Using GMF and M2M for Model-Driven Development

Tatiana Fesenko, Borland Software
Radomil Dvorak, Borland Software
Bernd Kolb
Markus Voelter
GMF Overview

“The Eclipse Graphical Modeling Framework (GMF) provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF.”

Runtime
• Binds EMF & GEF
• Notation metamodel
• Designed for extensibility

Generation (tooling)
• Models used to define graphics, tooling, mapping to domain
• Code generation targets runtime
• Promotes use of Domain-Specific Languages
GMF-Generated Diagram Editor
GMF Generation overview
Develop Tooling Definition

- Palette
- Tool Groups
- Creation Tool
Develop Graphical Definition

- Nodes
- Connection
- Compartment
- Labels
Mapping Definition

- Audit & metric definitions
- Domain constraints

Mapping model

Domain model

Graphical definition

Tooling definition

● Audit & metric definitions
● Domain constraints

Using GMF and M2M for Model-driven-development | © 2008 by Borland Software Corporation; made available under the EPL v1.0
Develop Mapping Definition
Create generator model

Mapping model -> «Create Generator Model...» -> Generator Model

Runtime options:
- Print support
- Validation support
- Diagram persistence

Code generation parameters:
- Plug-in provider name
- Plug-in ID
- Package namespace
Generate diagram plug-in and run diagram
GMF Generation is iterative

- Create GMF Project
  - *ecore
  - Domain Model(s)
- Develop Graphical Definition
  - *.gmfgraph
- Develop Tooling Definition
  - *.gmftool
- Develop Mapping Definition
  - *.gmfmap
- Adjust Generation Parameters
- Generate Diagram Plug-in
  - *.diagram
- Create Generator Model
  - *.gmfgen

The process is iterative, suggested by the cycle symbol (○).
Tooling Definition: Beautiful icons

• Set Beautiful icons
  - Delete default image
  - Create icon image

• Add 'Children' tool group
  - Create EntryAction
  - Create ExitAction
Graphical Definition: Intelligent figures

- Turn StartStateFigure into Ellipse, its background is black.
- Turn StopState figure into the Ellipse containing inner Ellipse.
- Create Figure Descriptor for StartStateName, create Label inside it.
- Create DiagramLabel referencing it. The label became external.
- Repeat with StopStateName.
- Set Arrow decoration for Transition
  - Create PolylineDecoration 'ConnectorTarget' in Figure Gallery
    Set TemplatePoints (-2,-1), (0,0), (-2,1).
  - Select Polyline Connection TransitionFigure inside Figure Descriptor TransitionFigure. Set choose 'ConnectorTarget' for TargetDecoration property.
Graphical Definition: SimpleState

- Turn SimpleStateFigure into rounded rectangle
- Add rectangles for Entry and Exit Action compartments inside SimpleAction.
- Create compartments for Entry and Exit Actions. Choose SimpleStateFigure in 'Figure' property, reference them to just created rectangles in 'Accessor' property.
Mapping Definition: Feature Initializers and OCL Constraints

- Automatically set default name for elements
  - Repeat with Start/Stop States

- Event-triggered transition
  - Create additional Link Mapping for Transition.
  - Set Constraints in order to distinguish links.
  - Create java FeatureValueInitializer. We will implement it to create and reference Event automatically.
Mapping Definition: Simple State

- Entry/Exit Actions compartment Mapping
- Distinguish Entry and Exit actions

- Set View Pattern 'ENTRY/{0}' for the MessageFormat parser of EntryAction
- Set View Pattern 'EXIT/{0}' for the MessageFormat parser of ExitAction
Generator model: Code generation parameters

- Make Entry EditPart class names intelligible
  - Rename LabelEditPart to EntryActionEditPart
- Repeat with Labels2EditPart (EntryActionEditPart)
Generated plugin: Code modification

- Generated code can be changed to implement domain-specific requirements

- Changed code is marked with 'generated NOT' tag.

  - Modify `EntryActionCreateCommand` and `ExitActionCreateCommand` in order to create them inside StateMachine and be referenced by SimpleState

  - Implement java `FeatureValueInitializer` for `EventTriggeredTransition` in `ElementInitializers` class.

```java
/*
 * @generated NOT
 */

private static Event createTriggeredByEvent(Transition self) {
  State targetState = self.getTargetState();
  if (targetState == null) {
    return null;
  }
  Event event = StateFactory.eINSTANCE.createEvent();
  targetState.getStateMachine().getEvents().add(event);
  return event;
}
```
StateMachine Diagram: Discover GMF Runtime Features

- Tool palette and overview
- Layout and selection tools
- Diagram image export (svg, bmp, jpeg, gif)
- Tabbed properties view
- Font and color options for selected element
- Link routing and style options
- Animated zoom and layout
- Diagram printing
Validation
Validation: Define rules

- Validation Rules are written in OCL
- They defined in GMF Map model.
  - State has 'States should have unique names' validation rule
  - Rules for source and target of a Transition
    - 'StopState cannot be a source for a transition' for StopState
    - 'StartState cannot be a target for a transition' for Transition
Enable and Run Validation

• Make validation enabled in GMFGen model:
  • 'Validation Enabled' of GenDiagram is 'true'
  • 'Validation Decorators' is 'true'
  • 'Validation Provider Priority' is 'Medium'

• Validation runs on diagram action:
  • Call 'Validate' action from the 'Diagram' tool menu.
Summary

• We created GMF-generated diagram editor for the StateMachine model.
  ✷ It was quick and easy

• Using GMF is an iterative process.
  ✷ We can modify selected tooling models and enjoy improvements in regenerated diagram

• Code GMF produces can be customized.
  ✷ We modified the generated code
Using GMF and M2M for Model-driven development

Thank you!

Questions?

http://www.eclipse.org/gmf