Eclipse Workbench Part Interaction FSML

Technical Report 2006-09
draft version 0.2
last update July 26, 2006

Michał Antkiewicz and Krzysztof Czarnecki
Generative Software Development Lab
University of Waterloo
`http://gp.uwaterloo.ca`
`{mantkiew,kczarnec}@swen.uwaterloo.ca`

Abstract

In this technical report we present the details of the Eclipse Workbench Part Interaction (WPI) FSML design and its prototype implementation. We describe the WPI domain, abstract syntax, mapping of the abstract syntax to the framework completion code, and agile round-trip engineering. Finally, we describe technical details of the WPI FSML prototype.

1 Introduction

Object-oriented frameworks are one of the most effective and widely used software reuse technologies today. Unfortunately, using a framework can be challenging. Application programmers need to know what framework classes to subclass, what interfaces to implement, and what methods to override or call in order to get the desired effect. Furthermore, they need to be able to see how the framework-provided concepts are instantiated and configured in the application code. The latter is challenging since some concepts, such as collaborations among objects, are usually scattered in the application code.

Framework-Specific Modeling Languages [1] address the problems related to framework instantiation and round-trip engineering. Models created using a FSML can express the design of applications based on the framework supported by the FSML. The round-trip engineering support of an FSML enables the automatic creation of application code from models and vice versa and keeping the models and code synchronized throughout the application development.

In this technical report we describe the design and prototype implementation of the Eclipse Workbench Part Interaction (WPI) FSML.

2 The Domain: Eclipse Workbench Part Interaction

Eclipse [2] is a universal, open-source platform for building and integrating tools, which is implemented as a set of Java-based object-oriented frameworks. Eclipse *Workbench* is the working area of an Eclipse user [3]. In Eclipse, tools are implemented as *plug-ins* and implementing a new tool is equivalent to implementing a set of Eclipse plug-ins. Each plug-in can contribute its functionality to the Eclipse Workbench by extending provided *extension points*. Plug-in contributions are declared in a special configuration file called *plugin.xml*.

In this technical report, we consider a particular part of the Eclipse Application Programming Interface (API), which is concerned with *workbench parts* and their interactions.

Workbench parts include *editors* and *views*. An editor is used for creating and modifying resources, referred to as *input resources*. Many different editors can be opened in the workbench simultaneously, but at most one can be *active* at any given time (i.e., have the focus). Furthermore, multiple instances of the same editor must have different input resources. An example of an editor is the Java editor included in the Eclipse Java Development Tools (JDT) [4]. Editors must implement the `IEditorPart` interface and can be contributed to the workbench by extending the *org.eclipse.ui.editors* extension point. Views are also used for presenting and editing information, but unlike editors, views are not associated with any particular input resource. Examples of standard Eclipse views are *Content Outline* and *Property Sheet* views. The Content Outline view is used to display an outline of a document opened in an active editor. The Property Sheet view is used to display the properties of the current selection and modify their values. Views must implement the `IViewPart` interface and can be contributed to the workbench by extending the *org.eclipse.ui.views* extension point.

Other elements of the workbench such as tool bars, menu bar, status line and perspectives are outside the scope of our example.

Workbench parts interact in various ways [5], and part interaction is the main area of concern of our example.

For example the Content Outline view listens to *part activation* events. When an editor, such as Java editor, is activated, the view asks the editor for a `IContentOutlinePage` adapter, which is used to display outline of the input resource of the activated editor. Furthermore, the Property Sheet view *listens to selections* and displays the properties of the element selected by the user anywhere in the workbench.

The Eclipse platform provides support for implementing various kinds of workbench part interactions. The Workbench runs many services, among which *Page Selection Service* and *Part Service* support part interactions. Page Selection Service allows workbench parts to register as selection listeners and receive *selection changed events*. Any part wanting to receive selection changed events must implement `ISelectionListener` interface and provide implementation for `selectionChanged(IWorkbenchPart, ISelection)` method, which will be called by selection provider (in this case Page Selection Service). Additionally, the part has to register with the Page Selection Service, by calling `ISelectionService.addSelectionListener(ISelectionListener)` method and deregister by calling `ISelectionService.removeSelectionListener(ISelectionListener)` upon its disposal.

Analogously, parts wanting to receive part events have to implement `IPartListener` interface, provide implementation for the interface methods such as `partActivated(IWorkbenchPart)` or `partClosed(IWorkbenchPart)`, and register and deregister with Part Service by calling `IPartService.addPartListener(IPartListener)` and `IPartService.removePartListener(IPartListener)` methods respectively.

The registrations are performed through a *workbench page*, which can be obtained by the part through its *part site* by calling `IWorbenchPart.getSite().getPage()`. Part site is an intermediate layer between a part and the workbench.

A part may also be the source of selection events for Page Selection Service, in which case the part has to implement the `ISelectionProvider` interface and register itself as a selection provider by calling `PartSite.setSelectionProvider(ISelectionProvider)` method.

Eclipse platform also provides support for interactions not involving any services. In our example we consider *listens to selection changed* and *requires adapter* interactions. Listens to selection changed interaction is an underlying mechanism used by Page Selection Service and can be implemented between any parts. In the latter case, a part being a selection provider is responsible for maintaining references to parts that registered as selection listeners. The difference is that selection listeners have to implement `ISelectionChangedListener` interface and provide implementation for `selectionChanged(SelectionChangedEvent)` method[1]. A part being a selection changed listener has to register directly with the provider.

Requires adapter interaction occurs between two parts when one part asks for an adapter of certain kind by calling `IAdaptable.getAdapter(Class)` and the other part provides the requested adapter. Every workbench part implements `IAdaptable` interface and therefore can take part in this type of interaction.

3 Definitions of framework-provided concepts

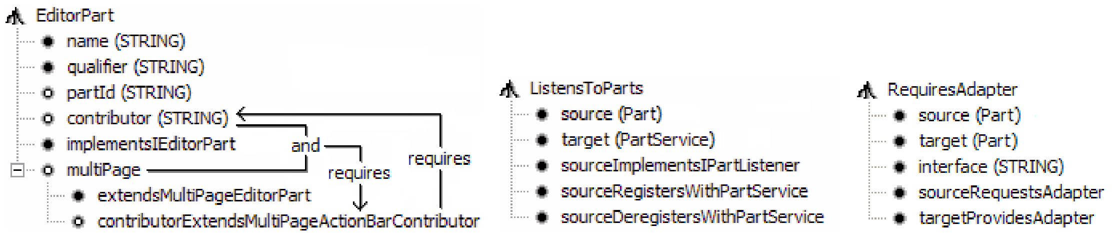
Framework-based application development involves writing *framework completion code*. Framework completion code implements the difference in functionality between the framework and the desired application.

[1]Note that this is different than `ISelectionListener.selectionChanged(IWorkbenchPart, ISelection)`.

The API of a framework provides a set of *domain concepts* for creating applications within a particular domain. Domain concepts for our example include *editor*, *view*, *listens to parts*, and *requires adapter*. Writing framework completion code amounts to instantiating and configuring domain concepts provided by the framework.

The framework also provides a set of implementation choices for each concept. For example, concept *editor* is instantiated by creating a class which implements the `IEditorPart` interface. Optionally, an editor can be contributed to the workbench and have a part id, have a contributor that contributes actions to menus and toolbars, or be a multi-page editor. For an editor to be a multi-page editor, it needs to extend framework-provided `MultiPageEditorPart` class and, if it has a contributor, the contributor has to extend another framework-provided class, namely `MultiPageActionBarContributor`.

The set of framework-stipulated implementation choices for a concept and the dependencies among these choices define all correct ways in which the concept can be instantiated as foreseen by the framework design. We can think of the implementation choices as *features* of a concept and formalize the concept's definition as a *feature model*. A feature model is a tree with the concept as its root and children representing its features [6]. Filled circles denote mandatory features and open circles denote optional features. A feature may have an attribute, which is denoted by its type shown in parenthesis. Additional dependencies between features can be expressed as constraints, such as *requires* or *excludes*. Conceptually, a feature model describes a set of all valid configurations (selections) of features.



(a) *editor* concept (b) *listens to parts* concept (c) *requires adapter* concept

Figure 1. Feature models describing some of the framework-provided concepts

For example, Fig. 1(a) shows a feature model describing the *editor* concept. Mandatory features have to be implemented by every instance of a concept, for example every editor has to have the `implementsIEditorPart` feature, meaning it has to implement the `IEditorPart` interface. Optional features, such as `multiPage`, are not required for every instance of a concept. Two additional constraints $contributor \wedge multiPage \Rightarrow contributorExtendsMultiPageActionBarContributor$ and $contributorExtendsMultiPageActionBarContributor \Rightarrow contributor$ are also represented on the figure.

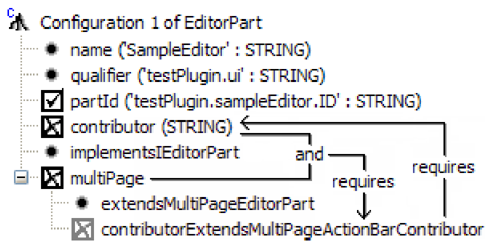
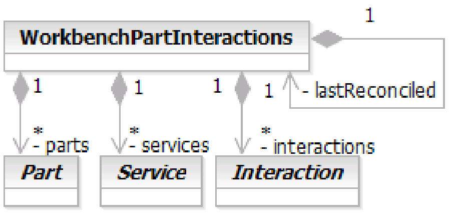

Configuration 1 of EditorPart
name ('SampleEditor' : STRING)
qualifier ('testPlugin.ui' : STRING)
partId ('testPlugin.sampleEditor.ID' : STRING)
contributor (STRING)
implementsIEditorPart
multiPage
extendsMultiPageEditorPart
contributorExtendsMultiPageActionBarContributor
and
requires
requires

Figure 2. Sample configuration of an editor concept instance

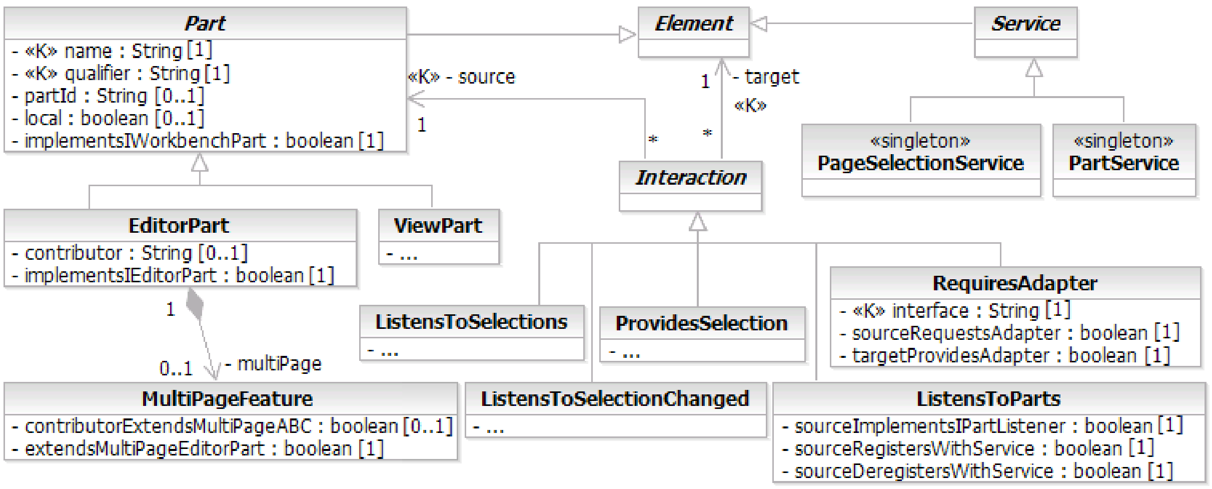
Fig. 2 presents a sample feature configuration for the instance of the *editor* concept, where some features have been selected (`partId`) and some eliminated (e.g., `multiPage`) and values of attributes have been specified (e.g., `'SampleEditor'` for `name`). The feature configuration satisfies all constraints implied by the feature model and, therefore, the implementation choices corresponding to the selected features (including the mandatory ones) are consistent. Note that recognizing the implementation of features of a given concept in the code also produces a configuration, which can then be checked for possible constraint violations.



(a) WPI Overview

Interaction	Target
`ProvidesSelection`	`PageSelectionService`
`ListensToSelections`	`PageSelectionService`
	`Part`
`ListensToParts`	`PartService`
`ListensToSelectionChanged`	`Part`
`RequiresAdapter`	`Part`

(b) Valid Interaction Targets


Part
- «K» name : String [1]
- «K» qualifier : String [1]
- partId : String [0..1]
- local : boolean [0..1]
- implementsIWorkbenchPart : boolean [1]
Element
Service
«K» - source
1
1
- target
«K»
*
*
Interaction
«singleton»
PageSelectionService
«singleton»
PartService
EditorPart
- contributor : String [0..1]
- implementsIEditorPart : boolean [1]
ViewPart
- ...
1
0..1
- multiPage
ListensToSelections
- ...
ProvidesSelection
- ...
RequiresAdapter
- «K» interface : String [1]
- sourceRequestsAdapter : boolean [1]
- targetProvidesAdapter : boolean [1]
MultiPageFeature
- contributorExtendsMultiPageABC : boolean [0..1]
- extendsMultiPageEditorPart : boolean [1]
ListensToSelectionChanged
- ...
ListensToParts
- sourceImplementsIPartListener : boolean [1]
- sourceRegistersWithService : boolean [1]
- sourceDeregistersWithService : boolean [1]

(c) WPI Concepts

Figure 3. WPI FSML Metamodel and Valid Interaction Targets

4 Eclipse Workbench Part Interaction (WPI) FSML

WPI FSML is a modeling language for modeling Eclipse workbench part interactions described in section 2.

4.1 Abstract syntax

Figure 3 presents the metamodel of WPI FSML. Class `WorkbenchPartInteractions` from Fig. 3(a) represents the whole model and contains parts, services and interactions. Figure 3(c) shows the details of the metamodel. It contains four abstract classes: `Part`, `Element`, `Interaction`, and `Service`. An `Interaction` can occur between a `Part` and an `Element`. Kinds of concrete workbench parts are `EditorPart` and `ViewPart`. Class `Service` is used to represent standard Eclipse Workbench services and has two concrete, singleton subclasses `PageSelectionService` and `PartService`.

WPI supports five different kinds of part interactions described in section 2. The interactions are represented using concrete subclasses of the `Interaction` class. Abstract syntax class diagram from Fig. 3(c) is supplemented with a set of constraints restricting possible interactions to occur only between valid elements as described in Table 3(b). Note, that interaction `ListensToSelections` occurs in two variants: *global* and *specific*. A part is a global selection listener if it registers with Page Selection Service using `ISelectionService.addSelection-Listener(ISelectionListener)` call. A part is a specific selection listener if it specifies the part id of the part which events it is interested in listening to. In that case, the part registers with selection service using `ISelectionService.addSelectionListener(String, ISelectionListener)` call.

Classes `Part` and `Service` are abstractions of existing framework concepts. Attributes `name`, `multiPage`, `qualifier`, and `partId` of class `Part` represent actual part attributes defined in the framework. Read-only, boolean attribute `local` is a derived attribute and is true if a class representing the part is a source class and belongs to the current project. Parts with attribute `local` valued false represent either binary classes or source classes but located in referenced projects. Only local parts can be modified using WPI FSML by changing their attribute values and/or specifying part interactions.

Note, that there is no one-to-one mapping between WPI FSML interaction concepts and framework classes. The interaction concepts are therefore abstractions of usages of the base framework. The framework only provides mechanisms for implementing the interactions by means of interfaces, adapters and services, but does not provide ready to use implementations.

Abstract syntax class diagram from Fig. 3 is derived from feature models presented in Section 3, Fig. 1. Features from concept feature models map to class diagram elements in various ways. Root features such as `EditorPart` (Fig. 1(a)) and `ListensToParts` (Fig. 1(b)) describe concepts and therefore map to classes.

Atomic subfeatures such as `name` or `partId` map to class attributes. Stereotype `<<K>>` indicates an attribute is a key and is used to unambiguously identify a concept. For example an EditorPart is identified by its `name` and `qualifier`, RequiresAdapter interaction is identified by `source`, `target` and `interface`.

Attribute multiplicities `1` and `0..1` indicate that an attribute is mandatory or optional, respectively. Key attributes are always mandatory. For example, attribute `partId` is optional, because an editor is not required to have a part id, in which case, the value of the `partId` attribute will be null and indicate the absence of the feature. Mandatory features which do not have any attributes are represented as boolean attributes. In this case, false indicates absence of the feature.

Composite features such as `multiPage` are represented by classes and containment association. Class `MultiPageFeature` corresponds to feature `multiPage`. Its attributes `extendsMultiPageEditorPart` and `contributorExtendsMultiPageABC` correspond to subfeatures of the `multipage` feature. Optional cardinality of feature `multiPage` is represented by multiplicity 0..1 of the `multiPage` containment association end.

4.2 Mapping abstract syntax to the framework API

Framework concept can be identified in the code by identifying all of its mandatory features. Analogously, a concept can be implemented in the code by implementing its features accordingly to feature configuration (i.e., a selection of features). Therefore the mapping between the model concepts and their implementation code is specified as mappings for individual features.

Features may have reverse and forward mappings. Reverse mapping defines how the value of attribute corresponding to a feature is computed. Forward mapping defines how the value of the attribute is translated into code. Forward mapping has three flavors: addition, refactoring and removal. Features without attributes can only be added or removed. Features with attributes can also be refactored, when their value changes. Read-only features have a reverse mapping only.

In our prototype, we have implemented the mappings in Java. For better presentation, however, here we define a number of *predicates* and *statements* that correspond to commonly used functions from our implementation.

Often, a statement performs addition, refactoring and removal, depending on values of arguments. For example, `null` argument forces removal, `non-null` argument forces addition if an instance of the feature does not exist in the code or refactoring if the instance exists.

For code generation we choose to use template syntax from Meta-AspectJ [7] as it allows us using AspectJ pointcuts, method introductions and intertype declarations to specify where the code should be woven. In short, `‘[<code>]` is the quote operator, `#<variable>` and `#[<expression>]` are the unquote operators. The unquote operator splices the value of a variable or an expression. Statement `WEAVE(template code)` executes the weaving of Meta-AspectJ template code specified as its argument. Consequently, `UNWEAVE(template code)` is used to execute reverse operation and is used for removal in forward mappings[2].

Table 1 shows forward and reverse mappings for the features of the EditorPart concept. The first row is a `mapping` declaration that binds two variables: `editor` to the code class and `ep` to the model editor part. Variable `editor` is used in reverse mappings and `ep` is used in forward mappings. The execution of the `EditorPart` mapping in a forward direction takes an instance of the abstract syntax class `EditorPart` as a parameter and creates, modifies or removes a class in the code. The execution in the reverse direction takes a code class and produces model element. Each row contains the definition of attribute corresponding to the feature which reverse mapping is indicated by $\hookleftarrow$, and forward mapping is indicated by $\mapsto$. The reverse mapping is an assignment to the attribute. Forward mapping is a sequence of statements executing code transformation.

[2]Not shown for brevity.

```
mapping EditorPart(EditorPart ep <-> Class editor);
key name
↩ ep.name = editor.name;
↦ RENAME(editor, ep.name);
key qualifier
↩ ep.qualifier = editor.package;
↦ MOVE(editor, ep.qualifier);
mandatory implementsIEditorPart
↩ ep.implementsIEditorPart = IMPLEMENTS(editor, IEditorPart);
↦ WEAVE('[ declare parents : #[ep.name] implements IEditorPart ]);
optional partId
↩ ep.partId = EDITORID(editor);
↦ EDITORID(ep.qualifier + "." + ep.name, ep.partId);
read-only, optional local
↩ local = editor.isSource() && INCURRENTPROJECT(editor);
optional contributor
↩ ep.contributor = CONTRIBUTOR(editor);
↦ CONTRIBUTOR(ep.qualifier + "." + ep.name, ep.contributor);
optional multiPage
↩ ep.multiPage = REVERSE(MultiPageFeature(ep <-> editor));
↦ FORWARD(MultiPageFeature(ep <-> editor));
```

Table 1. Definitions of features for EditorPart concept.

```
mapping MultiPageFeature(EditorPart ep <-> Class editor)
optional contributorExtendsMultiPageABC
↩ contributorExtendsMultiPageABC =
  IMPLEMENTS(ep.contributor, MultiPageActionBarContributor);
↦ if (ep.contributor != null) WEAVE('[declare parents : #[ep.contributor]
  extends MultiPageActionBarContributor;]);
mandatory extendsMultiPageEditorPart
↩ extendsMultiPageEditorPart = IMPLEMENTS(editor, MultiPageEditorPart);
↦ WEAVE('[declare parents : #[ep.name] extends MultiPageEditorPart;]);
```

Table 2. Definitions of features for the MultiPageFeature concept.

Predicate `IMPLEMENTS(x, type)` returns true if type `x`, directly or indirectly, implements interface `type` or extends class `type`. Parameter `x` can be either a class or a fully qualified name of the class.

Information about part ID and action bar contributor has to be retrieved from the plugin.xml file of the analyzed project. XML node containing information about editor can be accessed using XPath expression. Expression `/extension[point='org.eclipse.ui.editors']/editor[class='name']` selects `editor` node for an editor with fully qualified name `name`. The `editor` node is used by `EDITORID` and `CONTRIBUTOR` predicates and statements. Predicate `EDITORID(x)` returns the value of the `id` attribute for editor `x`. Statement `EDITORID(x, partId)` sets the value of the `id` attribute to the value `partId`. Predicate `CONTRIBUTOR(x)` returns the value of the `contributor` attribute for editor `x`. Statement `CONTRIBUTOR(this, c)` sets the value of the `contributor` attribute to the value `c`.

Statement `RENAME(x, name)` executes rename refactoring on class `x`. Statement `MOVE(x, package)` executes move refactoring on class `x`. Predicate `INCURRENTPROJECT(x)` returns true if class `x` resides in the current project.

Statement `FORWARD(feature)` executes forward mapping for given concept.

Table 3 shows the definitions of features of the RequiresAdapter interaction. The first row of the mapping definition binds two variables `source` and `target` if `Part(source)` and `Part(source)`, i.e., both `source` and

```
mapping RequiresAdapter(RequiresAdapter ra <-> Class source, Class target)
when Part(source <-> sourcePart) and Part(target <-> targetPart);
------------------------------------------------------------------------
mandatory sourceRequestsAdapter
↩ ra.sourceRequestsAdapter = CALLS(source, ‘[IAdaptable.getAdapter(?ra.interface)]);
↦ WEAVE(‘[
 private void #[sourcePart.name].request#[ra.interface]Adapter(IWorkbenchPart  part) {
   part.getAdapter(#[ra.interface].class);];
 } ]);
------------------------------------------------------------------------
mandatory targetProvidesAdapter
↩ ra.targetProvidesAdapter = RETURNS(‘[IAdaptable.getAdapter(Class)], target, ra.interface)
↦ WEAVE(‘[
 before(): returning #[targetPart.name].getAdapter(Class ?key) {
  if (#key.equals(#[ra.interface].class))
   return get#[ra.interface]Adapter();
  }
  private #[ra.interface] #[targetPart.name].get#[ra.interface]Adapter() {
   return null;
  } ]);
```

Table 3. Definitions of features for RequiresAdapter interaction concept.

`target` are workbench parts.

Predicate `CALLS(type, method template)` returns true if there exists a call to method `method template` within the supertype hierarchy of `type` (including the `type` itself). Any variables used in the template and prefixed a ? (question mark) are bound to actual method parameter values when the template is matched. Predicate `RETURNS(method, type, return type)` returns true if method `method` of type `type` returns an object of `return type`.

Note, that there is no direct definition for `interface` attribute, because its value is set when the method pattern in reverse definition of `sourceRequestsAdapter` attribute is matched.

Both forward definitions from table 3 introduce new methods, so that the requirements for requesting an adapter by the source part and providing the adapter by the target part are satisfied. Another function of the generated code is to provide sample code for the developers.

The ordering of the execution of reverse mappings for features from Table 3 is important. The mapping for feature `targetProvidesAdapter` has to be executed after the mapping for feature `sourceRequestsAdapter` because the latter binds the value of the `RequiresAdapter.interface` attribute (`?ra.interface`) which is than used in determining that the target indeed returns objects implementing the required interface. Note, that in this case the order of the execution of forward mappings does not matter.

The `before()` advice is used to add an additional `if` clause to the `getAdapter` method. In this case, the temporary variable `?key` matches the name of the parameter which is later used in generating the additional `if` clause (`#key.equals`).

Table 4 shows the definitions of features of the ProvidesSelection interaction.

Table 5 shows the definitions of features of the ListensToParts interaction.

Table 6 shows the definitions of features of the ListensToSelections interaction for a global listener.

Table 7 shows the definitions of features of the ListensToSelections interaction for a specific listener.

Table 8 shows the definitions of features of the ListensToSelectionChanged interaction.

```
mapping ProvidesSelection(ProvidesSelection ps <-> Class source)
when Part(source <-> sourcePart) and PageSelectionService(target <-> sevice);
--------------------------------------------------------------------------------
mandatory sourceImplementsISelectionProvider
↩ ps.sourceImplementsISelectionProvider = IMPLEMENTS(source, ISelectionProvider);
↦ WEAVE('[ declare parents : #[sourcePart.name] implements ISelectionProvider; ]);
--------------------------------------------------------------------------------
mandatory sourceRegistersWithPageSelectionService
↩ ps.sourceRegistersWithPageSelectionService =
  CALLS(source, ‘[PartSite.setSelectionProvider(ISelectionProvider)]);
↦ WEAVE(‘[
 after(): execution #[sourcePart.name].createPartControl() {
   getSite().setSelectionProvider(this);
 }]);
--------------------------------------------------------------------------------
```

Table 4. Definitions of features for ProvidesSelection interaction concept.

```
mapping ListensToParts(ListensToPart ltp <-> Class source)
when Part(source <-> sourcePart);
--------------------------------------------------------------------------------
mandatory sourceImplementsIPartListener
↩ ltp.sourceImplementsIPartListener = IMPLEMENTS(source, IPartListener);
↦ WEAVE('[ declare parents : #[sourcePart.name] implements IPartListener ]);
--------------------------------------------------------------------------------
mandatory sourceRegistersWithPartService
↩ ltp.sourceRegistersWithPartService =
  CALLS(source, ‘[IWorkbenchPage.addPartListener(IPartListener)]);
↦ WEAVE(‘[
 private void #[sourcePart.name].registerWithPartService() {
   getSite().getPage().addPartListener(this);
 }]);
--------------------------------------------------------------------------------
mandatory sourceDeregistersWithPartService
↩ ltp.sourceDeregistersWithPartService =
  CALLS(source, ‘[IWorkbenchPage.removePartListener(IPartListener)]);
↦ WEAVE(‘[
 private void #[sourcePart.name].deregisterWithPartService() {
   getSite().getPage().removePartListener(this);
 }]);
--------------------------------------------------------------------------------
```

Table 5. Definitions of features for ListensToParts interaction concept.

```
mapping ListensToSelectionsGlobal(ListensToSelections lts <-> Class source)
when Part(source <-> sourcePart) and PageSelectionService(target <-> service);
------------------------------------------------------------------------------
mandatory sourceImplementsISelectionListener
↩ lts.sourceImplementsISelectionListener = IMPLEMENTS(source, ISelectionListener);
↦ WEAVE(’[ declare parents : #[sourcePart.name] implements ISelectionListener; ]);
------------------------------------------------------------------------------
mandatory sourceRegistersWithPageSelectionService
↩ lts.sourceRegistersWithPageSelectionService =
  CALLS(source, ‘[IWorkbenchPage.addSelectionListener(ISelectionListener)]);
↦ WEAVE(‘[
 private void #[sourcePart.name].registerWithPageSelectionService() {
   getSite().getPage().addSelectionListener(this);
 }]);
------------------------------------------------------------------------------
mandatory sourceDeregistersWithPageSelectionService
↩ lts.sourceDeregistersWithPageSelectionService =
  CALLS(source, ‘[IWorkbenchPage.removeSelectionListener(ISelectionListener)]);
↦ WEAVE(‘[
 private void #[sourcePart.name].deregisterWithPageSelectionService() {
   getSite().getPage().removeSelectionListener(this);
 }]);
------------------------------------------------------------------------------
```

Table 6. Definitions of features for ListensToSelections (global) interaction concept.

```
mapping ListensToSelectionsSpecific(ListensToSelections lts  <-> Class source, Class target)
when Part(source <-> sp) and Part(target <-> tp);
--------------------------------------------------------------------------------------------
mandatory sourceImplementsISelectionListener
↩ lts.sourceImplementsISelectionListener = IMPLEMENTS(source, ISelectionListener);
↦ WEAVE(’[ declare parents : #[sourcePart.name] implements ISelectionListener; ]);
--------------------------------------------------------------------------------------------
mandatory sourceRegistersWithPageSelectionService
↩ lts.sourceRegistersWithPageSelectionService =
  CALLS(source, ‘[IWorkbenchPage.addSelectionListener(#[tp.partId], ISelectionListener)]);
↦ WEAVE(‘[
 private void #[sourcePart.name].registerForSelectionFrom#[tp.name]() {
   getSite().getPage().addSelectionListener(#[tp.partId], this);
 }]);
--------------------------------------------------------------------------------------------
mandatory sourceDeregistersWithPageSelectionService
↩ lts.sourceDeregistersWithPageSelectionService =
  CALLS(source, ‘[IWorkbenchPage.removeSelectionListener(#[tp.partId], ISelectionListener)]);
↦ WEAVE(‘[
 private void #[sp.name].deregisterFromSelectionsFrom#[tp.name]() {
   getSite().getPage().removeSelectionListener(#[tp.partId], this);
 }]);
--------------------------------------------------------------------------------------------
```

Table 7. Definitions of features for ListensToSelections (specific) interaction concept.

mapping `ListensToSelectionChanged(ListensToSelectionChanged ltsc <-> Class source, Class target)`
when `Part(source <-> sp) and Part(target <-> tp);`

---

mandatory *sourceImplementsISelectionChangedListener*

```
↩ ltsc.sourceImplementsISelectionChangedListener = IMPLEMENTS(source, ISelectionChangedListener);
↦ WEAVE('[ declare parents : #[sp.name] implements ISelectionChangedListener; ]);
```

---

mandatory *targetImplementsISelectionProvider*

```
↩ ltsc.targetImplementsISelectionProvider = IMPLEMENTS(target, ISelectionProvider);
↦ WEAVE('[ declare parents : #[tp.name] implements ISelectionProvider; ]);
```

---

mandatory *sourceRegistersWithTarget*

```
↩ ltsc.sourceRegistersWithTarget =
  CALLS(source, ‘[#[tp.name].addSelectionChangedListener(ISelectionChangedListener)]);
↦ WEAVE(‘[
 after(): execution #[sp.name].createPartControl() {
   target.addSelectionChangedListener(source);
 }]);
```

---

mandatory *sourceDeregistersWithTarget*

```
↩ ltsc.sourceDeregistersWithTarget =
  CALLS(source, ‘[target.removeSelectionChangedListener(ISelectionChangedListener)]);
↦ WEAVE(‘[
 before(): execution #[sp.name].dispose() {
   target.removeSelectionChangedListener(source);
 }]);
```

---

mandatory *targetNotifiesListeners*

```
↩ ltsc.targetNotifiesListeners =
  CALLS(target, ‘[ISelectionChangedListener.selectionChanged(ISelectionChangedEvent)]);
↦ WEAVE(‘[
 private void #[tp.name].provideSelectionChanged(ISelectionChangedListener listener,
         ISelection selection) {
   listener.selectionChanged(new SelectionChangedEvent(this, selection));
 }]);
```

Table 8. Definitions of features for ListensToSelectionChanged interaction concept.

4.3 Round-Trip Engineering

Our Eclipse WPI FSML supports full round-trip engineering, meaning that the model and the completion code can be edited independently and synchronized whenever desired. The result of synchronization is a model and code that are consistent. We also refer to this state of model and code as *reconciled.* The synchronization procedure follows a concurrent-versioning paradigm, which is inspired by the *Concurrent Versioning System* (CVS) and its Eclipse user interface [8]. Figure 4 shows the artifacts involved in the synchronization procedure. The intention of the synchronization procedure is to synchronize the current *asserted model* and the current *framework completion code.* In order to achieve this, the current *implementation model* is automatically derived from the current code. Furthermore, we assume that the *last reconciled model* has been archived (containment `lastReconciled` from Fig. 3(a)), which has been the result of the previous execution of the synchronization procedure before the model and/or the code were edited again. Special cases occur when last reconciled model, asserted model, and/or code are missing. These cases include situations where the code has to be first created from an existing model, the model has to be first created from existing code, or where independently created model and code need to be synchronized.

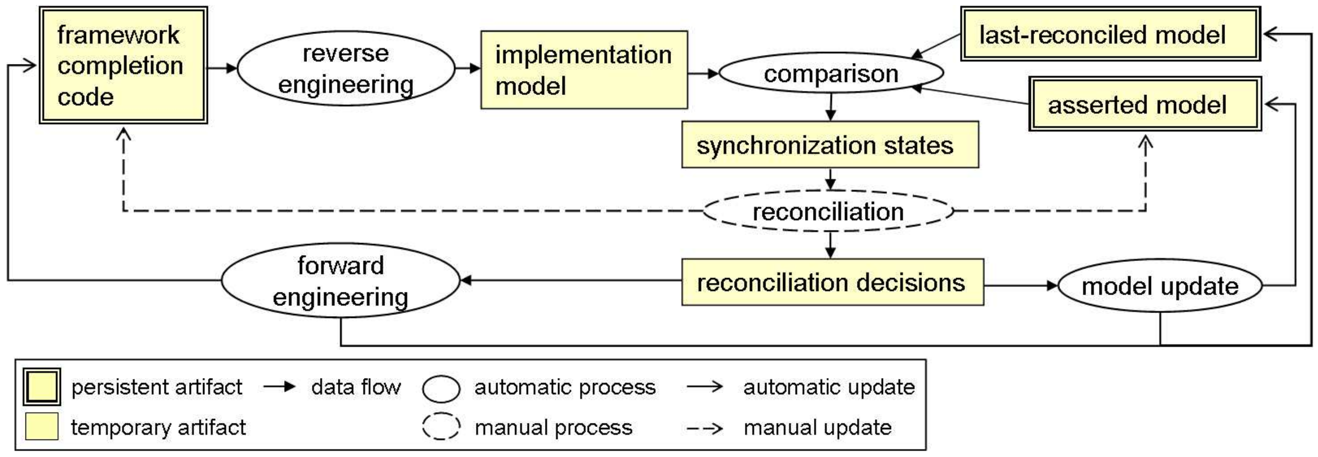


Figure 4. Artifacts and processes of agile round-trip engineering

Given an asserted model, a completion code, and a last reconciled model, the synchronization procedure involves the following steps:

1. Reverse engineering. The reverse mapping is executed on the code to create the implementation model. Consider the mapping from `EditorPart` to `Class` (Table 1) as an example. For every class in the code, the reverse mappings of the `EditorPart` features are executed, and an instance of `EditorPart` is created iff all mandatory features are present in the code or an `EditorPart` instance for that class already exists in the asserted model or in the last reconciled model. Reverse mappings for interactions are executed afterwards for exery pair of parts (e.g., `RequiresAdapter`) or for single classes (e.g., (ListensToParts)). In the case that there is no code, the implementation model is empty.

2. Comparison. This process compares the asserted model and the implementation model using the last reconciled model as a reference. The comparison involves establishing *correspondence links* among the corresponding concept instances in the three models. The correspondence among two or more concept instances is established if the values of their key attributes match. For every two corresponding concepts from the asserted model and the implementation model, an instance of a concept-correspondence object of type `ConceptSyncItem` is created (Fig. 5). The object keeps references to the concept instances from both the asserted and implementation models (if any). Furthermore, the concept-correspondence object contains other concept-correspondence objects for composite subfeatures and attribute-correspondence objects of type `AttributeSyncItem` for attributes[3]. All correspondence objects store the *synchronization state* and *reconciliation decision* for the particular elements they

[3]Note that the type of AttributeSyncItem.feature is Property. Property is the MOF type used for modeling of class attributes and allows AttributeSyncItems to 'point' at individual attributes.

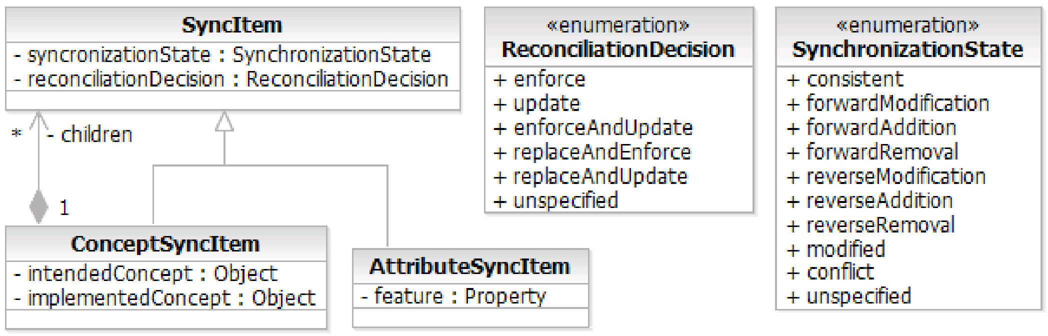


Figure 5. Correspondence Objects, Synchronization States and Reconciliation Decisions

link (see enumerations on Fig. 5). Synchronization state indicates what reconciliation actions have to be executed and in which direction in order to make concepts and attributes consistent.

Synchronization state `consistent` means that a concept or a feature from the asserted model is the same as its instance in the code. State `forwardModification` means that the value of a non-boolean feature attribute changed in the asserted model and needs to be propagated to the code. For the concept it means that concept's features have only forward or consistent states. State `forwardAddition` means that a new concept or a feature which didn't exist in the code appeared in the asserted model and therefore needs to be propagated to the code. State `forwardRemoval` means that a concept or a feature which did exist in the code was removed from the asserted model and therefore needs to be removed from the code. The reverse counterparts `reverseModification`, `reverseAddition` and `reverseRemoval` mean that a concept or a feature has been modified, added or removed in the code and these facts need to be reverse propagated to the asserted model. State `conflict` means that incompatible changes have been made to both the code and the asserted model. Synchronization state `modified` can only be applied to the concept and indicates that its subfeatures have at least one forward and at least one reverse synchronization state, but without any conflicits.

Using the feature modeling terminology from Sec. 2, modification is only possible for features having an attribute and/or having subfeatures; other features can only be added or removed.

The decision tables of how the synchronization states are established for attributes and than for concepts is given in Tables 9 and 10 respectively. Each row should be read from left to right. The last column, *attribute/concept state*, contains a synchronization state for an attribute or a concept when conditions from the preceeding columns have been satisfied. First three columns assert the existence of the concept containing the attribute. Symbols **m**, **c** and **r** stand for the existence of a concept from the asserted model, implementation model and last reconciled model respectively (1 - present, 0 - absent). Symbols **mv**, **cv** and **rv** stand for the existence of attribute's value in asserted model, implementation model and last reconciled model respectively. For boolean attributes 1 indicates true, 0 false. For other attributes 1 indicates non-null value, 0 null. Symbol **"-"** indicates the entry is irrelevant.

For example row #1 in Table 9 states that when a concept is present in both the asserted model and implementation model, and the attribute has a value in both models, and the values are equivalent then the attribute's synchronization state should be *consistent*. Row #6 illustrates the use of the last reconciled model as the point of reference. In this case, the values of an attribute exist both in the asserted model and in the last reconciled model, but is absent in the implementation model. Furthermore, both values are the same. That indicates, the code has been modified and the value was removed. Therefore, the synchronization state for the attribute should be *reverseRemoval*, that is, the model has to be updated to reflect the code. Row #7 is similar to row #6 and states that the value was modified in the asserted model (because it is different than the value in the last reconciled model) and removed in the code. Therefore, the synchronization state for the attribute should be *conflict* and the user should decide which value should be kept in the reconciled model and code.

Table 10 presents how the synchronization state for the concept depends on the synchronization states of its attributes. Symbol **ac** indicates that all attributes are consistent. Symbols **ncf**, **ncr** indicate that, among all attributes there is no conflict and at least one forward or reverse synchronization state respectively. Symbol **aloc** indicates that at least one attribute has *conflict* synchronization state.

For example, row #3 states that if a concept exists in both the asserted and the implementation models, and

#	m	c	r	mv	cv	rv	condition	attribute state
1	1	1	-	1	1	-	$mv == cv$	consistent
2				1	1	1	$mv! = cv\&\&mv == av$	reverseModification
3				1	1	1	$mv! = cv\&\&cv == av$	forwardModification
4				1	1	1	$mv! = cv\&\&mv! = av\&\&cv! = av$	conflict
5				1	1	0	$mv! = cv$	conflict
6				1	0	1	$mv == av$	reverseRemoval
7				1	0	1	$mv! = av$	conflict
8				1	0	0	-	forwardAddition
9				0	1	1	$cv == av$	forwardRemoval
10				0	1	1	$cv! = av$	conflict
11				0	1	0	-	reverseAddition
12				0	0	-	-	consistent
13	1	0	1	1	-	1	$mv == av$	reverseRemoval
14				1	-	1	$mv! = av$	conflict
15				1	-	0	-	reverseRemoval
16				0	-	-	-	consistent
17	1	0	0	1	-	-	-	forwardAddition
18				0	-	-	-	consistent
19	0	1	1	-	1	-	-	forwardRemoval
20				-	0	-	-	consistent
21	0	1	0	-	1	-	-	reverseAddition
22				-	0	-	-	consistent
23	0	0	-	-	-	-	-	consistent

Table 9. Attribute Synchronization State Decision Table.

#	m	c	r	ac	ncf	ncr	aloc	concept state
1	1	1	-	1	0	0	0	consistent
2				0	1	1	0	modified
3				0	1	0	0	forwardModification
4				0	0	1	0	reverseModification
5				0	0	0	1	conflict
6	1	0	1	-	-	-	-	reverseRemoval
7	1	0	0	-	-	-	-	forwardAddition
8	0	1	1	-	-	-	-	forwardRemoval
9	0	1	0	-	-	-	-	reverseAddition
10	0	0	-	-	-	-	-	consistent

Table 10. Concept Synchronization State Decision Table.

there is no conflicting attribute and at least one of the attributes has forward synchronization state, than the synchronization state for the whole concept should be *forwardModification*.

Row #6 states that if the concept exists in both asserted and last reconciled models but is missing in the implementation model (was removed from the code), than the synchronization state for the whole concept should be *reverseRemoval*.

Reconciliation decisions specify whether an addition, a removal, or a modification should be propagated form the model to the code or vice versa. They are determined in the following step.

3. Reconciliation. For all elements with synchronization state other than consistent, a reconciliation decision needs to be made by the user.

Table 11 summarizes possible reconciliation decisions for given synchronization states. For forward synchroniza-

Synchronization State	**Reconciliation Decisions**
`consistent`	
`forwardModification`, `-Addition`, `-Removal`	`enforce`, `replaceAndUpdate`
`reverseModification`, `-Addition`, `-Removal`	`update`, `replaceAndEnforce`
`modified`	`enforceAndUpdate`, `replaceAndUpdate`, `replaceAndEnforce`
`conflict`	`replaceAndUpdate`, `replaceAndEnforce`

Table 11. Synchronization states and available reconciliation decisions

tion states possible decisions are `enforce` and `replaceAndUpdate`, for reverse states are `update` and `replaceAnd-Enforce`, for conflicts only `replaceAndEnforce` and `replaceAndUpdate`. For modified state (both forward and reverse, but no conflicts) an additional `enforceAndUpdate` decision is available. Reconciliation may also require editing the asserted model, e.g., by providing new attribute values. At the end of the reconciliation step, the asserted model with the projected changes based on the reconciliation decisions needs to satisfy the well-formedness constraints of the FSML.

4. Forward engineering and asserted model update. Finally, any necessary changes are executed accordingly to the reconciliation decisions. Concepts and features with forward decisions (enforce, replaceAndEnforce and enforceAndUpdate) are enforced in the code by executing the corresponding forward mappings. Concepts and features with reverse decisions (update, replaceAndUpdate and enforceAndUpdate) are updated with values from the implementation model. The execution of the individual forward mappings needs to be properly scheduled in order to be correct.

5 Prototype Implementation

We developed a prototype of the WPI FSML as an Eclipse plug-in. Abstract syntax of the language, including well-formedness constraints, is implemented using Eclipse Modeling Framework (EMF) and its model validation framework. Reverse mappings use the AST, query, and pattern matching API of Eclipse's Java Development Tools (JDT) and type inference engine of the *Infer Generic Type Arguments* refactoring [9]. Forward mappings use Eclipse's JDT Java Model and AST rewriting API. The prototype supports agile round-trip engineering. The reverse mappings are completely implemented. To date, the forward mappings support the creation of classes with methods implementing the framework-stipulated behaviour, addition of interfaces and superclasses, and handling the plug-in manifest files. Weaving of *before* and *after* advices, and code fragment removal are not yet implemented.

An on-line demonstration of the prototype is available at our web page.

References

[1] Antkiewicz, M., Czarnecki, K.: Framework-specific modeling languages with round-trip engineering. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proceedings of MoDELS'06, Genoa, Italy. (2006)

[2] Eclipse Foundation: Eclipse. Available at `http://www.eclipse.org/` (2006)

[3] Xenos, S.: Inside the Workbench—A guide to the workbench internals. IBM. (2005) Available at `http://www.eclipse.org/articles/Article-UI-Workbench/workbench.html`. Last accessed March 17, 2006.

[4] Eclipse Foundation: Java Development Tools (JDT). Available at `http://www.eclipse.org/jdt/` (2006)

[5] Pandit, C.: Make your Eclipse applications richer with view linking. IBM India Software Labs. (2005) Available at `http://www.ibm.com/developerworks/java/library/os-ecllink/`. Last accessed March 17, 2006.

[6] Czarnecki, K., Kim, C.H.P.: Cardinality-based feature modeling and constraints: a progress report. In: International Workshop on Software Factories, San Diego, California (2005)

[7] Zook, D., Huang, S.S., Smaragdakis, Y.: Generating AspectJ programs with Meta-AspectJ. In: Generative Programming and Component Engineering: Third International Conference Proceedings. Volume 3286 of Lecture Notes in Computer Science., Springer (2004) 1 – 18

[8] Eclipse Foundation: Team CVS tutorial. (2006) Available in Workbench User Guide, Eclipse Help.

[9] Tip, F., Fuhrer, R., Dolby, J., Kieżun, A.: Refactoring techniques for migrating applications to generic Java container classes. IBM Research Report RC 23238, IBM T.J. Watson Research Center, Yorktown Heights, NY, USA (2004)