Developing User Interfaces with EMF Parsley

Lorenzo Bettini[*]
Dipartimento di Informatica, Università di Torino, Torino, Italy
bettini@di.unito.it

Keywords: EMF, Modeling, Eclipse, User Interface, DSL.

Abstract: In this paper we describe the main features of EMF Parsley, a new Eclipse project for implementing applications based on the Eclipse Modeling Framework (EMF). EMF Parsley aims at complementing the EMF reflective mechanisms with respect to rapidly creating user interfaces based on models, without having to deal with internal details and setup code. In particular, EMF Parsley uses injection mechanisms to easily customize all the aspects of such applications. Moreover, it provides a set of reusable user interface components like trees, tables and detail forms that manage the model with the introspective EMF capabilities, together with reusable views, editors and dialogs. Besides project wizards, to easily create projects based on EMF Parsley, the main developing tool is a DSL, implemented with Xtext/Xbase, that provides a rapid customization mechanism.

1 INTRODUCTION

The *Eclipse Modeling Framework* (EMF) (Steinberg et al., 2008) simplifies the development of complex software applications with its modeling mechanisms. However, the development of user interfaces based on EMF still requires an extensive setup and the knowledge of many internal details. For these reasons, in this paper, we present a new Eclipse framework, EMF Parsley; this has been recently approved as an official Eclipse project and it is in its incubation phase.

EMF Parsley provides a framework to easily develop user interfaces based on EMF models. The framework hides most of the complexity of internal details. Creating a JFace viewer and connecting it to an EMF resource, usually requires a few lines of code. Furthermore, the customization of specific behaviors is also easy thanks to the use of Google Guice, a *Dependency Injection* framework, and thanks to the polymorphic method dispatch mechanism that allows to write cleaner declarative code. This maximizes code reuse and promotes a programming style where the classes implemented by the programmer are usually very small and deal with only a few aspects. The framework comes with some UI widgets to be used out-of-the-box (including trees, tables, dialogs and forms, and view and editor parts).

We also provide a DSL, implemented in Xtext (Itemis, 2014; Eysholdt and Behrens, 2010; Bettini, 2013a), for making the use of our framework easier: customizations can be specified in a compact form in a single file.

EMF Parsley is the evolution of EMF Components (Bettini, 2012; Bettini, 2013b); EMF Components was a first experiment with building automatically applications based on EMF models. EMF Parsley is a huge evolution with this respect: we focused to make the initial setup easier (using project wizards); the same holds for customizations of all the behaviors, in particular the DSL helps a lot with this respect.

The paper is structured as follows: in Section 2 we introduce our motivations, while in Section 3 we describe the design choices and the main features of our framework by examples. Section 4 describes our DSL (and some examples) and Section 5 presents some additional features of EMF Parsley. Section 6 concludes the paper with related work and hints for future directions.

2 MOTIVATIONS

EMF.Edit (Steinberg et al., 2008) is an EMF framework with generic reusable classes for editing EMF models. The programmer can use these mechanisms to implement UI parts for the EMF models. The problem is that all these mechanisms have to be setup and

[*]The paper was partly supported by RCP Vision, www.rcp-vision.com.

```
adFact = new ComposedAdapterFactory(
        ComposedAdapterFactory.Descriptor.Registry.
        INSTANCE);
adFact.addAdapterFactory(new
        ResourceItemProviderAdapterFactory());
adFact.addAdapterFactory(new
        MyModelItemProviderAdapterFactory());
adFact.addAdapterFactory(new
        ReflectiveItemProviderAdapterFactory());

BasicCommandStack cmdStack = new
        BasicCommandStack();
cmdStack.addCommandStackListener(new
        CommandStackListener(){...});

edDom = new AdapterFactoryEditingDomain(
        adFact,cmdStack,...);

Tree tree = new Tree(composite, SWT.MULTI);
TreeViewer viewer = new TreeViewer(tree);

viewer.setContentProvider(new
        AdapterFactoryContentProvider(adFact));
viewer.setLabelProvider(new
        AdapterFactoryLabelProvider(adFact));
viewer.setInput(edDom.getResourceSet());

new AdapterFactoryTreeEditor(viewer.getTree(),
        adFact);
```

Listing 1: An example of typical use of EMF.Edit.

initialized correctly in order to achieve the desired features. This initialization phase takes many lines of code, and usually requires some deeper knowledge of EMF internals. In Listing 1 we show the typical Java code one needs to write to setup a viewer with EMF.Edit. As we will show in the next sections, our goal is to factor out this boilerplate code in such a way that UI components can be setup with only a few lines of code.

EMF has some generation mechanisms for the user interface. However, there are many problems when using this generated code; we will detail such problems in the following.

EMF generation mechanisms are based on Javadoc comments `@generated` for fields, methods and classes. Future generations will overwrite all the previously generated Java code elements unless the programmer removes that `@generated` from specific declarations (or replaces it with `@generated NOT`).

In Listing 2 we show an example of customization of labeling in a class generated by EMF (related to the classic EMF Library example). In order to specify the image for instances of `Book` we need to go into the generated `ItemProvider` class, and specify the path of the image (note the presence of other distracting and internal details like `overlayImage`,

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

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

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

`ResourceLocator`, etc.). Furthermore, if we want to customize the labels and images for all the elements of the model, we need to modify all generated `ItemProvider` classes. With this respect, we want to provide in EMF Parsley a much easier customization mechanism.

A problem with the `@generated` and `@generated NOT` mechanism is that it is difficult to keep track of custom modifications to the generated code. Indeed generated and custom code are mixed together in the same Java file. This makes such code much harder to maintain. A more appealing solution is 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. We will follow this pattern in the DSL (as we will show in Section 4). By using this pattern, generated and custom code are clearly separated in different files and are then much easier to maintain (in our case, they are also stored in different source folders). Keeping generated and custom code separate also makes it easier to store them in source control systems (e.g., Git, SVN, etc.); not to mention that when using a continuous integration build system, generated code needs not to be checked in the version control system, since it can be generated during the build. This significantly reduces the size of the source control system database.

For all these reasons we felt the need of creating a new framework built on top of EMF, to avoid all the above issues and to make the creation, customization and maintenance of EMF applications much easier, as we will show in the next sections.

3 EMF PARSLEY

In this Section we present the most relevant features of EMF Parsley through examples; this Eclipse project is still in the incubation phase, thus more features will be provided in the future (see also Section 6 for future work).

3.1 Overview

Our main design choice in developing EMF Parsley was to split responsibilities into small classes; this way, customizing a single aspect of UI parts does not require to subclass the parts themselves, but only to customize the class related to that specific aspect.

In order to handle the customized behaviors in a consistent way, we heavily use Google Guice, a *Dependency Injection* framework. With respect to manual implementation of existing patterns (Gamma et al., 1995), with dependency injection frameworks it is much easier to keep the desired consistency, and the programmer needs to write less code. Custom implementations of specific aspects are injected in the framework, so that all components using that aspect will be assured to use the custom version. Google Guice uses Java annotations, `@Inject`, for specifying the fields that will be injected, and a *module* is responsible for configuring the bindings for the actual implementation classes. We provide project wizards that perform all Guice injection initial setup.

The main inspiration for dealing with customized injected code in EMF Parsley comes from Xtext (Itemis, 2014; Bettini, 2013a), a framework for the development of programming languages, where Google Guice is heavily used. EMF Parsley uses the enhancements that Xtext added to Guice's module API: an abstract base class reflectively looks for methods with a specific signature in order to find declared bindings. These methods have the shape `bind<ClassName>` where `ClassName` is the name of the class for which we want to specify a binding. An example of custom bindings using this reflective mechanisms is shown in Listing 3. Of course, the programmer can also use the standard Google Guice mechanisms for specifying the bindings, if he prefers to. (When using the EMF Parsley DSL, all these bindings will be generated automatically.)

The main steps to use EMF Parsley can be summarized as follows:

1. Create an Eclipse project using our project wizards;
2. Create a standard Eclipse UI part, e.g., a view or an editor;

```
class MyModule extends EmfParsleyGuiceModule
      {
  public Class<? extends ResourceLoader>
        bindResourceLoader() {
    return MyResourceLoader.class;
  }
  public Class<? extends ILabelProvider>
        bindILabelProvider() {
    return MyLabelProvider.class;
  }
  public Class<? extends IContentProvider>
        bindIContentProvider() {
    return MyContentProvider.class;
  }
  ...
}
```

Listing 3: A Guice module with bindings.

3. "Inject" one of our classes;
4. Customize specific aspects and configure the Guice module to use the custom implementations.

In the rest of this section we will show some examples; in Section 4 we will concentrate on the use of our DSL to easily write customizations.

3.2 UI Components

JFace provides specific viewers for trees, tables, lists, etc. These viewer classes are not intended to be subclassed. Indeed, they are parameterized over providers. For example, the content provider specifies the contents based on the input; similarly, the label provider is used for getting a textual representation of the elements to be shown by the viewer.

In EMF Parsley we provide an `ViewerInitializer` to initialize a viewer with all the EMF mechanisms, but hiding the details from the programmer. In Listing 4 we show an example of a view with a tree viewer that uses an injected `ViewerInitializer`. It uses such instance to initialize the tree viewer (it can initialize it based on an EMF Resource, or an EObject). All the initialization details are carried on transparently by our framework. We invite the reader to compare this code with the one shown in Listing 1.

This can be seen as an "Hello World!" example for EMF Parsley; with only a few instructions we create an Eclipse view, that can be used in an existing application.

We provide other widgets to be reused in views and editors. We will concentrate only on some of them. One of the most useful is the *form* composite `FormDetailComposite` that shows the details of an EObject in a SWT form, and allows to edit such details.

```
public class MyView extends ViewPart {
  @Inject ViewerInitializer initializer;

  @Override
  public void createPartControl(Composite parent)
       {
    viewer = new TreeViewer(parent, ...);
    initializer.initialize(viewer, resource);
  }
```

Listing 4: Initialization of a viewer with EMF Parsley.

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

  @Inject EmfFormCompositeFactory factory;
  FormDetailComposite detailForm;

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

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

Listing 5: Using the `FormDetailComposite`.

In Listing 5 we show a possible use of `FormDetailComposite`: we create a view that 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 showing how easy it is to create this composite and set it up with EMF Parsley. All the other code in Listing 5 has to do with Eclipse and SWT. This view class is already part of EMF Parsley framework, since it is a typical view that can be reused in applications.

In Figure 1 we show a reusable editor provided by EMF Parsley and the form view implemented in Listing 5 (this shows the currently selected object fields for editing). 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). All these features are automatically handled by the EMF Parsley framework itself. Moreover, any change to the model in any view or editor that is connected to the same model resource will soon be

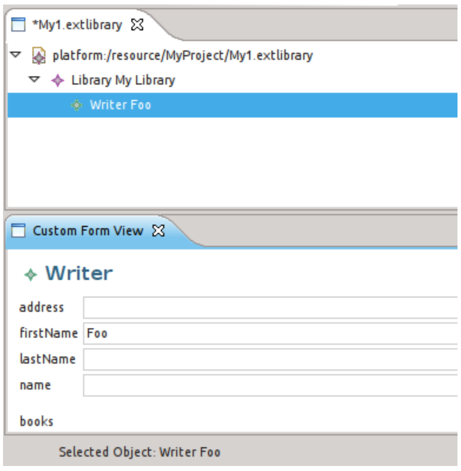

*My1.extlibrary
platform:/resource/MyProject/My1.extlibrary
Library My Library
Writer Foo
Custom Form View
Writer
address
firstName Foo
lastName
name
books
Selected Object: Writer Foo

Figure 1: A tree editor and a form view.

reflected in all the components using that resource; this takes place transparently, since our framework internally uses EMF Databinding, which connects the model and the user interface.

In Figure 1 the viewer and form contents are automatically created by EMF Parsley using EMF reflective capabilities; the programmer can then customize this behavior in many respects, as shown in the next section.

3.3 Customizations

In EMF Parsley we provide customization mechanisms (based on injection) that are easier to use than the standard EMF.Edit framework.

We use the class `PolymorphicDispatcher`, implemented in Xtext, 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] or *multi-methods* (Mugridge et al., 1991). This polymorphic dispatching mechanism does not require to implement a visitor structure (Gamma et al., 1995) since it inspects the available methods using reflection. With this mechanism we provide a declarative way of specifying custom behaviors according to the class of objects of an EMF model.

For example, by implementing a custom `ViewerLabelProvider` (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 as parameters. An example is shown in Listing 6 (using the

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

```
public class MyLabelProvider extends
      ViewerLabelProvider {
  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 6: An example of customization of labeling in EMF Parsley.

```
public class MyCustomModule extends
      EmfParsleyGuiceModule {
  public Class<? extends ILabelProvider>
        bindILabelProvider() {
    return MyLabelProvider.class;
  }
  ...
}
```

Listing 7: Binding the custom label provider.

EMF Library example). This code is much more readable and easier to write than the one of Listing 2 (note also the absence of internal details, especially for the images). Most of the customizations in EMF Parsley follow the same declarative pattern.

Injecting this customization is just a matter of defining the binding in the Guice module, as shown in Listing 7.

4 THE DSL

To enhance the usability of EMF Parsley we developed a DSL with Xtext (Itemis, 2014; Eysholdt and Behrens, 2010; Bettini, 2013a). Xtext is a framework for the development of programming languages: it generates all the typical artifacts for a fully-fledged IDE on top of Eclipse. With our DSL we can easily specify and customize the aspects of the viewers and composites of our framework without writing Java code and without writing the corresponding Guice module bindings.

This DSL uses Xbase (Efftinge et al., 2012), a reusable Java-like expression language, completely integrated with the Java type system (including generics). This means that all the existing Java libraries can be used in our DSL. From Xbase the DSL "inherits" a rich Java-like syntax for expressions; Java programmers will note the similarities between Xbase and Java, though Xbase removes most of the "syntactic noise" from Java (e.g., types of variable declarations can be automatically inferred) and provides more advanced features (e.g., lambdas). Xbase also provides some syntactic sugar, like extension methods and a concise getter/setter syntax[3].

```
module MyCustomModule {
  labelProvider {
    text {
      Book b -> "Book: " + b.title
      Borrower b -> "Borrower: " + b.firstName
    }
    image {
      Book -> "book.png"
    }
  }
  featureCaptionProvider {
    text {
      Person : firstName -> "First name"
      Person : lastName -> "Surname"
    }
  }
  featuresProvider {
    features {
      Library -> name, address
      Person -> firstName, lastName, address
      Writer -> firstName, lastName, books
    }
  }
}
```

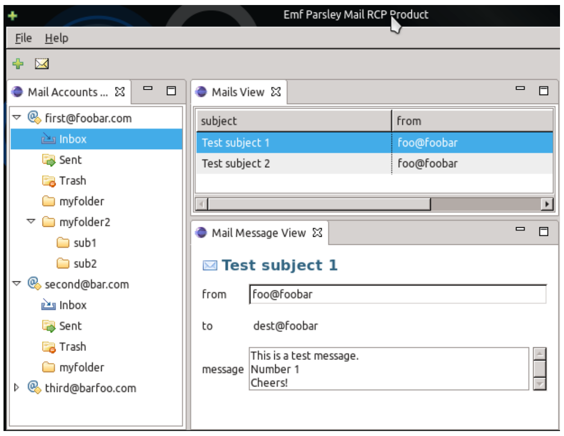
Listing 8: An example of module definition in EMF Parsley DSL.

With the EMF Parsley DSL we only need to define a **module** that will correspond to a Guice module in the generated Java code; inside this **module** we specify customizations (we described some in Section 3.3). The DSL will then generate the corresponding Java classes, and the corresponding custom bindings in the generated Guice module. This way, the customizations are specified in a much more compact form and they are all grouped together in a single file (instead of being spread into several Java classes).

Listing 8 shows some customizations using the DSL. With **featureCaptionProvider** we customize the representation of the features of the model, i.e., the captions of the fields in a form or the column headers in a table. Similarly, one might want to specify only a subset of features to be shown in the user interface; we do that using **featuresProvider**. The customization shown in Listing 8 should be self-explanatory.

Thanks to Xbase and its integration with the Java type system, the class names in Listing 8 actually refer to the corresponding Java classes of the model, and from the DSL editor one can directly navigate to their

[3]e.g., `foo(a, b)` can be written as `a.foo(b)`, `a.getFirstName()` as `a.firstName` and `a.setFirstName("foo")` as `a.firstName = "foo"`.



subject	from
Test subject 1	foo@foobar
Test subject 2	foo@foobar

Test subject 1

from foo@foobar

to dest@foobar

message This is a test message.
Number 1
Cheers!

Figure 2: An RCP application implemented with EMF Parsley and its DSL.

sources; the editor also provides syntax highlighting and code completion for all Java types (not to mention automatic import statements insertion). Moreover, the feature names in **featureCaptionProvider** and **featuresProvider** are bound to the corresponding EMF features; also in this case, one can navigate to those features and get code completion for the features of the corresponding EMF class. This implies that the DSL checks that the features specified actually belong to the corresponding EMF class.

As a final example, we show an RCP application implemented with EMF Parsley and its DSL; the resulting application is shown in Figure 2. This is similar to the classic Mail RCP application example that comes with Eclipse. It does not aim at implementing an actual email client, but it concentrates on the user interface part based on an EMF model.

The Eclipse projects are created using the EMF Parsley project wizards that take care of creating the initial Java code for Guice.

All the views, forms and tables shown in this example are provided by EMF Parsley; we only need to specify the customizations. EMF Parsley fosters the use of a separate project for each Guice module, so that all the classes in the same project use the same customizations. In the Mail application shown above, each view is implemented and customized in its own project.

Listing 9 shows the specification of the view for the accounts (the view on the left in Figure 2). In the **module** we also specify the **parts** section that will correspond to the Eclipse extension point for views; the DSL compiler will also generate the corresponding `plugin.xml` in the current project.

Thanks to Xbase, we are able to specify, for instance, the **labelProvider** with a Java-like syntax, as shown in Listing 9. Thus, using the DSL instead

```
module accountsview {
  parts {
    viewpart {
      viewname "Mail Accounts View"
      viewclass AccountsView
    }
  }
  labelProvider {
    image {
      Account -> "account.gif"
      Folder -> {
        switch (name) {
          case "Inbox" : "inbox.gif"
          case "Sent" : "sent.png"
          case "Trash" : "trash.png"
          default: "folder.gif"
        }
      }
    }
    text {
      Account -> email
      Folder -> name
    }
  }
  viewerContentProvider {
    children {
      Folder -> subfolders // don't show emails
    }
  }
...
```

Listing 9: Customization of the accounts view.

of Java does not prevent from writing involved customizations. This example also shows the customization of the content provider for the viewer; in this case we want to avoid the mails of a folder to be shown in tree, thus we specify that the children of a folder are only subfolders. Also in this case, this customization is easy and compact (and the editor will provide content assist for Java types).

The application then uses a table view provided by EMF Parsley; this view automatically shows the mails of the selected folder in the tree of the accounts view. By default, this view would show all the features of the mail objects. Since we want only the "subject" and the sender (feature "from") to be shown in this table, we specify this customization in the corresponding module:

```
  featuresProvider {
    features {
      Mail -> subject, from
    }
  }
```

We then use a form view that shows the currently selected mail in the table view. Also in this case we want to specify some customizations (Listing 10). For example, we want a different caption for the "recipi-

```
...
featureCaptionProvider {
  text {
    Mail : recipients -> 'to'
  }
}
formControlFactory {
  control {
    Mail : message -> {
        createText("", SWT.MULTI, SWT.
            BORDER, SWT.WRAP, SWT.
            V_SCROLL)
      } target observeText(SWT.Modify)
  }
}
...
```

Listing 10: Customization of the form.

ents" feature; we specify this using **featureCaptionProvider**. Moreover, we want to show the body of the mail (the feature "message") in a text area instead of a single line text. Note that in this case we have full control both on the creation of the SWT control and on the corresponding EMF Databinding mechanism (the **target** section is indeed used by EMF Parsley to perform databinding).

We refer to `http://projects.eclipse.org/projects/modeling.emf-parsley` for the complete source code of the example. Here, we only want to stress that for implementing this application we only had to write an average of 30 lines of EMF Parsley DSL code for each view (and in some cases a few small Java classes).

5 OTHER FEATURES

Teneo and CDO are two mainstream frameworks to persist EMF models on databases. Teneo resources can already be used seamlessly in EMF Parsley. On the contrary, CDO resources require some additional work. Indeed, CDO offers scalability, transactions capabilities, explicit locking, change notification, and branching/merging over models. EMF Parsley automatically recognizes CDO resource URIs and uses a specific resource loader that 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 with EMF Parsley, again, relieving the programmer from all the internal details.

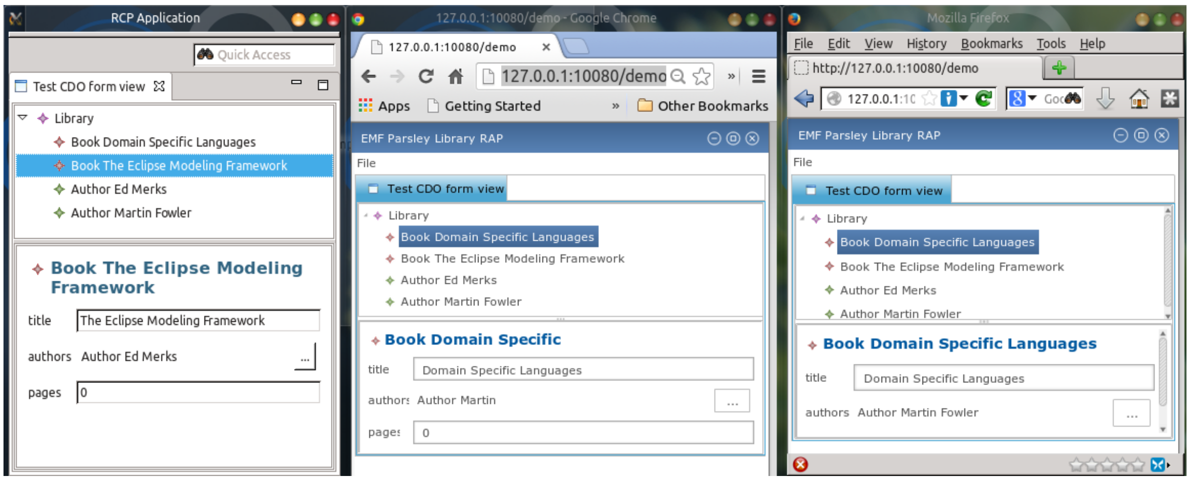
Another important feature of EMF Parsley is its support for RAP (Remote Application Framework) (Eclipse Foundation, 2014). RAP allows to seamlessly port an existing RCP application to the web. In particular, with its single-sourcing features the same application developed on the desktop can run in the browser (RAP supports all relevant web browsers, without any add-ons required). All EMF Parsley views, editors and internal mechanisms are implemented with single sourcing techniques so that they are available also for RAP applications. Figure 3 shows an RCP application implemented with EMF Parsley on the left; the same application (without any modification) is ready for RAP, it just needs to use the RAP version of EMF Parsley bundles; the figure shows the web version running on Chrome (center) and on Firefox (right). All of them are using a CDO resource, thus they act on the same data.

The use of Xbase implies another important feature: when running the generated Java code in debugging mode in Eclipse, we can choose to debug directly the original Parsley DSL code (it is always possible to switch to the generated Java code). Indeed a well-known problem of DSLs that generate Java code is that for debugging, the programmer has to debug the generated code that is usually quite different from the original program; our DSL implementation does not have this drawback.

6 RELATED WORK AND CONCLUSIONS

Differently from many generative frameworks for EMF (such as, e.g., JET or EEF and EGF, which are available in EMFT (Eclipse Modeling Framework Technology, 2012)), EMF Parsley does not generate code for the widgets and does not require the programmer to modify that generated code: the programmer uses the provided components and injects the customized code. The only point where we generate code is in the DSL (Section 4), but we generate code only for the customizations, not the code of the widgets.

EMF Parsley is also much different from GUI generators, like, for instance, WindowBuilder. WindowBuilder generates Java classes for GUI parts, and provides a visual editor with rich tooling to design GUI parts visually. Since WindowBuilder directly generates Java code, and not an intermediate representation of the GUI part, it is also capable of dynamically inspecting such Java code and updating the visual editor accordingly. Thus, the programmer can insert manually written sections in such generated Java code. As we stressed throughout the paper, EMF Parsley follows a completely orthogonal approach: it provides GUI parts that are ready to use, and allows the programmer to "inject" customizations of behavioral aspects. With that respect, EMF Parsley and its DSL share many design and goals with other reflective and



Library
Book Domain Specific Languages
Book The Eclipse Modeling Framework
Author Ed Merks
Author Martin Fowler

Book The Eclipse Modeling Framework
title The Eclipse Modeling Framework
authors Author Ed Merks
pages 0

Book Domain Specific
title Domain Specific Languages
authors Author Martin
pages 0

Book Domain Specific Languages
title Domain Specific Languages
authors Author Martin Fowler

Figure 3: An RCP application implemented with EMF Parsley and the same application ported to the web using RAP (in execution in two different browsers).

meta-model based frameworks such as, for instance, Magritte (Renggli et al., 2007) (although they target different programming languages, since Magritte is based on Smalltalk): customizations in one single place in the source-code (i.e., our DSL module) and easy abstractions without having to know internal details.

The work that is closest to EMF Parsley is the EMF Client Platform (ECP) (EMF Client Platform, 2014), a framework to build EMF-based client applications. ECP provides tools to create an application based on a given EMF model. The main difference between EMF Parsley and ECP is that we aim at providing many small blocks 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. Another important difference is that the customizations inside ECP are not based on dependency injection and they do not use the declarative style (based on polymorphic dispatching). We are currently investigating possible interactions and cooperations between EMF Parsley and ECP, for instance we are considering to rewrite some parts of ECP itself by relying on EMF Parsley and, on the other hand, we could reuse some of the ECP parts in EMF Parsley.

EMF Parsley can be used in existing applications, and it does not require to be used from the very beginning; in particular, EMF Parsley fosters an incremental adoption in existing applications. This is possible because EMF Parsley honors possible manual implementations that use EMF.Edit in legacy applications. Indeed EMF Parsley builds on top existing EMF mechanisms (like EMF.Edit) but it does not replace them.

We are also working on the integration of other EMF technologies in our framework, like queries, transactions and advanced validation mechanisms (though the standard EMF validation mechanisms are already handled inside EMF Parsley).

ACKNOWLEDGEMENTS

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

REFERENCES

Bettini, L. (2012). EMF Components - Filling the Gap between Models and UI. In *ICSOFT*, pages 34–43. SciTePress.

Bettini, L. (2013a). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing.

Bettini, L. (2013b). Rapidly Implementing EMF Applications with EMF Components. In *Software and Data Technologies*, Communications in Computer and Information Science. Springer.

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.

Eclipse Foundation (2014). RAP, Remote Application Platform. `http://eclipse.org/rap`.

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

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., and Hanus, M. (2012). Xbase: Implementing Domain-Specific Languages for Java. In *GPCE*, pages 112–121. ACM.

EMF Client Platform (2014). EMF Client Platform. `http://www.eclipse.org/ecp`.

Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In *SPLASH/OOPSLA Companion*, pages 307–309. ACM.

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

Itemis (2014). Xtext. `http://www.eclipse.org/Xtext`.

Mugridge, W., Hamer, J., and Hosking, J. (1991). Multi-Methods in a Statically-Typed Programming Language. In *ECOOP*, volume 512 of *LNCS*, pages 307–324. Springer.

Renggli, L., Ducasse, S., and Kuhn, A. (2007). Magritte A Meta-driven Approach to Empower Developers and End Users. In *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 106–120. Springer.

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

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