4.1.4 Synchronization

Layout can go a long way toward making diagrams more readable and can even convey semantic information. Keep in mind that diagrams represent a view of the underlying model. In the case of large, complex models, it's often best to have multiple diagrams, or views, of the model. You can use filters to accomplish these views, as mentioned earlier, or the views can be distinct diagram instances that the Practitioner creates. The question of synchronicity comes up frequently at this point because sometimes we want the diagram to update automatically based on changes to the underlying model, sometimes we want changes to be made only manually, and sometimes we want a hybrid approach in which elements on the diagram should update but no new elements should be added. We explore synchronization options in GMF in Section 11.4.2, "Synchronized."

4.1.5 Shortcuts

Toward the goal of creating specific views of our model, we often need to create shortcuts, or aliases, of model elements on diagrams that are essentially imported from another diagram or model. Support for shortcuts on diagrams is common, as is the capability to have more than one notation element represent the same underlying domain model element. We explore these options as we develop our sample diagrams. Shortcuts are supported in diagrams generated with GMF and are covered in Section 11.4.2, "Contains Shortcuts To and Shortcuts Provided For."

4.2 Graphical Modeling Framework

Before GMF, many had undertaken the task of binding the model aspect of the GEF's Model-View Controller (MVC) architecture to an EMF model. An IBM Redbook was written [43], a sample was provided by the GEF project, and numerous commercial and academic institutions implemented solutions, some of which included a generative component. GMF came about as the result of this need for an easier way to develop graphical editors using GEF and an underlying EMF model.

Today GMF consists of two main components: a runtime and a tooling framework. The runtime handles the task of bridging EMF and GEF while providing a number of services and Application Programming Interfaces (API) to allow for the development of rich graphical editors. The tooling component provides a model-driven approach to defining graphical elements, diagram tooling, and mappings to a domain model for use in generating diagrams that leverage the runtime.

4.2.1 GMF Runtime Component

GMF has two runtime options. The first is commonly referred to as just the runtime; the second is referred to as the "lite" runtime. The former provides extensive capabilities for extension, and the latter focuses on providing a small installation footprint and is largely generative. These two runtimes represent two distinct philosophies of how to provide a diagramming runtime. Even more fundamentally, perhaps, they illustrate two approaches to Model-Driven Software Development (MDSD) in general.

The full runtime was originally developed as an extensible framework for creating diagrammatic editors on Eclipse Modeling Framework (EMF) and GEF. This runtime was originally designed and developed to provide rich extensibility options for clients. It includes a rich set of APIs, extension-points, a service layer, and many enhancements to the underlying EMF and GEF runtimes. The full runtime can be used with or without the tooling and generation features of GMF. The default target of the tooling and generation component is the full runtime.

Details on the GMF runtime, its APIs, and extension-points are described in Chapter 10, "Graphical Modeling Framework Runtime." Although it's not necessary to understand the inner workings of the GMF runtime during the initial phase of development using the tooling and generation component, you will eventually need to provide functionality that goes beyond what is generated.

The GMF Lite Runtime

Whereas the full runtime provides a rich published API, numerous extension-points, a service provider layer, and more, the lite runtime is just the opposite. The motivation for the lite runtime was to provide as much of a generated implementation for diagramming as possible, with a minimal runtime code base. The current implementation of the lite runtime consists of a single runtime plug-in, with a single extension-point for supporting diagram shortcuts.

To provide compatibility with GMF diagrams created for the full runtime, the lite runtime uses the same notation model. In theory, a diagram produced with an editor generated using the lite runtime option will open in an editor generated to the full runtime. Some missing features in the lite runtime prevent full interoperability, but it does work, to an extent.

To target the lite runtime when using the tooling, you must first deselect the **Utilize Enhanced Features of the GMF Runtime** and **Use IMapMode** options when creating the generator model. When generating diagram code from the generator model, use the **Generate Pure-GEF Diagram Code** option.

The lite runtime requires a single `org.eclipse.gmf.runtime.lite` plug-in for deployment, along with its dependencies, which include the Eclipse platform core, EMF, GEF, EMF Transaction, and tabbed properties view. Although

it is not as feature rich as the full runtime, it offers a core set of runtime capabilities, including diagram properties, preferences, shortcuts, and validation. For tooling, the lite runtime has its own set of Xpand templates for generation, found in the `org.eclipse.gmf.codegen.lite` plug-in, available in the GMF Experimental SDK feature.

4.2.2 GMF Tooling Component

As you will see, GMF was itself developed as a DSL Toolkit. From the beginning, it was decided that the tooling for GMF would be as model driven and bootstrapped as possible. In short, a diagram is defined using a collection of models (DSLs) that drive code generators targeting either the full runtime or the lite runtime. One of the remaining tasks to complete the story is to use Query/View/Transformation (QVT) in the transformation from its mapping model to generation model, to considerably improve the extensibility of GMF's tooling.

Figure 4-1 illustrates the main components and models used on the tooling side of GMF. To begin, a GMF project is created and references a domain model. A graphical definition model designs the figures (nodes, links, compartments, and so on) that will be used to represent domain model elements on the diagram surface. A corresponding tooling definition model supports palette tool definition and other tooling for use in diagramming. The mapping model binds elements from the graphical and tooling definitions to the domain model. A transformation from the mapping model to the generator model is followed by the generation of a diagram plug-in.

Note that it is possible to design and run GMF diagrams without a domain model, which can be useful for those who want to experiment with notation design and not be burdened with mapping it to a domain. Each of these models is described in some detail in the following sections, and Chapter 11, "Graphical Modeling Framework Tooling," includes a complete reference for each model. Following the basic overview of each model, we turn to learning more about them in the context of developing our sample application diagrams.

Graphical Definition Model

The graphical definition model consists of two parts and defines the graphical elements found on a diagramming surface. The first part is a Figure Gallery, which defines figures (shapes, labels, lines, and so on) that the Canvas elements later reference to define nodes, connections, compartments, and diagram labels. An important point is that figure galleries can be reused. Many diagrams require similar-looking elements, such as a rounded rectangle with center label, or connections that are a solid line with open arrowhead decoration on the target end.

Defining a number of figures and sharing galleries within your organization or larger community means less time spent reinventing the wheel. For UML2, a set of figures are defined and available for reuse from the UML2 Tools component of the Model Development Tools (MDT) project.

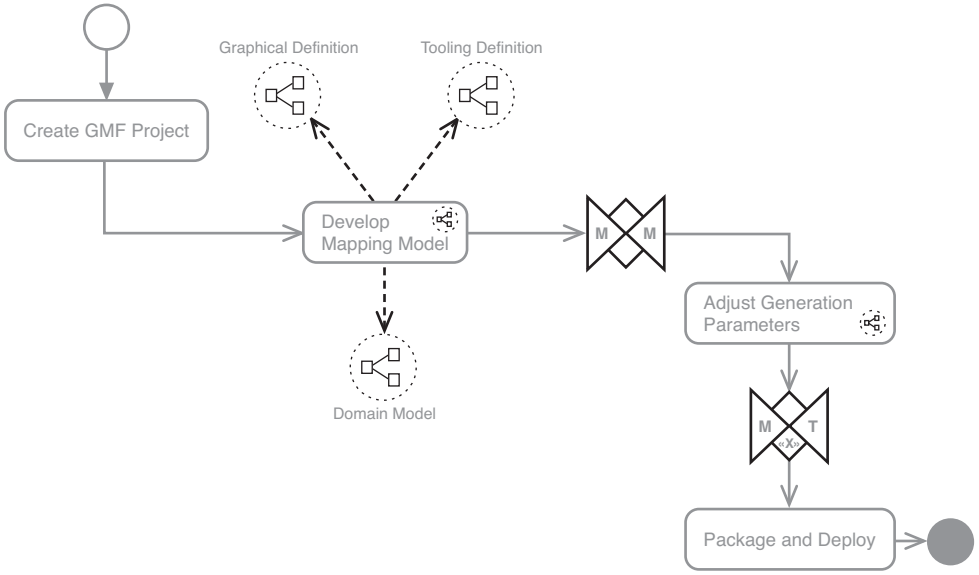

Figure 4-1 Graphical Modeling Framework workflow

The mapping model references figures defined in the gmfgraph model. When the mapping model is transformed to the generator model, figure code is generated and included within the gmfgen model itself. When code is generated, edit parts will contain figures as inner classes. This is the default behavior when working with GMF, although it is not necessarily the recommended approach.

Another lesser-known feature of the graphical definition model is the capability to export figures to a standalone figure plug-in. This can also satisfy reuse because these plug-ins can be shared by several diagrams and among a community as a binary form of the figure gallery. To create a figure plug-in from a gmfgraph model, either use the **Generate Figures Plug-In** context menu action, or start by creating a new plug-in project and select the **Figure Definitions Converter** template in the plug-in project wizard. Section 4.5.5, "Generating the Figures Plug-In," covers the use of a standalone figure plug-in.

A complication that arises when using a standalone figures plug-in is that it creates a "mirror" of the gmfgraph model. This adds one more model to the

picture that needs to be synchronized, so it's recommended that you first define all your figures and generate the plug-in that contains the associated `mirror.gmfgraph` model. The mapping model uses the mirrored model instead of the original graphical definition model. With this approach, the mapping model passes class name references to the generator model, not actual classes. The generated code simply references the figure classes in their own plug-in and is not written out as inner classes within generated edit parts. This process imposes the additional step of regenerating the figures and mirrored model upon a change to the graphical definition. A future version of GMF will hopefully do away with the serialized class method and make generating a standalone figures plug-in work more seamlessly in the workflow.

To ease the design of figures, a WYSIWYG (what you see is what you get) style of editor is included in the experimental Software Development Kit (SDK). It is not complete, but it illustrates the bootstrapping of GMF and a method for customization using a decorator model described in detail in Section 4.2.3, "Dynamic Templates." Use of the graphical model editor is also helpful for understanding layouts and how they work when composing complex figures.

Tooling Definition Model

Diagrams typically include a palette and other periphery to create and work with diagram content. The purpose of the tooling definition model is to specify these elements. The tooling model currently includes elements for the palette, the toolbar, and various menus to be defined for a diagram. Unfortunately, in the current release of GMF, the generator uses only the palette element. If additional capabilities are required until this functionality is completed, advanced properties view UI elements can be designed using an extension to the generator model and custom templates, as discussed in Section 4.6.6, "Color Preferences." Note that it is also possible to exclude the palette altogether from a diagram definition, thereby creating a read-only diagram. Of course, the pop-up bars and connection handles features should be disabled as well in this case.

Mapping Model

Perhaps the most important of all models in GMF is the mapping model. Here, elements from the diagram definition (nodes and links) are mapped to the domain model and assigned tooling elements. The mapping model represents the actual diagram definition and is used to create a generator model. Typically a one-to-one mapping exists among a mapping model, its generator model, and a particular diagram.

The mapping model uses Object Constraint Language (OCL) in many ways, including initializing features for created elements, defining link and node

constraints, and defining model audits and metrics. Audits identify problems in the structure or style of a diagram and its underlying domain model instance, and metrics provide measures of diagram and domain model elements.

Generator Model

As mentioned in the overview, the generator model adds information used to generate code from the mapping model and is somewhat analogous to the EMF genmodel. Both can be reproduced and reloaded from their source models, although the EMF genmodel is a true decorator model. The GMF generator model is more of a many-to-one model transformation than a decorator model.

As a mapping model is transformed into a generator model, it loses knowledge of the graphical definition and gains knowledge of the runtime notation model. This minimizes the number of dependencies linked from the generator model and separates concern among the models. Currently, the transformation is performed using Java code, but it is planned to be reimplemented using QVT to give Toolsmiths easier customization, as mentioned earlier.

A trace facility exists in the experimental SDK to aid in generating visual IDs when new nodes are added and the generator model is updated. A reconciler preserves other user-modified elements in the generator model upon retransformation from the mapping model. Many of the commonly modified properties are preserved, although not all of them are, so be aware of this when making changes to the generator model.

As with EMF, you can use custom code-generation templates in GMF. The main difference here is that EMF uses Java Emitter Template (JET) as its template engine, and GMF uses Xpand. Chapter 14, "Xpand Template Language," covers Xpand, which also is used throughout Chapter 7, "Developing Model-to-Text Transformations." You can find information on how to use dynamic templates in GMF in Section 4.2.3, "Dynamic Templates," and in our sample diagram in Sections 4.3–4.6.

When using the full runtime as a generation target, a number of extension-points are contributed to in the generated diagram code. You will likely want to explore the generated plug-in manifest and source code.

TIP

Sometimes you must open and modify GMF definition models in a text editor. When doing so, add new elements that are part of a list of items to the end of the list, because GMF models use relative position references. For example, if you're copying a figure from one .gmfgraph model to another, add it after the last descriptors element in the file.

4.2.3 Customization Options

You can extend the GMF generator in several ways, all of which are analogous to how you can use and provide for extensibility in your DSL tooling. The next sections discuss code modification, extension-points, dynamic templates, and decorator models, and illustrate them in the sample applications.

Code Modification

As GMF utilizes JMerge to protect Toolsmith modifications of generated code from being overwritten, the same practice of placing `NOT` after `@generated` tags in code can be used as with EMF. Additionally, GMF provides merge capabilities for `plugin.xml` and `MANIFEST.MF` files, which is a nice feature that EMF should consider adopting.

Extension-Point

When targeting the full runtime for generation, provided extension-points can be used to extend diagrams generated using the tooling component of GMF. This approach has the benefit of being completely separate from the generated diagram and code. For example, a parser provider for our color modeling diagram's attribute elements is provided in this manner, as discussed in Section 4.6.7, "Custom Parsers." The service-provider aspect of the runtime allows for the addition or overriding of behavior in diagrams, such as the addition of EditPolicies to an EditPart, as illustrated in Section 10.9.3, "Custom EditPolicy."

Dynamic Templates

Also as in EMF, you can leverage dynamic templates to provide customized output from GMF code generators. You can extend or override both the templates used to generate figure code and the templates used to generate diagram code using so-called dynamic templates.

GMF uses Xpand extensively. To override a template for diagram generation, you must put it in the same directory structure (namespace) that GMF uses. The easiest way to see the templates and their structure is to import the `org.eclipse.gmf.codegen` plug-in into your workspace using the **Import As → Source Project** option from the Plug-Ins view. Note also that GMF templates contain «`DEFINE`» entries for `extraMethods` and `additions` with corresponding «`EXPAND`»s to allow for extensibility. When using the «`AROUND`» construct for aspect-oriented features of Xpand, GMF requires placing these templates under an `/aspects` folder below the root in order to be found. GMF recently added a new "composite template" approach that makes it possible to augment

an existing template if found in the same namespace, effectively merging its content with the original. Sections 4.6.5, "Gradient Figures," and 4.6.6, "Color Preferences," describe the use of custom templates with GMF in detail.

Decorator Model

A more advanced—and possibly most conceptually "pure"—method for customizing or extending the output of GMF code generators is to use a decorator model. Basically, the GMF generator model is wrapped in a root XML Metadata Interchange (XMI) element to allow additional decorator model instances to coexist and to enable elements of the generator model to reference them. Xpand templates used to generate diagram code are augmented with custom templates that are invoked when these references are encountered.

The GMF graphical definition model has a bootstrapped diagram editor to allow for WYSIWYG-style figure development. It was implemented using custom templates and also includes a decorator model for use in defining its form-based properties view. This serves as an example of how to use decorator models in the context of GMF, but also for any other occasion in the context of using a DSL Toolkit where extensions are required to an existing model used for generation. Section 4.6.6, "Color Preferences," covers the steps in using decorating models in GMF.

Model Extension

With the addition of the child extenders feature in EMF 2.4, it's possible to have your contributed model elements of customizations to the GMF models available in the default editor. GMF 2.1 has been regenerated with these generator model settings, thereby allowing your extensions to contribute to GMF editors. The UML2 Tools project has extensions defined for GMF models and makes use of this new capability.

4.2.4 Dashboard

GMF comes with a dashboard view that streamlines the workflow of dealing with its collection of models. The dashboard is available from **Window → Show View → Other → General → GMF Dashboard** (Ctrl+3 → gmfd), or you can open it when creating a new GMF project. Figure 4-2 shows the dashboard used in the context of the mindmap diagram sample project.

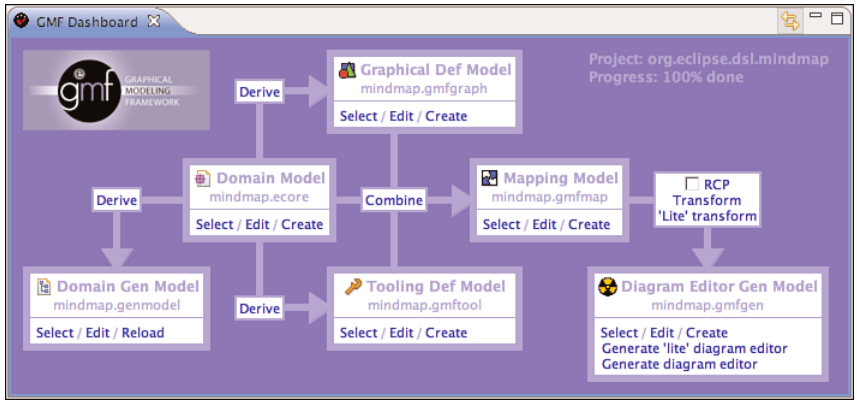

Figure 4-2 GMF Dashboard view

Each model can be selected, created, or edited within the dashboard, including EMF *.`ecore` and *.`genmodel` models. Invoking GMF and EMF wizards is accomplished using the hyperlink actions throughout, making the dashboard helpful not only in understanding the workflow, but also in streamlining the invocation of transformation and generation actions during diagram development.

4.2.5 Sample Application Diagrams

The best way to learn how to use GMF is by example, as with most new technology. The following sections explore most aspects of GMF-based diagram definition in the context of our sample projects, by design. Comments throughout should illustrate the techniques for developing diagrams, enumerate their relative pros and cons, and provide the basis for becoming well versed in GMF tooling.

Because diagramming is central to mindmaps, I pay special attention to this diagram, particularly layout and other usability elements. The requirements dependency diagram is similar to the mindmap, but the underlying model structures are different, so we explore how this impacts our mapping model. The scenario diagram enables us to explore the concept of diagram partitioning. Finally, our business domain modeling diagram explores compartments, customization, and more advanced labeling techniques.

4.3 Developing the Mindmap Diagram

Our diagram for the mindmap DSL defined earlier is rather straightforward: It is a simple "box and line" style of diagram, but one that serves us well in introducing GMF. We start out with a simple default diagram definition to first understand