OAG	Open Applications Group			

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see "How to get IBM Redbooks" on page 229. Note that some of the documents referenced here may be available in softcopy only.

► *Patterns: Implementing an SOA using an Enterprise Service Bus,* SG24-6346

► *Patterns: SOA with an Enterprise Service Bus in WebSphere Application Server V6,* SG24-6494

Other publications

These publications are also relevant as further information sources:

► Alexander, Christopher. *The Timeless Way of Building.* Oxford University Press, August 1979. ISBN 0-195-02402-8.

► Alexander, Christopher. *A Pattern Language Towns, Buildings, Construction.* Oxford University Press, August 1977. ISBN 0-195-01919-9.

► Greenfield, Jack; Short, Keith; Cook, Steve; Kent, Stuart; Crupi, John. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* John Wiley & Sons, August 2004. ISBN 0-471-20284-3.

► Hubert, Richard. *Convergent Architecture: Building Model-driven J2EE Systems with UML.* John Wiley & Sons, Inc., November 2001. ISBN 0-471-10560-0.

► Mellor, Stephen J. and Balcer, Marc J. *Executable UML: A Foundation for Model-Driven Architecture.* Addison Wesley, May 2002. ISBN 0-201-74804-5.

► Mellor, Stephen J.; Scott, Kendall; Uhl, Axel; Weise, Dirk. *MDA Distilled: Principles of Model-Driven Architecture.* Addison Wesley Professional, March 2004. ISBN 0-201-78891-8.

- Raistrick, Chris; Francis, Paul; Wright, John; Carter, Colin; Wilkie, Ian. *Model-Driven Architecture with Executable UML.* Cambridge University Press, March 2004. ISBN 0-521-53771-1.

Online resources

These Web sites are also relevant as further information sources:

- MDD Manifesto

  http://www.ibm.com/software/rational/mda/papers.html

- New Roles in MDD

  http://www.cs.kent.ac.uk/projects/kmf/mdaworkshop/submissions/Aagedal.pdf

- Pragmatics of MDD - Selic

  http://www.computer.org/software/homepage/2003/s5sel_print.htm

- MDA Guide by OMG (especially see Section 3.10.4)

  http://www.omg.org/docs/omg/03-06-01.pdf

- The UML profile for testing

  http://www.omg.org/cgi-bin/doc?ptc/2004-04-02

- The UML profile for software services

  http://www.ibm.com/developerworks/rational/library/05/419_soa/

- The UML profile for real-time and schedulability

  http://www.omg.org/technology/documents/formal/schedulability.htm

- MSDN Software Factories home page

  http://lab.msdn.microsoft.com/teamsystem/workshop/sf/default.aspx

- Automated Generation and Execution of Test Suites for DIstributed Component-based Software

  http://www.agedis.de/

- Model-driven testing: IBM Haifa home page

  http://www.haifa.il.ibm.com/projects/verification/mdt/tools.html

- Model-driven testing: IBM Haifa presentation

  http://heim.ifi.uio.no/~janoa/wmdd2004/presentations/alan.pdf

► Other perspectives on Model-Driven Architecture and UML

   – Martin Fowler

     http://www.martinfowler.com/bliki/ModelDrivenArchitecture.html

   – Steve Cook

     http://www.bptrends.com/publicationfiles/01-04%20COL%20Dom%20Spec%20Mode
ling%20Frankel-Cook.pdf

   – Dave Thomas

     http://www.jot.fm/issues/issue_2003_01/column1

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications, and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at the following Web site:

    ibm.com/redbooks

Help from IBM

IBM Support and downloads

    ibm.com/support

IBM Global Services

    ibm.com/services

Index

IBM

Redbooks

Patterns: Model-Driven Development Using IBM Rational Software Architect

Patterns: Model-Driven Development Using IBM Rational Software Architect

IBM®

Redbooks

Learn how to automate pattern-driven development

Build a model-driven development framework

Follow a service-oriented architecture case study

"The convergence of patterns, models and tooling sets the scene for major increases in application development productivity by 2010. Now is a good time to get on board."
Jonathan Adams, Distinguished Engineer,
IBM Academy of Technology

You may read this IBM Redbook for a number of reasons. Perhaps you are already familiar with the ideas behind model-driven development (MDD), and you want to learn about how to put those ideas into practice and how to convince others in your organization to adopt the approach. Maybe you heard something about the benefits of MDD but want to learn more about it before you are convinced that it is suitable for your project. Or you recently joined an MDD project and need to understand what it is all about.

This IBM Redbook is written for technical practitioners and project managers who want to learn more about MDD in practice. It will help you understand how to put the ideas of MDD into practice using Unified Modeling Language (UML). You will learn how to articulate the advantages of MDD to both project managers and technical colleagues. You will see how the MDD software life cycle differs from other approaches and how you can effectively plan and manage an MDD project. If you are already working on an MDD project, you will learn