the eclipse series

eclipse Modeling Project

A Domain-Specific Language (DSL) Toolkit

Richard C. Gronback

SERIES EDITORS ▸ Erich Gamma · Lee Nackman · John Wiegand

Library of Congress Cataloging-in-Publication Data:

Gronback, Richard C.
Eclipse modeling project : a domain-specific language (DSL) toolkit / Richard C. Gronback.
p. cm.

ISBN 0-321-53407-7 (pbk. : alk. paper) 1. Computer software—Development. 2. Eclipse (Electronic resource) 3. Programming languages (Electronic computers) I. Title.
QA76.76.D47G785 2009
005.1—dc22

2008050813

ISBN-13: 978-0-321-53407-1
ISBN-10: 0-321-53407-7
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing March 2009

Associate Publisher
Mark Taub

Acquisitions Editor
Greg Doench

Managing Editor
Kristy Hart

Project Editor
Jovana San Nicolas-Shirley

Copy Editor
Krista Hansing Editorial Services, Inc.

Indexer
Erika Millen

Technical Reviewer
Simon Archer

David Orme
Daniel Holt

Publishing Coordinator
Michelle Housley

Cover Designer
Sandra Schroeder

Compositor
Nonie Ratcliff

Foreword

Just like a pearl, the Eclipse Modeling Project has grown organically as layers around a central core. From the humble beginnings of the Eclipse Modeling Framework (EMF) (initially part of the Eclipse Tools Project) along with the Graphical Modeling Framework (GMF) and the Generative Modeling Tools (GMT) (both initially part of the Eclipse Technology Project), the Modeling Project coalesced to become one of Eclipse's most exciting and diverse projects. The depth and breadth of its technology is vast and even its rate of growth continues to increase. The Eclipse Modeling Project has truly become a Swiss Army knife for state-of-the-art model-driven software development.

The sheer volume of useful modeling technologies that the Eclipse Modeling Project includes makes mastering a significant portion of it a daunting task. Even determining which specific available technologies are useful for solving any particular problem is a challenge exacerbated by the fact that, as a rule, the documentation tends to lag far behind the development work. As such, this book fills a fundamentally important need in the modeling community: a coherent vision of how all this powerful technology can be best exploited to build domain specific languages (DSLs). In other words, the focus of this book is on pragmatic applications illustrated by way of concrete examples rather than on abstract modeling concepts and theories. This pragmatic focus reflects that of the Modeling Project overall—that is, a focus on building powerful frameworks that real programmers use every day. I'm sure this influential book—with its interesting examples and its excellent reference material—will become a key part of every toolsmith's technical arsenal.

EMF provides a sound basis for abstract syntax development and even includes a crude but effective XML-based concrete syntax. But, that is only the start of the journey, and the second edition of the *Eclipse Modeling Framework* book covers this basic material well. This book effectively picks up where EMF leaves off with an in-depth exploration of alternative forms of concrete syntax, particularly graphical syntax, model-to-model transformation (such as Query

View Transformation Operational Mapping Language), and model-to-text transformation (such as Xpand). It rounds out the DSL picture with a holistic view of everything it takes to build a complete model-based product line.

It has been my great pleasure and honor to work closely with Richard C. Gronback as the Modeling Project coleader for the past few years. He has a keen mind and a sharp wit. This book reflects it well. I've learned a great deal from him, and I'm sure readers of this book will as well.

—Ed Merks, Ph.D.
President, Macro Modeling

Preface

About This Book

This book covers a relatively new collection of technologies that focus on developing domain-specific languages (DSLs) using the Eclipse Modeling Project, offering a first look at a range of Eclipse projects that have not yet been covered in detail within this context. Although the core of these technologies has been available for several years in the Eclipse Modeling Framework (EMF), diagram definition and model transformation are emerging technologies at Eclipse. These are complemented by upcoming textual syntax development frameworks, which likely will be covered in detail in subsequent editions of this book.

This book delivers a pragmatic introduction to developing a product line using a collection of DSLs. A model-based, largely generative approach is designed to accommodate future adjustments to the source models, templates, and model transformation definitions, to provide customized solutions within the context of the product line. To illustrate this approach, this book presents a set of sample projects used to define a requirements product line.

Audience

This book targets developers and architects who want to learn about developing DSLs using Eclipse Modeling Project technologies. It assumes a basic understanding of the Java programming language, Eclipse plug-in development, and familiarity with EMF. This book's target audience are those interested in learning about the Eclipse Graphical Modeling Framework (GMF), Model-to-Model Transformation (M2M) Query/View/Transformation Operational Mapping Language (QVT OML), and Model-to-Text Transformation (M2T) Xpand project components.

The book is divided into introductory, hands on, and reference sections. Readers who want an overview of the Eclipse Modeling Project and development of DSLs in the context of an Eclipse-based product line should read Part I, "Introduction." Readers who want to follow along in a tutorial fashion to learn

how to use the projects listed earlier should read Part II, "Developing Domain-Specific Languages." Readers also can reference the sample project solutions in this section to get an overview of the techniques. Part III, "Reference," serves as a resource for readers who want a deeper understanding of Graphical Editing Framework (GEF), GMF, Xpand, and QVT OML while they are completing Part II or developing their own DSL-based projects.

Readers who want to experience the benefits of a commercial version of the technologies presented here can download the Borland Together product. There they will find enhanced domain modeling, refactoring, diagram development, transformation authoring and debugging, workflow, and generation capabilities in a well-integrated DSL Toolkit.

Sample Code

The Modeling Amalgamation Project (Amalgam) at Eclipse holds the sample code from this book and is available as sample projects in the DSL Toolkit download. This package also includes all the prerequisites required for developing the sample applications.

Visit the Amalgam project Web site for more information on obtaining the DSL Toolkit: www.eclipse.org/modeling/amalgam.

Feedback

The examples in this book are maintained within the Modeling Amalgamation Project at Eclipse. Feedback on their content—and, therefore, this book's content—is welcome on the project newsgroup, http://news.eclipse.modeling.amalgam. Alternatively, feel free to contact the author directly at richard.gronback@gmail.com.

CHAPTER 3

Developing a DSL Abstract Syntax

In this chapter, we walk through the development of a domain-specific language (DSL) using the Eclipse Modeling Framework (EMF) and supporting components. Specifically, we develop the DSL's abstract syntax using the Ecore metamodel. But first we cover some basics on what to consider when creating a DSL and the different implementation strategies you might want to employ when using EMF. Next, we provide an overview of EMF, leaving detailed information to the book [38] dedicated to this purpose. We cover some additional components of EMF and Model Development Tools (MDT) that enable you to further refine DSLs, and we develop a series of domain models for use in the sample projects.

DISCLAIMER

The domain models developed as samples are constructed to illustrate certain features of the associated tooling and, as such, should not necessarily be considered "best practices" in some cases.

3.1 DSL Considerations

Many considerations are involved in creating a DSL. Does a model already exist that is close enough? If so, can an existing model be extended, or is it fixed? Does the model need to be based on a standard? Does the DSL lend itself to graphical display and editing? Does the DSL require a textual syntax and editor? Will a

product line be built on the DSL, and perhaps others? Is the Ecore metamodel expressive enough to suit your needs for a DSL? How can you model dynamic behavior?

BEST PRACTICE

Leverage existing models, when appropriate. XML Schema Definition (XSD) and EMF are very popular technologies, and EMF can import just about any XSD, so search for existing domain models before you reinvent the wheel. Also consider publishing your domain model if you think that others might find it useful, if only as part of your application's API to aid in integration.

A key consideration is the amount of flexibility you need or will tolerate in the DSL. As you can see in the examples, sometimes a change in the domain model makes your transformation definitions much easier to write. Also, frameworks such as GMF have certain limitations—or, rather, were designed with particular use cases in mind. Your particular style of modeling might not lend itself well to graphical representation, but a few changes might allow mapping to diagram elements much easier. For example, certain mappings in Query/View/Transformation (QVT) and template expressions can be facilitated by adding derived features or methods to the domain model. Complex queries using Object Constraint Language (OCL) (and, therefore, useful ones in QVT and Xtend) can be added to the domain model with code generated for their implementation at runtime. Having a feature available in the model will greatly simplify transformations and templates that access them.

TIP

Don't be afraid of modifying your domain model to make working with templates, transformations, and diagram definitions easier. Unless you're using a model that cannot be altered, the Toolsmith will appreciate being able to make certain design decisions in the domain model to suit the tooling, instead of having to create workarounds or write custom code to use the tooling with a domain model.

This is not to say that you should let the tooling influence your DSL to an extent you are not comfortable with. The question is, how do you maintain a satisfactory level of "purity" in your DSL when considering the additional

complexity associated with developing and maintaining the other Model-Driven Software Development (MDSD) artifacts? In general, the more complex the metamodel (DSL) is, the more complex the transformation definitions, templates, and diagram definitions are.

A set of conventions and best practices for the definition of DSLs, transformations, and templates likely will arise, as it has for Java and other popular programming languages. With conventions and best practices comes tooling to support refactorings, static analysis, and cleanup. At this stage of the Modeling project's evolution, operations are still quite manual and even error prone. As an open source project that forms the basis for commercial products, Eclipse eventually will see more advanced features pushed down into it, thereby improving the Toolsmith experience.

3.2 Eclipse Modeling Framework

From the project description, EMF is "a modeling framework and code generation facility for building tools and other applications based on a structured data model." This pretty much sums it up, but there's a lot to know about EMF. I highly recommend that you first read, or at least have available, the book on EMF [38] to better understand its use in the context of this book. Alternatively, reading through the online documentation and tutorials on EMF should make its use in this book easy to follow. In other words, the examples in this book only scratch the surface of what is possible using EMF.

You can create models using EMF in many ways. You can use the provided editor (a tree with properties view) or import a set of annotated Java classes. An Ecore diagram is available from the EMFT project. If you have the XSD component installed, you can import an XSD file. If you have the Unified Modeling Language (UML) version 2 (UML2) component installed, you can import a UML2 model. If you have Graphical Modeling Framework (GMF) installed, you can use its example Ecore diagram editor. If you download and install Emfatic [42], you can work in a textual syntax and synchronize with your Ecore model. In the future, you will be able to design your own concrete textual syntax for Ecore, or any model, using the Textual Modeling Framework (TMF) project.

Regardless of the method you choose for importing or working with your domain model, you will find an `.ecore` file in your workspace—that is, unless you take a purely programmatic approach. If you open this file in a text editor, you will see that it is an XML Metadata Interchange (XMI) serialization of your Ecore-based model. By default, EMF enables you to edit models in a basic (generated) tree editor with a Properties view. You can easily generate a similar editor for your own model.

Before getting into more detail, let's take a look at the Ecore metamodel.

3.2.1 Ecore Metamodel

The EMF book describes the Ecore metamodel in detail, but here you find a simplified diagram for reference (Figure 3-1), along with some discussion of the more relevant aspects used as we develop our DSL abstract syntax. It's a fairly simple model, which is part of its strength. In most cases, you can compensate for the lack of features in Ecore by using some of the more advanced modeling techniques, which are discussed in the EMF book. A longstanding topic of debate among EMF users is the lack of an EAssociation class, but we don't get into that here.

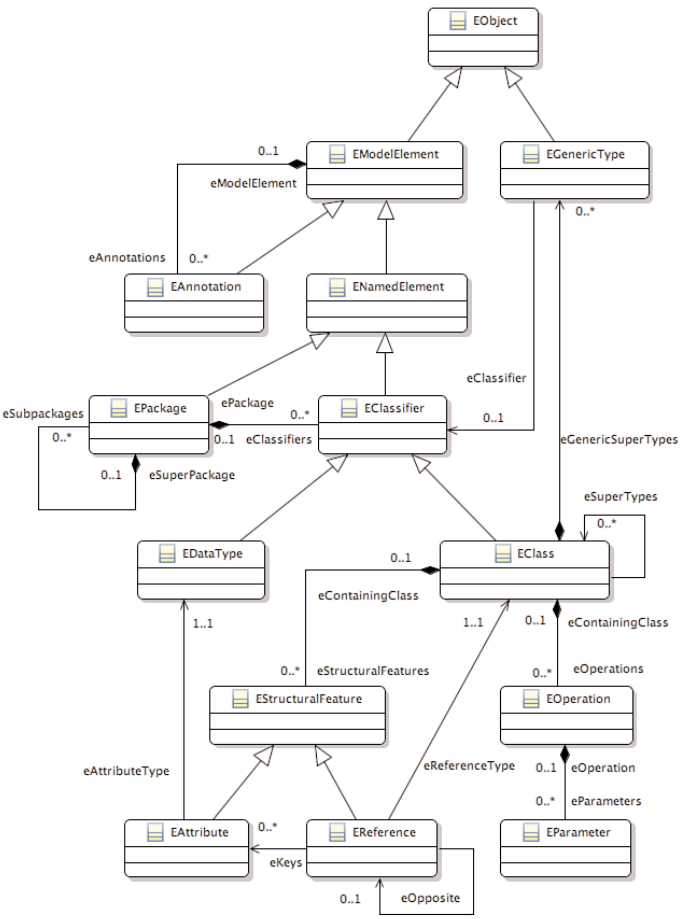

Figure 3-1 Ecore model

Annotations

Sometimes it's important to add information to a model element for documentation purposes, or to provide parameters to be considered during transformation or generation. EAnnotations provide these for all model elements in EMF. An EAnnotation has a **Source** field, which serves as a key, and a list of **References**. An EAnnotation may have zero or more **Details Entry** children, which have **Key** and **Value** properties. This simple capability of annotating models is quite flexible and turns out to be useful for many purposes, including XSD support.

Another particularly useful application of annotations is to declare OCL constraints, method bodies, and derived feature implementation, as discussed in Section 3.2.4, "Applying OCL."

3.2.2 Runtime Features

The EMF runtime includes facilities for working with instances of your models. No strict dependencies exist on the Eclipse platform for the runtime and generated model and edit code, so these bundles can be used outside of the Eclipse workbench. As bundles, they can be deployed in any Equinox OSGi container, even within a server environment.

The generated code for your model has a dependency on the underlying EMF runtime components. A significant benefit is gained from the generated Application Programming Interface (API) and model implementation working with the provided runtime features. An efficient observer pattern implementation is provided to alert listeners to model change events. A generated reflective API provides an efficient means of working with models dynamically. In fact, EMF can be used in a purely dynamic fashion, requiring neither an `.ecore` model nor code generation. Finally, it's possible to have static registration of a dynamic package, but that's an advanced use case left to the EMF documentation.

When working with model instances, changes can be recorded in a change model that provides a reverse delta and allows for transaction support. A validation framework provides for invariant and constraint support with batch processing. The Model Transaction and Validation Framework components provide enhanced transaction and validation support, respectively.

For persistence of models, the EMF runtime provides a default XML serialization. The persistence layer is flexible, allowing for XMI, Universally Unique Identifiers (UUIDs), and even a zip option. A resource set consists of one or more resources, making it possible to persist objects in multiple files, including cross-containment references. Proxy resolution and demand loading improve performance when working with large models across resources. Additionally, use of EMF Technology (EMFT) components Teneo and CDO allow for the persistence of models to a relational database.

The generated editor for EMF models includes a multipage editor and properties view. Drag-and-drop support is provided, as is copy/paste support. A number of menu actions are available in the generated editor, including validation invocation and dynamic instance model creation. Each generated editor comes with a default creation wizard. Figure 3-2 shows an example of the editor, including a context menu showing options to create new elements, cut, copy, paste, delete, validate, and so on.

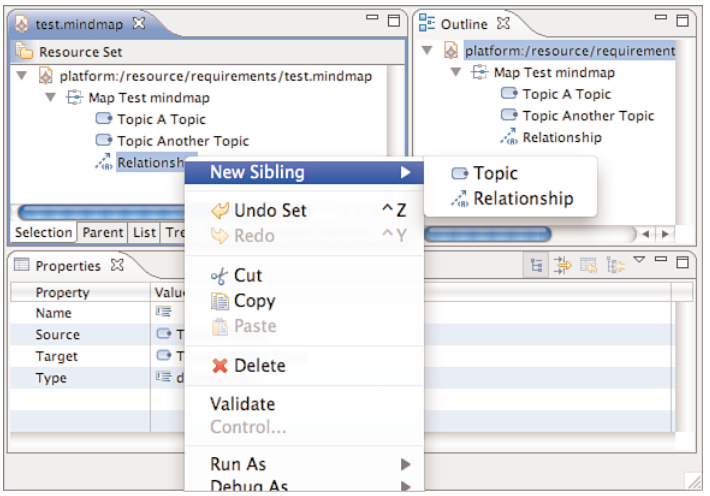

Figure 3-2 EMF-generated editor

3.2.3 Code Generation

From an `*.ecore` (Ecore) model, you need to produce a generator model and supply additional information required for code generation. An EMF generator model has a `*.genmodel` file extension and is essentially a decorator model for a corresponding `*.ecore` model. This generator model is fed to Java Emitter Templates (JETs) that are used to write Java and other files. JET is the Java Server Pages (JSP)-like technology used by default when generating text from Ecore models. This book does not cover it in detail, but a tutorial is available online [51] if you want to know more.

You can customize the existing generation output using custom templates. Furthermore, you can extract constraint, pre-/post-condition, and body implementations from OCL annotations for use in generation and invocation at runtime. This is not a native EMF capability, but you can add it using the MDT OCL component. You will use this technique in the context of the sample projects.

When regenerating code, the JMerge component is used to prevent overwriting user modifications. Generated Java code is annotated with `@generated` javadoc style tags to identify it and distinguish it from user code. Removing the tag or adding `NOT` after the tag ensures that JMerge will not overwrite the modified code. Typically, using `@generated NOT` is preferred because it allows the Toolsmith to identify code that was generated and modified, as opposed to newly added code. Note that not all code benefits from merging. Specifically, `plugin.xml` and `MANIFEST.MF` files need to be deleted before an update can occur.

3.2.4 Applying OCL

Many opportunities arise for using OCL in EMF models. Class constraints, method bodies, and derived feature implementations can all be provided using MDT OCL and EMF dynamic templates. The approach of using OCL and custom templates in this book comes from an Eclipse Corner article [44] and has been modified only slightly to conform to a similar approach taken to leverage OCL added to models in QVT, as discussed in Section 6.5.6, "Leveraging OCL in EMF Models." The templates are generic and can easily be added to any project that needs to provide OCL-based implementations in its generated model code. It is also worth noting that GMF uses OCL extensively in its models, employing an EMF Validator to maintain the integrity of its models.

To add an OCL expression to a model element, we begin by adding a normal EAnnotation. For the `Source` property, enter `http://www.eclipse.org/2007/OCL`. This URI allows our custom templates and QVT engine to recognize this annotation as OCL, where it can expect to find `Details Entry` children of type `constraint`, `derive`, or `body`. Note that the original article [44] used `http://www.eclipse.org/ocl/examples/OCL` as the URI.

Depending on the context, add the appropriate `Key` (EMF `constraint` key, `derive`, or `body`) to a child `Details Entry` of the EAnnotation and specify the OCL in the `Value` property. For invariant constraints, the OCL annotations complement the normal EMF constraint annotations by providing implementation for the validation framework to enforce constraints.

TIP

To test your OCL, it's helpful to use the Interactive OCL Console with a dynamic instance of your model, as discussed in Section 1.5.4, "Object Constraint Language." Be sure to select the proper model element for the expression, as well as the proper metalevel in the console.

To invoke the provided OCL at runtime, you must use custom JET templates for your domain model. The generated code retrieves the OCL statement from the model element and invokes it, evaluating the result. An alternative to this is to generate a Java implementation of the OCL during generation and avoid invoking the OCL interpreter at runtime.

The referenced article covers the details of the custom templates, so they are not covered here. Also, the templates are included in the book's sample projects and are touched upon during the development of the sample projects. For now, we take a look at just the derived feature implementation, both before and after using the OCL with a custom template approach. First, consider the default generated code for a derived reference—in this case, the `rootTopics` reference from the `MapImpl` class in our mindmap example.

```
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public EList<Topic> getRootTopics() {
  // TODO: implement this method to return the 'Root Topics'
  // reference list
  // Ensure that you remove @generated or mark it @generated NOT
  // The list is expected to implement
  // org.eclipse.emf.ecore.util.InternalEList and
  // org.eclipse.emf.ecore.EStructuralFeature.Setting
  // so it's likely that an appropriate subclass of
  // org.eclipse.emf.ecore.util.EcoreEList should be used.
      throw new UnsupportedOperationException();
}
```

Let's add the following OCL statement to the derived feature using the previous convention. Here we see the annotation within the mindmap.ecore model in its native XMI serialization. Note that this OCL statement could be simplified by using the parent `eOpposite` relationship on our Topic's subtopics reference, which was added to facilitate the diagram definition of Section 4.3.5, "Subtopic Figure."

```
<eStructuralFeatures xsi:type="ecore:EReference"
  name="rootTopics" upperBound="-1" eType="#//Topic" volatile="true"
  transient="true" derived="true">
  <eAnnotations source="http://www.eclipse.org/2007/OCL">
    <details key="derive"
      value="let topics : Set(mindmap::Topic) = self.elements->
                 select(oclIsKindOf(mindmap::Topic))->
                 collect(oclAsType(mindmap::Topic))->asSet() in
```

```
                topics->symmetricDifference(topics.subtopics->
                asSet())"/>
  </eAnnotations>
</eStructuralFeatures>
```

Before regeneration, we need to make some changes in the genmodel. To allow the OCL plug-in to be added to our generated manifest dependencies, we need to add `OCL_ECORE=org.eclipse.ocl.ecore` to the `Model Plug-in Variables` property of the genmodel root. Also, we need to set the `Dynamic Templates` property to `true` and enter the templates path (such as `/org.eclipse.dsl.mindmap/templates/domain`) to the `Template Directory` property. After we generate, we can see the following implementation in our `MapImpl` class.

```
private static OCLExpression<EClassifier> rootTopicsDeriveOCL;

private static final String OCL_ANNOTATION_SOURCE =
  "http://www.eclipse.org/2007/OCL";

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public EList<Topic> getRootTopics() {
  EStructuralFeature eFeature =
    MindmapPackage.Literals.MAP__ROOT_TOPICS;

  if (rootTopicsDeriveOCL == null) {
    Helper helper = OCL_ENV.createOCLHelper();
    helper.setAttributeContext(MindmapPackage.Literals.MAP, eFeature);

    EAnnotation ocl = eFeature.getEAnnotation(OCL_ANNOTATION_SOURCE);
    String derive = (String) ocl.getDetails().get("derive");

    try {
      rootTopicsDeriveOCL = helper.createQuery(derive);
    } catch (ParserException e) {
      throw new UnsupportedOperationException(e.getLocalizedMessage());
    }
  }

  Query<EClassifier, ?, ?> query =
    OCL_ENV.createQuery(rootTopicsDeriveOCL);

  @SuppressWarnings("unchecked")
  Collection<Topic> result = (Collection<Topic>) query.evaluate(this);

  return new EcoreEList.UnmodifiableEList<Topic>(this, eFeature,
    result.size(), result.toArray());
}
```

The generated code checks to see if the `OCLExpression` for this derivation has been created already; if not, it initializes it by retrieving the statement from the `EAnnotation` and its detail with key `derive`. Then the expression is evaluated and the list of `Topic` elements is returned.

As mentioned in the article, some improvements could be made to this approach, but it illustrates the usefulness of adding OCL statements to your EMF models. It's not hard to imagine how a significant portion of an application could be generated from a model adorned with OCL for invariant constraints, method bodies, and derived features. In GMF, we can see how OCL is used to augment the diagram-mapping model to provide for constraints, feature initialization, audit definition, and model metric definition.

BEST PRACTICE

Adding constraints and validation is essential in model-driven software development. Although you can place validation code within QVT, Xpand templates, and so on, it's most useful to ensure that your model instance is well formed when created, or before moving to a model transformation.

3.2.5 Dynamic Instances

A powerful feature of EMF, and one that is useful to a Toolsmith developing a new DSL, is the capability to create dynamic instances of a model. The reflective framework of EMF is leveraged to allow instantiation of a model element without generating code beforehand. This can be done from within the default Ecore editor by selecting an element and choosing the `Create Dynamic Instance` context menu item. The instance is stored in an XMI file within the development workspace, so the generation or launch of plug-ins is not required to test a model or, more importantly, to test Xpand templates and QVT transformations under development. This is one important distinction when comparing JET to Xpand. Dynamic instances are used in the context of our sample projects.

BEST PRACTICE

Use dynamic instance models for development as much as possible. Xpand templates, QVT transformations, and the OCL console can all work with dynamic instance models and avoid making Toolsmiths generate code and

invoke a runtime instance to test their work. GMF diagrams still require code generation to develop effectively, although generated diagrams are capable of working with dynamic instances.

Figure 3-3 is an example of a dynamic instance model for our mindmap domain model, along with the Properties view. It's similar in functionality to the generated EMF editor, although it requires the metamodel to be loaded and reflected upon, as you can see from the loaded `mindmap.ecore` resource file.

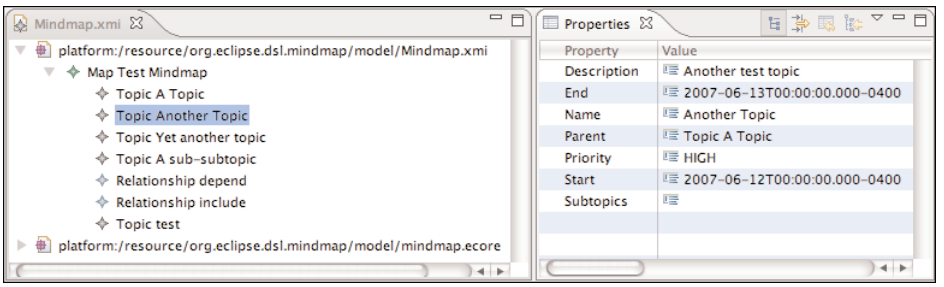

Figure 3-3 Mindmap dynamic instance model

TIP

You can view any Ecore model using the **Sample Reflective Ecore Model Editor**, so there's little need to generate the EMF .editor plug-in. This applies to XMI dynamic instances, such as GMF-based diagrams files where both the domain and notation models are persisted in a single file. Simply right-click the file and select **Open With** → **Other** → **Internal Editors** → **Sample Reflective Ecore Model Editor**.

3.3 Developing the Mindmap Domain Model

We develop a simple mindmap DSL and use it throughout the book to provide an example of how to use components of the Modeling project as a DSL Toolkit. This model forms the base of our fictitious Requirements Elicitation Project (REP).

This is the beginning of those sections in the book that you can follow in a tutorial fashion. Solution projects are available to save time, although you should be able to begin from scratch and produce these solutions on your own. It's up to you to follow along or simply review the solutions as we proceed.

Figure 3-4 is a diagram of the basic mindmap DSL we create in this section. Not much explanation should be required here because you can easily see that a `Map` class serves as the container for `Topics` and `Relationships`, which both extend from `MapElement`. The following sections provide details on setting up a DSL project and creating this model, along with the other DSL artifacts associated with the project.

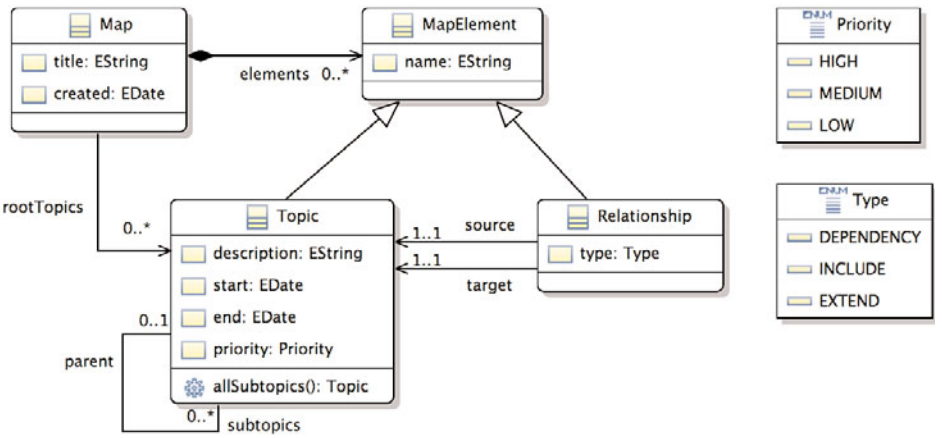

Figure 3-4 Mindmap domain model

3.3.1 Project Setup

Before getting started defining our mindmap domain model, we need a new project. Although EMF and GMF provide their own project wizards, we use the DSL Project Wizard provided by the Amalgam project to hold our DSL artifacts. You can create an equivalent project by starting with a plug-in project and adding the required dependencies, natures, and builders. The DSL project is also a plug-in project, as we'll eventually want to deploy the project to facilitate revisioning and extension. Furthermore, Xpand and workflow files currently need to be located in source paths to be developed, so we need a Java project anyway. In the future, this should not be required.

For our mindmap project, switch to the DSL perspective and use **File → New → DSL Project** to create a new project named `org.eclipse.dsl.mindmap`. The wizard creates a set of folders: `/model`, `/diagrams`, `/templates`, `/transformations`, and `/workflows`. Not all of these folders are required for each DSL project, but we use them for our mindmap. The wizard also adds natures and builders for QVT and Xpand/Model Workflow Engine (MWE).

3.3.2 Creating the Mindmap Domain Model

As mentioned earlier, creating an Ecore model involves many starting points. If we had an existing XML Schema for our domain, we could import it and EMF would take care of serializing documents conforming to the schema. If we used the UML2 project and associated the UML2 Tools class diagram to model our domain, we could import it to create an EMF model. We begin using "classic" EMF to create our mindmap DSL from scratch.

Typically, we'd begin with **File → New → Other → Eclipse Modeling Framework → Ecore Model** (Ctrl+3 → Ecore Model). However, the DSL Toolkit from Amalgam provides some wizard redefinitions to facilitate DSL development and defines capability definitions to hide the original UI contributions from various Modeling projects. To create our model, we select the `/model` folder and use the **File → New → Domain Model** (Ctrl+3 → Domain Model) wizard, which is really just the GMF Ecore diagram wizard. Name the model and diagram files `mindmap.ecore` and `mindmap.ecore_diagram`, respectively. Optionally, you can use the Ecore Tools component, available from the EMFT project. It provides some capabilities beyond those that the GMF example editor provides.

Before we begin to model the domain, we need to set some defaults in our mindmap Ecore model. First, right-click on the blank diagram surface and select **Show Properties View**. This shows the properties for the root package in our new Ecore model. Each Ecore model has a single root package, under which we can create a number of additional subpackages. In our case, we set the properties accordingly: `name` and `Ns Prefix` should be set to `mindmap`; `Ns URI` should be set to `http://www.eclipse.org/2008/mindmap`.

Using Figure 3-4, model the mindmap domain using the palette and Properties view. It's a straightforward model to implement, with only a couple noteworthy items: First, the `MapElement` class is abstract; second, the `rootTopics` relationship is derived, transient, and volatile. We implement this derived reference using OCL in Section 3.3.5, "Adding OCL."

The diagram surface has many features to explore, as discussed in Section 10.1, "Overview." You should note a few things, however, when using the Ecore diagram to create the mindmap domain model:

Aggregation links create a reference with the `Containment` property set to `true`, in contrast with Association links, which are noncontainment references.

Setting the upper bound property of a link to –1 creates a **many** relationship and causes the link to be displayed with the familiar 0..* notation.

References with `eOpposites` are shown in Figure 3-4 as a single connection, whereas the Ecore diagram shows two separate links.

3.3.3 Creating the Mindmap Generator Model

With our `mindmap.ecore` model complete, we can validate it and create a generator model. To validate it, open the model in the **Sample Ecore Model Editor** and right-click on the root package. Select **Validate** and confirm that no errors exist. If there are errors, correct them and continue. We look into adding validation for our mindmap model later, which leverages a similar validation framework provided for all Ecore models.

To create `mindmap.genmodel`, right-click the `mindmap.ecore` file in Explorer view and select **New → Other → Domain-Specific Language → Domain Generator Model** (Ctrl+3 → Domain Gen). Note that the original EMF wizard is found in **New → Other → Eclipse Modeling Framework → EMF Model** (Ctrl+3 → EMF Model). We started by selecting our `mindmap.ecore model`, so the wizard defaults to the same folder and provides the name we want. It also recognizes that we are importing an Ecore model, but we have to load it ourselves, curiously. We have only one root package, so we can finish the wizard and take a look at the generator model properties.

EMF generator models include several properties to consider. For our mindmap, we need to change only a couple from their default settings. In the root, change the `Compliance Level` from 1.4 to 5.0 (if it's not already set to 5.0) and change the `Model Directory` to be `/org.eclipse.mindmap/src`. (Note that this changes the edit, editor, and tests properties as well.) We need to manually change the `Model Plug-in ID` property to `org.eclipse.mindmap`, however. In the properties for the Mindmap root package, we need to set the `Base Package` property to `org.eclipse` to match our new plug-in namespace.

This gets us started, so we can move on to code generation. Later, we return to our mindmap model and add constraints, validation, and other enhancements.

3.3.4 Generate and Run

The last thing to do is generate our mindmap plug-ins and code. Technically, we don't need to generate code at this time because we plan to leverage dynamic instances as long as we can in the development of our DSLs. However, for those new to EMF, it's worthwhile to continue with generation at this point to see how things work. This is accomplished by right-clicking the root of the `mindmap.genmodel` in the editor tree and selecting **Generate All**. This generates our model code, edit code, editor code, and test skeletons, each in their own plug-in projects. We don't need the generated editor code because a diagram provides our primary means of working with mindmap instance models. For now, we can continue by running the generated editor to verify our model and configuration.

To run our plug-ins and test the functionality of our editor, we need to be in the Plug-in Development Environment perspective to gain access to the appropriate Run option. Select **Run → Open Run Dialog** (Ctrl+3 → Run C) and create a new Eclipse Application run configuration named `requirements` in a directory named `runtime-requirements`. Figure 3-5 is an image of this dialog. Figure 3-6 shows the **Arguments** page with some arguments for launching on Mac OS X. We use this launch configuration throughout our development of the sample projects, hence the general `requirements` name.

TIP

If you get tired of adding arguments to your launch configurations each time you create one, navigate in the Preferences dialog to **Plug-In Development → Target Platform → Launching Arguments** and enter them in the field provided. These values will be copied into any new launch configuration you create.

Run this configuration to launch a new instance of Eclipse with the new plug-ins installed. We could trim the plug-in list to launch only those plug-ins we need for our application. This makes launching faster and keeps us aware of our underlying plug-in dependencies. In Chapter 8, "DSL Packaging and Deployment," we fine-tune our launch settings before creating our product configuration.

In the runtime workbench, create a new project and follow **New → Example EMF Model Creation Wizards → Mindmap Model**, giving it whatever name you want and selecting `Map` as the Model Object. The default EMF-generated editor appears upon finish, ready for you to create new `Topic` and `Relationship` instances within the map.

You again need to open the Properties view to set model element properties and establish subtopics and relationship links. Notice that validation is available for our model instances and enforces the basic structural features defined in our model. For example, we declared 1 for the `upper` and `lower bounds` on `source` and `target` references of our `Relationship` class. Creating a new Relationship instance in our model and invoking the **Validate** menu option brings up a dialog that points out that these required features were not set. As we enhance our model further, EMF and the Validation Framework will provide additional validation, as used by GMF for diagram validation.

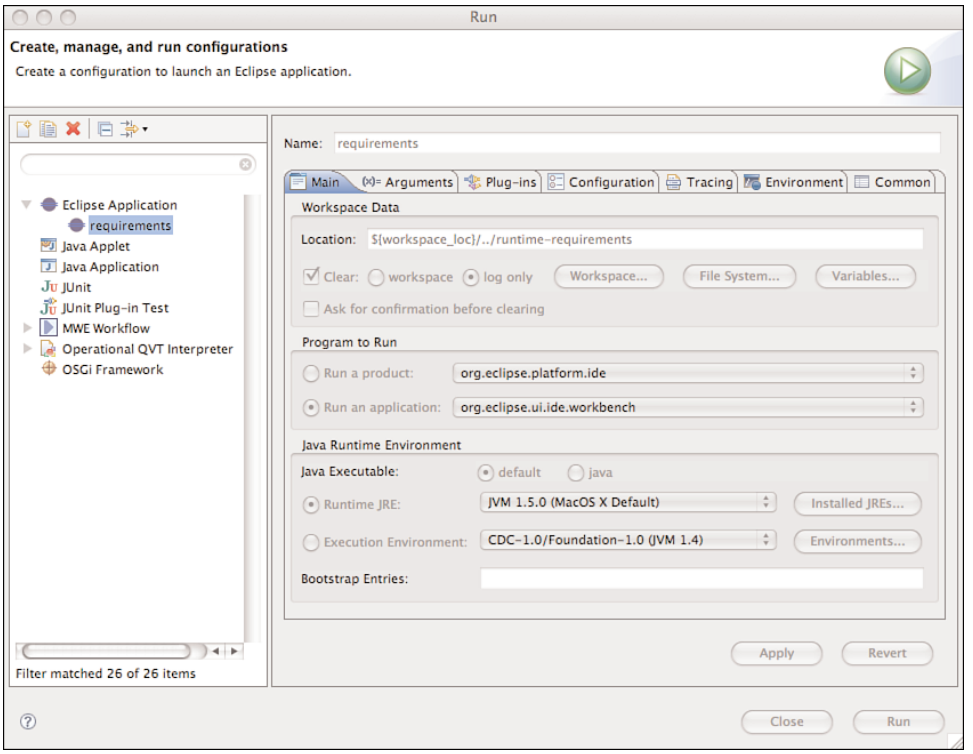

Figure 3-5 Requirements launch configuration

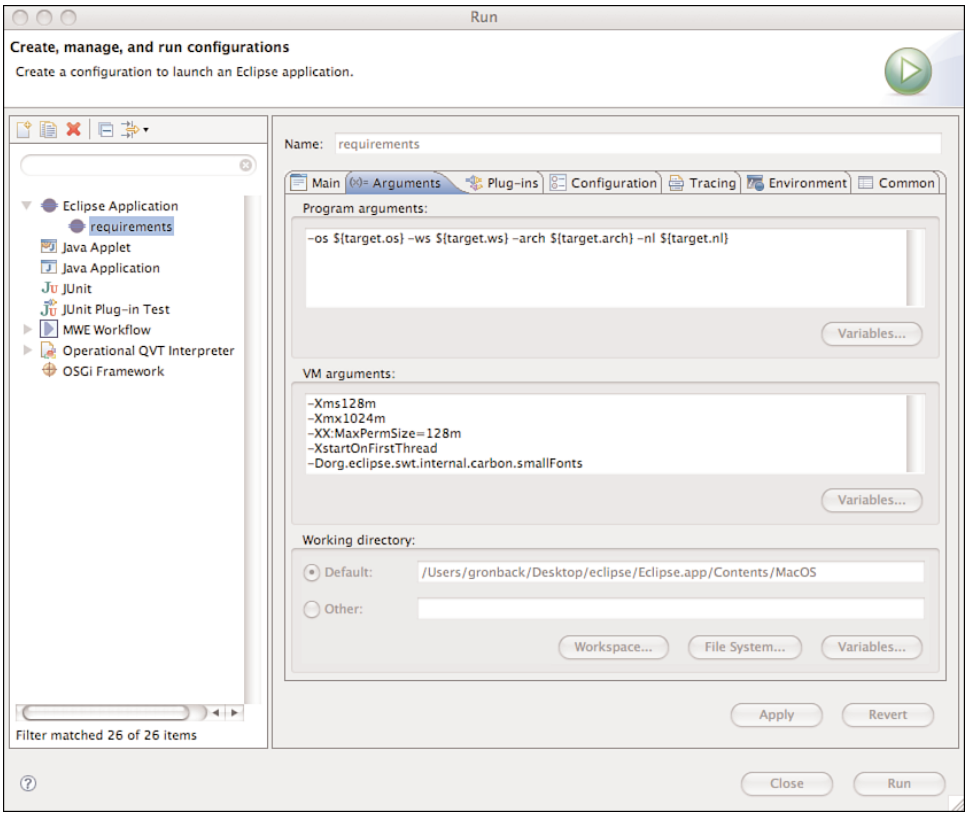

Figure 3-6 Requirements launch configuration arguments

3.3.5 Adding OCL

As you should recall, we added a derived, transient, volatile `rootTopics` reference in our `Map` class. Section 3.2.4, "Applying OCL," described the basics of adding OCL and using dynamic templates to generate implementations for invariant constraints, method bodies, and derived features. The example in that section covered the `rootTopics` implementation using OCL and used a set of dynamic templates that we use in this context as well. At this time, rename the default templates folder to be a templates-domain source folder in the mindmap project, and copy the templates provided in the solution into this folder. We'll have additional templates later for deployment, so we can separate them into different root folders. Each DSL project that uses OCL to refine its domain model

will reuse this set of templates. Then return to Section 3.2.4 and configure the `mindmap.ecore` model to use OCL to implement the `rootTopics` feature.

We can leverage OCL in our model in additional places to provide an implementation and avoid having to modify our generated code. Let's begin by adding a method that returns the full set of subtopics for a given `Topic`.

Finding All Subtopics

Currently, our model has a `subtopics` reference on each `Topic`, along with a method, `allSubtopics()`, that is intended to return a list of all of a `Topic`'s subtopics—that is, its subtopics, all of their subtopics, and so on. All methods declared in an Ecore model require an implementation to be provided, so we turn to OCL, where the implementation of this method is trivial, thanks to the non-standard `closure` iterator in MDT OCL:

```
self->closure(subtopics)
```

We need to add an `EAnnotation` to the method in our model with `Source` equal to `http://www.eclipse.org/2007/OCL`. A child `Details Entry` is added to the annotation with the previous expression as its `Value` property and with a `Key` value of `body`. When we regenerate our model code, we can see that our implementation is provided:

```
/**
 * The parsed OCL expression for the body of the
 * '{@link #allSubtopics <em>All Subtopics</em>}' operation.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @see #allSubtopics
 * @generated
 */
private static OCLExpression<EClassifier> allSubtopicsBodyOCL;

private static final String OCL_ANNOTATION_SOURCE =
  "http://www.eclipse.org/2007/OCL";

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public EList<Topic> allSubtopics() {
  if (allSubtopicsBodyOCL == null) {
    EOperation eOperation =
      MindmapPackage.Literals.TOPIC.getEOperations().get(0);
    OCL.Helper helper = OCL_ENV.createOCLHelper();
```

```
      helper.setOperationContext(MindmapPackage.Literals.TOPIC,
        eOperation);
      EAnnotation ocl = eOperation.getEAnnotation(OCL_ANNOTATION_SOURCE);
      String body = ocl.getDetails().get("body");

      try {
        allSubtopicsBodyOCL = helper.createQuery(body);
      } catch (ParserException e) {
        throw new UnsupportedOperationException(e.getLocalizedMessage());
      }
    }

    Query<EClassifier, ?, ?> query =
      OCL_ENV.createQuery(allSubtopicsBodyOCL);

    @SuppressWarnings("unchecked")
    Collection<Topic> result = (Collection<Topic>) query.evaluate(this);

    return new BasicEList.UnmodifiableEList<Topic>(result.size(),
      result.toArray());
  }
```

Again, here we see the boilerplate OCL code that configures an `OCLExpression` if it's the first invocation, and then it invokes the expression obtained from the annotation. We leave the mindmap model at this point and move on to develop the second domain model in our product line.

3.4 Developing the Requirements Domain Model

In a similar fashion to our mindmap model, we create a new `org.eclipse.dsl.requirements` DSL project here to hold our requirements model. This forms the base of our fictitious Requirements Management Project (RMP). We create the new `requirements.ecore` model using the Domain Model Wizard and GMF Ecore diagram, and we complete it to match the diagram and description of Figure 3-7.

Basically, a model contains a collection of `RequirementGroups`, which contain a number of children groups and `Requirements`. `Requirements` have child references and contain Version and optional Comment elements. A number of enumerations for `Priority`, `State`, `Type`, and `Resolution` are also in the model. A `Requirement` can also have a number of dependent requirements, which become the basis for our dependency diagram. Note that the author attributes are simple strings. We could create a Team model and reference these elements to assign to our requirements and comments. We also could have a separate Discussion model to use here and in our mindmap, as a topic might have an associated discussion thread. Many possibilities exist, but for the purposes of our sample application, we keep it simple.

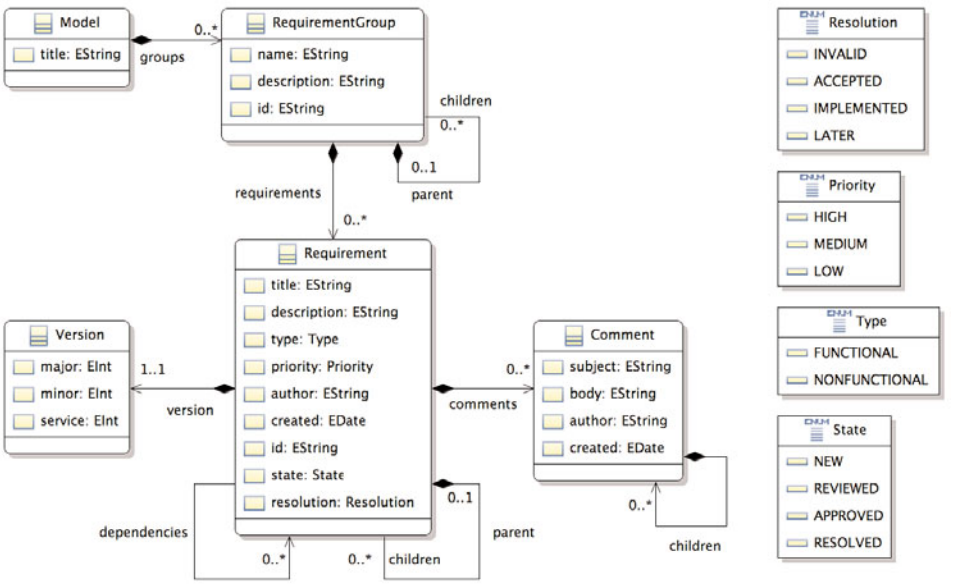

Figure 3-7 Requirements domain model

3.4.1 Requirements Generator Model

We create a `requirements.genmodel` in the usual manner, using the new **Domain Generator Model** (**Ctrl+3 → Domain Gen**) wizard and selecting our `requirements.ecore` model as the input. We'll make some adjustments to this genmodel and to the generated Edit code because we intend to use the generated EMF editor as part of our solution.

For the display string of a requirement in the editor selection tree, we want to have it be `id (major.minor.service):title`, where `major`, `minor`, and `service` are from the contained `Version` element. We'll be using the Properties view to edit the details of the requirement, so we'll have plenty of horizontal space to use in the editor, allowing even longer requirement titles to fit. Another option is to navigate using the Project Explorer view, but this is narrow and does not allow for much information display. Furthermore, we'll have a second tab in the editor to display a requirements dependency diagram, which will also require a bit of editor space. To accomplish the task, we'll select the requirement element in the genmodel and change its `Edit Label Feature` to be our `id:EString` attribute. Unfortunately, we cannot set two attributes to display for the label, as we can for GMF diagrams. This means we have to modify the generated code.

Before generation, we need to check the other properties and make changes accordingly. As with the mindmap and other models, we want to generate our model, edit, and editor code to their own projects, so we can change the **Model Plug-in ID** and **Model Directory** properties to be `org.eclipse.requirements.model`. We generate the three plug-ins and open the `org.eclipse.requirements.provider.RequirementItemProvider` class from our Edit plug-in. Modify the `getText()` method as seen next. Note that if we wanted to preserve the generated method to allow the label feature of the generator model to have an effect, we could use the `getTextGen()` method approach, as described in the EMF documentation.

```
/**
 * This returns the label text for the adapted class.
 * Modified to show id (major.minor.service) : title
 *
 * @generated NOT
 */
@Override
public String getText(Object object) {
  StringBuilder sb = new StringBuilder();
  sb.append(((Requirement)object).getId());
  sb.append(" (");
  Version version = ((Requirement)object).getVersion();
    if (version != null) {
      sb.append(((Requirement)object).getVersion().getMajor());
      sb.append(".");
      sb.append(((Requirement)object).getVersion().getMinor());
      sb.append(".");
      sb.append(((Requirement)object).getVersion().getService());
    } else {
      sb.append("0.0.0");
    }
    sb.append(") : ");
    sb.append(((Requirement)object).getTitle());
    String label = sb.toString();
    return label == null || label.length() == 0 ?
      getString("_UI_Requirement_type") : label;
}
```

We've eliminated the redundant `Requirement` prefix from our label because we're using a custom icon to distinguish `Requirements` from `RequirementGroups`, `Comments`, and so on. For our `RequirementGroup` element, we can similarly modify the `getText()` method to display only the `name` attribute; we can modify the `Comment` element to display `created`, `author`, and `subject`.

3.5 Developing the Scenario Domain Model

Because we're basing the notation for our Scenario diagram on the Business Process Modeling Notation (BPMN) standard, we could simply use its description of the underlying domain model and semantics to develop our DSL. A better approach would have been to find an XSD for BPMN and simply import it into EMF. Unfortunately, no such schema is published with the specification—even worse, a new specification from the OMG, the Business Process Definition Metamodel (BPDM), is slated to provide a domain model for BPMN2. Also unfortunate is that this specification has no serialized format that we can use and is overly complicated for our Scenario DSL. This leaves us to create our own scenario model.

In a new `org.eclipse.dsl.scenario` project, we can create our `scenario.ecore` model as the base of our fictitious Requirements Scenario Project (RSP) project. Figure 3-8 is the model to create using our Ecore diagram.

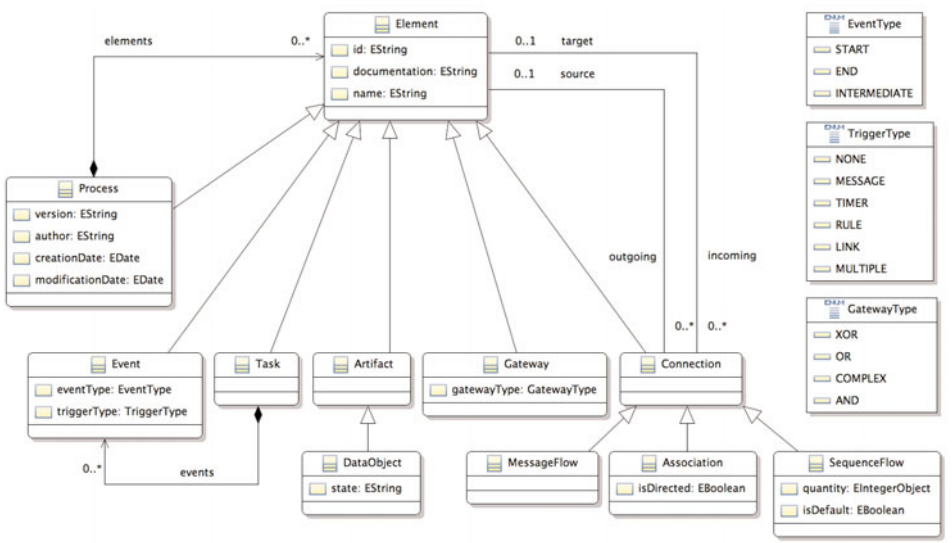

Figure 3-8 Scenario domain model

Elements of a scenario model are maintained in the `Process` class, which itself extends `Element`. A `Connection` maintains target and source references for `Elements` that are connected in `Sequence` or `Message` flows. An `Association` also connects elements. `Elements` come in a variety of types, including `Tasks`, `Events`, `DataObjects`, and `Gateways`. These elements all map in a straightforward manner to notation elements because the model is

inherently graphical in nature. The model is actually similar to the description provided in the BPMN specification, although it is a subset.

3.6 Developing the Business Domain Model

Plenty of options exist for developing the domain model that will form the base of our fictitious Business Domain Modeling (BDM) project. We want something less complicated than the Unified Modeling Language (UML) model for structural class modeling, but something expressive enough to generate code either directly or through an intermediate model. Also, the model should constrain the user to the supported methodology and domain of business models. For the purposes of this book, the four archetypes described for business domain modeling in *Java Modeling in Color with UML* [46] seem like a good option. Figure 3-9 is a partial image of the Domain-Neutral Component (DNC) model, created with the editor we develop in Section 4.6, "Developing the Color Modeling Diagram." Of course, a black-and-white rendering of a color modeling diagram in the printed form of this book is not very compelling.

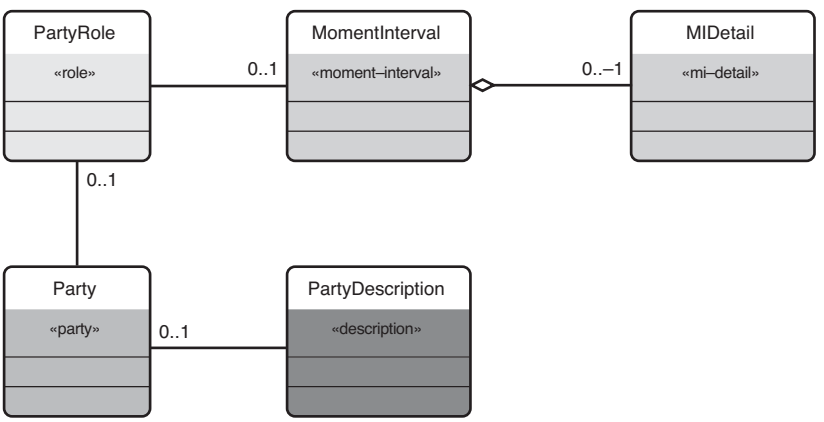

Figure 3-9 Color archetypes

Basically, a set of archetypes is used to model moment-interval, role, party, place, thing, and description elements and their relationships. The relationships and several described patterns of interaction are provided in the book, which we want to facilitate in our modeling environment. First, however, we need an underlying model (a DSL).

This DSL is strongly rooted in a general object-oriented DSL, so we begin with just that. Figure 3-10 is an `oocore.ecore` model that we extend to add our

archetypes and other DNC elements. Why begin with a general object-oriented DSL? Well, we might decide to use this model as the basis for another DSL in the future. Why not simply extend Ecore itself, you might ask? Well, it contains elements that we really don't need or want, leaving us with all those *E*-prefixed elements and their properties. Besides, it's straightforward to develop our own object-oriented DSL. We can use the Java EMF Model (JEM) as a transformation target, giving us a chance to see what a model that extends Ecore is like.

Adventurous types can create a new `org.eclipse.dsl.oocore` DSL project and create the `oocore.ecore` model, as we have done previously. Complete the model using Figure 3-10 as a reference. Otherwise, simply copy the `oocore.ecore` domain model from the solutions into your project. Finally, create a new `org.eclipse.dsl.dnc` DSL project to hold our `dnc.ecore model` that will extend our core model.

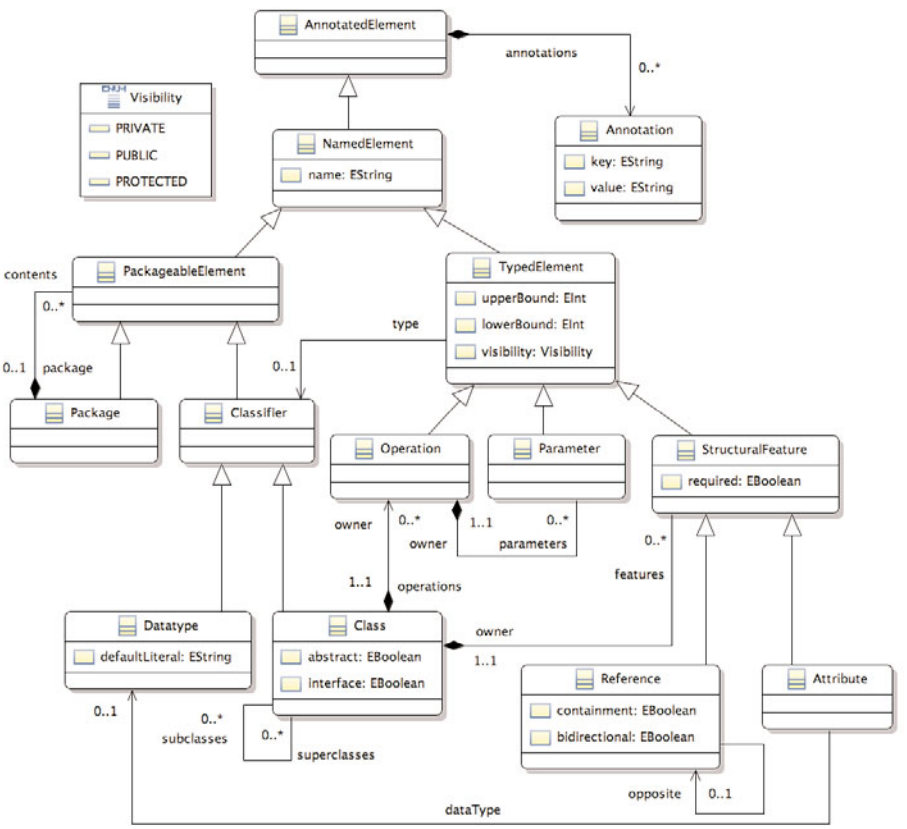

Figure 3-10 Object-oriented core domain model

With our base model complete, we can create our `dnc.ecore` model. To reference our `oocore.ecore` model in our newly created `dnc.ecore` model, we need to use the EMF **Load Resource** context menu in the default EMF editor. Fortunately, the dialog that appears now contains options to **Browse Registered Packages**, **Browse File System**, and **Browse Workspace**. At one time, you needed to enter `platform:/` URIs into the field to load a registered package. In our case, the `oocore.ecore` model is easily found in our workspace.

In creating our DNC model (Figure 3-11), several options exist, as always. You've seen that using an enumeration to define the type is one solution, as was used in the Mindmap domain model's `Relationship` class and `Type` enum. This approach has some drawbacks, including the loss of polymorphic behavior in our templates and transformation scripts. To illustrate the differences, let's go ahead and create an `Archetype` abstract class that extends our `oocore::Class` class. Each of our archetypes will extend the `Archetype` class. We also add an `Association` class that extends `oocore::Reference` and add a property to signify aggregation. Although it is not a true `Association` class in the UML sense, it aids us in developing our diagram and hiding some complexities of the underlying model to the Practitioner. As we develop the diagram and other DSL artifacts, we'll revisit this model and refine it as necessary, potentially pulling up some functionality into the domain model to aid in our color modeling and model transformations.

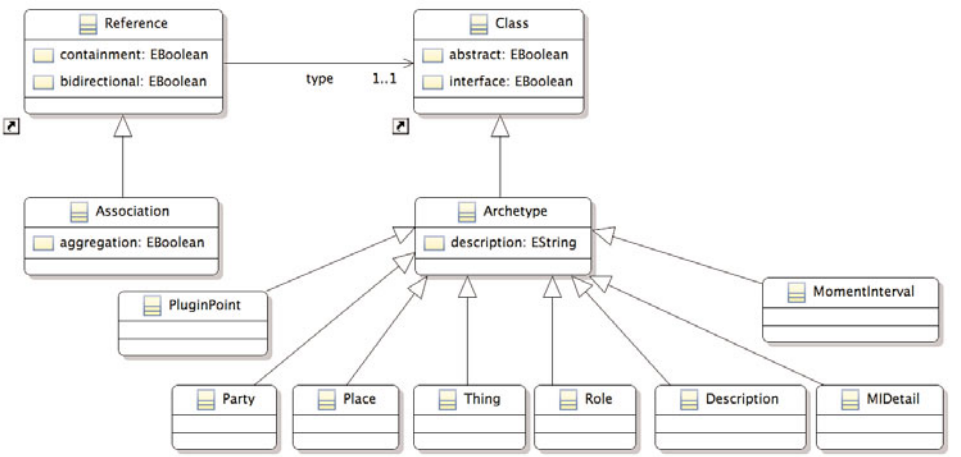

Figure 3-11 Domain-neutral component domain model

3.7 Summary

This chapter explored the capabilities of EMF as the means of describing our DSL abstract syntax. Although we leave the details of EMF to its own book, we covered enough to get started developing our sample projects. The benefits of leveraging a common underlying metamodel and generation capabilities should become clear as we continue to develop the DSL projects.

At this point, we have starter domain models for our fictitious ERP, plus a reference to a fifth oocore DSL. We now move on to describing how to create graphical concrete syntax for those we want to provide diagrams for, understanding that we will most likely revisit and enhance the EMF models we created in this chapter.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X-Y-Z