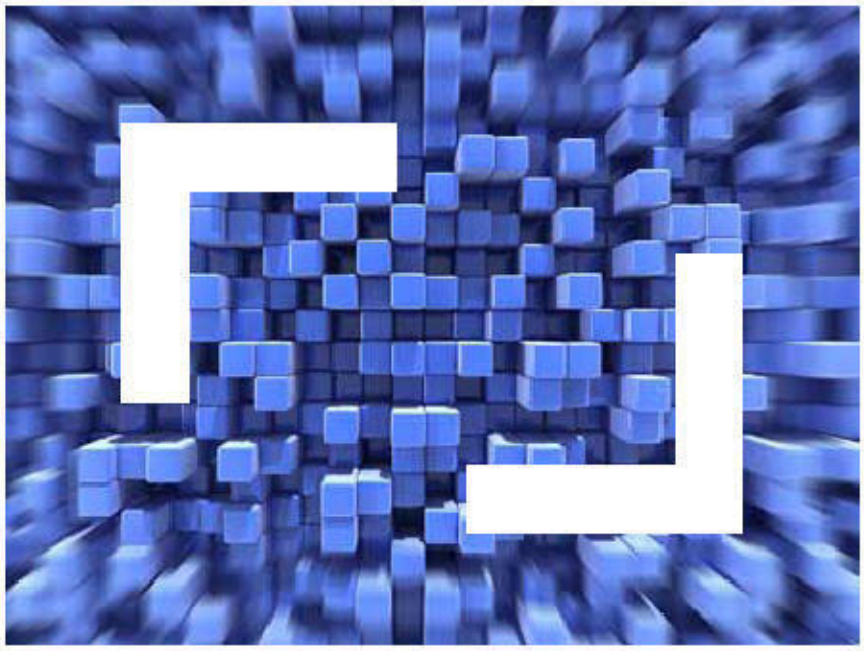

Borland Together 2008

Borland Together DSL Toolkit Guide

Borland®
(A MICRO FOCUS COMPANY)

MICRO FOCUS®
Leading the Evolution™

Borland Software Corporation
4 Hutton Centre Dr., Suite 900
Santa Ana, CA 92707

Contents

Getting Started with Together DSL Toolkit

Together DSL Toolkit Overview

Welcome to the Borland® Together DSL Toolkit. The Together DSL Toolkit provides the advantages of domain-specific languages, including increased productivity, faster turnaround time, and adherence to the guidelines and constraints specified in the DSL.

The following resources offer additional information and assistance:

- *Borland Together Home Page*
- *Borland Together Documentation*
- *Borland Product Support*

For information on how to use this Help system, see Help on Help in the Related Concepts.

Note: If your Internet access is limited by network security, or if your computer is protected by a personal firewall, the Web-based links in this Help system might not function properly.

Note: Since Together 2008 Release 3, the feature set DSL Toolkit is not required to be installed. When not present, the corresponding parts of the user interface and functionality are not available. Refer to the installation instructions in the Release Notes document for additional info about the product installation process.

Related Topics

Related Resources

The following Web resources offer additional information to supplement the *DSL Toolkit Guide* and *Modeling Guide* Help.

Note: If your Internet access is limited by network security, or if your computer is protected by a personal firewall, the Web-based links in this Help system might not function properly.

- *Eclipse Modeling Project*
- *Eclipse Graphical Framework Modeling Project*
- *Eclipse Modeling Framework (EMF) Project (includes Query, Transaction, Validation)*
- *Eclipse Model-to-Model Transformation (M2M) Project (includes Procedural QVT [Operational] and Declarative QVT [Core and Relational])*
- *Eclipse Model-to-Text (M2T) Project (includes Xpand and JET)*
- *Eclipse Model Development Tools (MDT) - UML2*
- *Eclipse Model Development Tools (MDT) - UML2Tools*
- *Eclipse Model Driven Development integration (MDDi)*
- *Eclipse Generative Modeling Technologies (GMT)*
- *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*

- *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*
- *OMG UML Resource Page*
- *OMG Meta-Object Facility Query-View-Transformations (MOF QVT) Final Adopted Specification, 1.0*
- *Object Constraint Language OMG Available Specification Version 2.0*
- *OMG Business Process Modeling Notation (BPMN) Specification, 1.1*
- *Borland's Model Driven Development Resource Center*

The following newsgroups, mailing lists, and forums offer user discussions and additional insights into modeling within an Eclipse environment.

- *Forum: Together*
- Eclipse Platform:
  - *Eclipse NewsPortal - eclipse.platform*
  - *Eclipse NewsPortal - eclipse.platform.swt*
  - *Eclipse NewsPortal - eclipse.platform.rcp*
  - *Eclipse NewsPortal - eclipse.platform.ua*
  - *Mailing list: platform-dev*
  - *Mailing list: platform-ant-dev*
  - *Mailing list: platform-compare-dev*
  - *Mailing list: platform-core-dev*
  - *Mailing list: platform-cvs-dev*
  - *Mailing list: platform-debug-dev*
  - *Mailing list: platform-doc-dev*
  - *Mailing list: platform-ide-dev*
  - *Mailing list: platform-releng-dev*
  - *Mailing list: platform-scripting-dev*
  - *Mailing list: platform-search-dev*
  - *Mailing list: platform-swt-dev*
  - *Mailing list: platform-team-dev*
  - *Mailing list: platform-text-dev*
  - *Mailing list: platform-ua-dev*
  - *Mailing list: platform-ui-dev*
  - *Mailing list: platform-update-dev*
  - *Mailing list: platform-webdav-dev*
  - *Eclipse NewsPortal - eclipse.platform*
  - *Eclipse NewsPortal - eclipse.platform.swt*
  - *Eclipse NewsPortal - eclipse.platform.rcp*
  - *Eclipse NewsPortal - eclipse.platform.ua*
  - *Mailing list: platform-dev*
  - *Mailing list: platform-ant-dev*
  - *Mailing list: platform-compare-dev*
  - *Mailing list: platform-core-dev*
  - *Mailing list: platform-cvs-dev*
  - *Mailing list: platform-debug-dev*
  - *Mailing list: platform-doc-dev*
  - *Mailing list: platform-ide-dev*
  - *Mailing list: platform-releng-dev*
  - *Mailing list: platform-scripting-dev*
  - *Mailing list: platform-search-dev*

    - *Mailing list: platform-swt-dev*
    - *Mailing list: platform-team-dev*
    - *Mailing list: platform-text-dev*
    - *Mailing list: platform-ua-dev*
    - *Mailing list: platform-ui-dev*
    - *Mailing list: platform-update-dev*
    - *Mailing list: platform-webdav-dev*
- Eclipse Modeling Framework (EMF)
    - *Eclipse NewsPortal - eclipse.technology.emft*
    - *Eclipse NewsPortal - eclipse.tools.emf*
    - *Mailing list: emft-dev*
    - *Mailing list: emf-dev*
- Graphical Modeling Framework (GMF)
    - *Eclipse NewsPortal - eclipse.modeling.gmf*
    - *Eclipse NewsPortal - eclipse.technology.gmf*
    - *Mailing list: gmf-dev*
- Model Development Tools (MDT)
    - *Eclipse NewsPortal - eclipse.modeling.mdt.eodm*
    - *Eclipse NewsPortal - eclipse.modeling.mdt*
    - *Eclipse NewsPortal - eclipse.modeling.mdt.ocl*
    - *Eclipse NewsPortal - eclipse.modeling.mdt.uml2*
    - *Eclipse NewsPortal - eclipse.modeling.mdt.uml2tools*
    - *Eclipse NewsPortal - eclipse.modeling.mdt.xsd*
    - *Mailing list: mdt.dev*
    - *Mailing list: mdt-eodm.dev*
    - *Mailing list: mdt-ocl.dev*
    - *Mailing list: mdt-uml2.dev*
    - *Mailing list: mdt-uml2tools.dev*
    - *Mailing list: mdt-xsd.dev*
- M2* Transformations
    - *Eclipse NewsPortal - eclipse.modeling.m2t*
    - *Mailing list: m2t-dev*
    - *Eclipse NewsPortal - eclipse.modeling.m2m*
    - *Mailing list: m2m-dev*
    - *Mailing list: m2m-atl-dev*

Concepts

Domain-Specific Languages

A domain-specific language (DSL) is designed to accomplish a specific task or type of task. A well-designed DSL provides an environment that closely matches a task's specific needs. Domain experts can typically be more effective because the DSL uses the vocabulary and concepts of the domain. As a result, domain experts can more quickly build proper models without having to learn a different taxonomy because the models map to the domain concepts. DSLs are typically designed at a high level of abstraction, such as process modeling, and used to generate supporting software artifacts, such as source code and auxiliary text files.

The term *language* in this context can refer to both programming languages and modeling languages. Both types of languages are built upon syntactic and semantic rules. In the context of DSLs, both types of languages can be used. In fact, it is common for a DSL to include several languages for its various components. It might use a modeling language for its models, a programming language for its implementation and a transformation or template language to produce desired output.

General-purpose languages are designed to handle many different types of tasks. As such, they typically provide generalized and extensible capabilities that are well-suited for many tasks. The Unified Modeling Language (UML) is an example of a general-purpose modeling language.

Note: The UML can be seen as a collection of DSLs that include class modeling and activity modeling. As a result, the UML is ideal for modeling parts of applications, such as class models or state machines. At the same time, the UML can be confusing due to its sheer size and complexity, especially for those who fail to understand the boundaries between those DSLs.

Related Topics

DSL Capabilities on page 11

DSL Toolkit Workflow on page 12

DSL Toolkit Usage Scenarios on page 18

DSL Toolkit Best Practices on page 14

Domain-Specific Modeling

By nature of their extensibility, some programming languages like LISP, Smalltalk and Ruby can be tailored to provide a programming environment that matches the needs of particular types of tasks. This approach has limitations because the resulting DSL, which Martin Fowler calls an *internal DSL*, is constrained by the features and capabilities of the general-purpose programming language. The problem domain does not need many capabilities that the general-purpose programming language provides. This makes the DSL more difficult to learn and use.

An external DSL is written in a language other than the language the DSL itself uses. While their use has certain advantages in text-based DSLs, external DSLs can cause difficulties in using and understanding the DSL as well as extending or modifying the DSL when necessary. Potential disadvantages include the lack of a proper semantic editor and the lack of debugging facilities.

The DSL Toolkit takes a model-centric approach to DSLs. This approach lets the Toolsmith focus on accurately modeling the domain. The domain model is the longest-lived and most valuable part of the DSL. This model-driven approach insulates the domain model from changes in the underlying technologies. As technologies evolve, the domain model can be retargeted with alternate transformations and templates.

The model-driven approach offers the following advantages:

- Because the domain model uses the vocabulary of the domain and accurately captures the entities, relationships, and rules, a single source can generate the rest of the DSL.
- Because the DSL is tightly coupled to the domain, a Practitioner can use the domain terminology and concepts with ease.
- A necessary framework for data integrity on the technology side and high usability on the human side is provided.

The success of any DSL depends on how easily its users can apply it. Text-based approaches require extreme care during editing and provide little assistance in avoiding or detecting errors. The model-based approach of the DSL Toolkit uses familiar concepts such as models and diagrams integrated into the already familiar Eclipse workbench. The domain metamodel and Object Constraint Language (OCL) constraints help prevent errors.

DSL Toolkit

A well-designed DSL meets the specific needs and requirements of the domain while providing an environment that is more concise and easier to use. Key benefits of a DSL include increased productivity, faster turnaround time, and adherence to its own guidelines and constraints.

The DSL Toolkit provides such an environment in the Eclipse Modeling Framework (EMF), which is the standard framework for models in the Eclipse platform. It includes a rich set of tools and technologies to manage models and integrate them in the platform. DSL models use EMF and are used to capture the domain semantics in a well-formed model complete with OCL constraints. This model-driven approach ensures the quality of the model data and creates a more productive workflow.

DSLs involve the following two primarily roles:

- The Toolsmith *creates* the DSL.
- The Practitioner *uses* the DSL.

Elements of a DSL

Domain Model

The domain model provides the basis for the DSL. This model defines the syntax and semantics as well as captures the domain taxonomy. The domain model is the metamodel. Instance models are derived from the domain model and contain data.

Diagram

The DSL can provide an optional diagram that creates and edits instance models. This familiar diagramming interface enables practitioners to easily and quickly create well-formed models.

Model Transformation

Model transformations can be used for a number of functions, including generating output models, modifying existing models, and transforming model elements into another model's elements. Model transformations are typically created using the Operational QVT transformation language.

Code Generation Template

A template language is often used to produce textual output. These templates are typically used to generate source code or auxiliary deployment files.

Report

Reporting is an important part of any DSL. It is often required to visualize model data as text. The DSL Toolkit contains features to help with generation of these reports.

UI contributions

A unique aspect of the DSL Toolkit is its deep integration into the Eclipse run time. The ability to extend the workbench's menus demonstrates this integration and lets practitioners easily execute model transformations and apply templates.

Conclusion

The DSL Toolkit provides the Toolsmith with a variety of tools to make Model-Driven Development (MDD) and generative approaches easier. The Practitioner benefits from a native Eclipse run-time environment that is immediately familiar and integrated into the platform.

Although the creation of a DSL requires time and effort, the return on this investment is a higher quality product with fewer bugs because the DSL provides a constrained input model and generated artifacts. The DSL eliminates the need to constantly debug and test the generated artifacts and can result in a time savings when reused across multiple projects.

A DSL can include several components, including domain models that describe abstract and concrete syntaxes), diagrams, transformations, reports, and templates. Several of these components are optional, depending on the usage of the DSL.

DSL Capabilities

The DSL Toolkit within Together enables the development of a domain-specific language (DSL). The following table identifies the capabilities that a Toolsmith uses to construct DSLs.

Domain modeling	The DSL Toolkit supports the creation of new domain models as well as the importing of existing ones. The Toolsmith uses the domain model editing features to capture domain concepts and vocabulary within this model. Every Domain model uses the Eclipse Modeling Framework (EMF), which enables compatibility across the Eclipse ecosystem.
Custom diagramming	Diagrams can be easily created for the domain model. As a result, diagrams provide a standard, easy-to-use graphical interface for editing domain instance models.

	Common diagram features include the drawing surface, property inspectors, diagram palette, and layout, which are generated automatically. More advanced features that can be added programmatically include custom graphic elements and custom layout algorithms.
Model transformations	The DSL Toolkit uses Operational QVT (Query/View/Transformation), which provides a standard means to transform, modify and generate models. In addition to the QVT features provided by the open source M2M.QVT Eclipse Project (such as the runtime engine and the editor with codesense and syntax highlight functionality), the DSL Toolkit provides the Operational QVT Debugger, the transformation creation wizard, and QVT refactoring and enhancements to the editor.
Templates	The Xpand template language primarily handles code generation, but JET is also supported. The Xpand template language provides an easy and flexible means of generating textual content, such as source code or auxiliary files. Templates are the preferred approach to generating code. They provide flexibility and portability far beyond handwritten Java. However, it might be prudent in some cases to use Java for generation of certain artifacts because of existing code or possible integration with a system requiring Java implementation.
Transformation sequences	Some simple DSLs accomplish their tasks with a single transformation or template. More complex DSLs often require multiple steps to produce the desired outputs. For this reason, the DSL Toolkit provides the ability to apply multiple transformations and templates in a sequence.
Figure Galleries	Figure galleries provide a simplified way to add custom figures or graphical elements to diagrams. Reuse figure galleries across DSL projects.
User interface contributions	DSLs can extend the Eclipse user interface. Common extensions, such as adding items to a diagram's popup menu, are supported. The DSL Toolkit generates some DSL actions to launch transformations or templates. Other user interface contributions get added programmatically.
DSL deployment	DSLs are deployed as Eclipse plug-ins, allowing users to install and manage DSLs just like any other Eclipse plug-in.

DSL Toolkit Workflow

This section describes the workflow of an end-to-end DSL development scenario. Some of the steps are optional. At the core of a DSL is the domain model. It typically also provides custom diagrams, editors, model transformations, diagram reports, code generation templates, and packaging for deployment.

- *Domain-Specific Languages* on page 9
- *DSL Toolkit Usage Scenarios* on page 18

Related Topics

Create DSL Project

The first step is to create a new DSL project. The **New DSL Project** wizard guides you through the basic project settings and displays the DSL Editor. A DSL project contains the following artifacts that separately implement the DSL:

- Domain models
- Diagrams
- Transformations
- Templates
- Reports

A DSL project also operates as the central hub for related, generated implementation projects. Alternatively, you can import existing DSL projects into the workspace using **File ➤ Import... ➤ Plug-in Development ➤ Import DSL Project from Platform** from the main menu. The DSL Editor manages all the content and related projects.

Domain Modeling

In the context of the DSL Toolkit, Toolsmiths use domain modeling to capture fundamental domain entities and relationships. In this case, the metamodel is created and can be augmented with constraints to help ensure model integrity. In fact, the output of domain modeling produces the most important part of the DSL. The domain model provides the basis for all generated artifacts and is used to generate the interface for all model transformations. The DSL Toolkit provides excellent tools for domain modeling. The Toolsmith can create new domain models or import existing ones.

Object Constraint Language (OCL) constraints can optionally be added directly to the domain model's diagram. Toolsmiths can then define rules for domain models to ensure data integrity. Additionally, OCL provides the implementation for derived features and operation bodies.

Diagram Development

A DSL can use a diagram as the primary model editor. The DSL Toolkit visual diagram editor provides a unified tool to create custom diagrams and work with diagram definitions. The diagram definition provides a standard means to visualize a model and to control user interface (UI) controls on the diagram. The underlying technology uses a collection of models to define the diagram.

The graphical definition model defines the nodes, links, and their corresponding visual attributes for a diagram. The tooling definition model specifies the elements that are displayed on the diagram palette, creation tools, and any contributed UI actions. The mapping definition model binds together the domain, graphical definition, and tooling definition models. As a by-product of the these models, the generator model generates the code to instantiate the DSL.

Note: While most DSLs can be depicted graphically, it is entirely possible to produce a DSL without a diagram.

This step is optional.

Author Model Transformations

The DSL Toolkit can use Operational QVT transformations to generate, modify and transform models. Use the provided QVT development environment (that includes QVT Debugger, refactoring and advanced editor) to create and debug QVT transformations and libraries.

Create Code Generation Templates

DSLs typically produce textual content of some sort. Sometimes this content is source code, but it can also be produced as text files, such as XML. The DSL Toolkit supports Xpand and JET templates for use in code generation.

This step is optional.

Construct Transformation Sequence

Practitioners find transformation workflows useful in orchestrating the execution of several transformations, including both model-to-model and model-to-text. Practitioners can save this set as an Ant script and even attach it to a user interface action.

This step is optional.

Design DSL Report

The DSL Toolkit includes a report designer for domain models and diagrams that lets the Practitioner generate reports of model content. These reports leverage the Eclipse BIRT reporting engine.

This step is optional.

Test DSL

While most testing can be done in the development workspace using dynamic instance models, the DSL Toolkit provides a run-time environment in which you can test your DSL. The Eclipse runtime configuration replicates the DSL run time for developer testing. More robust quality assurance uses replicated install images of the end-user Eclipse environment. Familiar unit tests can be developed to provide automated testing of DSL generated artifacts.

Deploy DSL

Because DSLs created by the DSL Toolkit are deployed as Eclipse plug-ins and wrapped by an optionally generated feature, packaging and installation for DSLs is simplified.

DSL Toolkit Best Practices

DSL development crosscuts several software-development disciplines. This collection of best practices concentrates on usage of the DSL Toolkit within Together. Best practices pertaining to metamodeling, modeling, and DSL in general are available elsewhere.

Success with the DSL Toolkit typically depends on expertise with the following platform underlying technologies:

- Eclipse
- plug-in development
- Eclipse Modeling Framework (EMF)
- Graphical Modeling Framework (GMF)
- Eclipse Graphical Editing Framework (GEF)
- *Creating a Dynamic Instance Model* on page 29

General

Follow the best practices for base technologies such as EMF, GEF and GMF. Mastery of these technologies is essential.

Naming Conventions

When applying naming conventions to your DSL projects, keep in mind the following guidelines:

- Prevent clashes in project names by inserting `.dsl.` between the organization segment and the domain name. For example, when creating a DSL for a Mindmap application, the recommended DSL Project name is `com.borland.dsl.mindmap`. This convention allows for the generation of a `com.borland.mindmap` branding plug-in to correspond with a `com.borland.mindmap-feature` project.
- Use clear and consistent conventions for naming DSLs and their artifacts.
- The domain model name must be a noun that clearly describes the domain.
- Avoid changing the default file name extensions for DSL artifacts.
- Namespace URI (NS URI) must include the base package, year, and model name, such as `http://www.borland.com/2007/mindmap`. Alternatively, you can use the version number instead of the year value.
- Use unique, intelligible file extensions for DSL domain files. A good approach is to use the name of the DSL as the extension, such as `mindmap`.
- Name model-to-model transformations by using <inputModel>2<outputModel>.qvto as a template, such as `Mindmap2XHTML.qvto`.
- Append `_diagram` to the DSL name as the diagram file extension, such as `mindmap_diagram`.
- Regardless of the naming approach you choose, consistent naming practices make development easier. Consistency makes it simpler to locate files, understand a particular file's usage, and help others use your work.

You should use Java naming conventions for elements and attributes in domain models. For information on Java naming conventions, see the information found at *http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html#367*

For more information, visit the naming conventions found at *http://wiki.eclipse.org/Naming_Conventions*.

Project Artifacts

The DSL Toolkit leverages EMF but changes some of the default namings. For example, EMF by default generates plug-in code to the same project where the model—that is, the `*.ecore` file—is found. The DSL Project is intended to contain all artifacts and not be the generation target. Therefore, when you use domain models in the DSL Toolkit, the default namespace for model code generation is the name of the DSL Project with a `.model` suffix but without a `.dsl.` segment. For example, the DSL Project described previously for a

Mindmap application (`com.borland.dsl.mindmap`) has model code generated to `com.borland.mindmap.model`. This guideline applies to generated diagram projects as well.

Use the **DSL Editor Contents** page to manage DSL artifacts, such as domain models, diagram definitions, and report designs.

Domain Modeling

When working with domain models, keep in mind the following guidelines:

- Leverage existing models, when appropriate. Because XSD and EMF are popular technologies, and EMF can import most XSDs, search for domain models that already exist before attempting to create a new one. Also, consider publishing your domain model if you feel others might find it useful.
- Test and refine the domain model as you develop it. Create a dynamic instance, and edit the XMI file by using the generated editor to avoid errors in the model syntax. Xpand templates, QVT transformations, and Report templates can utilize dynamic instance models for design-time testing.
- Consider the structure of the domain model and how it maps to the wizard-automation features in the DSL Toolkit. You can save yourself time and effort by using a structure similar to that provided in the mindmap sample. For example, use a top-level node with `1..*` containment links to diagram elements. You can use other model structures as well.
- Modify your domain model to simplify working with templates, transformations, and diagram definitions. Except when they use a model that cannot be altered, the Toolsmiths find it advantageous to make certain design decisions in the domain model to suit the tooling rather than to create workarounds or write custom code to use the tooling with a domain model.

Diagrams

When working with diagrams, keep in mind the following guidelines:

- In the mapping model, order matters, which is obvious in feature initializers but not so obvious in the case of node labels. For example, if a node has two labels and the top one is a read-only stereotype label, the default generated code does not enable the second node name label to be activated with the in-place editor when the node is created on the diagram. Reverse the order of the label mappings to achieve the effect you want.
- When working with complex figures, add a line border to see how layout and placement work.
- Compartments require the proper layout in the parent figure for proper visual appearance. The parent figure must use a Flow Layout with vertical orientation and the force single-line option for compartments to be displayed consistently with other UML Class nodes.
- Use the default size facet of a node element to give it the appropriate size upon creation. In the figure definition, you can also set the maximum, minimum, and preferred sizes.
- Leverage known notations where possible. With the popularity of certain modeling notations, certain shapes and figures already have meaning to a number of people. Also, try not to provide a diagram element when it serves no purpose in recognition or semantic meaning for the model. Textual elements like external labels might be the best means to provide the required information, and they do not always require a border.
- A *phantom node* leaves the Containment Feature blank for the top-level node and sets the Target Feature of a Link Mapping to the containment reference for the node. In this manner, the node is stored properly when linked but is displayed temporarily as a phantom node on the diagram until the link is made.
- To use a custom Scalable Vector Graphic (SVG) file as the graphic for a figure, set its type to `CustomFigure` in gmfgraph and then add the code to the appropriate `[figureName]EditPart.java` file. Locate the **createNodeShape** method and modify it to resemble the following code.

```
/**
```

```
 * @generated NOT
 */
protected IFigure createNodeShape() {
  URL url = FileLocator.find(BpmnDiagramEditorPlugin.getInstance()
  .getBundle(), new Path("images/circle.svg"), null); //$NON-NLS-1$
  return new ScalableImageFigure(RenderedImageFactory.getInstance(url),
    true, true, true);
}
```

Model Transformations

When working with model transformations, keep in mind the following guidelines:

- You can facilitate certain mappings in QVT and template expressions by adding derived features or methods to the domain model. For example, you can add some complex queries using OCL to the domain model with code generated for their implementation at run time. Making a feature available in a model simplifies transformations and the templates that access them.
- Use enumerations instead of improvised keywords. To reduce the chance of user input errors, base transformation or generation logic on constrained input values.
- Consider using incremental model-to-model transformations instead of complex, one-step code generation. Transforming input models to intermediate models can reduce the overall complexity of DSL transformation sets.
- Use QVT libraries for common queries and simple mappings.
- When a model's URI is passed as an argument to the transformation, the QVT engine takes into account any URI mappings. Define these mappings by choosing either **Project ➤ Properties ➤ DSL Toolkit ➤ URI Mappings** or **Window ➤ Preferences ➤ DSL Toolkit ➤ URI Mappings**. Evaluate the QVT script by using the **Evaluate** context menu option in the **Transformation Sequence** section of the DSL Editor's **Transformations** tab. With a successful evaluation, the QVT script can operate with metamodels that use logical URIs defined in the project's URI mappings.
- Leverage the extensibility options of Xpand to allow for the easy modification or enhancement of your templates. By adding placeholder **DEFINE** elements for select extensibility points, a Toolsmith later can use the **AROUND** element to leverage Xpand's aspect-oriented capabilities.

Figure Galleries

Use figure galleries for common, reusable diagram-element-notation definitions.

Consider using the gmfgraph definitions included with the Eclipse UML2 diagrams.

URI Mappings

If your domain model uses or extends elements from another model, referencing that model in the **URI Mappings** field of the Domain Model Editor is essential to managing workspace configuration issues and discrepancies that could otherwise arise. This field maps the URIs of resources rather than the URIs of model elements.

Related Topics

Common URIs on page 78

DSL Toolkit Usage Scenarios

The DSL Toolkit within Together enables the development of a domain-specific language (DSL). This section describes common usage scenarios for a Toolsmith who uses the DSL Toolkit features.

Given the broad definitions for DSL, these scenarios omit several legitimate usage models that the Toolsmith often uses. This section does not include scenarios that involve creating profiles for the UML, developing internal DSLs, or extending development tools.

Related Topics

Complete DSL Development

In some domains, DSLs provide an entire solution and operate standalone. As a result, the DSL provides all the following components of a DSL:

- Domain model
- Model transformations
- Reports
- Code generation templates

An example of DSL development is a process definition DSL that includes a process model, process diagram, and code generation.

You could deploy this type of DSL as an Eclipse Rich Client Platform (RCP) application.

DSLs can be deployed as Eclipse plug-ins to extend the standard workbench. However, it might be advantageous to deploy a smaller application with fewer features than the standard Eclipse workbench installation. In these cases, the DSL may be deployed as an Eclipse Rich Client Platform (RCP) application. This approach deploys only the essential features and functionality.

DSL as Part of a Larger Application Framework

In many solutions a DSL is combined with more traditional approaches. Many examples of small DSLs exist within larger frameworks, such as Enterprise Java Beans (EJBs) deployment descriptors. In this example, the EJB deployment descriptor provides a simple means to describe how to deploy the EJB. The development of application features requires more traditional software code.

This scenario often leverages build automation technologies, such as Apache Ant, but can also be a manual workflow.

A simplistic version of this scenario is the addition of a custom diagram to an existing domain model.

You can use a DSL in the context of a workflow (for example, a model-to-model transformation can be followed by a model-to-text transformation). In addition, you can use the DSL Toolkit in a manner in which both types of transformations are used for activities like integration, export and import, and codegeneration. Use workflow scripts to orchestrate everything.

Create Model Transformations for Existing Domain Models

Using the import features of the DSL Toolkit, you can import existing models and author model-to-model transformations. This scenario uses only a subset of features but leverages the authoring, debugging, packaging and deployment capabilities.

Note: This approach can also use the transformation projects available in Together.

Diagram definition

If a domain model already exists, you can define a diagram for the domain model and deploy it for the practitioner.

Text generation

Create code generation templates

This scenario is similar to the previous one. Code generation templates are developed and applied to the input model.

Note: This approach can leverage the model-to-text transformations in Together.

Provide reporting capabilities

You can augment a domain model with reporting capabilities. This is actually a model-to-text transformation.

Special Considerations for C# Projects

The Together C# code-generation feature is a first-of-its-kind component of the DSL Blueprint that lets customers experience out-of-the-box functionality without having to rely on a Toolsmith. C# code generation is a basic model-to-text (M2T) transformation that uses a QVT script and Xpand engine functionality, including customizeable templates.

- *Generating C# Code* on page 31

Related Topics

Project configuration issues

The Xpand template that is used for C# code generation is deployed as part of the `com.borland.dsl.csharp.codegen` plug-in. Do not make any changes to it.

The following two options of C# code generation are configurable:

- File placement
- Code formatting

Use the lesser general public license (LGPL) C# parser provided by the LLK open source parser generator to format code.

Note: If the code returns errors from a file that has been modified, the parser does not produce an abstract syntax tree (AST) and merge. Problems with error recovery are also possible.

This parser provides a merging capability. If an AST is successfully produced for both old and new code, merging starts. Merging replaces all nodes—including namespaces, classifiers, members, and accessors—from the previous version of the file that have the `<generated/>` tag with generated nodes that have the same signature. The parser analyzes all nodes with the `<generated/>` signature for merging. If no `<generated/>` tag in the node's documentation comment exists, the node and its children remain unchanged even if the generator can create a replacement for it with the same ID.

Common Workarounds

Users who generate C# code might encounter the following problems:

- When C# code is generated into workspace resources but the generated files are edited with an external tool, the Eclipse internal file system can potentially fail to register those changes, and the merge completes with inappropriate data. **Workaround:** Press the refresh action **F5** key to explicitly refresh your workspace resources.
- The parser registers user changes as inappropriate and fails to parse the previously generated file. **Workaround:** Manually replace the file's content with the newly generated content.

Procedures

Creating a DSL Toolkit Project

To create a new DSL Toolkit project

1. Choose **File ➤ New ➤ Project...** from the main menu. After the New Project window opens, expand the **DSL Toolkit** node and select **DSL Project**. Click **Next**.

   The **New DSL Project** wizard opens.

2. Specify the project name and click **Next**.

   Note: To adjust the default project location settings, uncheck **Use default location** and select the appropriate project location. To change the default project file name, uncheck **Use default DSL Project file name** and type the appropriate project file name. This file name should end with a `.dsl` extension.

3. Review the **Project Settings**. Typically, you specify the **Branding provider** with your company URL in reverse, such as `com.borland`. You can modify the project setting defaults in the DSL Toolkit preferences by choosing **Window ➤ Preferences ➤ DSL Toolkit ➤ Project Editor**. If no default package prefix is specified in the DSL Toolkit preferences and the project name does not contain any periods (**.**), **org.eclipse** is used as the default base package prefix. A **Create Plug-in project** option is available and checked by default. You can even convert nonplug-in projects by using a **Convert to Plug-in project** context menu option.
4. To create a new domain model or import an existing one, click **Next**. Select **Domain Model** to create a new model. Select **Import Domain Model** to import an existing ecore or genmodel file. Review the subtasks later for documentation about creating or importing domain models.
5. Click **Finish** to create the project.

The **DSL Editor** opens.

Related Topics

DSL Editor on page 44

DSL Explorer View on page 66

Domain-Specific Language Preferences on page 81

Creating a Domain Model on page 21

DSL Toolkit Best Practices on page 14

Creating a Domain Model

To create a Domain Model

1. Create a DSL project.

2. Open the **DSL Editor**.

   Within the **Getting Started** group, click the **Domain Model** link.

   Note: Alternatively, you can open the **DSL Explorer**, right-click the project root, and choose **New ➤ Domain Model** from the context menu.

   The **Domain Model Creation** wizard opens.

3. Choose **Create new Domain Model** and click **Next**.

   The **Domain Model Properties** screen opens.

4. Enter properties for the Domain Model.

   Note: Specify the domain name in lowercase letters, such as `mindmap`, for the **Model name** field. Specify the **Base package** by commonly reversing the URL of your organization, such as `com.borland`. Using a format such as `http://www.borland.com/2007/mindmap`, indicate the Uniform Resource Identifier (URI) in the **Namespace URI** field. By appending the domain model name to the base package, such as `com.borland.mindmap.model`, supply a **Plug-in ID**.

5. Click **Finish** to create the Domain Model.

   The **Domain Editor** opens.

Tip: If you decide to use artifacts from another project, import the appropriate model from the project development environment (PDE) platform.

Related Topics

Importing Models from the PDE Platform on page 38

DSL Explorer View on page 66

DSL Toolkit Best Practices on page 14

Importing an Existing Domain Model

1. On the Artifacts screen of the **New DSL Project** wizard, check the **Create artifact in new DSL Project** check box and select **Import Domain Model** from the **Available artifacts** field. Click **Next**.
2. On the Domain Model screen, specify the domain name in lowercase letters, such as `mindmap`, for the **Model name** field. The location of the model is listed for your reference but is read-only. Uncheck **Use default file name** to make changes to the default model properties. You can modify the file name but leave its extension as `.domain`.

   Click **Next**.

3. Select the model to import.

   Note: The **Errors** button becomes enabled if the selected model causes any validation errors. These errors might indicate missing model references, among other things. It is possible to import the model anyway by checking **Ignore errors**. However, you might need to add the missing models later in the DSL Editor.

4. Click **Next**.

5. Select the model element that will function as the top-level node in the domain model.
6. Click **Finish**.

   The **Domain Editor** opens.

   Note: To open **DSL Editor**, locate the DSL file in the **DSL Explorer** view. Double-click to open the editor.

Loading a Model from a PDE Platform

1. After you create a model like an Ecore model, deploy the model as a plug-in and add the plug-in to the target platform.
2. Configure the PDE platform with any additional plug-ins or other dependencies to match the target platform. Plug-ins can be grouped by location or name.
3. Before importing the plug-in, uncheck the **Use generation model** checkbox.

   In doing so, you avoid warnings about missing genmodels from the Domain Editor if you did not use a genmodel when the plug-in was created.
4. To address plug-ins from the target platform when loading the original model, specify a `target:/plugin/` URI scheme in the **Model URI:** field.

   This scheme is similar to the `platform:/plugin` scheme except that an instance set up with the target platform gets looked up for plug-ins rather than a running Eclipse instance looking up plug-ins.
5. Click **OK**.

   After you load the model by choosing **Load Resource...** from the context menu, the model is available for tasks, such as creating shortcuts. After a dependency shortcut has been created, the model for the second subsystem can use concepts and elements from the original model.
6. Replace all configuration-specific URIs with unique URIs to avoid referencing problems caused by URIs that reflect different environments.

   For example, `NsURI` is a good choice for Ecore models. You might need to establish other URI conventions for other types of models, such as `platform:/plugin/`, which works well for the deployed model. Any unique string that lets a client map to the correct URI is appropriate.
7. Configure **URI Mappings** by substituting the **Resource** mapping with the **Actual URI**.

   The Actual URI points to the model from the target platform configuration.
8. Click **Save**.

Related Topics

Creating a DSL Toolkit Project

To create a new DSL Toolkit project

1. Choose **File ➤ New ➤ Project...** from the main menu. After the New Project window opens, expand the **DSL Toolkit** node and select **DSL Project**. Click **Next**.

   The **New DSL Project** wizard opens.

2. Specify the project name and click **Next**.

   Note: To adjust the default project location settings, uncheck **Use default location** and select the appropriate project location. To change the default project file name, uncheck **Use default DSL Project file name** and type the appropriate project file name. This file name should end with a `.dsl` extension.

3. Review the **Project Settings**. Typically, you specify the **Branding provider** with your company URL in reverse, such as `com.borland`. You can modify the project setting defaults in the DSL Toolkit preferences by choosing **Window ➤ Preferences ➤ DSL Toolkit ➤ Project Editor**. If no default package prefix is specified in the DSL Toolkit preferences and the project name does not contain any periods (**.**), **org.eclipse** is used as the default base package prefix. A **Create Plug-in project** option is available and checked by default. You can even convert nonplug-in projects by using a **Convert to Plug-in project** context menu option.

4. To create a new domain model or import an existing one, click **Next**. Select **Domain Model** to create a new model. Select **Import Domain Model** to import an existing ecore or genmodel file. Review the subtasks later for documentation about creating or importing domain models.

5. Click **Finish** to create the project.

The **DSL Editor** opens.

Related Topics

DSL Editor on page 44

DSL Explorer View on page 66

Domain-Specific Language Preferences on page 81

Creating a Domain Model on page 21

DSL Toolkit Best Practices on page 14

Adding Database Persistence Support

The following procedure is one method of adding database persistence support for your Domain Model project.

Note: For more information on using Hibernate facilities, see *http://www.hibernate.org/5.html#A7*

To download Teneo and Hibernate

1. Download Teneo from the download page at the *http://www.eclipse.org/modeling/emf/downloads/?project=teneo* Web site.

2. Download the following Hibernate libraries:

   - `hibernate-annotations-..*.GA.tar.gz`
   - `hibernate-..*.ga.zip`
   - `hibernate-entitymanager-..*.GA.tar.gz`

3. Install Teneo from the location that you downloaded it.

4. In Together, create a new plug-in project by choosing **New ➤ Project... ➤ Plug-in Development ➤ Plug-in from existing JAR archives**.

   For example, the project can be named `HBs` and contain the following files:

   - `hibernate3.jar`
   - `hibernate-annotations.jar`
   - `hibernate-entitymanager.jar`
   - any required JAR files from the `hibernate*/lib` directory
   - JDBC driver

5. In the project folder, add the following text to the `manifest.mf` file:

```
Eclipse-BuddyPolicy: dependent
```

Related Topics

To add database persistence support manually

1. On the **Overview** page of the Domain Editor, check the **Edit** and **Editor** check boxes and click **Generate** to generate your model.
2. Add the following text to your project's `plugin.xml` file:

```
<extension
        point="org.eclipse.emf.ecore.extension_parser">
 <parser

class="org.eclipse.emf.teneo.hibernate.resource.HibernateResourceFactory"
          type="file_extension">
 </parser>
</extension>
```

3. Add the following text to the `manifest.mf` file:

```
 org.eclipse.emf.teneo.hibernate;bundle-version="0.8.0",
 org.eclipse.emf.ecore.xmi;visibility:=reexport,
 HBs;bundle-version="1.0.0";visibility:=reexport
```

To add database persistence support using the Domain Editor

1. On the **Advanced** page of the Domain Editor, check the **Use dynamic templates** check box.
2. Check the **Use advanced templates** check box.
3. In the **Project with Hibernate libraries:** field, specify your Hibernate project name.
4. Click **Populate templates...**.
5. On the **Overview** page of the Domain Editor, check the **Editor** check box and click **Generate**.

To connect to the database

1. The practitioner must start a new instance and create an EHB file with the following content:

```
#Example (for MySQL):
name=test4teneo
nsuri=http://www.example.org/2008/test4teneo
editorextension=test4teneo
dbname=dsldomain
# database should exist
dburl=jdbc:mysql://127.0.0.1:3306/dsldomain
dbdialect=org.hibernate.dialect.MySQLInnoDBDialect
dbuser=user
dbpassword=pass
dbdriver=com.mysql.jdbc.Driver
```

2. Right-click the EHB file you created and choose **Teneo ➤ Open resource**.

   The opened editor will be empty because the database does not contain elements.

3. Using the New Model wizard, select the root metaclass.

Related Topics

Domain Model Editor on page 49

Creating a DSL Diagram Definition

To create a DSL Diagram Definition

1. Open an existing DSL project.
2. Open the **DSL Editor**.

   Within the **Getting Started** group, click the **Diagram Definition** link.

   Note: Alternatively, you can open the **DSL Explorer**, right-click the project root, and choose **New ➤ Other...**. Expand the **DSL Toolkit** node in the tree view list, select **Diagram Definition**, and then click **Next**.

   The **Diagram Definition** page opens.

3. Edit the model name and location as necessary. Uncheck the **Use default location** check box to enable editing of the location. Similarly, uncheck the **Use default file name** check box to edit the file name. In most cases, use the default values. Click **Next**.

   The **Source Model** screen opens.

   Note: Because the list of models is loaded asynchronously to increase wizard responsiveness, all models might not be readily viewable.

4. You can choose from ECORE, DOMAIN, and GMFMAP models. Of these, only DOMAIN models inherently reference GenModel files, in which case you can continue with the next step. For ECORE and GMFMAP models, you can specify the GenModel file by selecting the model and clicking **Next** to access the **Domain Genmodel** page.

   Tip: In cases when the GenModel file is out of sync with the model, the wizard can fail to produce a GMFGEN model. Check the **Reload** check box on the **Domain Genmodel** page to ensure that the GenModel file is not stale.

5. Select the source model and click **Next**.

   Note: In most cases, select the main domain model associated with the DSL. However, it is possible to create the diagram definition for any model in the workspace and those models that are deployed to the Eclipse installation.

   Note: The **Errors** button becomes enabled if the selected model causes any validation errors. These errors might indicate missing model references, among other things. It is possible to import the model anyway by checking **Ignore errors**. However, you might need to add the missing models later in the DSL Editor.

   The **Graphical and Tooling Definition Models** screen opens.

6. Use the **Add** and **Remove** buttons to manage available figure galleries. Adjust the **Canvas** and **Palette** selections as necessary.

   Click **Next**

   The **Diagram element** screen opens.

7. Select the top-level element for the diagram canvas. The canvas typically maps to the domain model's **EClass** root element, which should contain all instances of EClasses present on the diagram, directly or indirectly.

   Click **Next**.

   The **Diagram structure** screen opens.

8. Configure the diagram structural elements by declaring that an element is either a Node, Link, or Label.

   Although the wizard makes these initial selections, review them and change them as necessary. You can restore the wizard's initial selections by clicking the **Defaults** button.

   Typically, leave the **Diagram Element** as `Create...` unless you need to assign a previously added figure gallery to the figure. The **Creation Tool** specifies whether the element will be displayed on the diagram's palette.

   Note: In the list box, enable only those model elements that you want visually displayed on the diagram. Even if they are not enabled here, all model attributes are available in the **Properties** view. These check boxes enable *visualization* on the diagram.

   Note: If an existing diagram canvas was chosen, all model elements and the palette must map to the existing diagram elements and palette. A GMFGRAPH model is not created in this case.

9. Click **Finish** to create the diagram definition.

   The **Diagram Editor** opens.

Related Topics

Creating a Figure Gallery on page 28

Printing DSL Toolkit Diagrams on page 28

Diagram Definition Editor on page 56

DSL Explorer View on page 66

Generating the Composite Editor

To generate the Composite Editor

1. In the DSL Editor, open up the **Contents** page of your `.dsl` project.
2. Select your diagram listed under the **Diagrams** tab. The **Generate Composite Editor** check box is displayed to the right.
3. Click the **Generate Composite Editor** check box.

   A combined editor is generated into your DSL project, which is defined in the Project Editor. The Composite Editor has two pages. The first page graphically displays a synchronized diagram of the root element. The second page displays a tree-view EMF editor. You can use both to make simultaneous modifications to your topic map.

Note: The Composite Editor is registered on all files that have your model's extension. The combined editor expects the diagram file to have the same name and diagram extension as the model file. If the diagram file is not found near the model file, the Composite Editor creates the diagram file.

Related Topics

DSL Editor on page 44

Printing DSL Toolkit Diagrams

- To print from Windows, make sure that the **org.eclipse.gmf.runtime.common.ui.printing.win32** plug-in is installed.
- To print from Together, make sure that the **Printing Enabled** option on the **Advanced** page of the diagram editor is selected.
- To print from GMF, make sure that the **printingEnabled property** of **GenPlugin** is set to true and that diagram code is regenerated.

Note: The Java print library cannot update the standard printer settings. If necessary, set the paper and orientation settings to match the Together Print Preferences.

Related Topics

Creating a DSL Diagram Definition on page 26

Creating a Figure Gallery on page 28

DSL Explorer View on page 66

Creating a Figure Gallery

To create a figure gallery

1. Open an existing DSL project.

2. Open the **DSL Explorer**, right-click the project root, and choose **New ➤ Other...**.

   Expand the **DSL Toolkit** node in the tree view list, select **Figure Gallery**, and then click **Next**.

   The **Figure Gallery** wizard opens.

3. Type a name for the figure gallery in **Model name**.

   Verify the path in **Location**.

   Uncheck **Use default location** to edit **Location**.

   Uncheck **Use default file name** if you want to use alternative names.

4. Click **Finish**.

   The Figure Gallery editor opens.

Related Topics

Diagram Definition Editor on page 56

DSL Explorer View on page 66

Figure Gallery Editor on page 71

Creating a Dynamic Instance Model

To create a dynamic instance model

1. Open an existing domain model.
2. Open the **Domain model editor**.

   Within the **Diagram** page, locate the top-level class in the domain model. Right-click that class and choose **Create Dynamic Instance**.

   The **Dynamic Model** wizard opens. Enter an appropriate name and choose the location for the XMI file.

   The **XMI editor** opens.

   Tip: Within the **XMI editor**, you can create instance models with sample data. Instance models can be very useful in developing DSLs.

   Note: Creating a dynamic instance is useful when developing a DSL because it contains sample model data for use when developing reports, templates and transformations. It is an XMI file derived from the domain model.

Related Topics

Domain Model Editor on page 49

Domain-Specific Language Glossary on page 89

Creating a DSL Transformation

To create a DSL Transformation

1. Open an existing DSL project.
2. Open the **DSL Editor**.

   Within the **Getting Started** group, click the **Model transformation** link.

   Note: Alternatively, you can open the **DSL Explorer**, right-click the project root and choose **New ➤ Transformation**.

   The **New Operational QVT Transformation** wizard opens.
3. Ensure **Source container** is set to the proper location.

   If you choose to, type a value for **Namespace**.

   Type a name for the transformation in **Module Name**.

   Typically, a transformation has one input model parameter and one output model parameter, although Operational QVT allows any number of input, output, and inout models for transformation.
4. Click the **Add** button. Set **Direction** as `inout` or `in`.

   Edit **Name**.

   Click the button in **Metamodel** and select the appropriate input models.

   Select the **Entry point type** to the desired type.

   Tip: You can type the first letters of the metamodel and the Entry point names and press `Ctrl-Space` to use the code completion feature.
5. Click the **Add** button.

   Leave **Direction** as `out`.

   Edit **Name**.

   Click the button in **Metamodel** and select the appropriate output models. Select the **Entry point type** to the desired type.

   Tip: Use **Move up** and **Move down** to add more models and adjust their order. Review the Transformation signature preview.
6. Click **Next**.

   The **Import metamodels** screen opens.
7. Select the additional metamodels to import and click **Next**.

   The **Import libraries** page opens.
8. Select the additional libraries or transformations to import and click **Next**.

   The **Initial mapping operations** screen opens.
9. Create initial mappings as desired and click **Finish**.

   The **QVT Editor** opens.

Note: Manually register shared metamodels when they are imported to avoid losing registration information.

Related Topics

Manually Registering a Metamodel for Use with QVT (Together Modeling Guide)
DSL Editor on page 44
DSL Explorer View on page 66
QVT Operational Developer Guide

Generating C# Code

You can transform a UML model into a C# model and generate C# code from the transformation. The cheat sheets provided with Together include an example of how to generate C# code from a model.

To access the C# cheat sheet

1. On the main menu, click **Help ➤ Cheat Sheets...**.
2. Expand the **Together C# modeling** node.
3. Select **Generate C# code from model** and click **OK**.

Related Topics

Special Considerations for C# Projects on page 19

Creating a DSL Transformation Library

To create a DSL Transformation Library

1. Choose **File ➤ New ➤ Other...** from the main menu.

   Expand the **DSL Toolkit** node in the tree view list, select **QVT Library**, and then click **Next**.

   The **New Operational QVT Library** wizard opens.
2. Ensure **Source container** is set to the appropriate location.

   If you choose to, type a value for **Namespace**. Type a name for the library in **Module Name**.

   Click the **Next** button.

   The **Import metamodels** screen displays.
3. Select the additional metamodels to import and click **Next**.

   The **Import libraries** screen opens.
4. Select the additional libraries or transformations to import and click **Finish**.

   The **QVT Editor** opens with the library declaration and imported members.

Related Topics

DSL Editor on page 44

QVT Operational Developer Guide

Creating a DSL Template

To create a DSL template (model-to-text transformation)

1. Open an existing DSL project.
2. Open the **DSL Editor**.

   Navigate to the **Transformations** page and expand **Template Collections**.

   Note: Alternatively, you can open the **DSL Explorer**, right-click the project root and choose **New ➤ Other...**. Expand the **DSL Toolkit** node in the tree view list, select **Xpand Template**, and then click **Next**. The **New Xpand Template** wizard opens.

3. Click **Template Collection**.

   Type a name and click **OK**.
4. Click **Xpand Template**.

   You might need to expand the new template collection node. Double-click the new template file to open its editor.

   Note: It is also possible to create a JET template.

Related Topics

JET Tutorial Part 1 (Introduction to JET)

DSL Editor on page 44

DSL Explorer View on page 66

Xpand Language Guide on page 79

Creating a DSL Transformation Workflow

To create a DSL Transformation Workflow

1. Open or create a DSL project.
2. In the **DSL Editor**, navigate to the **Transformations** page and expand the **Workflows** section.
3. Click **New...**. In the wizard dialog, accept the default **Script Model** name or provide an alternate. You can optionally change the location and file name.

   Note: The Workflow Editor has both a **Diagram** and a **Workflow** page for defining execution scripts. Both offer the same creating and editing capabilities, although the **Diagram** page is graphical and

might be easier to work with. The palette groups on the diagram correspond similarly to the items found within each combo box grouping on the form editor.

4. Set up shared elements—that is, template roots—in addition to inputs, outputs, and transformations to define a transformation sequence.

   Note: The text fields in the property pages of shared elements support clipboard actions, including **Ctrl-C** and **Ctrl-Insert** for copying, **Ctrl-X** and **Shift-Delete** for cutting, and **Ctrl-V** and **Shift-Insert** for pasting.

Related Topics

DSL Editor on page 44

Creating a DSL Transformation on page 30

Creating a DSL Template on page 32

Declaring Shared Elements for a DSL Transformation Workflow

1. Create a new **Template Root** by selecting the script root element on the **Workflow** page and then an element under the **Shared Elements** combo box on the form. Alternatively, select a tool in the **Shared Elements** palette group on the **Diagram** page.

   Note: For more advanced scenarios, use the query-based (QVTO or OCL) template roots and class-loader context options available in the shared elements category. A simple selection of a declared template root usually suffices for workflow definition.

2. Click **Pick...** to choose a template root that has been already declared in the project.

   Note: By specifying the name of the template root, you enable the translation of this location into the appropriate `platform:/` URI when the transformation script is executed. You can also specify a full `platform:/resource` or `platform:/plugin` URI.

Specifying an Input for a DSL Transformation Workflow

Specifying an input

1. If you are using the form editor, select the script root element.
2. If you are using a file that has a known location at execution time, create a **Resource** input or specify a full URI.

   Note: If you specify a selected file in the workspace as an input, create the **Context** element and accept the default **Selection** option.

Related Topics

Configuring an Xpand Invocation

1. Select the script root element using the form editor.
2. In the **Invocation** group, select the **Xpand** element. **Input** and **Output** elements are also created.
3. Select the template root shared element that you created previously.
4. Click **Pick...** to select the domain model, and select `DEFINE` from the **Choose template** field.
5. In the **Diagram** page, use the Input link tool to allow the **Xpand Template** element's input pin to be connected to its input. For deployments, you can use the **Selection Context** input that you created previously. A **Resource** input is useful in local testing.
6. Next to the input selection, check the **Chain** check box.
7. Type the correct Imperative OCL statement in **Expression**, such as `self->first()`, to select the root element of the input model.
8. Select the **Output** pin if you are using the diagram. Set the Regular file **Name** property to an appropriate file name and extension.

   Note: An OCL expression can be used to form complex output paths, which typically use the `platform:/` URI scheme. In the case of multiple input elements, as determined by the input query, multiple outputs can be generated, which likely means that you need to express a model property or counter in the output path.

Configuring a QVT Invocation

1. Select the script root element using the form editor.
2. In the **Invocation** group, select the **QVT** element. **Input** and **Output** elements are also created.
3. Click **Browse** to select the transformation QVTO file.

   Note: In the **Transformation** field, use a project-relative path so that this location can be translated into the appropriate `platform:/` URI scheme when the transformation script is executed. Alternatively, specify any valid URI. By default, the path takes the form `transformations/<myscript>.qvto`, where `<myscript>` is the name of the QVT script file.

4. In the **Diagram** page, use the Input link tool to allow the **QVT** element's input pin to be connected to its input. For deployments, you can use the **Selection Context** input that you created previously. A **Resource** input is useful in local testing.
5. Next to the input selection, check the **Chain** check box. When you use the **Diagram** view, select the input pin in order to view these properties.
6. Type the correct Imperative OCL statement in **Expression**, such as `self->first()`, to select the root element of the input model.
7. Select the **Output** pin, if you are using the diagram.

Set the Regular file **Name** property to an appropriate file name and extension.

Note: An OCL expression can be used to form complex output paths, which typically use the `platform:/` URI scheme.

Related Topics

Validating a Workflow

1. On the **Workflow** page, select the root of the script in the list box.
2. Right-click the root of the script in the list box and choose **Validate**.
3. Fix any errors so that you can achieve a clear validation before proceeding.

Evaluating a Transformation Sequence

1. On the **Workflow** page, select the root of the script in the list box.
2. Right-click the root of the script in the list box and choose **Evaluate**.

   Note: For convenience, use **Evaluate** with a local dynamic instance model as a **Resource** input when developing workflows. Replace the local resource with a **Selection Context** before generating and deploying the script.

Related Topics

Generating an Ant Script for a Transformation Sequence

1. On the **Workflow** page, select the root of the script in the list box.
2. Right-click the root of the script in the list box and choose **Generate Ant Script**. This creates an Ant `build.xml` file in the workspace. You can run the Ant script locally, or the Practitioner can deploy it for execution.

   Note: You can choose between Ant and Java to execute your workflow scripts. Both are available when defining script invocation UI contributions in the DSL Editor.

Creating a DSL Report

To create a report for your domain model, you need to create a plugin project using existing `DSL Report Example` template

1. Create a new plug-in project. (Click **File** > **New** > **Project**, and choose **Plug-in Project**,and then click **Next**.)
2. On the **Plug-in Project** page, specify the name for your project , and check the box for **Create a Java Project** (this should be the default). Leave the other settings on the page with their default settings, and then click **Next** to accept the default plug-in project structure.
3. On the **Plug-in Content** page, make sure that **Generate an activator...** and **This plug-in will make contributions to the UI** boxes are checked (this should be the default). Click **Next**.
4. On the **Templates** page, check the box for **Create a plug-in using one of the templates**. Then select the **DSL Report Example** template and then click **Next**.
5. On the **DSL Repoirt Example Generator** page, specify the required report properties:

• **Report Name**	is the name for the report.
Input Model	is the domain model for the report.
Model File Extension	is the instance model's file extension. Typically, this value remains as the default value.
Master Element	is the root element of the report.
Master Name	is chosen among EString properties of metaclass specified as master element.
Child Elements	are chosen from master element children.
Child Name	is chosen among EString properties of metaclass specified as child element.
Instance URI	is the URI of the input model instance.

Note that all fields are required except for **Instance URI**; it can be omitted, but specifying it will help to design the report because it includes sample model data.

The plugin generates a stub report (`model_report.rptdesign`) for the domain model and registers an action on files with extension specified as *`Model File extension`* that runs the report; generated stub can be opened in BIRT report editor and modified appropriately. See *BIRT Report Developer Guide* for more info. Note that DSL Toolkit adds EMF Datasource to the list of BIRT datasources, making report designing for emf models easier.

Note that the steps above can better be made a cheatsheet, in this case the procedure should just reference the corresponding cheatsheet; existing models like CSharp can be used as example.

Related Topics

DSL Explorer View on page 66

Creating a Textual Notation for Your Domain Model

1. Open your domain model.
2. From the main menu, choose **File ➤ New ➤ Other**.
   The **Select a wizard** page appears.
3. Under the **DSL Toolkit** node, select the **Textual notation** wizard and click **Next**.
   The **Textual Notation** page appears.
4. Type a location and name for your notation, or accept the default values, and click **Next**.

The **Source Model** page appears.

5. Select the existing Genmodel or Domain and click **Next**.
   The **GenPackage** page appears.
6. Select the main genpackage and click **Finish**.
   A `<modelname>.tnt file` opens in your workspace.
7. As necessary, adjust the settings on the Textual Notation editor's **Overview**, **Language**, **Advanced**, and **Text** pages.
8. On the **Overview** page, click **Generate**.
   A console appears displaying a log of the generation results, including all conflicts that need to be resolved. A generated plugin will also be listed in your **Package Explorer** window.
9. In the Console view, ensure that no conflicts occurred:

```
........
        [exec] Number of Shift-Reduce conflicts: 0
        [exec] Number of Reduce-Reduce conflicts: 0
        [exec] Number of Keyword/Identifier Shift conflicts: 0
        [exec] Number of Keyword/Identifier Shift-Reduce conflicts: 0
        [exec] Number of Keyword/Identifier Reduce-Reduce conflicts: 0
.......
```

Importing a Figure Gallery

To import a figure gallery

1. Open an existing DSL project.
2. Choose **File ➤ New ➤ Other...**.

   Expand the **DSL Toolkit** node in the tree view list, select **Import Figure Gallery**, and then click **Next**.

   The **Figure Gallery** wizard opens.
3. Type a name for the figure gallery in **Model name**.

   Verify the path in **Location**.

   Uncheck **Use default location** to edit **Location**.

   Uncheck **Use default file name** if you want to use alternative names.
4. Click **Next**.

   The **Canvas** page opens.
5. Select the figure gallery to import.

   Choose the diagram canvas, and click **Finish**.

The Figure Gallery editor opens. The linked gmfgraph models are included on the **Tree** page.

Related Topics

Figure Gallery Editor on page 71

DSL Artifacts on page 85

Importing DSL Projects from an Existing Platform

Together provides a wizard that allows you to import other DSL projects from an existing platform into your workspace.

To import a DSL project from an existing platform

1. From the Together main menu, select **File ➤ Import...**.
2. In the Import dialog, select the **Import DSL Project from platform** under the **Plug-in development** node.
3. The **Plug-ins and Fragments Found** field displays all available DSL projects in the platform.
   - Select the projects you want to import into your workspace and click **Add**.
   - To add all available projects, click **Add All**.
   - To delete projects you have already added, select those projects in the **Plug-ins and Fragments to Import** field and click **Remove** or **Remove All**.
   - Select a project in each field and click **Swap** to switch their places in one step.
   - Click **Required Plug-ins** to populate the **Plug-ins and Fragments to Import** field with generated projects associated with selected DSLs.
4. Filter the list in the **Plug-ins and Fragments Found** field further by optionally selecting the check box next to the following options, all of which are enabled by default:
   - Include fragments when computing required plug-ins
   - Show latest version of plug-ins only
   - Show DSL-related plug-ins only
5. Click **Finish**.

Related Topics

Importing Models from the PDE Platform on page 38

Importing Models from the PDE Platform

In some cases, users on a project might want to make use of artifacts from another project. To do so, those artifacts must be imported.

To import a model from the project development environment (PDE) platform

1. Select the model with the artifacts you want to import.
2. Making sure that all model files—including all the files in the `META-INF` and `model` folders—are selected to be included in the binary build, deploy the model as an Eclipse plug-in. For example, choose **Export ➤ Deployable plug-ins and fragments**. If another build infrastructure is in place, adjust these selections accordingly.
3. Specify a location on your local file system as the destination directory for your build artifacts. This action deploys the binary version of the plug-in as a regular JAR file.
4. Clear a new workspace for your target platform by choosing **File ➤ Switch workspace**).

5. Although the target platform contains its own subsystem, dependencies must be established to other parts of the common product. Unzip the binary build to a local file system and configure the PDE for additional plug-ins. You can optionally group the plug-ins by location; if this option is not selected, plug-ins are grouped by name.
6. Before importing the plug-in, disable the **Use generation model** option to avoid warnings about missing genmodels from the Domain Editor in case a generation model was not used when the plug-in was created.
7. Load the original plug-in to the target platform by choosing **Load Resource...** from the context menu.

   The **Model URI** field in the **Select Model** dialog indicates a `target:/plugin/` scheme.

   After it is loaded, the model is available for tasks such as creating shortcuts. You can link the models of the two subsystems so that concepts and elements can be shared.

The model file stores model references as full URIs, which can cause problems if the client environment differs from the local machine's environment. For example, another team might use both subsystems in its workspaces and address them differently. To avoid this issue, replace configuration-specific URIs with other unique URIs. For Ecore models, `nsURI` is the best choice. You might need to establish a URI convention for other types of models. Any unique string through which each client maps to the correct URI is appropriate. If possible, use a form that works without any mapping. For Ecore models, `nsURI` works; for other models, `platform:/plugin/` might be a good choice because it form fits the deployed product well.

Related Topics

Migrating from the Eclipse Modeling Project

Users who are familiar with the Eclipse Modeling Project (EMP) might decide to migrate to the DSL Toolkit in order to take advantage of its functionality. The DSL Toolkit comes with commercial features that are not available in such open source projects as Eclipse Modeling Framework (EMF), Graphical Modeling Framework (GMF), Generative Modeling Technologies (GMT), Model Development Tools (MDT), Model-to-Model Transformation (M2M), and Model To Text (M2T).

When migrating from the Eclipse Modeling Project open source project to the Together DSL Toolkit, users can access a number of import wizards that make the transition straightforward. The following scenarios contain best practice considerations when transitioning to the DSL Toolkit.

When migrating to a DSL Project

1. In the Together main menu, choose **File ➤ New ➤ DSL Project**.
2. Using the **New DSL Project** wizard, create a DSL project for each domain model and its corresponding artifacts. You can combine the artifacts from the following project types into a single DSL project:
   - **EMF**
   - **GMF**
   - **QVTO**
   - **oAW**

Related Topics

DSL Toolkit Best Practices on page 14

Importing a Figure Gallery on page 37

Importing Using Xpand

When importing using Xpand

Note: The underlying expression language used by Together's implementation of Xpand changed from the proprietary language that the Xtend language had built upon. Together's Xpand is based on the OCL and QVT Operational Mapping Language standards. Some noteworthy variations between the two implementations exist, including the following differences:

- Together Xpand accesses language features as properties instead of operations (for example, `isEmpty()` instead of `isEmpty`).
- There is no metaType property available in Together's Xpand (instead, use `eClass()` for elements that extend `EObject`).
- There are no `counter` or `counter0` properties on iterators in Together Xpand.

For more information on OCL and QVT Operational Mapping Language standards, refer to the *Object Constraint Language OMG Available Specification Version 2.0* and the *MOF QVT Final Adopted Specification*.

1. Replace the use of «`FILE`» with workflow-based persistence declaration of files.

   The Xpand version in Together views templates as string builders, and the persistence of the generated stream is handled external to the file. Typically, an OCL expression accomplishes this persistence in the template invocation, though the specification of a simple file name can also be used.
2. Migrate MWE files to Together workflow scripts (`*.exec` models).

Related Topics

Importing GMF Artifacts for the Diagram Editor

When importing GMF artifacts for the Diagram Editor

1. To import each graphical definition model to create a Figure Gallery, choose **File ➤ New ➤ Import Figure Gallery**.
2. Import each `*.gmfgraph` model to create a new Figure Gallery.
3. To import each GMF mapping model to create a Diagram Definition, choose **File ➤ New ➤ Diagram Definition**.
4. Import each `*.gmfmap` model to the `*.diagram` model for use in the Diagram Definition Editor.

Importing ecore for the Domain Model Editor

Related Topics

1. To import metamodels into a domain, choose **File ➤ New ➤ Import Domain Model**.
2. Import each `*.ecore` or `*.genmodel` to create a `*.domain` model for use in the Domain Model Editor.

   Note: The Domain Model Editor supports both `*.ecore` and `*.genmodel` diagrams. Updates to both are made simultaneously.

Migrating Xtend-Based Templates to QVTO-Based Xpand Templates

The migration tool to migrate from QVTO-based Xpand comes included with the DSL Toolkit as part of GMF 2.2M4.

To migrate to QVTO-based Xpand

1. Ensure that legacy Xtend-based templates are available in the workspace and can be compiled.

   You might have to create a valid `.xpand-root` file in the root of the project with templates so that your project can point local templates to their external template roots in other projects or plug-ins.
2. Right-click the legacy project in your workspace and choose **Migrate to new QVTO-based xpand**.

   A new folder with the QVTO-based templates is created as a sibling of each template root specified in the `.xpand-root` file, which also references newly created `*.migrated` template folders. In addition, new Java source root (`*.qvtlib`) is created, QVTO native extensions are registered in the `plugin.xml` file, the new template builder (`org.eclipse.gmf.xpand.xpandBuilder`) is registered, and the `org.eclipse.m2m.qvt.oml.project.TransformationNature` and `org.eclipse.m2m.qvt.oml.QvtBuilder` are installed on the project.
3. Open the `.xpand-root` file in a text editor and verify that all external QVTO-based template roots specified there are available in your configuration and were updated during migration.
4. Examine the QVT source container, generated by `org.eclipse.m2m.qvt.oml.project.TransformationNature` and `org.eclipse.m2m.qvt.oml.QvtBuilder`, for errors in the `*.qvto` files.

   The current QVT builder allows only one root to be specified as the QVT source container. By default, the first migrated template root is designated as the QVT source container. To see QVT build errors for other template roots, specify the appropriate folder as the QVT source container in the corresponding project properties.
5. Add the `org.eclipse.m2m.qvt.oml` plug-in to the list of required plug-ins. This resolves references to `org.eclipse.m2m.qvt.oml.blackbox.java.Operation`, which generated Java code imports for Java annotations.

6. Repeat the preceding steps for each template project as necessary. If you have cross-project dependencies, migrate the project with the most commonly referenced templates first.

Note: Existing projects with legacy templates cannot be compiled correctly using the new builder until they are migrated. This includes independent template projects. Before the migration templates project gets compiled with the legacy Xtend-based builder, the builder of the project gets replaced with the new QVTO-based project builder. Note that only `*.xpt` and `*.ext` files are migrated to the template root. All other files from the legacy template root are copied into the same relative directory beneath the template root.

Running a DSL

To run a DSL

1. Create a new run configuration by choosing **Run ➤ Open Run Dialog...** from the main menu.

   The **Run** manager dialog opens.

2. Right-click **Eclipse Application** and choose **New**.

   Type a descriptive title in the name text box and click **Apply**.

Note: You might need to adjust the default configuration to improve memory management. Remove unnecessary plug-ins on the **Plug-ins** tab and adjust the VM arguments on the **Arguments** tab. Modify settings like `-Xms128M -Xmx512M -XX:PermSize=64M -XX:MaxPermSize=128M` to match your system.

Related Topics

Regenerating a DSL

To regenerate a DSL

1. Open the **DSL Editor**.

   Within the **Generation** group, click **Generate**.

2. Optional: Depending on the complexity of your edits, you might want to click **Regenerate** in the **Export** group. This action regenerates the required plug-ins.

Tip: Under some circumstances, you might need to delete the generated projects to clear compiler errors. After manually editing the gmfgraph model, for instance, you might need to delete the generated diagram project.

Caution: Exercise caution when deleting generated projects because any manual edits are lost.

Related Topics

DSL Toolkit Workflow on page 12

DSL Editor on page 44

Deploying a DSL

To deploy a DSL

1. Open the **DSL Editor**.

   Within the **Export** group, click the **Regenerate** link.

2. Open the **Deployment** page and ensure that the feature is configured and the plug-ins or features that you want are selected.

   Under normal circumstances, this content is managed automatically. Some cases require manual editing.

3. Open the **Overview** page and click **Export Wizard**.

   This action launches the standard Eclipse export wizard to guide you through the deployment process. The exported feature is ready for installation.

Related Topics

DSL Editor on page 44

Reference

DSL Perspective

The Domain-Specific Language perspective configures the Eclipse interface for DSL development. The following table describes elements of the perspective:

DSL Explorer	Provides a logical view of each DSL project in terms of artifacts and lets you manage them.
Wizards	The Domain-specific language perspective provides several wizards. These wizards create new DSL projects, domain models, diagram definitions, figure galleries, model report definitions, operational QVT transformations and libraries, and Xpand templates.

To choose the domain-specific language perspective:

From the main menu, choose **Window ➤ Open Perspective ➤ Other**.

The **Select Perspective** dialog box opens.

Select the Domain-specific language perspective from the list and click **OK**.

DSL Editor

- *Domain-Specific Languages* on page 9
- *Creating a DSL Toolkit Project* on page 23
- *Creating a Domain Model* on page 21
- *Creating a DSL Diagram Definition* on page 26
- *Creating a DSL Diagram Definition* on page 26
- *Creating a DSL Template* on page 32
- *Creating a DSL Report* on page 35
- *Workflow Editor* on page 67
- *Template Explorer* on page 69

Overview Page

This page provides access to common DSL editing tasks.

Getting Started

Contains a collection of links to DSL Toolkit wizards.

New Domain Model creates a new domain model.

Imported Domain Model creates a domain model based on an existing EMF model or XSD schema.

Textual Notation creates a textual representation of the domain model.

Dynamic Instance selects a class to instantiate.

Diagram Definition creates a new diagram definition.

New Figure Gallery creates a new figure gallery.

Imported Figure Gallery creates a new figure gallery based on an existing gmfgraph model.

Model Transformation creates a new QVT transformation.

Report Definition creates a new domain model report (ecore).

Diagram Report Definition creates a new diagram model report (gmfgraph) that includes embedded diagrams saved as image files. Instead of the `.domain` model as the input, the `.diagram` model is used so that the diagram file and its content is available in the report definition.

Template Collection creates a new template collection.

Workflow creates a new Script model.

General Properties

Contains fields for configuring general DSL Project options for the creation of new artifacts.

User-Friendly name specifies an easily identifiable name for your project.

Branding Provider specifies the branding provider for newly created artifacts and generated plug-ins. For example, `Borland Inc.`.

Base Package specifies your project's base package. For example, `com.borland.mindmap`.

Validation and Generation

Provides validation and generation options for the DSL.

Validate checks the integrity of your DSL project models and configuration.

Generate creates the necessary plug-ins and source code for the DSL.

Generate included domains enables generation of code for domain models included in this DSL.

Generate included diagrams enables generation of code for diagrams defined within this DSL.

Navigation

A set of links to pages in this editor.

Contents manages artifacts included in this DSL.

Transformations configures transformations for this DSL.

User interface generates custom UI extensions.

Deployment manages deployment of the DSL.

Export

Provides links to assist in packaging and exporting the DSL.

Contents

This page manages the artifacts included in the DSL.

Domain Models

New creates a new domain model and adds it to this DSL.

Import creates a new domain model based on an existing EMF model or XSD schema and adds it to this DSL.

Add contributes an existing domain model to this DSL.

Open domain diagram opens the diagram for the selected domain model.

Configure edits the list of referenced models.

Generates Model and Edit code indicates which plug-ins are generated for the domain.

Expose to M2M transformations makes the selected domain model usable by QVT transformations.

Textual notations

New creates a new textual notation and adds it to this DSL.

Add contributes an existing textual notation to this DSL.

Diagrams

New creates a new diagram definition and adds it to this DSL.

Add contributes an existing diagram definition to this DSL.

Generate Composite Editor combines the Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF) editors into a single editor for simultaneous modifications to this DSL.

Dynamic Instances

New creates a new metaclass of dynamic instance that is available across your entire DSL project instead of for a specific domain model only.

Add lets you select other project-wide dynamic instances, which are subsequently listed in a folder under a DSL Project root in the DSL Explorer for viewing and organizing.

Audits and Metrics

New creates a new audit or metric and adds it to this DSL.

Add contributes an existing audit or metric to this DSL.

Transformations

This page manages transformations, templates and transformation sequences.

Transformations

Model transformations defined for the DSL.

New creates a new QVT transformation and adds it to the DSL.

Add chooses an existing QVT transformation and adds it to the DSL.

Launch opens the **Run configuration** dialog box with pre-populated values for the selected transformation.

Template Collections

Lists template collections (folders) with their contents (Xpand or JET templates).

Xpand template enabled only when a template collection is selected. Creates a new Xpand template.

JET template enabled only when a template collection is selected. Creates a new JET template.

New Group enabled only when a template collection is selected. Creates a template collection nested under the selected template collection.

Remove deletes the selected node as well as any children of that node.

Template Collection creates a new template collection folder.

Design-time contains custom templates for UI and feature projects generation. Use this option with **Show templates view** to override templates.

Show templates view opens the **Templates Explorer** view.

Force deployment forces the template collection to be copied into a UI plug-in. The template collection is otherwise used in action contributions or in workflow scripts.

Workflow

Lists workflow scripts associated with this DSL.

New creates a new Workflow script and adds it to the DSL.

Add chooses an existing Workflow and adds it to the DSL.

Right-clicking a listed workflow script and selecting the **Open** menu item (or double-clicking the script) opens the selected workflow in its editor.

Right-clicking a listed workflow script and selecting the **Unlink** menu item removes the selected workflow from the list.

Right-clicking a listed workflow script and selecting the **Delete with resource(s)** menu item removes the selected workflow from the list and deletes the corresponding `*.exec` and `*.exe_diagram` files.

User Interface

UI Contribution Plug-in

Offers options to extend the Eclipse runtime UI.

Enable enables generation of the UI plug-in, which has its content described on this page, and the editing of the **Action Contributions** group.

Plug-in ID identifies the UI plug-in ID (`*.ui`).

Version identifies the plug-in version.

Plug-in Name identifies the UI project name.

Activator class identifies the UI activator class (`*.ui.Activator`).

Localization class identifies the UI localization class (`*.ui.Messages`).

Generate Activity specifies whether to automatically generate an activity, and whether to generate activity that hides all contributions of this DSL project. These contributions include domains, diagrams, and the UI project. Each activity's category is visible from **Window ➤ Preferences ➤ General ➤ Capabilities** menu. To access the activities themselves, you must follow the same navigation and then click **Advanced...**. When you expand the categories, the activities are displayed.

ID identifies the activity's category ID (`*.activity`).

Name specifies the name of the DSL.

Description describes the activity.

Category ID identifies the activity's category ID.

Category Name identifies the activity's category name.

Category Description describes the activity's category.

Enable by default sets generated activity to be enabled by default.

Action Contributions

Lists actions associated with this DSL. The actions are context-menu options for the DSL models when deployed.

Invoke template adds a menu item to execute an Xpand template.

Run transformation creates a menu item to apply a model transformation.

Run report lets users run a defined report for the model.

Run script provides a menu item to apply a transformation sequence.

Generate Project Wizard

Offers options to generate a wizard to create new instances of the DSL as well as a new Eclipse perspective.

Generate wizard enables generation of a project wizard for this DSL.

ID sets the identifier for the wizard.

Name provides a name for the wizard.

Description describes the wizard.

Class Name assigns a class name for the wizard.

Category ID provides a category for the wizard.

Category Name names the wizard category.

Generate icon for wizard enables generation of an icon for the wizard.

Generate perspective enables generation of a new perspective for the DSL.

ID sets the identifier for the perspective.

Name provides a name for the perspective.

Description describes the perspective.

Class assigns the class for the perspective.

Deployment

Manages DSL deployment artifacts.

DSL Feature

Enables generation of a feature plug-in.

Generate feature enables generation of a feature project.

Project name specifies the name for the feature project.

ID sets the identifier value for the feature.

Version defines the version number.

Name names the feature.

Provider sets the provider value.

Generate feature project executes the generation of the feature.

Generate stale files

Lists files that can safely be deleted. Stale files are artifacts that are leftover after a model changes to such a degree that regeneration no longer produces those artifacts. The list is populated automatically after each successful regeneration. After you review the files, you can click **Remove** or **Remove all** to delete the stale files.

Note: If your custom file is listed among the stale files, remove all `@generated` comments from it.

Delete stale files as soon as possible.

Plug-ins and Features

These lists define additional plug-ins to include into feature definition. Domain, Diagram, and UI plug-ins are included automatically. You can add your own custom plug-ins to these lists.

Related Topics

Domain Model Editor

The Domain Model Editor defines the abstract syntax of a domain-specific language. A domain model is defined in terms of Eclipse Modeling Framework's (EMF's) Ecore metamodel, which is closely aligned with the Object Management Group's (OMG's) Essential Meta Object Facility (EMOF) specification. While traditional EMF-based development involves the creation and synchronization of a model (ECORE file) and a corresponding generator model (GENMODEL file), the Domain Editor lets you work with both simultaneously using a feature-rich diagram editor.

Overview Page

This page contains general information about this domain model and its generated plug-in.

General Properties

Domain Name is the name of the domain model.

NS URI specifies the namespace URI. Use this URI when importing the model in templates or during model transformations.

NS Prefix is the namespace prefix.

Generation Options

Manages generation of support code for the domain model

Plug-in ID names the plug-in identifier.

Base Package specifies the base package name used in generating code.

Validate verifies the integrity of the domain model.

Generate generates artifacts for the domain model.

Use generation model is enabled by default, which indicates that the generation model (GenModel) is persistent.

Generation model file is normally left as the default.

Show generation properties shows or hides the generation properties.

Use advanced merge capabilities merges `plugin.xml` and `MANIFEST.MF` when generating code and keeps them synchronized throughout the development process. Disable this option if you prefer to manually manage them.

Generate all referenced models includes all referenced generator models in the generation of this domain model.

Stale Files

Lists files that can safely be deleted. Stale files are artifacts that are leftover after a model changes to such a degree that regeneration no longer produces those artifacts. The list is populated automatically after each successful regeneration. After you review the files, you can click **Remove** or **Remove all** to delete the stale files.

Navigation

Contains links to the other pages in this editor, as follows:

- **Diagram**
- **References**
- **Advanced**
- **Tree**
- **Generation**

Diagram

Domain model diagram lets you visually model the domain by using standard diagram concepts.

Note: When you use the **Properties** view to review model and element properties, you can set negative values as necessary. However, when you use the in-place editor, which you can access by pressing the **F2** key, all negative values less than `-2` are recognized as `-2`, and the `unspecified` label indicates `-2` multiplicity. Only elements of the Domain models, Generator models, and Ecore models listed in the Domain Model Editor's **References** page that are compatible with the corresponding property type are displayed in the **Properties** view.

The palette of the **Diagram** page supports graphical representation of Object Constraint Language (OCL) annotations, such as the Constraint option in the OCL group. This feature visualizes the predefined semantics of OCL annotations. Each visual notation maps to a particular EAnnotation structuring.

Derived Feature and **Operation Body** graphical notation options are also on the palette. OCL `derive | body` constraints can be used as EAnnotation keys with a value of OCL expression. Code can then be generated based on these annotations, although code generation does not depend on annotations.

Note: With QVT projects, OCL annotations can be used for the derived features and operation bodies of dynamic instances without prompting code generation. Continue to synchronize dynamic and generated instances because such OCL constraints can be overridden in the proper runtime contexts. OCL annotations apply to dynamic instances only. Generated instances come with appropriate OCL-related runtime applications, and in these cases the code generator is responsible for specifying that OCL declarations are applied. Other clients, including the **Dynamic Instance** editor, do not suport OCL annotations at all.

Text

The Text page of the Domain Editor contains text representation of the domain model.

The following example demonstrates an Ecore textual notation of a zoo metamodel.

```
package zoo {
 documentation : "This metamodel is a sample of domain textual notation";
 documentation : "http://www.wwf.org/";
 nsPrefix : "zoo";
 nsURI : "http://www.domain.example.org/2009/zoo";

 interface Creature {
  documentation : "documentation is special annotation, which is also used in
Properties view for 'Documentation' tab. \n"
```

```
   "All text values in text notation can have multiple lines";
  reference cage : Cage<Creature> opposite content {
   documentation : "Reference with opposite feature";
  }
  operation eat(food : Food) throws BadFoodException {
   documentation : "Sample of operation that throws an exception";
  }
 }

 class Zoo {
  documentation : "This class contains special annotation for EMF constraints\n"

   "Body of 'constraints' annotation contains a list of validation names with
space or line feed";
  constraints : "EmfConstraint1 EmfConstraint2 EmfConstraint3";
  containment animals : Animal[0..*] {
   documentation : "Multiplicity in square brackets";
  }
  containment birds : Bird[0..*];
  containment cagesForAnimals : Cage<Animal>[0..*] {
   documentation : "Containment reference has type with type argument";
  }
  containment cagesForBirds : Cage<Bird>[0..*];
  operation getAllCreatures : Creature[0..*] {
   documentation : "Operation with ocl implementation";
   operationbody :
"birds.oclAsType(Creature)->asOrderedSet()->union(animals)->asOrderedSet()";
  }
 }

 interface Cage<T extends Creature> {
  documentation : "Class with type parameter";
  containment content : T opposite cage {
   documentation : "Containment reference with 'type parameter' as type";
  }
  operation canContain(t1x : T, t2x : T) : EBoolean;
  operation canContain as canContain1(t : T[1..*]) : EBoolean {
   documentation : "'canContain1' is ID of the operation. \n"
    "If the name of an element is unique in model, it is not necessary to add
additional ID, \n"
    "name of element will be used as element identifier.\n";
  }
  operation put(creature : T) throws InvalidCreatureException<T>;
 }

 class InvalidCreatureException<CR extends Creature> {
  reference creature : CR;
 }

 interface Animal extends Creature {
  documentation : "'interface' is a class with interface attribute";
  ~unique attribute size : EDouble {
   documentation : "All datatypes declared in Ecore metamodel are available by
short name";
  }
 }

 class StBernard extends Animal {
  attribute name : EString = "Beethoven" {
   documentation : "Attribute with the default value";
  }
```

```
}

class Doghouse {
 reference dogs : StBernard[0..*] {
  keys : name;
 }
}

abstract class AbstractCage<Y extends Creature> extends Cage<Y> {
 documentation : "Sample of the derived feature and its implementation";
 attribute length : EDouble;
 attribute width : EDouble;
 attribute height : EDouble;
 derived volatile attribute volume : EDouble {
  featurebody : "length * width * height";
 }
}

class Canines {
 documentation : "Sample of ocl constraint for 'length' feature";
 oclconstraint : "-- valid length of canines\n"
  "lenght > 0" {lenght};
 attribute lenght : EInt;
}

class SuperCage<E extends Creature> extends Cage<E> {
 documentation : "Class with implementations for abstract operations from super
class";
 operationbody : "true" {canContain1};
 operationbody : "true" {canContain};
}

interface Bird extends Creature;

interface Predator extends Animal {
 containment canines : Canines[1..1];
}

interface Dinosaur extends Creature {
 documentation : "@deprecated";
}

datatype BadFoodException {
 documentation : "Datatype with instance class name attribute";
 instanceClassName : "org.example.domain.zoo.BadFoodException";
}
datatype Food {
 documentation : "This datatype contains special annotation \n"
  "'extendedMetaData' - annotation with fixed source:
'http:///org/eclipse/emf/ecore/util/ExtendedMetaData'";
 extendedMetaData {
  "baseType" -> "http://www.example.org/2000/FoodSchema#food";
 }
 instanceClassName : "org.example.domain.zoo.Food";
}

package bestiary {
 documentation : "Example of subpackage";
 annotation "superSource" {
  documentation : "Example of list with line wrapping";
  references : Dinosaur, Bird, Predator, InvalidCreatureException,
```

```
BadFoodException, Canines, SuperCage, Cage, AbstractCage,
    Doghouse, Creature, zoo;
  }
  nsPrefix : "bestiary";
  nsURI : "http://www.domain.example.org/2009/zoo/bestiary";
  class Chimera;
 }
}
```

References

This group provides editors for managing referenced models. If your domain model uses or extends elements from another model, include a reference in the appropriate list.

Domain Models adds and removes domain models (`*.domain`)

Generator Models adds and removes generator models (`*.genmodel`)

Ecore Models adds and removes ecore models (`*.ecore`)

Advanced

This page provides additional model and generation properties.

Other Properties

Compliance level sets the compliance level for the generated code.

Copyright text defines the copyright text to use in the generated artifacts.

Advanced generation properties

Use dynamic templates enables use of dynamic templates for code generation.

Path to dynamic templates is the location of the dynamic templates.

Show templates view allows you to browse the hierarchy of EMF's built-in JET templates used for model code generation. This option also lets you override necessary template files by copying them into the folder you specified in the **Path to dynamic templates:** field. You can then customize their content. Code generators can be dynamically created from those templates.

Use advanced templates enables use of advanced dynamic templates for code generation. Because the editor does not populate these templates automatically, click **Populate Templates...** to load the available templates before generating the code.

Use UUIDs lets you initialize the **Resource Type** for the genpackage as `XMI`. When the code is generated, the editor overrides the `*ResourceImpl.useUUIDs()` method for all genpackages with `type != XMI`. XMI resources can use unique identifiers, which assign unchangeable unique IDs to entities such as GMF diagrams. Consequently, an entity's ID remains the same even if the name of the entity changes. However, resource text readability can suffer.

Generate database support lets you persist Domain Models in a Hibernate database library. When this option is checked, a dialog box lets you initialize the **Resource Type** for the genpackage as `none`. Database persistence allows you to retain your Domain Model work in a database library so that it exists beyond your program execution. Hibernate library features include support for relational databases, OR Mapping of class hierarchies, data query and retrieval, HQL query language, and second-level caching. A validation warning is also generated.

In the **Project with Hibernate libraries:** field, select the name of the Eclipse project with the Hibernate binary library.

Dynamic Instances

Lists the dynamic instances for this model.

Add opens a dialog box that lets you create a dynamic instance for the model element you select.

Remove deletes the reference to the dynamic instance without deleting the XMI file.

Validate verifies the integrity of the dynamic instance.

Cleanup removes dynamic instances that are no longer part of the project.

Tree

A tree view and editor of the domain model. Edits in this view are synchronized with the diagram view.

Tip: Use the **Properties** view to review model and element properties. The **Properties** view displays only elements of the Domain models, Generator models, and Ecore models listed in the Domain Model Editor's **References** page that are compatible with the type of corresponding property.

Generation

A table view/editor of the domain model

Group	Property	Type
All	Bundle Manifest	Boolean
	Compliance Level	1.4, 5.0 or 6.0
	Copyright Fields	Boolean
	Copyright Text	string
	Model Name	string
	Model Plug-in ID	FQN
	Non-NLS Markers	Boolean
	Runtime Compatibility	Boolean
	Runtime JAR	Boolean
Edit	Creation Command	Boolean
	Creation icons	Boolean
	Edit Directory	location
	Edit Plug-in Class	FQN
	Provider Root Extends Class	FQN

Group	Property	Type
Editor	Creation Submenus	Boolean
	Editor Directory	location
	Editor Plug-in Class	FQN
	Rich Client Platform	Boolean
Model	Array Accessors	Boolean
	Binary Compatible Reflective Methods	Boolean
	Containment Proxies	Boolean
	Feature Delegation	[none, reflective, virtual]
	Generate Schema	Boolean
	Minimal Reflective Methods	Boolean
	Model Directory	location
	Model Plug-in Class	FQN
	Model Plug-in Variables	string[]
	Suppress Containment	Boolean
	Suppress EMF Metadata	Boolean
	Suppress EMF Model Tags	Boolean
	Suppress GenModel Annotations	Boolean
	Suppress Interfaces	Boolean
	Suppress Notification	Boolean
Model Class Defaults	Public Constructors	Boolean
	Root Extends Class	FQN
	Root Extends Inteface	FQN

Group	Property	Type
	Root Implements Interface	FQN
	Static Packages	list
Model Feature Defaults	Boolean Flags Field	string
	Boolean Flags Reserved Bits	integer
	Feature Map Wrapper Class	FQN
	Feature Map Wrapper Interface	FQN
	Suppress EMF Types	Boolean
	Suppress Unsettable	Boolean
Templates and Merge	Code Formatting	Boolean
	Dynamic Templates	Boolean
	Facade Helper Class	FQN
	Force Overwrite	Boolean
	Redirection Overwrite	Boolean
	Template Directory	string
	Template Plug-in Variables	list
	Update Classpath	Boolean
Tests	Tests Directory	location
	Test Suite Class	FQN

Diagram Definition Editor

- *Template Explorer* on page 69
- *Model Audits and Metrics Descriptions*
- *Creating a DSL Diagram Definition* on page 26
- *Regenerating a DSL* on page 42

Overview Page

The main page contains several groups to manage general configuration properties and provide generation actions.

General Properties

This group lets you modify plug-in attributes. It is recommended that you accept the default values.

ID specifies the identifier for the plug-in, which is used to name the generated diagram-implementation project).

Version sets the version number.

Name is a descriptive name for this diagram definition.

Provider is typically the name of the organization.

Activator specifies the name of the generated diagram plug-in's activator class.

Generation Options

Configures options for the diagram, such as file extensions for the domain model and diagram files.

Model ID is the identifier for the domain model.

Domain File Extension displays the file extension for corresponding domain model instances.

Diagram File Extension specifies the file extension for instances of this diagram.

Package Name Prefix configures package naming for the generated code.

Generation

Provides actions to assist with generating implementation for the diagram definition.

Generate generates code for the diagram definition.

Validate checks the integrity of the diagram definition.

Update will update the generator model with content changes.

You can further define update reconciliation properties by optionally selecting the **Always update generator model before generation** and **Reconcile changes in generator model** check boxes.

Note: The DSL Toolkit contains a trace facility to help with reconciling changes made to the generator model when retransforming from the mapping model. It is recommended that you use this facility if you plan to make changes or augment the generator model in any way.

Navigation

A collection of links to open the other pages available in this editor.

The **Advanced** page defines advanced properties, such as user-defined templates.

The **Palette** page lets you customize your diagram palette.

The **Content** page defines mapping properties between a domain element and its representation.

The **Audits And Metrics** page lets you add audits and metrics to the diagram.

The **Tree** page lets you browse applicable models using a traditional tree structure.

Figure Galleries

Lists figure galleries in use by this diagram definition.

If you select the **Embed figures code in generator model** check box, figure code is inserted into the generator model and placed within the **EditPart** class. Alternatively, a figure gallery can generate a standalone figure plug-in that is referenced by the generator.

Add imports an existing figure gallery for use.

Remove removes the reference to the selected figure gallery but does not delete the figure gallery.

Advanced

Provides editing of advanced properties.

Advanced Properties

Diagram plug-in properties.

Enable Print support activates the ability to print the diagram at runtime.

Same file for diagram and model enables physically storing model content and diagram layout information in the same file at runtime.

Use the **Copyright text** field to define the copyright text to include in generated artifacts.

Use dynamic templates enables usage of dynamic templates in code generation. Use the **Path:** field to specify the location of the dynamic templates. You can browse the hierarchy of XPand templates used by GMF for diagram generation, and copy (override) or aspect (extend) necessary template files (or just their `<define>` parts) into a specified Template folder. There you can customize the template content. Modified templates can then be used on the next code generation.

Mapping to Generator Transformation

Properties to configure transforming the mapping model to the generator model.

Use IMapMode configures usage of IMapMode in generated GEF code. It is recommended to leave this property enabled.

Utilize enhanced features of GMF runtime enables the generator to target the full GMF runtime instead of the “lite runtime.” It is recommended to leave this property enabled.

Generate RCP instructs the generator to use the minimal dependencies in order to provide a lightweight diagram editor for the Eclipse platform.

Property Sheet

Properties to configure the generated property view for the diagram.

Needs Caption generates a properties view with a label.

Read Only prevents editing the properties view.

Package Name: specifies the package for the generated property sheet.

Label Provider Class Name: names the class that provides the label for the property sheet.

Shortcuts

Properties to manage model and diagram references.

Referenced models specifies models whose elements can be referenced in this diagram.

Referencing diagrams specifies other diagrams that might reference this diagram.

Palette

This page is an editor for the diagram's tooling palette and its elements. You can customize your palette by clicking one of the following links and filling in the property fields.

Creation tool includes elements for instantiating nodes and links.

Standard tool includes standard elements such as Select, Zoom In, and Note.

Generic tool includes generic elements for use with custom tool classes.

Group arranges several palette elements together.

Separator graphically separates elements or groups of elements.

delete removes the selected elements.

Properties lets you view and optionally edit key properties for the selected element. Note that you can use the Eclipse Properties view as well to view and edit all available properties. Configurable properties include:

Title:	Specifies the name of the tool or group.
Description:	Specifies the function of the tool or group.
Large Icon:	Lets you specify whether a palette tool is displayed as a large icon by default. **Tip:** Click the image plate (the blank square beneath the **Default** check box) to access a popup dialog that lets you specify a bundle.
Small Icon:	Lets you specify whether a palette tool is displayed as a small icon by default. **Tip:** Click the image plate (the blank square beneath the **Default** check box) to access a popup dialog that lets you specify a bundle.
Grouping:	Lets you specify whether a palette group is displayed as stacked or collapsible (available when a palette group is selected).
Class:	Lets you specify the class of a new tool.
Kind:	Lets you specify whether a new tool is SELECT, SELECT_PAN, MARQUEE, ZOOM_PAN, ZOOM_IN, or ZOOM_OUT.

References specifies how a selected element is used in the diagram.

Details provides a Reconcile action on a selected element that launches a wizard to update the palette model based on changes in the domain model.

Mapping

This page provides for the mapping of diagram graphical elements to their domain and tooling elements. The layout of this page is designed to guide the Toolsmith through actions that create mappings in a diagram-centric manner.

Following the creation of a diagram definition using the provided wizard, the Content page will contain a Canvas mapping. The domain model has its elements listed in the **Domain Element** section. These elements are mapped by selection in the context of the currently selected Diagram Element. The **Actions** section provides links to create or remove elements for the selected context.

If a Canvas is selected:

- In the **Actions** section, the **Add node** action initializes a new diagram node on the Canvas.
- Properties can be specified in the **Properties** section, and selections can be made in the **Figure** section and **Creation Tool** section:
    - A Domain Element is selected to map the new node to a class from the domain it is to represent.
    - The **Containment Reference** field is used to select the domain model reference where objects of this type are stored.
    - The **Children Reference** field is used to select a reference from which children are retrieved. This property can be left blank, because it applies only to the domain modeling pattern whereby a single containment reference is used to store all objects of a general type. Separate read-only references are used to retrieve objects of a particular type.
    - **Related Diagrams** can be added or removed. By declaring the Canvas of a diagram definition as related, you can double-click this node to open diagrams of that type.
    - An **Initializer** can be defined to set properties (such as an attribute value or reference) of the newly created domain element.
    - A **Specialization** can be defined using one of several languages (such as OCL, Java, regexp, or nregexp) to more strictly define the domain elements that are being mapped. For example, if multiple mappings apply to a single domain element, the value of a feature can be used to distinguish between them.
    - The **Figures** section provides a list of eligible figures from the associated galleries. The **Creation Tool** section displays the available tools from the Palette, with actions to add a new tool or group without first navigating to the Palette page of the editor.

If a node is selected:

- **Remove the node** deletes the node mapping from the diagram definition.
- **Add node reference** enables this node to specify a child node to be reused from another node mapping. This avoids duplication of mappings throughout the definition if there are many similar nodes. A node reference element has **Containment Reference** and **Children Reference** properties, and its **Actions** section provides the ability to **Remove the node reference**.
- **Add a node** creates a child node below the current node and displays all the properties for a node. An infinite level of children nodes can be mapped in this manner, as dictated by the graphical notation.
- **Add compartment** provides a compartment in which a node can be defined. The **Child Nodes** property lets you select nodes found within the compartment, and the **Figures** section lets you select from available compartment figures in the referenced galleries. A **Remove the compartment** action is also available.
- **Add label** creates a mapping for an optionally `Read Only` label using the label selected in the **Figure** section. Note that unlike an attribute-based label mapping, this is a static label mapping that does not provide text based on a feature. A design label is used when diagrams are defined without a domain model. A **Remove this label** action becomes available when the label is selected in **Diagram Element** as well.
- The **Add attribute-based label** action creates a label mapping where one or more domain element attributes can be selected for edit and display on the diagram. The following properties are available for attribute-based labels:
    - **Read Only** prevents the Practitioner from modifying this label's value on the diagram.
    - Add or remove Attributes using hyperlinks. The **Add** button opens a dialog with the available attributes of the parent node's mapped domain model class.
    - **View Pattern** defines how the attribute is seen when the label is not in edit mode. By default, the MESSAGE_FORMAT View Method is used, which allows for the specification of patterns using the Java

MessageFormat class. See the JavaDoc on this class for more information. Optionally, select the NATIVE (Java), REGEXP, or PRINTF patterns for **View Method** and **Edit Method**.
    - **Edit Pattern** defines the format of the label when the parser accepts it after changes are made. For example, if the **View Pattern** displays several attribute values separated by commas, you can type each value into the in-place editor of the label using colon delimiters.

- The **Add link** action initializes a new diagram link. Actions available with a link selected include **Remove the link**, **Add label**, and **Add design label**. Select figures and palette tools using the corresponding sections in the editor. Additionally, the following Properties are available for a link mapping:
    - **Containment Reference** refers to the location that stores new instances of link elements (for example, when a domain element class is used to represent a link, and is selected using the **Domain Element** section). In this case, a **Source Reference** and **Target Reference** mapping will be defined as well. Links that represent domain model references require only the **Target Reference**. **Specialization** and **Initializer** properties (described previously) are also available for link mappings.**Note:** In order to indicate where they are stored in the domain model, some nodes need to be first created and then linked. In these cases, a **Node** element is created without a **Containment Reference** specified. Instead, a link mapping is created and specifies the containment to be used for storing the node in its **Target Reference** property.
    - **Source Constraint** and **Target Constraint** elements can be added to a link to restrict how a Practitioner creates links between elements on the diagram. OCL, Java, regexp, and nregexp languages can be used to define constraints.

Validation

This page lets you define the audits and metrics of diagrams and domain models. Use the audit to detect the deviation from a best practice or to validate the content of a model prior to some processing (for example, transformation or code generation). Use metrics to gather numeric data on model content (for example, the number of elements of type *X*).

From the **Mapping** root element in the tree, use the **New Child** right-click menu action to add **Audit Container** and **Metric Container** elements. These elements have the following properties that open in the **Details** section.

Note: The **Metric Container** has no properties of its own but can contain child **Metric Rule** elements.

- **Metric Rule** elements have **Name**, **Description**, **Key**, **Low Limit**, and **High Limit** properties. A **Value Expression** child can be added, which allows for the specification of an OCL, Java, Regexp, or NRegexp expression that defines the rule. Along with a **Value Expression**, a **Domain**, **Diagram**, or **Notation Element** is declared as the context for the expression.
- **Audit Container** has **Id**, **Name**, and **Description** properties. Child **Audit Container** elements can be created. **Audit Rule** elements can also be added and, like **Metric Rule** elements, consist of a Constraint and an associated context model element. Additionally, Audit definitions can report on Metric results.

Tree

This page provides a composite viewer and editor for all models involved in this diagram.

Tip: This tree provides easy access to the gmfgraph model. Common graphical customizations are made on the **Tree** page, such as visual attributes for diagram elements. Enable **Always update generator model before generation** on the **Overview** page to reconcile changes made here.

Related Topics

Textual Notation Editor

Textual notation is a DSL artifact that comes with its own editor, the Textual Notation editor, and generates a single plug-in.

The Textual Notation editor contains four pages: Overview, Language, Advanced, and Text.

Related Topics

Overview Page of Textual Notation Editor on page 62
Language Page of Textual Notation Editor on page 63
Advanced Page of Textual Notation Editor on page 65
Text Page of Textual Notation Editor on page 65

Overview Page of Textual Notation Editor

General Properties

The **General Properties** node contains the following fields that let you define and configure your generated textual notation plug-in, and also provides generation options.

Plug-in ID	Specifies the plug-in ID of the textual notation.
Base package	Specifies the root package for generation.
Text File Extension	Specifies the extension of the file in which the model is persisted in textual form.
Plug-in Name	Specifies the name of the textual notation.
Plug-in Provider	Specifies the plug-in provider.
Custom Templates	Lets you browse for dynamic templates to customize generation. Click on the **Custom templates** link to open the **Template Explorer**.
Custom Transformation	Lets you specify a path to a QVT file transformation for creating a generation model.
Custom Factory Class	Lets you specify a Java class to customize generated code. The **Custom Factory Class** link opens the selected class in the editor or prompts you to create a class if one does not already exist (alternatively, click the **Browse** button to select an existing class).

Source packages

The **Source packages** node lets you select source language metamodels for your textual notation. By clicking the **Add** button, you specify which source models and Genpackages to contribute.

Uncheck the **Use default file extension** check box to specify a different extension to the model file (the default value is taken from the domain model). The file extension is used to register converter actions and editor for XMI resource.

Validation and Generation

The **Validation and Generation** node contains the following options and buttons for enabling features and generating.

Actions to convert between text and model	Check this check box if you want to be able to convert between textual and XMI representations.
Register resource factory for text file	Check this check box if you want to be able to work with the text file in the same way you work with the usual model.
Register Text editor for XMI model resource	Check this check box if you want to be able to edit XMI files as text (this option transparently converts between textual and XMI representations upon each save/load).
Generate	Click this button to generate your textual notation.
Validate	Click this button to validate your textual notation before generating.

Related Topics

Textual Notation Editor on page 62

Language Page of Textual Notation Editor

Textual Notation

The Language page lets you define the syntax and formatting of your textual notation. It contains a single node, the **Textual Notation** node, which displays the filtered model tree and lets you specify how each of your model elements contribute to your notation. Drag and drop features to reorder them.

In addition to the **Expand** and **Collapse** display options, use the following icons on the upper right of the window to filter the content of the tree:

- Show Model Types and Features (default)
- Show All Inherited Features Without Excluded
- Hide Excluded Features and Types
- Show Only Excluded Features and Types

Depending on the language element selected, different configurable options appear that let you control how selected elements appear in the notation, including a Preview window that displays how your settings affect the notation.

The following configurable options appear:

Class options By default, all classes in your model begin with a reserved word, possibly followed by its instance identifier. All attributes are enumerated one by one in curly braces (class body).

Custom Literal	Lets you customize the reserved word, which is the metaclass name by default.
Literal	Space-separated list of tokens and formatting rules, including:

identifier	Introduces soft keyword.
!identifier	Introduces hard keyword.

sequence of control characters	Standard tokens, such as comma and semi-colon.
_	Space in the text.
[NL]	New line in the text.

Element body options Lets you customize the class body.

Custom identity attribute Lets you select an identifying attribute that makes references to the class more friendly. This option is inherited by subclasses.

Show in Outline Lets you check the appropriateness of the class in the outline.

Feature options

Feature options include the following:

Custom Literal Lets you customize the reserved word (the default is the feature name).

Literal Space-separated list of tokens and formatting rules, including:

identifier	Introduces soft keyword.
!identifier	Introduces hard keyword.
sequence of control characters	Standard tokens, such as comma and semi-colon.
_	Space in the text.
[NL]	New line in the text.

Location Defines the location of the feature within the class.

List options (for multi-valued features only) Lets you customize list syntax and formatting.

Show in Outline Shows the feature's value in the outline.

Default containment (for containment features only) Inserts the feature value into the class body without preceding it with the literal.

Feature color Lets you select a color (created on the Advanced page).

Enumeration literal options

Enumeration literal options include the following:

Custom Literal Lets you customize the textual representation of the literal (the default is the literal name).

Literal Space-separated list of tokens and formatting rules, including:

identifier	Introduces soft keyword.
!identifier	Introduces hard keyword.
sequence of control characters	Standard tokens, such as comma and semi-colon.
_	Space in the text.
[NL]	New line in the text.

Related Topics

Textual Notation Editor on page 62

Advanced Page of Textual Notation Editor

Preferences

The **Preferences** node contains the following check boxes and fields that let you set preference page options.

Custom editor's preference page name	Select this check box to specify a different name for your editor's preference page from the default name.
Category name	Specify the custom name for your editor's preference page here.
Custom category for preference page	Select this check box to specify a custom parent category for your preference pages (appear in the root of preferences by default).
Parent category ID	Specify the ID of the category where you would like to place your preference pages.
Syntax coloring preference page name	Specify the name for your coloring preference page here.

Notation defaults

The **Notation defaults** node lets you change Doctype and maximum line width default values in the generated editor.

Additional highlighting

The **Additional highlighting** node lets you add colors for custom semantic highlighting. When you add a color, options appear that let you specify the color's ID, title, and color. You can also specify whether the text for the ID you specify has bold, italic, strikethrough, or underline highlighting.

Created colors can be used on the Language page. Select any feature to see the **Semantic Highlighting** option to the right.

Related Topics

Textual Notation Editor on page 62

Text Page of Textual Notation Editor

The **Text** page of the Textual Notation editor is a full-featured text editor that contains options from the Textual Notation editor's first three pages in a text form and includes the following features:

- copy/paste
- syntax highlighting
- code completion
- error highlighting
- outline
- quick outline
- reference navigation

All changes you make are applied to the underlying model incrementally when you save or switch to another page.

The editor validates the text as you type, using EMF validation. To customize how often text is validated, choose **Window ➤ Preferences ➤ DSL Toolkit ➤ Config Textual Editor**. To customize colors, choose **Window ➤ Preferences ➤ DSL Toolkit ➤ Config Textual Editor ➤ Syntax Coloring**.

Related Topics

Textual Notation Editor on page 62

DSL Explorer View

For enhanced navigation of your DSL projects, DSL Toolkit offers a specialized DSL Explorer view. The DSL Explorer provides a logical view of each DSL and lets you perform a variety of DSL tasks. Using the DSL Explorer, you can create DSL projects, fill them with artifacts (including domain models, diagrams, textual notations, report definitions, transformations, templates, and transformation sequences), transfer artifacts between existing projects, validate projects and their contained artifacts, use templates and create new template collection folders, and generate all the DSL code.

The DSL Explorer is similar to the more general Project Explorer view, both of which are based on the Eclipse Common Navigator Framework. While both views are available, the Project Explorer no longer provides DSL navigation. View any DSL project content by using the DSL Explorer.

Note: The DSL Explorer does not yet support any Java contributions that you can browse. This issue stems from Eclipse Common Navigator Framework's inability to support two or more major contributors, such as Java, in one view. To browse Java projects, it is necessary to switch from you DSL Explorer to your Package Explorer.

Importing DSL Projects from an Existing Platform on page 38

Features of the DSL Explorer

The following DSL Explorer features enhance navigation by supplying a more specialized, logical view of DSL projects:

- The DSL Explorer is always synchronized with all of the project editors. When you add a domain in an editor, the domain is immediately propagated to the DSL Explorer. Conversely, domains added through the DSL Explorer are immediately propagated to the editor.
- You can navigate all DSL projects that have been contributed into the platform. These projects are listed under the DSL Explorer's **Platform** node. To enable this node, you must first turn off the **Platform DSL Projects** filter by clicking the **DSL** icon (**Show Platform DSL Projects**) on the toolbar. Alternatively, choose **Customize View** on the **View Menu** and ensure that the **Platform DSL Projects** option is not selected.
- To import DSL projects from the platform into your workspace, choose the **Import DSLs...** context menu option. Alternatively, choose **File ➤ Import...** from the main menu and select the **Import DSL Project from Platform** option under the **Plug-in Development** node.
- The DSL Explorer supports working sets. From the toolbar menu, choose **Top Level Elements ➤ Working Sets** to create a new set, reorder existing sets, or input more projects into a set. Drag and drop projects to move them between sets.
- You can view all of your project resources under the DSL Explorer's **Resources** node. Although this node is not displayed by default, you can turn it on by choosing **Customize View... ➤ Resources in DSL Project**.
- When you drop a DOMAIN, TNT, DIAGRAM, DSLDESIGN, or QVTO file into a DSL project, an artifact is automatically added to the DSL project and stored in its DSL file. In DSL Explorer, this reference is listed

under the project's appropriate category. For example, `.domain` references are displayed under **Domain Diagrams**. When no editor is opened for the DSL file, any added or imported artifacts are stored directly to disk.

- When you use the DSL Editor to add project-wide dynamic instances, the DSL Explorer lists them in a folder under the DSL Project root. From there you can view and organize all dynamic instance models that you have specified in the project.
- The **Model Refactor** dialog box is automatically displayed when a selection contains at least one model file. From the toolbar, select **Window ➤ Preferences ➤ DSL Toolkit ➤ Model Refactor** and check the **Search/Update references (in file types selected below)** option. Whenever you drag and drop a resource or template of a model, the **Model Refactor** dialog box is automatically displayed. Dragging and dropping nonmodel files moves the files but does not open the **Model Refactor** dialog box.
- You can add new templates and template groups in the templates subtree. After you create a new template or template group, that template or template group is selected in the subtree.
- You can remove artifacts from the project while keeping their resources on the disk. To remove artifacts along with their resources, choose the context menu option **Delete with resource(s)**. An **Unlink from Project and Delete Resources** confirmation dialog box with a list of artifact resources is displayed, and you can select which of them you want to delete.
- You can create an Audits and Metrics model by choosing **New ➤ Audits and Metrics** from the context menu. A folder for your Audits and Metrics models is created in your DSL project. Your DSL project editor contains references of Audits and Metrics models, in addition to tools for creating Audits and Metrics models.
- Use the DSL Explorer to generate and validate your DSL project code.
- The DSL Explorer is the default view to the left of the editor in the DSL Perspective.

Workflow Editor

The Workflow Editor manages transformation workflow scripts, which are used to define model transformation executions that are invoked independently or in chained sequences. Scripts are maintained in `*.exec` model files (listed under the editor's **Workflow** and **Tree** pages) and have corresponding `*.exec_diagram` files (listed under the editor's **Diagram** page). Workflows can be defined using either or both the diagram and form-based user interfaces.

Path definitions in script elements can be project-relative, or they can be in the `platform:/resource` or `platform:/plugin` URI scheme, depending on the execution scenario requirements.

Execute scripts using generated Java code or using a generated Ant script. The scripts can also be evaluated in-place during development for testing purposes using dynamic instance models.

DSL Editor on page 44

Toolbar Actions

The following toolbar actions are available for use with the Workflow Editor:

Generate	Invokes an evaluation of the script. Contextual inputs will be satisfied by a pop-up dialog that allows for selection of workspace files. Alternatively, this can be done using the **Evaluate** action available from a script root on the **Workflow** page.
Validate	Invokes a validation of the workflow script to ensure its correctness.

Diagram Page

The diagram surface is populated with nodes and links available from the following groups of palette tools.

Input Group

The Input group of tools can be used for each of the following input types used in a workflow definition.

Resource creates a resource element with a value that can be any valid `platform:/` URI path.

OCL Query defines an input based on an OCL expression.

Context defines contextual input, with a provided default value of `Selection` that corresponds to a Practitioner's resource selection in the deployed workspace.

Chain defines an input based on the output of another transformation, especially useful in model-to-text following model-to-model transformation sequences.

Proxy defines a new input pin on an existing model-to-model transformation where multiple input models are used.

Invocations Group

The Invocation group of tools can be used for each of the following model transformation types.

QVT creates a new invocation element representing an operational QVT transformation (`*.qvto`) file, preconfigured with an input and output pin. Additional input pins can be added using the **Proxy** tool.

Xpand Template creates a new Xpand template invocation element, preconfigured with an input and output pin.

JET Template creates a new JET template invocation element, preconfigured with an input and output pin.

Composite creates a new transformation group element to hold template roots for invocation.

Note: The **Composite** tool provides a compartment in which you can add elements into the compartment if you start drawing the corresponding element near the top of the compartment. However, there exists a GMF limitation against adding elements inside non-list compartments.

Shared Elements Group

The Shared Elements group of tools can be used to define shared elements for use in input definitions.

Resource Template Root creates a new template root element for use within the context of a model-to-text template invocation.

OCL Query-based Template Root creates a model-defined template root that allows for an OCL expression language value, condition, and input context.

Composite Template Root creates a composite template root element.

Classload Context creates a classload context element with a list of bundle IDs that specify an execution context.

Output Group

The Output group of tools can be used to specify output elements for a transformation.

Plain File specifies an output file name, which can be specified using a `platform:/` URI.

OCL-based allows an output to be defined using an OCL expression.

Feature-based allows for a feature-based path declaration.

Memory defines a memory output when chaining transformations without an intermediate file.

Links Group

Used for making a connection between workflow elements.

Input links transformation input pins to **Input** elements. Dependencies are indicated by links drawn in the direction of the pin to the input.

Workflow Page

This page presents a form-based user interface that includes similar functionality as the diagram page. Tool groups on the diagram palette correspond roughly to combo boxes on the form (for example, the Input group's **OCL Query** tool on the **Diagram** page corresponds to the **OCL Query** input resource on the **Workflow** page, and the **Model Root** Shared element form on the Workflow page corresponds to the **OCL Query-based Template Root** tool on the Diagram page).

The lists in the **Workflow** page interface do allow for the evaluation of workflow scripts, validation, and ant file generation. In addition, the following functionality is available on the **Workflow** page of the editor:

Post Processors	Lists the available post processors that you can add to a transformation output.
Java Merge	Lets you specify a path to merge settings used with JMerge when generating Java code.
Java Formatter	Formats Java code following generation.
Import Organizer	Cleans up import statements in generated Java code, and shortens fully qualified Java class names in the body into import statements.

Tree Page

Displays the `*.exec` model files of your scripts in a tree view.

Related Topics

Template Explorer

The Template Explorer lets you view a set of templates by their root, including an overlay of how local templates can override or extend a configured base set of templates. This view is particularly helpful when working with a complex set of templates arranged in a hierarchy, such as EMF and GMF code generation templates. Both JET and Xpand templates are supported.

To access the Template Explorer, create a Template Collection directory within the DSL Editor's **Template Collections** tab. When you select the directory, click the **Show Templates View** link that is displayed. Alternatively, select **Window ➤ Show View ➤ Other** and click the **Template Explorer** check box under the **DSL Templates** node.

The Template Explorer has the ability to display Base Template Sets and Configured Template Contexts. Use the Properties View to display template information when selecting elements in this view.

Using the Template Explorer, you can extend templates so that generated code capabilities are improved, add dependencies to base templates, compare templates that have local overrides with their base (parent) templates, and even compare directory structures.

Note: The **Compare With** context menu option is available when templates are selected in the Template Explorer. The feature opens a two-layered display with a Structure Compare (for comparing entire folders) pane on top and an empty pane on the bottom. Double-click the item in the top pane in order to textually compare the template files in the bottom pane.

Note: The Generic Templates Browser and Dynamic Templates view that were available in earlier versions of the DSL Toolkit have been deprecated. However, you can activate these capabilities by accessing the **Window ➤ Preferences** menu item, clicking the **Capabilities** node under **General**, and then clicking the **Advanced...** button. When you expand the **Model to Text Transformation** capability, you can enable the Generic and Dynamic Template features by selecting the **Template Exploring (Legacy)** node.

Base Template Sets

Base Template Sets show the template roots for a set of configured sources that can be overridden within the context of a DSL Project. This view is visible when you select **All** from the main toolbar. Elements display a *tooltip* when the mouse hovers over them. The tooltip itself will display various options when the mouse hovers over it, including **Open in Editor**, **Close**, and **Open Properties View**.

Base Template Sets can also be searched, collapsed, or expanded using right-click context menus.

Configured Template Contexts

Configured Template Contexts display the set of template sets found in projects, with the active template shown in boldface. Use the right-click context menu on each template context to access the **Activate**, **Collapse All**, **Remove**, **Search**, **Refresh**, **Expand All**, and **Collapse All** options.

Activating a template context also causes the view to show only this root and its base overlay. Font coloring shows the difference between the local template structure and that of its base, including overrides and extension. Additionally, a **Customize this Template** menu item is available to allow the Toolsmith to conveniently import a template for modification into the DSL project. When in the contextual view model, you can also select from several overlay options from the top of the view.

Toolbar Menu Items

The Template Explorer has toolbar items that provide navigation, filtering, and configuration functions:

Import template roots from your projects	Displays project template roots for you to select in the Template Explorer. During import, template roots and base sets can be configured, as they can in the **Xpand Roots** section of the project properties dialog.
Show History List	Lets you select previously browsed template roots, view the entire history, switch between different configured contexts, or clear the history list.
Open In Editor	Opens the editor that originally configured the selected template root.
Refresh Structure	Updates template structure for nonworkspace templates (for example, the Base Template Sets from the platform plug-ins) and updates all configured templates at the same time.
Expand All...	Expands the current selection or root.
Collapse All...	Collapses the current selection or root.

All	Displays the Base Template Sets and Configured Template Context roots and their contents within the Template Explorer. The active configured template context is displayed in boldface. This option is available in the **Active** mode.
Active	Causes the view to show only the configured template context that is active. This option is available in the **All** view mode.
Back	Returns you to the previously viewed template structure.
Go Into	Focuses the view on the current selection's content only.

View Menu Items

The Template Explorer has the following view menu items:

View Preferences	Lets you modify the color settings for base templates, aspects, arounds, arounds errors, base templates calling local, and local templates.
Folder Presentation	Lets you control whether a folder has a flat or hierarchical display.

Figure Gallery Editor

Descriptions of the Figure Gallery pages.

Creating a Figure Gallery on page 28

Overview Page

Configures basic details for the Figure Gallery.

General Information

Lets you modify plug-in attributes. Typically, accept the default values.

ID sets the identifier for the plug-in.

Version specifies the version number.

Name provides a name for the Figure Gallery.

Provider specifies the provider name.

Package defines the package for the generated code.

Figure Gallery Generator

Configures generator options.

Use IMapMode By default, the DSL Toolkit uses **HiMetricMapMode** where coordinate points are 0.01 of an inch. IMapMode uses **IdentityMapMode** where coordinate points are equal to pixels.

Utilize enhanced features enables the use of enhanced merge capabilities.

Template Directory specifies custom code generation templates.

Figures Plug-in

Generate generates implementation of this figure gallery.

Validate verifies figure gallery model integrity.

Figures Page

Figures provides a tree view of the figure gallery model.

Details displays key properties for the selected element. Access full properties from the Properties view.

Elements Page

Diagram Elements provides a tree view of the figure gallery model.

Details displays key properties for the selected element. Access full properties from the Properties view.

Tree Page

This page provides a composite viewer and editor for all models involved in this figure gallery.

Related Topics

DSL Toolkit Ant Support

- *Model Transformation Support*
- *Creating Model-To-Text Transformations*
- *Creating a Model-To-Model Transformation*
- *EMF API for Together Profiles*

Ant Editor

The Ant editor provides basic editing features for Ant scripts. **Content Assist**, which you can access by pressing and holding Ctrl and spaceaccessible via ctrl+space, provides a list of proposals including DSL Toolkit tags.

Build Script Overview

The following code snippets will provide a basic tour of an Ant script used for a template sequence.

```
<project name="build.xml" default="invokeAll" basedir="."
xmlns:xpt="borland:xpand.exec.dsl/2007" xmlns:dsl="borland:exec.dsl/2007"
xmlns:fs="borland:resource.exec.dsl/2007" xmlns:qvt="borland:qvt.exec.dsl/2007"
>

 <dsl:templateset id="tset1"
url="platform:/resource/org.eclipse.dsl.mindmap/templates"/>

 <fs:input id="input1"
uri="platform:/resource/org.eclipse.dsl.mindmap/model/Map.xmi"/>

 <target name="invokeAll">
  <antcall target="invoke1"/>
  <antcall target="invoke2"/>
 </target>
```

```
 <target name="invoke1">
     <xpt:template name="mindmap2csv::CSVFile" templateset="tset1" >
      <?together generated="true"?>
   <xpt:selector refid="input1" expression="first()" id="input2"/>
         <fs:destfile
value="platform:/resource/org.eclipse.dsl.mindmap/out/mindmap.csv"/>
     </xpt:template>
 </target>


 <target name="invoke2">
     <qvt:transform
script="platform:/resource/org.eclipse.dsl.mindmap/transformations/mindmap2requirements.qvto">

      <?together generated="true"?>
   <xpt:selector refid="input1" expression="first()" id="input3"/>
         <fs:destfile
value="platform:/resource/org.eclipse.dsl.mindmap/out/mindmap.requirements"/>
     </qvt:transform>
 </target>

</project>
```

This build script defines the following items:

- a template set (tset1)
- input (input1)
- Xpand template (invoke1)
- QVT (invoke2)

xpt:template

Defines namespaces for Xpand template invocation.

Example:

```
<xpt:template name="RubyGen" xmlns:xpt="borland:xpand.exec.dsl/2007"
xmlns:fs="borland:resource.exec.dsl/2007" xmlns:dsl="borland:exec.dsl/2007"
xmlns:model="borland:model.exec.dsl/2007"
/>
```

Attribute	Value	Description
name	string	Specifies the Xpand template name.

jet:template

Defines namespaces for Java Emitter Template (JET) invocation.

Example:

```
<jet:template name="Jet2Merlin" xmlns:jet="borland:jet.exec.dsl/2007"
xmlns:xpt="borland:xpand.exec.dsl/2007" xmlns:fs="borland:resource.exec.dsl/2007"
 xmlns:dsl="borland:exec.dsl/2007" xmlns:model="borland:model.exec.dsl/2007"
/>
```

Attribute	Value	Description
name	string	Specifies the JET template name.

xpt:context

Defines the Xpand context.

Example:

```
<xpt:context id="thisContext">
 <xpt:bundle name="bundleName"/>
</xpt:context>
```

Attribute	Value	Description
id	string	Specifies the Xpand context.
xpt:bundle	string	Specifies the Xpand bundle.

xpt:selector

Defines the Xpand selector.

Example:

```
<xpt:selector id="thisSelector" refId="theRefId" expression="first()"/>
```

Attribute	Value	Description
id	string	Specifies the Xpand selector.
refId	string	Specifies the Xpand reference identifier.
expression	string	A valid query that produces the result set used as input to the template.

model:selector

Defines the model selector.

Example:

```
<model:selector id="thisSelector" refId="theRefId" type="eClass"/>
```

Attribute	Value	Description
id	string	Specifies the model selector.
refId	string	Specifies the model reference identifier.

Attribute	Value	Description
`type`	string	A valid Eclipse Modeling Framework (EMF) type.

model:selector

Defines the model selector.

Example:

```
<model:selector id="thisSelector" refId="theRefId" type="eClass"/>
```

Attribute	Value	Description
`id`	string	Specifies the model selector.
`refId`	string	Specifies the model reference identifier.
`type`	string	A valid Eclipse Modeling Framework (EMF) type. Use either the `type` or `feature` attribute but not both.
`feature`	string	A valid EMF feature. Use either the `type` or `feature` attribute but not both.

fs:input

Defines a resource input.

Example:

```
<dsl:templateset id="tplRoot" url="templates"/>
```

Attribute	Value	Description
`id`	string	Specifies the template root identifier.
`url`	string	Specifies the template root URL.

collect-stale-files

Cleans stale files.

Example:

```
<collect-stale-files id="cleanId" destLocation="/temp" log="WARNINGS"
```

```
exclude="/custom"/>
```

Attribute	Value	Description
`id`	string	Specifies the identifier.
`destLocation`	string	Specifies the location in which to clean stale files.
`log`	string	Specifies the logging level.
`exclude`	string	Specifies the locations to avoid.

tg-merge-xml

Merges `Plugin.xml`.

Example:

```
<tg-merge-xml file1="${in1}" file2="${in2}" target="${target}"/>
```

Attribute	Value	Description
`file1`	string	Specifies the first `Plugin.xml` file.
`file2`	string	Specifies the second `Plugin.xml` file.
`target`	string	Specifies the target file that contain the merged contents of `file1` and `file2`.

tg-merge-manifest

Merges `MANIFEST.MF`.

Example:

```
<tg-merge-manifest file1="${in1}" file2="${in2}" target="${target}"/>
```

Attribute	Value	Description
`file1`	string	Specifies the first manifest file.
`file2`	string	Specifies the second manifest file.
`target`	string	Specifies the target file that contain the merged contents of `file1` and `file2`.

Code Generation Ant Tasks

Use the custom Ant tasks in this topic to generate code. Generate tasks to invoke the Xpand (`borland:xpand.exec.dsl/2007:template`) and Java Emitter (`borland:jet.exec.dsl/2007:template`) templates automatically by selecting the **Generate Ant Task** action in the context menu of the `com.borland.dsl.exec.Script` element.

- *QVT Operational Developer Guide*
- *Xpand Language Guide* on page 79
- *DSL Toolkit Ant Support* on page 72
- *QVTO Ant Tasks (Together Modeling Guide)*

Xpand Template Task

An example of an Xpand template task follows:

```
<xpt:template name="Uml2Java::Model" templateset="uml2java">
<global name="INPUT" value="this"> <context refid="uml"> <xpt:destfile
expression="this.eClassifiers.name"> <xpt:selector refid="UMLPackage"
expression="this.eClassifiers">
</xpt:template>
```

Java Emitter Template Task

An example of a Java Emitter template task follows:

```
<jet:template name="rdb/Column.javajet" templateset="rdb2ddl">
<global name="INPUT" value="this"> <context refid="rdb"> <fs:destfile
value="/testProject/output.ddl"> <model:selector refid="rdbCore" type="Column"
imports="http://www.eclipse.org/qvt/1.0.0/Operational/examples/rdb">
</jet:template>
```

Attributes

The attributes for both templates are as follows.

Attribute	Value	Description
`name`	string	Specifies the template name relative to the template root (`templateset`).
`templateset`	string	Specifies the identifying reference on the root in which templates are located. This attribute is not defined.
`context`	string	Specifies the identifying reference on a classload context and indicates which bundles to use in the classpath during code generation.

Nested Elements

The nested elements for both templates are as follows.

Attribute	Value	Description
`parameter`	string	Specifies the template parameter.
`templateset`	string	Specifies the identifying reference on the root in which templates are located. This attribute is not defined.
`context`	string	Specifies the identifying reference on a classload context and indicates which bundles to use in the classpath during code generation.
`global`	string	Specifies the global variable.
`destfile`	string	Specifies the destination to which the template evaluation result is written.
`input selector`	string	Specifies the template **Input Selector**.

Related Topics

Common URIs

DSL Toolkit supports the use of ubiquitous URIs, which are URIs that are not tied to a specific Eclipse workspace location or run-time configuration.

Note: To ensure that the URI support feature is turned on, open the **bundles.info** file in the **eclipse/configuration/org.eclipse.equinox.simpleconfigurator** directory and verify that the line with the `com.borland.dsl.ubiquity` plug-in entry ends with `true`. For earlier versions of DSL Toolkit, open the **config.ini** file in the **eclipse/configuration** directory and verify that the `osgi.bundles` line has configured `com.borland.dsl.ubiquity@start`.

Advantages of Common URIs

Although DSL Toolkit supports conventional file-like URIs such as `platform:/plugin/<bundlename>/<path-to-model-file>` and `platform:/resource/<projectname>/<path-to-model-file>`, using ubiquitous URIs avoids the following constraints:

- File-like URIs do not allow you to switch model locations easily. The ability to switch becomes important when working in environments that contain many models but only a few of which are actually being modified or accessed at any given time. For example, suppose a modeler is developing two models in a workspace, Model A and Model B, and Model B references elements in Model A by using a `platform:/resource` reference. If another modeler is developing a Model C that depends on Model B—and, implicitly, Model A—any attempt to access elements from Model B requires that all dependencies of Model A are resolved and checked into the workspace before work can continue with Model C.
- Until the incorporation of ubiquitous URIs, no mechanism was available for models to access models and other artifacts from a target platform. In some environments, not all the modules that a developer uses as

artifacts are necessary in the running platform, `platform:/`, but some modules are more applicable to target platforms like `target:/` in which components for an application are developed. Target platforms that are easily configurable and updateable allow users to include in the workspace only those plug-ins that are essential to their subsystems. You can access other dependencies, such as the entire product itself or other needed subsystems, through the `platform:/` mechanism. This way, you can easily access updates and install them as new targets.

Supported URI Syntaxes

In addition to `platform:/plugin/` and `platform:/resource/` schemes, the following ubiquitous URI syntaxes are supported in DSL Toolkit. In these schemes, `<bundle-id>` refers to the Bundle-SymbolicName entry in the bundle or plug-in manifest file (`MANIFEST.MF`), and `<token>` is the identifier for a model that maps to a file within the specified bundle by using a `models.tg` file.

`target:/plugin/<bundle-ID>/<path-to-model-file>`

Allows users to reference a plug-in with model files eithe
can then be saved as JAR files like the platform's other p

`model:/plugin/<bundle-ID>/<token>`

Looks up bundle or plug-in first in workspace, then in tar
through a `../../../` type directory structure, uses the
`META-INF/` directory of a plug-in and is deployed with the
`relative-path-to-model-file>`, as in the following

`ecore-baseline-version=/model/execute.ecor`

With this syntax, tokens provide more information for the
model were deployed in a `/model` folder in a bundle nar
must contain the `framework=model/framework.uml`

A path within a model can also be specified, following th
specify the model as either of the following paths.`model`
`model:/plugin/com.borland.framework/framew`

`model:<token>@<bundle-ID>`

Similar to the `model:/plugin/<bundle-ID>/<toke`
`model:latest-domain@com.borland.dsl.exec` w
the workspace, that model is used. If it is not in the works
running Eclipse instance.

Xpand Language Guide

Introduction

Xpand is the template language that the DSL Toolkit uses to generate textual output from models. This section provides an overview of the syntax and semantics of the language.

Xpand instructions are surrounded by guillemet (« and ») characters. The Xpand editor provides Content Assist entries to aid in writing Xpand templates. Press the Content Assist key combination (`CTRL+SPACEBAR` by default) to display the window.

Note: Mac users must set their workspace encoding settings to ISO-8859-1 when working with Xpand templates in the editor. Open Eclipse **Preferences**. Expand **General** and then **Workspace**. Select **Other** under **Text file encoding**. Choose "ISO-8859-1" from the list. Click **OK**.

Xpand files must contain only letters, numbers and underscores.

Simple Xpand template

The following example template illustrates the basic structure of the language.

```
«IMPORT "http://org.example/ant/2007"»

«DEFINE buildxml FOR BuildScript»
<?xml version="1.0"?>
<project name="«name»" default="«defaultTarget»" basedir="«basedir»">
«ENDDEFINE»
```

Syntax

COMMENTS

Comment tags, legal only outside of other tags. Comments can span multiple lines.

```
«REM»commentary«ENDREM»
```

DEFINE

A tag, essentially a method, for an element. It includes a name, parameter list, and the name of the domain model element.

```
«IMPORT "http://www.example.org/2007/ant"»
«DEFINE buildxml FOR BuildScript»
«EXPAND genProject FOR this.project»
«ENDDEFINE»
«DEFINE genProject FOR Project»
<?xml version="1.0"?>
<project name="«this.name»" default="." basedir="«this.basedir»">
</project>
«ENDDEFINE»
```

EXPAND

This statement expands another `DEFINE` method in the current context and redirects its output at the current location.

```
«IMPORT "http://www.example.org/2007/ant"»
«DEFINE buildxml FOR BuildScript»
«EXPAND genProject FOR this.project»
«ENDDEFINE»
«DEFINE genProject FOR Project»
<?xml version="1.0"?>
<project name="«this.name»" default="." basedir="«this.basedir»">
</project>
«ENDDEFINE»
```

EXTENSION

Declares an import for an Extend file.

```
«EXTENSION example::ExtenderClass»
```

FILE

The `FILE` directive is not supported. Instead you can configure file output as part of a transformation sequence or a UI contribution.

FOR

Applies the designated Expand method against the result of the expression.

```
«EXPAND genProject FOR this.project»
```

FOREACH

Applies the designated Expand method against each element in the collection.

```
«EXPAND myDef FOREACH entity.allAttributes»
```

Alternatively, you can use `FOREACH` to iterate inplace.

```
«FOREACH targets AS t» <target name="«t.name»"> </target>
«ENDFOREACH»
```

IF

Defines conditional expansion statements. `ELSEIF` and `ELSE` are optional.

```
«IF private» private «ELSEIF public» public «ELSE» protected «ENDIF»
```

IMPORT

Imports a namespace.

```
«IMPORT "http://www.borland.com/2007/mindmap"»
```

LET

Defines a local variable for use with the expression.

```
«LET "<target name=\"" + t.name + "\"/>" AS tag» «tag» «ENDLET»
```

Expressions language guide

For more information about the expressions language, refer to *http://www.eclipse.org/gmt/oaw/doc/4.1/r10_expressionsReference.pdf*.

Related Topics

Domain-Specific Language Preferences

Note: A single workspace-wide registry of URI mappings, which is stored in the workspace preferences, can be used to resolve any cross-model references. You can use the project preferences to override the default URI map for each project.

DSL Toolkit

Save all modified resources automatically prior to code generation Enable this setting to save modified files and avoid warning messages when generating code.

Model Refactor

These settings assist with maintaining consistent URI references among DSL models during refactoring operations.

	Participate in workspace resource refactorings enables model-refactoring features. **Add history script as project file artifact** enables generation of a file to track refactoring history. Adjust the file extension as desired. **Search/Update references** enables file types for management during refactorings. Use the **Add** and **Remove** buttons to modify the list of file types. **Search for references from runtime platform** Expands detection to references from installed plug-ins. Click **Configure** to select which plug-ins to include.
Project Editor	**Show model URIs in DSL Explorer view** Adds URI to model nodes in the **DSL Explorer**. **Default version for new projects** Set this value to configure the version number for new DSL projects. **Default prefix for base package** Configure this option for your organization's base package, which will automatically appear in new DSL artifacts plug-in configuration. **Default Branding Provider for new projects** Set this value for new DSL projects. For example, `Borland Inc.`.
Report Definition Editor	**Open the associated perspective when generating a report** Select a value of **Always**, **Never** or **Prompt** to adjust behavior when working with reports. **Overwrite user changes in .rptdesign when generating a report Select a value [Always, Never or Prompt]** to adjust behavior when generating reports.
URI Mappings	Maps logical URIs and physical URIs. You can add, clone, remove, edit, move, import, or export entries.

Related Topics

DSL Explorer View on page 66

DSL Toolkit Activities

The DSL Toolkit includes traditional open-source Eclipse features as well as activities configured specially for DSL Toolkit that extend upon those open-source features.

The activities configured specially for DSL Toolkit are enabled by default. Turn all of these features on or off by selecting **Window ➤ Preferences... ➤ General ➤ Capabilities** and then clicking **Advanced**. The different features are listed as nodes under the **DSL Development** feature.

Tip: The capabilities configured specifically for DSL Toolkit and enabled by default provide the most useful implementation of domain-specific language and model-driven development. As a best practice, avoid enabling the open-source features, such as **Eclipse Graphical Modeling Framework** and **Eclipse Modeling Framework** listed under the **Development** capabilities group, unless first disabling those features enabled by default. Enabling all the capabilities at the same time can lead to confusion with cluttered menu items and wizards of the UI.

Diagram Definition	Capabilities associated with developing a diagram definition for a domain model.

Domain Modeling	Capabilities associated with developing a domain-specific model.
DSL Project	Support for DSL-specific project structures.

Related Topics

DSL Explorer View on page 66

Domain Diagram Preferences

The following table describes the global settings and saving options for domain diagrams.

Global settings	**Enable animated layout** – Enhances the perception of a diagram's spacial movement by animating the layout. **Enable animated zoom** – Enhances the perception of a diagram's spacial movement by animating zoom level changes. **Enable anti-aliasing** – Minimizes the distortion of high-level resolution signals at a lower resolution. **Draw diagram nodes shadows** – Creates a shadow effect beneath diagram nodes. **Draw classes with gradient** – Creates labels with gradient background filling for class nodes. **Show name of abstract classes as italic** – Displays abstract class names in italics. If this option is not selected, class names can still be in italics if the corresponding class node has an option for italics. **Show operation parameters** – Enables all operation parameters, such as `name:type`, to be visible on a diagram.
Saving options	**Validate on save** – Validates the domain model and EMF generator model automatically when saving. This option can slow down diagram editor performance in larger models. **Generate on save** – Generates relevant code automatically when saving. This option can slow down diagram editor performance in larger models. **Automatically save diagram** – Saves the domain editor automatically after each modification.
Code generation options	**Delete stale files after code generation** – Automatically removes leftover artifacts that are no longer updated with each regeneration.

Related Topics

DSL Model Overview

Descriptions of the DSL models.

DSL model

At the heart of any DSL project is the DSL model. This model is basically a wrapper for all content that the DSL manages. It contains basic DSL information as well as references to domain models, diagrams, templates, transformations, reports and transformation sequences.

Domain model

The domain model typically captures the concepts and vocabulary of the problem domain. It not only provides the structure for the input models but also contributes to the generation of the DSL implementation.

Diagram definition model

The diagram definition model references the GMFMAP and GMFGEN models and provides options for transforming them. It lets you specify which figure galleries to use. A single diagram definition can contain references to multiple figure galleries. By pointing to the target GMFGEN model, the diagram definition model lets you tweak that model's advanced options. Multiple diagrams or views can reside within a single DSL.

Graphical definition model (GMFGRAPH)

The graphical definition model, `gmfgraph.ecore`, comes from the GMF project. Its constructs closely resemble those of the Graphical Editor Framework (GEF). It is advisable to become familiar with those concepts for success in using the DSL Toolkit.

The graphical definition model has its own code-generation templates, which are invoked either during the creation/recreation of the GMF generator model or the generation of a stand-alone figures plug-in. Some models contain figure definitions for class and state machine diagram elements in the `org.eclipse.gmf.graphdef` plug-in. Other figures are defined as part of the `org.eclipse.uml2.diagram.def` UML2 diagram definition plug-in.

The root element of a graphical definition model is the *canvas*. It contains references to figure galleries, nodes, connections, compartments, and labels. Note the distinction between figures and diagram elements. In a GMF graphical definition, a figure is defined within the Figure Gallery, which is then referenced by **Node**, **Connections**, **Compartment**, and **Label** diagram elements. These diagram elements are siblings to the Figure Gallery elements. Because they can reference figure definitions from other graphical definition models, you can reuse figure definitions across DSLs.

A Figure Gallery contains figures, figure descriptors and an optional implementation bundle. The implementation bundle is used only when generating standalone figure plug-ins. This property is not intended to be used by the Toolsmith directly.

Figures are typically generated as subclasses of `org.eclipse.draw2d.Figure`. It is also possible to define a figure by using an Scalable Vector Graphics (SVG) file.

Tooling definition model (GMFTOOL)

The tooling model is one of the simplest GMF models. It is primarily used to define items on a diagram's palette. This model's tools create nodes and links.

The palette can contain icon images, palette separators, tool groups, and creation tools. A tool group logically organizes creation-tool entries, such as nodes or links. A separator can be added between tools and results in a horizontal line on the palette. A creation tool has image child elements, which are large and small icon bundles that can provide either the default EMF icons or can use provided icon graphics files.

Mapping definition model (GMFMAP)

The mapping model transforms the graphical and tooling models into one or more generator models that drive templates for code generation. The quality of the mapping model depends on the quality of the input models. As a result, Toolsmiths must use the validation facilities provided for each of the input models.

The canvas-mapping element is required and represents the diagram canvas. You can create the top node references on the diagram canvas. The domain model must contain the elements. The containment feature defines where these objects are added.

Each top node reference contains a single node mapping. This mapping binds together a diagram node, tool, and domain model element. The diagram node is from the graphical definition model. The tool comes from the tooling definition model, and the element itself is from the domain model.

Generator model

The GMF generator model is the largest model used in GMF and the one most likely to be extended to provide customizations, likely using a decorator model. Much of the model does not need to be covered in detail because most elements are prefixed with `Gen` from the input model.

A trace facility reconciles changes made to the generator model when retransforming from the mapping model. It is recommended that you use this feature if you plan to make changes or augment the generator model in any way.

The generator model has references to the EMF domain and generator models for use in generating the diagram code. The figures used in the diagram are either serialized into fields in the generator model during the transformation from GMFMAP to GMFGEN or are referenced by class name from their corresponding generated figure plug-in.

Related Topics

DSL Artifacts on page 85

DSL Artifacts

Name	Type	Description
`.dsl`	model	DSL definition
`.ecore`	model	EMF metamodel
`.genmodel`	model	EMF generator model
`.domain`	model	domain model
`.exec`	model	workflow script model
`.domain_diagram`	diagram	domain diagram
`.exec_diagram`	diagram	workflow diagram
`.diagram`	diagrams model	DSL diagram definition
`.gmfgen`	diagrams model	GMF generator model

Name	Type	Description
`.gmfgraph`	diagrams model	GMF graphical model
`.gmfmap`	diagrams model	GMF mapping model
`.gmftool`	diagrams model	GMF tooling definition
`.qvto`	transformations	model-to-model transformation
`.tnt`	model	Textual Notation Configuration
`.tmfgen`	model	Textual Notation Generator Model
`.xpt`	templates	Xpand template
`.dsldesign`	reports	report design

Related Topics

DSL Model Overview on page 83
DSL-Generated Artifacts on page 86

DSL-Generated Artifacts

Name	Type	Description
DSL project	project	main project for DSL
`project.model`	generated project	model implementation
`project.model.diagram`	generated project	diagram implementation
`project.edit`	generated project	model edit implementation
`project.editor`	generated project	model editor implementation
`project.text`	generated project	text editor implementation

Name	Type	Description
`project.ui`	generated project	UI contributions implementation
`project-feature`	generated project	DSL feature plug-in
`.qvtotrace`	transformations	used internally for model-to-model transformation
`.trace`	diagrams model	used internally for regenerating genmodel
`.tnt`	textual notation	textual representation of the domain model
`.xmi`	XMI	mapping

Related Topics

DSL Model Overview on page 83

DSL Artifacts on page 85

Stale Files Tasks

Syntax:
It is important to keep track of stale files during project generation. <varProduct/> enables you to collect a list of all generated files prior to running a code generation. This way, you can note which files have been regenerated and which files have been unrevised. Then you can remove specific files you do not want or all of them.

Selectively deleting stale files avoids having to completely delete the existing project prior to regeneration (which has been the regular approach to ensure cleanly generated code). However, deletion of an entire project risks the erroneous removal of essential hand-written code.

Collect Stale Files

By using the `collect-stale-files` Ant script, stale files are collected from the `destlocation` folder that was created prior to running this task. These files can subsequently be deleted by running `clean-stale-files`. For an example of this task and its associated attributes, refer to *DSL Toolkit Ant Support*.

Report Stale Files

By using the `report-stale-files` Ant script, you can print the names of the files that were collected by the `collect-stale-files` task.

The attributes for this task are as follows:

Attribute	Value	Description
`refid`	String	Specifies the reference ID of the `collect-stale-files` task.
`exclude`	String	A pattern that specifies which files are to be excluded from the list of stale files. These files will not be reported. The pattern may contain '?' and '*' symbols.

Delete Stale Files

By using the `clean-stale-files` Ant script, you can delete the files that were collected by the `collect-stale-files` task.

The attributes for this task are as follows:

Attribute	Value	Description
`refid`	String	Specifies the reference ID of the `collect-stale-files` task.
`exclude`	String	A pattern that specifies which files are to be excluded from the list of stale files. These files will not be deleted. The pattern may contain '?' and '*' symbols.

Alternatively, you can select specific files among the stale files listed in the Deployment task of the DSL Editor and click the **Remove** button, or click **Remove All** to remove them all.

Descriptor Files Merging Tasks

Syntax:
You can use the following custom Ant tasks to merge Eclipse descriptor files.

Merge Manifest Task

By using the `tg-merge-manifest` Ant script, you can merge manifest.mf files by directing generated output to a file next to manifest.mf (if one exists). For an example of this task and its associated attributes, refer to *DSL Toolkit Ant Support*.

Merge Plug-in Descriptor Task

By using the `tg-merge-xml` Ant script, you can merge generated plugin.xml code for a diagram plug-in so that values you input are not reset to default values when you regenerate the diagram. For an example of this task and its associated attributes, refer to *DSL Toolkit Ant Support*.

Related Topics

Domain-Specific Language Glossary

This section provides a description of domain-specific language terminology.

Diagram Definition

A model, also referred to as a *http://www.computer-dictionary-online.org/index.asp?q=concrete%20syntax*, that references a number of other models used to define a diagram for a domain-specific language. This model's filename extension is `.diagram` and is located by default in the `/diagrams` directory of a DSL project.

Domain Model

The central model of any DSL, which is sometimes referred to as an *http://www.computer-dictionary-online.org/index.asp?q=abstract%20syntax* or a metamodel. It defines the elements, attributes and relationships of the domain. It also captures the domain vocabulary in the naming of model members and can ensure data integrity through constraints. The model contains references to the `.ecore` and `.genmodel` models. This model's filename extension is `.domain` and is located by default in the `/model` directory of a DSL project.

DSL

A Domain-Specific Language or DSL is a language designed for a particular and singular domain.

DSL Toolkit

The DSL Toolkit takes a model-centric approach to DSLs. It extends Eclipse and several key Eclipse projects, such as EMF and GMF, to assist the Toolsmith in creating DSLs and artifacts used in model-driven development scenarios.

Dynamic Instance Model

An XMI file that aids in developing a DSL because it can contain sample model data for use when developing reports, templates and transformations without the need to generate code or launch a runtime workbench.

EMF

Eclipse Modeling Framework is a modeling framework and code-generation facility for building tools and other applications based on a structured data model. For more information about EMF, see *http://www.eclipse.org/modeling/emf/*.

Figure

A figure is the term used in GEF terminology for graphical elements on a DSL diagram.

Figure Gallery

A model that defines graphical elements for use in DSL diagrams. Figure galleries can be reused across multiple diagrams.

GEF

Eclipse Graphical Editing Framework allows developers to create a rich graphical editor from an existing application model. For more information about GEF, see *http://www.eclipse.org/gef/*.

GMF

Eclipse Graphical Modeling Framework provides a generative component and run-time infrastructure for developing graphical editors based on EMF and GEF. For more information about GMF, see *http://www.eclipse.org/gmf/*.

OCL

Object Constraint Language is a declarative language used for describing model rules and queries. The specification is available at *http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL*.

Practitioner

The role in a software development team that uses the DSL created by the Toolsmith.

QVT

A model transformation language, Query / Views / Transformations (QVT) is language designed to facilitate model-to-model transformations. It is an *http://www.omg.org/spec/QVT/1.0/* based heavily on OCL constructs. The DSL Toolkit, supports operational QVT.

RCP

Eclipse Rich Client Platform is a platform for deploying GUI applications. It includes an extensible component framework, which provides the ability to deploy to a variety of desktop operating systems.

Report

Reports can be included as part of a DSL to provide documentation generation for domain models and diagrams.

Template

A template provides model to text transformation. In the DSL Toolkit, Xpand and JET are used for generating textual content from models.

Toolsmith

The member of a software development team who creates and extends software tools. The Toolsmith often creates DSLs and provides tool configurations, customizations and extensions.

Transformation

A transformation is the generation of one software artifact from another. Typical tranformations include model-to-model or model-to-text.

Workflow

A series of transformations, both model-to-model and model-to-text, defined for execution in a script. Workflow models are maintained in `*.exec` model files and have corresponding `*.exec_diagram` files.

Xpand

This template language is used to generate textual content from models.

Related Topics

Domain-Specific Languages on page 9

GMF Programmer's Guide

DSL Editor on page 44

Index