EMF Components
Filling the Gap between Models and UI[*]

Lorenzo Bettini
Dipartimento di Informatica, Università di Torino, Torino, Italy

Keywords: EMF, Modeling, Eclipse, User Interface.

Abstract: The Eclipse Modeling Framework (EMF) provides code generation facilities for building tools and applications based on structured data models. Most of the Eclipse projects which somehow have to deal with modeling data are based on EMF since it simplifies the development of complex software applications with its modeling mechanisms. However, we argue that from the user interface point of view, the development of applications based on EMF could be made simpler, with more code reuse, and without having to deal with too many internal details. For this reason, in this paper, we propose Emf Components, a lightweight framework that allows easy and quick development of applications based on EMF and it can be configured to use all kinds of EMF persistence implementations (e.g., XMI, Teneo, CDO). It aims at providing a set of reusable components like trees, tables and detail forms that manage the model with the introspective EMF capabilities. The framework provides basic UI implementations which are customizable in a declarative way and with dependency injection mechanisms.

1 INTRODUCTION

The *Eclipse Modeling Framework* (EMF) (Steinberg et al., 2008) provides code generation facilities for building tools and applications based on structured data models. The model specification can be described in XMI, UML or annotated Java, and from this specification EMF produces a set of Java classes for the model, along with a set of adapter classes for editing facilities.

EMF simplifies the development of complex software applications with its modeling mechanisms. However, we argue that from the user interface point of view, the development of applications based on EMF could be made simpler, with more code reuse, and without having to deal with too many internal details. For this reason, in this paper, we propose the Emf Components framework.

Emf Components is a lightweight framework that allows easy and quick development of applications based on EMF. It can be configured to use all kinds of EMF persistence implementations (e.g., XMI, Teneo, CDO). It aims at providing a set of reusable components like trees, tables and detail forms that manage the model with the introspective EMF capabilities. Using these components one can easily build more complex widgets, editors and applications. The framework provides basic UI implementations which are customizable with injection mechanisms (based on Google Guice (Google, 2012), a Dependency Injection framework (Fowler, 2004)). The Emf Components framework is available as an open source project at http://code.google.com/a/eclipselabs.org/p/emf-components.

EMF already provides some mechanisms for generating user interface code, in particular, it can also generate an editor (based on a tree viewer) for an EMF metamodel. Indeed, besides the classes for the model, EMF can generate an additional plugin with an editing framework, i.e., classes not directly dealing with UI functionalities, but that can be used for generic editing features; then, it generates a plugin which directly deals with the UI part, in the shape of an editor and some wizards, which rely on the generated editing framework for the metamodel.

In particular, EMF.Edit (EMF.Edit, 2004; Steinberg et al., 2008) is a framework which is part of EMF that includes generic reusable classes for building editors for EMF models. In particular, it provides the content and label provider classes, property source

[*]The paper was partly supported by RCP Vision, www.rcp-vision.com, and by MIUR (Project PRIN 2008 DISCO).

Bettini L..
EMF Components - Filling the Gap between Models and UI.
DOI: 10.5220/0004027100340043
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 34-43
ISBN: 978-989-8565-19-8

support, and other convenience classes that allow EMF models to be displayed using standard desktop (JFace) viewers and property sheets. Moreover, it also provides a command framework, with a set of generic command implementation classes for building editors that support fully automatic undo and redo. These functionalities are dealt with by relying on an *editing domain* which manages a self-contained set of interrelated EMF models and the commands that modify them.

Most of these functionalities can be tied together by using *adapter factories* which can transparently bind the mechanisms of the EMF.Edit framework together with the UI functionalities, so that, for instance, labels for the viewer's elements can be retrieved automatically. The main problem for the programmer is still that all these mechanisms have to be setup and initialized correctly in order to achieve the desired functionalities; and this initialization might still be a burden for the programmer since it requires many lines of code, which are recurrent, require some deeper knowledge, and tend to fill the code with too many distracting details. In Listing 1 we show the typical Java code which is very recurrent in applications that use EMF models and UI functionalities (based on the above describe EMF.Edit framework). We note that such code is full of too many recurrent code with many internal details. As we will show in the next sections, our goal is to factor out this recurrent code in such a way that UI components (such as the viewer in the Listing) can be setup with only a few lines of code (see Listing 6, Section 3.1). Note that from Listing 1 we intentionally omitted all the instructions to load the actual resource of the EMF model, which requires some additional (and recurrent) effort. Finally, while the code presented in Listing 1 deals with a viewer with editing functionalities, still, even in the case of a viewer in read-only mode, many instructions would be required.

```
1  adapterFactory = new ComposedAdapterFactory(
       ComposedAdapterFactory.Descriptor.Registry.
       INSTANCE);
   adapterFactory.addAdapterFactory(new
       ResourceItemProviderAdapterFactory());
3  adapterFactory.addAdapterFactory(new
       MyModelItemProviderAdapterFactory());
   adapterFactory.addAdapterFactory(new
       ReflectiveItemProviderAdapterFactory());
5
   BasicCommandStack commandStack = new
       BasicCommandStack();
7  commandStack.addCommandStackListener
     (new CommandStackListener() {...});
9
   editingDomain = new AdapterFactoryEditingDomain(
       adapterFactory, commandStack, ...);
11
   Tree tree = new Tree(composite, SWT.MULTI);
13 TreeViewer viewer = new TreeViewer(tree);
15 viewer.setContentProvider(new
       AdapterFactoryContentProvider(adapterFactory)
       );
   viewer.setLabelProvider(new
       AdapterFactoryLabelProvider(adapterFactory));
17 viewer.setInput(editingDomain.getResourceSet());
19 new AdapterFactoryTreeEditor(viewer.getTree(),
       adapterFactory);
```

Listing 1: An example of typical recurrent use of EMF functionalities.

EMF standard generation mechanisms are based on specific Javadoc comments, i.e., `@generated`, for fields, methods and classes. The idea behind these generation mechanisms is that future generation will overwrite all the previously generated Java code elements unless the user removes that `@generated` from specific declarations (or replaces it with `@generated NOT`); thus, for instance, if in a generated Java class we want to modify a specific generated method with a custom implementation, we can remove that `@generated` from the method's Javadoc, and the next time we run EMF generation, that method will not be overwritten.

An example of customization of labeling in a class generated by EMF into the edit plugin (related to the classic EMF Library example) is shown in Listing 2. We observe that even for specifying the image for instances of `Book` we need to go into the generated `ItemProvider` class, and specify the path of the image, but also other distracting details (like `overlayImage`, `ResourceLocator`, etc.). Furthermore, if we want to customize the images for all the EClasses of our model, we need to modify every generated `ItemProvider` classes. Instead, we would like to have a more direct mechanisms to specify these customizations, and possibly grouped in one place, easier to maintain. We will show our customization strategies in Section 3.2.

Concerning the code generated for the model interfaces and classes, instead of using this technique for replacing generated code, we could use specific annotation in the ecore metamodel so that we can also specify the implementation of specific methods for the classes of the metamodel.

However, this is not easily applicable to code generated for the UI classes, thus one should use the `@generated` mechanism described above for customizing the generated editor.

Another problem with this `@generated` and

```
1  public class BookItemProvider extends ... {
     /** @generated NOT */
3    @Override
     public Object getImage(Object object) {
5      return overlayImage(object,
         getResourceLocator().getImage("mypath/
             custom_book.png"));
7    }

9    /** @generated NOT */
     @Override
11   public String getText(Object object) {
       return "Book: " + " " + ((Book)object).
           getTitle();
13   }
```

Listing 2: An example of typical customization of EMF labeling.

`@generated NOT` mechanism is that it might easily get difficult to keep track of custom modifications to the generated code. Moreover, the editor generated by EMF is not effectively designed to promote code reuse: the generated editor is a monolithic class consisting of more than two thousand lines of code, with many inner classes. Indeed, if one compares the editors generated for two different metamodels, the differences are really minimal, thus the classes generated by EMF for the UI components leads to a huge amount of duplicated code.

A more appealing solution would be something similar to the *Generation Gap* (Vlissides, 1996), a pattern that solves the problem of maintainability of coexisting generated and manual code by relying on class inheritance: there will be a class that encapsulates generated code and another one class that encapsulates modifications.

However, it is not our main goal to change the mechanisms of code generation of EMF; instead we want to provide new UI classes that are more modular: we tried to split responsibilities into different classes and to then to use delegation as much as possible, following the *Single Responsibility Principle* (Martin, 2003).

The paper is structured as follows: in Section 2 we introduce our design choices, while in Section 3 we describe some components of our framework, with some examples. Section 4 concludes the paper with related work and hints for future directions.

2 DESIGN CHOICES

The main inspiration for dealing with customized injected code in Emf Components comes from XTEXT (Itemis, 2012), a framework for the development of programming languages as well as other domain-specific languages (DSLs): it provides high-level mechanisms that generate all the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse. All these mechanisms can be customized by injection using Google Guice (Google, 2012).

As hinted in the previous section, our main design choice in developing Emf Components is to split responsibilities into small classes, so that customizing a single aspect of UI components should not require to subclass the components themselves, but only to customize the class that deal with that specific aspect. Then, our custom version of that specific aspect will be injected in the framework, so that all components relying on that aspect will be assured to use our specific version. In order to deal with the consistent use of customized behaviors, we rely on Google Guice (Google, 2012), a dependency injection framework.

The *Dependency Injection* pattern (Fowler, 2004) is used to "inject" actual implementation classes into a class hierarchy in a consistent way. This is useful when classes delegate specific functionalities to other classes: messages are simply forwarded to the objects referenced in fields (which abstract the actual behavior). These fields will then be instantiated through injection mechanisms so that we do not have the implementation classes' names hardcoded in the code of the classes that will use them, and we are sure that, if we switch the implementation classes, we will do that consistently throughout the code. Typically the same goal can be achieved manually by implementing *factory method* and *abstract factory* patterns (Gamma et al., 1995), but with *dependency injection frameworks* it is easier to keep the desired consistency, and the programmer needs to write less code.

Google Guice (Google, 2012) provides a framework for dependency injection that makes all the injection mechanisms really easy to use, and relieves the programmer from the most internal details of dependency injection. For instance, Guice relies on Java annotations (Sun Microsystems, Inc., 2007), in particular, `@Inject`, for specifying the fields that will be injected, and on a *module* which configures the bindings for the actual implementation classes; from the module we can build an *injector* which will be used to create the actual instances of classes of a framework, where the fields will be actually injected according to our bindings. An example is shown in Listing 3; note that we do not need to create a custom subclass of `Service`: we only need to provide custom implementations for classes that `Service` relies on. These custom implementations will be consistently injected

```
class Service {
  @Inject protected Log log;
  @Inject protected Processor p;
  ...
}

class MyModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Log.class).to(MyLog.class);
    bind(Processor.class).to(MyProcessor.class);
  }
}

Injector injector = Guice.createInjector(new
    MyModule());
Service service = injector.getInstance(Service.class);
...
```

Listing 3: An example of Google Guice usage.

```
<extension point="org.eclipse.ui.editors">
 <editor
  class="
   EmfComponentsExecutableExtensionFactory:
    EmfTreeEditor"
  id="it.rcpvision.emf.components.editors.
      treeEditor"
  name="EMF Tree Editor">
 </editor>
</extension>
```

Listing 4: An example of extension point using Google Guice.

throughout the application with Google Guice injection mechanisms.

In the context of Eclipse plugins (Clayberg and Rubel, 2008), these injection mechanisms cannot be used directly, since in Eclipse, most UI components, such as editors and views, are not usually instantiated programmatically: they are declared as *extension points*, i.e., through XML elements. Thus, the actual instantiation of UI components is done internally by Eclipse itself. However, when specifying the Java class in an extension point, we can prefix it with an `IExecutableExtensionFactory`, which allows to customize the instantiation of that class, and in our case, we can use Guice injection mechanisms. We then provide a base implementation of `IExecutableExtensionFactory`, `EmfComponentsExecutableExtensionFactory`, which relies on a Google Guice module that can be customized by the programmer (an example of extension point using this mechanism is shown in Listing 4, where, for simplicity, we omitted the Java packages names).

The programmer has to declare a subclass of `EmfComponentsExecutableExtensionFactory`, a subclass of `EmfComponentsGenericModule` with custom bindings, make the subclass of `EmfComponentsExecutableExtensionFactory` refer to the subclass of `EmfComponentsGenericModule`, and make sure to prefix the classes of the Emf Components framework with the custom executable extension factory, in the extension points. Indeed, the programmer does not have to perform all these initial setup operations: we provide a wizard that creates a new plugin project with all these classes, and the programmer will only have to provide the custom injection bindings.

```
public class MyEmfComponentsModule extends
    EmfComponentsGenericModule {

  public Class<? extends ResourceLoader>
      bindResourceLoader() {
    return MyResourceLoader.class;
  }

  public Class<? extends EmfViewerMouseAdapter
      > bindEmfViewerMouseAdapter() {
    return MyEmfViewerMouseAdapter.class;
  }

  public Class<? extends FeatureLabelProvider>
      bindFeatureLabelProvider() {
    return MyFeatureLabelProvider.class;
  }
}
```

Listing 5: A module with bindings.

Emf Components relies on the enhancement that Xtext added to Guice's Module API: Xtext provides an abstract base class, which reflectively looks for certain methods in order to find declared bindings. The most common kind of method that we rely on is of the shape `bind<ClassName>` where `ClassName` is the name of the class for which we need to specify a binding. An example of custom bindings relying on this reflective mechanisms is shown in Listing 5. Of course, the programmer can also use the standard Google Guice mechanisms for specifying the bindings, if he prefers to.

3 EMF COMPONENTS

In this Section we present the most important UI components provided by the Emf Components framework; this project is still in the early stage, thus more components will be provided in the future (see also Section 4 for future work).

```
public abstract class MyView extends ViewPart {

  @Inject ViewerInitializer initializer;

  @Override
  public void createPartControl(Composite parent) {
    ...
    viewer = new TreeViewer(parent, ...);
    // initialize with an EMF Resource
    initializer.initialize(viewer, resource);
    // or alternatively, if you have an EObject
    initializer.initialize(viewer, eObject);
  }
```

Listing 6: Initialization of a viewer with Emf Components.

3.1 Viewers

Eclipse JFace UI components often rely on viewers (e.g., tree viewers, table viewers, etc.). JFace provides specific viewers for trees, tables, lists, etc. These viewer classes, as specified in the Javadoc, are not intended to be subclassed. Indeed, they are parametrized over the content provider, which takes care of providing the contents based on the input, and other providers for the layout, for instance the label provider, which provides a textual representation of the elements to be shown by the viewer.

Thus, if one wants to create a viewer (e.g., a tree viewer) and initialize it, we provide an `ViewerInitializer`, providing many overloaded methods, which can be used to initialize the viewer with all the EMF mechanisms, but hiding the details from the programmer. In Listing 6 we show an example of a view with a viewer which relies on an injected `ViewerInitializer`. It uses such instance to initialize the tree viewer (it can initialize it based on an EMF Resource, or an EObject).

We invite the reader to compare the initialization code in Listing 6 with the one which is typically used without using Emf Components, in Listing 1. All the initialization details are carried on transparently by Emf Components classes. As we said, there are several initialization methods available, and if the programmer needs it, he can also specify all the providers for the viewers. In particular, if the programmer needs access to the `EditingDomain`, he can create it and pass it to the initialization; but also in this case, we provide utility mechanisms for creating (and initializing) an `EditingDomain` without having to worry about initializing the adapter factories for the editing domain itself (see the manual initialization in Listing 1).

3.2 Customizations

If one has used the EMF standard generation mechanisms to generate the edit plugin, and possibly customized some behavior in the edit plugin, all these customizations will be honored by Emf Components UI widgets.

However, in Emf Components, we also rely on the customization mechanisms (based on injection) which aim at making the customization of editing functionalities easier than the standard EMF.Edit framework.

The Xtext framework (on which we rely to implement some internal mechanisms of Emf Components) provides mechanisms, through the class `PolymorphicDispatcher`, for performing (overloaded) method dispatching according to the runtime type of arguments, a mechanism known as *dynamic overloading* (Castagna, 1997; Bettini et al., 2009)[2]. Note that this polymorphic dispatching mechanism provided by Xtext does not require a visitor structure (Gamma et al., 1995) since it inspects the available methods in a class and selects the method using reflection. By relying on this polymorphic dispatch mechanism we can provide a declarative way of specifying custom behaviors according to the class of objects of an EMF model (though the internal details about the use of `PolymorphicDispatcher` are hidden to the programmer).

For instance, by implementing a derived `CompositeLabelProvider` (a label provider of our framework), the programmer can specify the text and image for labels of the objects of the model by simply defining several methods `text` and `image`, respectively, using the classes of the model to be customized. An example is shown in Listing 7 (applied to the classic EMF Library example). These methods will be used internally by our label provider to implement the `LabelProvider`'s methods `getText` and `getImage`. Compare this code, which allows to customize in only one place the representation of the elements of a model, with the code in Listing 2, which refers to one single customization. With respect to Listing 2 we would also like to observe the absence of inner details (especially for the images).

Injecting this customization in the framework is just a matter of defining the binding in the Guice module, as shown in Listing 8 (we also refer to Section 2).

Another thing that the programmer might want to customize is the representation of the `EStructuralFeature`s of the model, i.e., the descriptions of

[2]While most compiled or statically-typed languages (such as Java) determine which implementation to call at compile-time.

```
public class CustomLabelProvider extends
      CompositeLabelProvider {

  public String text(Book book) {
    return "Book: " + book.getTitle();
  }

  public String image(Book book) {
    return "book.png";
  }

  public String text(Borrower b) {
    return "Borrower: " + b.getFirstName();
  }
  // other customizations
}
```

Listing 7: An example of customization of labeling in Emf Components.

```
public class MyCustomModule extends
    EmfComponentsGenericModule {
  public Class<? extends CompositeLabelProvider
      > bindCompositeLabelProvider() {
    return CustomLabelProvider.class;
  }
  ...
}
```

Listing 8: An example of bindings of customization of labeling.

the fields of the elements of the model. Again, we provide a declarative way to do this, through a custom `FeatureLabelProvider`, as illustrated in Listing 9; note that this time, the name of the method for customizing the text of the feature must contain the name of the EClass and the name of the specific feature of such EClass.

Similarly, we might want to specify only a subset of features to be shown in the UI components of a model; for this task, the Emf Components framework relies on `EStructuralFeaturesProvider`; the

```
public class CustomFeatureLabelProvider extends
    FeatureLabelProvider {

  public String text_Person_firstName() {
    return "First name";
  }

  public String text_Person_lastName() {
    return "Surname";
  }

}
```

Listing 9: An example of customization of EStructuralFeatures label representations in Emf Components.

```
import static org.eclipse.emf.examples.extlibrary.
    EXTLibraryPackage.Literals.*;

public class CustomEStructuralFeaturesProvider
    extends
    EStructuralFeaturesProvider {
  @Override
  protected void buildMap(
      EClassToEStructuralFeatureMap map) {
    super.buildMap(map);
    map.mapTo(LIBRARY,
        LIBRARY__NAME,
        ADDRESSABLE__ADDRESS);
    map.mapTo(PERSON,
        PERSON__FIRST_NAME,
        PERSON__LAST_NAME,
        ADDRESSABLE__ADDRESS);
    map.mapTo(WRITER,
        PERSON__FIRST_NAME,
        PERSON__LAST_NAME,
        WRITER__BOOKS);
  }
}
```

Listing 10: An example of customization of EStructuralFeatures to be represented in Emf Components.

programmer can customize it and specify the features to be shown for some of the classes of the model; the customization shown in Listing 10 should be self-explanatory[3].

Note that the customizations are not limited to classes belonging to a single EMF metamodel package: in one single place we can provide customization for every EMF classes that we intend to use with Emf Components widgets.

All these classes are used internally by the Emf Framework to represent models in the widgets; thus a custom injected implementation will be used consistently throughout the application for all the UI components created with the same `IExecutableExtensionFactory`. For instance, the custom implementations shown in Listing 9 and 10 will be used when building the forms (described in Section 3.3) and the column headers for table viewers (as shown later in Figure 2).

3.3 Composites

The smallest components which Emf Components provides as reusable units are Eclipse `Composites`. We provide composites to be reused in views and ed-

[3]Due to lack of space, we do not show here the corresponding mechanism dealing with the customization of EMF generated code; the interested reader can compare our solution with the typical example of EMF: the Library example.

itors based on several viewers. Also in this case, the setup instructions for these components is minimal.

The problem with SWT Composites is that they do not expose a default constructor; this is crucial for dependency injection to work. We think that this problem might disappear in the new version of Eclipse e4 (see also Section 4). However, for the moment, we provide factories to be used to actually create these composites; the idea is that you pass to the factory methods the arguments for the constructor, and, internally, the factory will also setup all the injection mechanisms, so the programmer will not have to deal with these details.

We will not describe here all the composites provided by the Emf Components framework; we will concentrate only on some of them. The first composite we describe is the *form* composite `FormDetailComposite` which shows the details of an EObject in a SWT form, and allows to edit such details. This is something that was currently missing in the EMF framework itself: the EMF framework only provides mechanisms to edit details of an EObject in the Eclipse standard *Properties* view, which might be limited in its functionalities. The SWT form toolkit instead provides richer UI features.

In Listing 11 we show a possible use of `FormDetailComposite`: we create a view which reacts on selections from other elements of the workbench, and if the selected element is an EObject it shows its details in the form. We highlighted the two relevant lines in the Listing which show how easy is to create this composite and set it up. All the other code in Listing 11 has to do with Eclipse and SWT, not with Emf Components itself. Indeed, we showed the code in Listing 11 as an example of use of Emf Components; however, this view class is already part of Emf Components framework, and it can already be reused in applications.

In Figure 1 we show a reusable editor provided by Emf Components and the form view implemented in Listing 11 (which shows the currently selected object fields for editing). Note that in the form we modified one feature of the selected writer, and the editor sensed this change and went into "dirty" state (the * in the editor title). This was possible thanks to the internal use of an editing domain; however, the programmer did not have to deal with this: everything is handled by the Emf Components framework itself. Moreover, any change to the model in any view or editor which is connected to the same model resource will soon be reflected in all the components using that resource: this takes place transparently, since the Emf Components widgets rely on EMF Databinding (Schindl, 2009), which connects the model and

```
public abstract class MyView extends ViewPart
  implements ISelectionChangedListener {

  @Inject EmfFormCompositeFactory factory;
  Composite detail;
  FormDetailComposite detailForm;

  @Override
  public void createPartControl(Composite parent) {
    ...
    detail = new Composite(this, SWT.BORDER);
  }

  @Override
  public void selectionChanged(
      SelectionChangedEvent event) {
    EObject selectedObject =
        getFirstSelectedEObject(event.getSelection
        ());
    if (selectedObject != null) {
      if (detailForm != null)
        detailForm.dispose();

      // relevant lines
      detailForm = factory.
        createFormDetailComposite(
          detail, SWT.BORDER);
      detailForm.init(selectedObject);
    }
  }
  ...
}
```

Listing 11: Using the `FormDetailComposite`.

the user interface.

In Figure 2 we show the same components, after applying the customizations of Listing 9 and 10. The changes of Listing 9 can be seen in the form where the labels for the features `firstName` and `lastName` have been changed; the changes of Listing 10 can be seen again in the form: only the features `firstName`, `lastName` and `books` are represented (see the last mapping in Listing 10). In Figure 2 we also show a tabular view (another example of use of Emf Components framework which we did not illustrate in this paper); note that since the same customized `EStructuralFeaturesProvider` was used, the headers of the columns in the table for the selected writer respect the customizations of Listing 9 and 10.

Finally, in Figure 3 we show how we can reuse the tree editor and the form editor together in a single view (in this case separated by a sash that allows the user to drag the outline). Again, this can be achieved with a minimal number of Java code, since the components of the framework are already implemented with compositionality in mind. Note that all the components shown here come with pre-

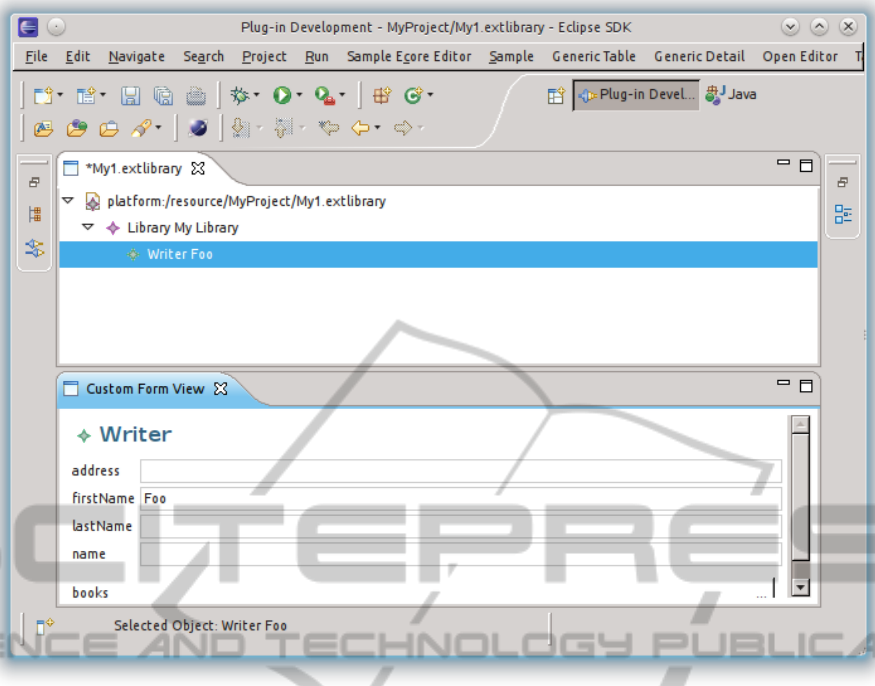

Figure 1: A tree editor and a form view.

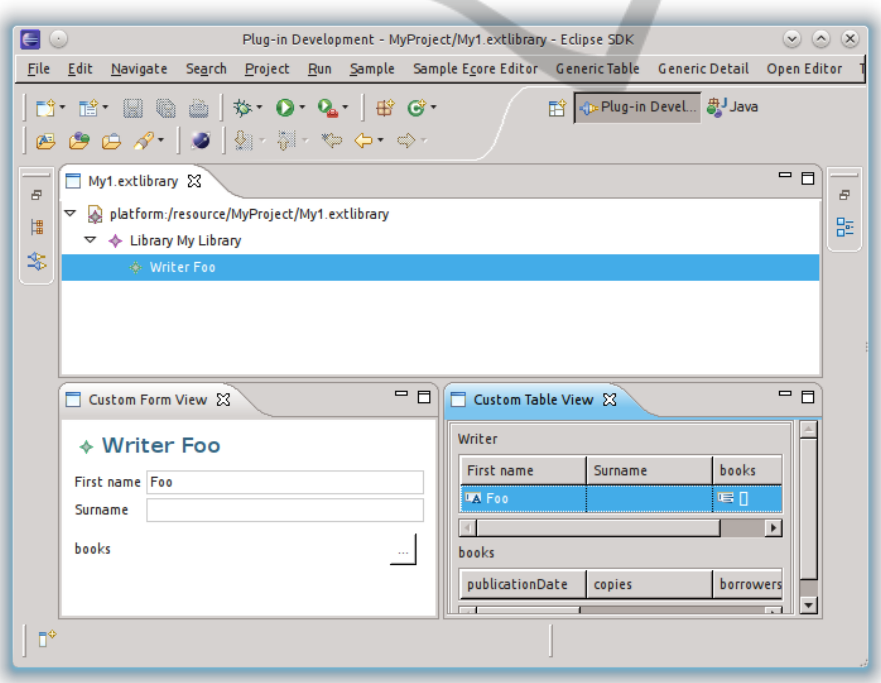

Figure 2: A tree editor, a form view and a tabular view with customizations.

defined SWT styles, but they can be configured later with custom styles.

3.4 Databases

There are already existing technologies to persist EMF models on databases. The most used ones are

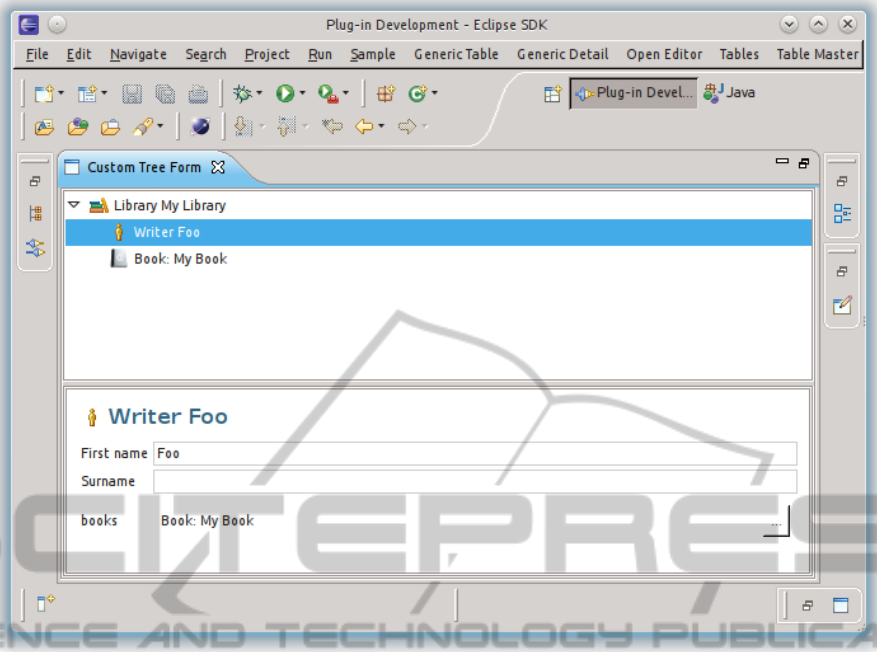

Figure 3: A view with a nested tree editor and a form.

Teneo and CDO. Teneo EMF resources can already be used seamlessly with existing EMF editing and UI technologies, thus, Teneo resources can be integrated in Emf Components transparently as well.

Things are instead more complicated for CDO resources. Indeed, CDO aims at scalability (CDO also offers transactions capabilities, explicit locking, change notification, and branching/merging over models), thus being able to deal with models of arbitrary sizes, and for this reason setting up a CDO resource requires more work. In order to deal with this, Emf Components internally recognizes CDO resource URIs and uses a dedicated resource loader which automatically creates a transaction (or a view, in case we use read-only widgets) and returns the associated CDO resource. Thus, also CDO resources can be used transparently inside Emf Components, relieving the programmer from all the internal details.

Simply by using a CDO URI format for a resource, all existing code implemented using Emf Components can already be reused for models stored using CDO. Indeed Emf Components was designed with test driven development (Beck, 2003) and agile programming (Martin, 2003) in mind; thus, it must be easy to switch from a development/testing environment to a production one. With the mechanism described above to hide the details of the actual storage of a model, it is straightforward to use, for instance, an in memory model during development stage, and an actual CDO model for the production.

4 RELATED WORK AND CONCLUSIONS

Most of the projects that deal with customization in the context of EMF are based on the generative approach. Indeed, EMF itself relies on JET (a template system) for code generation. Thus, the programmer could also provide custom templates to drive the generation of Java code performed by EMF. Other similar technologies (like EEF and EGF), which are available from Eclipse Modeling Framework Technology (EMFT) (Eclipse Modeling Framework Technology, 2012) are based on the generative approach. The Emf Components framework can be seen as complementary to the generative approaches. Our framework is also orthogonal to GUI builders, since Emf Components is more related to bridging the models to the abstract editing parts, and not to the actual GUI framework.

The work that is closest to our proposal is the EMF Client Platform (ECP) (EMF Client Platform, 2012), a framework to build EMF-based client applications. The goal of ECP is to provide a very quick mechanism to create an application based on a given EMF model. The main difference between our framework and ECP is that Emf Components aims at providing

many fine grain components to build an application based on EMF, while ECP already provides an application ready to use. With this respect, our customizations, and our components, are easier to reuse since they are smaller. With Emf Components the programmer has the complete freedom on how to build the application, while ECP provides a more rigid template. Furthermore, the customizations inside ECP do not rely on dependency injection and the declarative style (based on polymorphic dispatching). We are currently investigating about the reuse of our Emf Components inside ECP, and also about the possibility of rewriting some parts of ECP itself by relying on Emf Components.

We believe the use of a dependency injection is not an optional choice: the degree of reuse and customization is achieved without sacrificing consistency throughout the instances of the application, since the dependency injection framework itself will make sure that the instances of classes will be created according to the specified bindings. Furthermore, also the new forthcoming version of Eclipse (e4) will rely on dependency injection, thus, we are confident that some drawbacks in usage of UI classes (like the `Composite`, as illustrated in Section 3.3) will be avoided.

We are currently developing in Xtext (Itemis, 2012) a DSL for the customization aspects of Emf Components. The idea is that writing a single configuration file with this DSL the programmer can customize all the aspects of Emf Components applications (presented in Section 3); then, the framework will generate the corresponding Java files, Guice module bindings and the executable extension factory. This should speed even more the development of EMF applications using the Emf Components framework.

We are also planning to integrate other EMF technologies in the Emf Components framework, like queries, transactions and advanced validation mechanisms (though the standard EMF validation mechanisms are already handled inside Emf Components).

ACKNOWLEDGEMENTS

The author is grateful to all the people from RCP Vision for their help, support and contribution to the development of Emf Components.

REFERENCES

Beck, K. (2003). *Test Driven Development: By Example*. Addison-Wesley.

Bettini, L., Capecchi, S., and Venneri, B. (2009). Featherweight Java with Dynamic and Static Overloading. *Science of Computer Programming*, 74(5-6):261–278.

Castagna, G. (1997). *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser.

Clayberg, E. and Rubel, D. (2008). *Eclipse Plug-ins*. Addison-Wesley, 3rd edition.

Eclipse Modeling Framework Technology (2012). Eclipse Modeling Framework Technology (EMFT). http://www.eclipse.org/modeling/emft/.

EMF Client Platform (2012). EMF Client Platform. http://www.eclipse.org/emfclient.

EMF.Edit (2004). EMF.Edit. http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.Edit.html.

Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection pattern. http://www.martinfowler.com/articles/injection.html.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Google (2012). Google Guice. http://code.google.com/p/google-guice.

Itemis (2012). Xtext. http://www.eclipse.org/Xtext.

Martin, R. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

Schindl, T. (2009). EMF Databinding. http://www.eclipse.org/resources/resource.php?id=511.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition.

Sun Microsystems, Inc. (2007). JSR 308: Annotations on java types. http://jcp.org/en/jsr/detail?id=308).

Vlissides, J. (1996). Generation Gap [software design pattern]. *C++ Report*, 8(10):12, 14–18.