The EMF Parsley DSL for Developing EMF Applications

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

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

Abstract: The Eclipse Modeling Framework (EMF) is the official Eclipse modeling framework. It provides code generation facilities for building tools and applications based on structured data models. The Eclipse project EMF Parsley enhances the EMF reflective mechanisms in order to make the development of EMF applications easier by hiding most EMF internal details and by using dependency injection for customizing all the aspects of such applications. In this paper we show the main features of the EMF Parsley DSL that aims at making the development of EMF applications even easier and faster. The DSL is built with Xtext and Xbase, thus it features full Eclipse IDE support and full interoperability with Java.

1 INTRODUCTION

The *Eclipse Modeling Framework* (EMF) (Steinberg et al., 2008) is the official Eclipse modeling framework with code generation facilities for building tools and applications based on structured data models.

EMF Parsley, an official Eclipse project[2], is a framework to quickly and easily develop user interfaces based on EMF models. EMF Parsley aims at filling a few gaps in EMF development in general: the development of user interfaces based on EMF still requires an extensive setup and the knowledge of many internal details. On the contrary, EMF Parsley hides most of the complexity of such internal details. The framework provides some UI components that can be used out-of-the-box (including trees, tables, dialogs and forms, and views and editors). Programmers can use these widgets as the building blocks for their applications and customize them, or use them as a reference implementation to build their own components based on our framework. Specification of custom behaviors is based on the types of elements of the EMF model, using polymorphic method dispatch mechanism that allows to write cleaner and declarative code. All custom code is then "injected" in the framework with Google Guice, a *Dependency Injection* framework.

*Work partially supported by MIUR (proj. CINA), Ateneo/CSP (proj. SALT), and ICT COST Action IC1201 BETTY. The paper was partly supported by RCP Vision, www.rcp-vision.com.

[2]https://www.eclipse.org/emf-parsley.

The very first version of EMF Parsley was first presented in (Bettini, 2014). The framework has evolved a lot since then, concerning its usability, mostly thanks to the EMF Parsley DSL, implemented in Xtext: customizations can be specified in a compact form in a single file.

This paper extends previous presentations since it concentrates on the DSL and all its new features that have been recently added. The paper also describes in more details the architecture of EMF Parsley in such a way that we think its design choices could be reused also in other frameworks targeting the same goals. Since the DSL is implemented with Xtext (Bettini, 2013a), it comes with a powerful and professional Eclipse IDE support.

Moreover, we also leverage Xbase (Efftinge et al., 2012), a reusable expression language that can be used in Xtext DSLs.

In particular, Xbase's type system is compliant with the Java type system (including generics). This also implies that a DSL that uses Xbase will have access to all the existing Java libraries.

The paper is structured as follows: in Section 2 we describe the architecture and the main ingredients of our framework. Section 3 describes the EMF Parsley DSL with some examples. Section 4 describes some related work. Section 5 concludes the paper and hints for future directions.

Bettini, L.
The EMF Parsley DSL for Developing EMF Applications.
DOI: 10.5220/0005743803010308
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 301-308
ISBN: 978-989-758-168-7
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 EMF PARSLEY

In this Section we describe the overall architecture of EMF Parsley. This Eclipse project is still in the incubation phase and it will ship the first stable release soon. The main design choice of EMF Parsley is to split responsibilities into small classes in order to make it easy and straightforward to customize each single aspect of an EMF application. Differently from the standard EMF application development workflow, in EMF Parsley the programmer is not required to modify monolithic generated classes.

Dependency Injection. In order to handle the customized behaviors in a consistent way, we heavily use Google Guice, a *Dependency Injection* framework.

Custom implementations of specific interfaces and classes are injected in the framework's classes, so that all the other classes in the framework 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 took inspiration from Xtext, where Google Guice is heavily used. EMF Parsley uses the enhancements that Xtext added to Guice's module API: Google Guice bindings are specified by declaring methods that have the shape `bind<ClassName>` where `ClassName` is the name of the class for which we want to specify a binding.

Of course, the programmer can also use the standard Google Guice mechanisms for specifying the bindings. As we will see later, when using the EMF Parsley DSL, all these bindings will be generated automatically.

EMF.Edit. EMF.Edit (Steinberg et al., 2008) is the part of the EMF framework with generic reusable classes for editing EMF models.

It provides a command framework, with a set of generic command implementation classes for building editors that support undo and redo mechanisms.

The problem is that all these mechanisms have to be setup and initialized correctly in order to achieve the desired behavior. This initialization phase takes many lines of code, and usually requires some deeper knowledge of EMF internals.

For this reason, EMF Parsley is built on top of EMF.Edit, but it hides all the above mentioned internal details and it automatically sets all EMF.Edit mechanisms. In particular, the customizations that are specified using EMF Parsley have the precedence, and, in the absence of customizations, the default behaviors are delegated to EMF.Edit.

```
@Inject ViewerFactory initializer;

TreeViewer viewer = new TreeViewer(parent, ...);
initializer.initialize(viewer, resource);
```

Listing 1: Initialization of a viewer with EMF Parsley.

```
public class MyLabelProvider extends ViewerLabelProvider {
  public String text(Book book) {
    return "Book: " + book.getTitle();
  }
  public String image(Book book) {
    return "book.png";
  } // other customizations
}
```

Listing 2: An example of customization of labeling in EMF Parsley.

This has several benefits: (i) We do not need to reimplement all the reflective mechanisms of EMF.Edit, which already provides good defaults; (ii) EMF Parsley applications seamlessly interoperate with existing EMF.Edit customizations; (iii) This interaction fosters an incremental or partial porting of an EMF application to EMF Parsley. Initializing a tree viewer with EMF Parsley is just a matter of a few lines of code, Listing 1.

Declarative Customizations. In EMF Parsley we provide declarative customization mechanisms 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*.

This enables 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 2 (using the EMF Library example).

This code is much more readable and easier to write than the usual `instanceof` cascades and casts. Note also that we only need to specify the image file name, and EMF Parsley will automatically retrieve such file from the "icons" directory of the containing Eclipse project and create the image object. 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 3.

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

Listing 3: Binding the custom label provider.

Reusable UI Components. EMF Parsley provides an infrastructure of Java classes and the injection mechanisms to configure any Eclipse UI component, e.g., views and editors. It also provides some of these components that can be used out-of-the-box and that can be easily customized. We provide Eclipse editors for editing EMF models. We also provide views that can also edit EMF models.

Such views can represent EMF models with a tree, a table, a form, or a combination of them. Similarly, we provide views that react on selection, that is, they show an EMF object that is selected in another view or editor.

Our implementations also take care of notifications occurring in the underlying EMF model, so that we can handle "dirty" state in the editors and views and we also ensure that all the UI elements are automatically updated if they share the same EMF model.

Undo and Redo mechanisms are also automatically handled by EMF Parsley.

3 THE DSL

Although EMF Parsley can be used directly from Java, the DSL makes the use of the framework and the development of EMF applications with EMF Parsley even easier.

With our DSL we can easily specify and customize the aspects of UI components of our framework without writing Java code and without writing the corresponding Guice module bindings. The DSL compiler will then generate the corresponding Java classes.

Xbase. The DSL uses Xbase (Efftinge et al., 2012) for the expression language. Xbase is a reusable Java-like expression language, completely interoperable with the Java type system (including generics). This means that all the existing Java libraries can be used in our DSL. Java programmers will be able to easily learn the Xbase language. Although Xbase expressions are Java-like, Xbase removes most of the "syntactic noise" from Java, with type inference, syntactic sugar for getters/setters and *extension methods*, which simulate adding new methods to existing types without modifying them. Xbase provides *lambda expressions*, which have the shape `[ param1, param2, ... | body ]` The types of parameters can be omitted if they can be inferred from the context. For these reasons, Xbase expressions are quite compact, very readable, and look like they are written in an untyped language.

```
module org.eclipse.emf.parsley.examples.librarytreeform {
  parts {
    viewpart views.librarytreeform {
      viewname "My Library Tree Form"
      viewclass SaveableTreeFormView
    }
  } ...
```

Listing 4: An example of module definition in EMF Parsley DSL.

Specification Structure. A specification in EMF Parsley DSL, i.e., an input file, consists of a `module` that will correspond to a Guice module in the generated Java code. Inside this `module` we specify customizations (corresponding to the Java customizations we sketched in Section 2). Each customization has its own specific section, as we will see in the rest of the section.

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.

EMF Parsley provides some Eclipse project wizards to create projects with an initial setup for the DSL. These wizards also provide initial templates for specific views (e.g., tree, tree with form, table, etc.).

Besides customizations, in the module one can specify also sections corresponding to Eclipse parts (i.e., views and editors). From such section, the DSL compiler will generate the `plugin.xml` with the Eclipse *extension points*.

In Listing 4 we show an example of module specification (together with a view part specification) written in the EMF Parsley DSL. These initial contents are created by our project wizard, mentioned above, selecting the "tree form" template.

The part specification resembles the extension point for Eclipse view parts. The "viewclass" refers to the Java type `SaveableTreeFormView` which is one of the views that come with the EMF Parsley distribution (see Section 2). The specification in Listing 4 is enough to have a working tree form view that edits an EMF resource, as shown in Figure 1. The form allows the user to edit the element selected in the tree. In this case, we delegate to EMF.Edit standard reflective behavior, as described in Section 2.

Actually, we still need to tell such view which EMF resource to represent. In order to do that, we do not need to subclass `SaveableTreeFormView` and

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

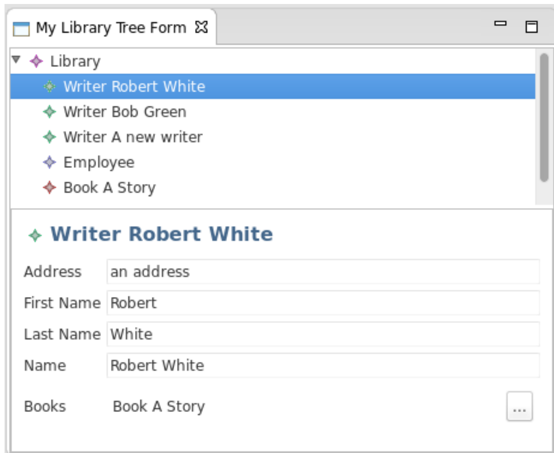

Figure 1: The tree form view.

```
import org.eclipse.emf.common.util.URI
  ...
  configurator {
    resourceURI {
      SaveableTreeFormView -> {
        return URI.createFileURI( System.getProperty("user.home
          ") + "/MyLibrary.library" );
      }
    }
  }
```

Listing 5: Configuring a view.

override a specific method. In fact, such views are configured with an injected `Configurator` and in the DSL specification we only need to declaratively specify this information for such view using a specific section, as shown in Listing 5.

Note that the code block is an Xbase expression, which refers to standard Java classes (the DSL also supports Java-like imports).

Customizing the UI. In the module there are several sections to specify customizations. These sections correspond to the Jface standard components like label provider and content provider and to other specific EMF Parsley components.

For example, we customize the label provider using the section `labelProvider` in the module specification, as shown in Listing 6. Also in the DSL we follow the declarative mechanisms described in Section 2 in a more compact form. Note that the `Writer`, `Book` and `Library` are references to Java types, i.e., the EMF classes for the Library model. This is evident also in the import statement[3]. In the code blocks we specify Xbase expressions. In particular, we use the syntactic sugar and `name` is actually a reference to the method `getName()` in the type `Writer`. This

[3]From now on, we will omit the import statements in the listings.

```
import org.eclipse.emf.examples.extlibrary.*
  ...
  labelProvider {
    text {
      Writer -> { name }
      Book -> { title }
    }
    image {
      Library -> { "library.png" }
      Book -> { "book.png" }
      Writer -> {
        if (books.empty)
          "writer-nobooks.png"
        else
          "writer.png"
      }
    }
  }
```

Listing 6: Customizing the label provider.

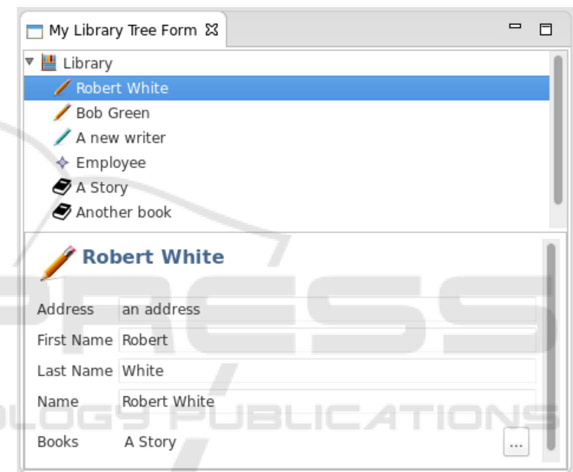

Figure 2: The tree form view with the custom label provider.

syntactic sugar, together with the implicit scope resolution that we enforce in the code blocks in the DSL, enabling compact specifications. In Listing 6 we also show an example of image specification based on the state of the `Writer` object, depending on whether the writer has any associated book. With such customization, the view becomes as shown in Figure 2.

We also stress that all the code blocks are statically typed, thus specifying a method that is not part of the corresponding Java type will result in an error reported by our DSL editor. Similarly our DSL Eclipse editor provides full support for Java integration, such as content assist for Java method references.

There are other sections available in the DSL for customizing the aspects of the UI parts. For example, in Listing 7 we use the section `featureCaptionProvider` to customize the captions of the fields in a form. The result is shown in Figure 3.

Customizing the contents. The EMF Parsley DSL provides specific sections for customizing the

```
featureCaptionProvider {
  text {
    Book : author -> "Written by"
  }
}
```

Listing 7: Customizing the captions.

Figure 3: The caption for a book's author customized.

contents of the EMF model that must be shown in the UI components. By default, we show all the features of an EMF object, in the order they are specified in the EMF model definition. The features and their order can be customized for each single Java type using the `featuresProvider` section. An example is shown in Listing 8 and the result is shown in Figure 4, where only the features of a book specified in `featuresProvider` are shown, in the specified order (differently from the default behavior shown in Figure 3). Also in this case, the feature names in `featuresProvider` are bound to the corresponding EMF features. This implies that the DSL checks that the features specified actually belong to the corresponding EMF class.

We can also customize the contents that are shown in the tree viewer. In Jface such contents are specified using a "content provider". In the EMF Parsley DSL we provide a specific section for customizing the content provider in a declarative way. For example, in Figure 1 we see that by default, the tree viewer shows all the contents of a Library, including employees. If we want to restrict the contents to writers and books we define the `viewerContentProvider` section as shown in Listing 9. Note that we use another Xbase feature: the operator + is overloaded for collections.

```
featuresProvider {
  features {
    Library -> name
    Writer -> firstName, lastName, address, books
    Book -> author, title, category
  }
}
```

Listing 8: Customizing the features to be shown.

Figure 4: The features of a book shown in the form after the customization.

```
viewerContentProvider {
  children {
    Library -> { writers + books }
  }
}
```

Listing 9: Customizing the content provider.

The `viewerContentProvider` can also specify the root objects of the tree. In the Jface content provider these root objects are specified in the `getElements()` method. In the `viewerContentProvider` section this can be achieved using `elements`. For example, in Listing 10 we show an alternative implementation of the content provider and in Figure 5 we show the resulting tree. We specify the root elements of an EMF resource using Xbase lambdas and its additional utility methods: we take the library objects (filtering by type), for each library we take the writers and the books without author (filtering by predicate) and we flatten the result (passing from a collection of collections to a collection). Moreover, we show the books with an author as children of such author.

We would like to stress the simplicity of such specification. Of course, the same result could be achieved by using standard EMF and EMF.Edit mechanisms, but that would require much more code, spread in several Java classes.

Customizing editing mechanisms. EMF Parsley viewers are already configured with context menus. The default behavior is to create the context menus us-

Figure 5: The contents of the tree customized in Listing 10.

```
viewerContentProvider {
  elements {
    Resource -> {
      contents.filter(Library).
        map[writers + books.filter[author == null]].
        flatten
    }
  }
  children {
    Writer -> {
      books
    }
  }
}
```

Listing 10: Customizing the content provider (alternative).

```
menuBuilder {
  menus {
    Writer w -> #[
      submenu("Edit", #[
        actionCopy,
        actionCut,
        separator,
        actionPaste
      ])
    ]
  }
  emfMenus {
    Writer w -> #[
      actionChange("New book", w.eContainer as Library,
      [
        library |
        val book = EXTLibraryFactory.eINSTANCE.createBook
        library.books += book
        book.title = "A new book"
        book.author = w
      ])
    ]
  }
}
```

Listing 11: Customizing the context menu.

ing EMF.Edit reflective capabilities, so that it is possible to create new children of the current object or new siblings.

The EMF Parsley DSL provides a specific section, `menuBuilder`, for easily customizing the context menus according to the currently selected object. An example of customization for the tree nodes representing library writers is shown in Listing 11 and the result is shown in Figure 6.

In `menuBuilder` there are two sections: one for standard editing menu items and one for menu items that are specific of EMF. Menus can be nested, and must be returned in the shape of a list. We use the Xbase syntax for defining lists `#[...]`. We provide default menu entries for “Copy”, “Paste”, etc. and for menu separators. Content assist is available also in this case. For EMF menus we allow the programmer to specify the implementation of a menu, in the shape of an Xbase lambda, using the method `actionChange`. This way, it is easy to specify a menu entry that is meant for changing the model, and, most of all, EMF Parsley will take care to automatically implement Undo/Redo accordingly. In the example of Listing 11 our intention is to provide a menu entry for a selected writer that adds a book to the library with an initial title and sets the selected writer as its author. The second argument passed to the method represents the model object whose changes have to be recorded. The third argument is a lambda that gets as argument the model object. All the changes performed in such lambda can be undone and redone (using the standard “Undo”/”Redo” menu entries). In particular, EMF Parsley will take care of implementing such mechanisms.

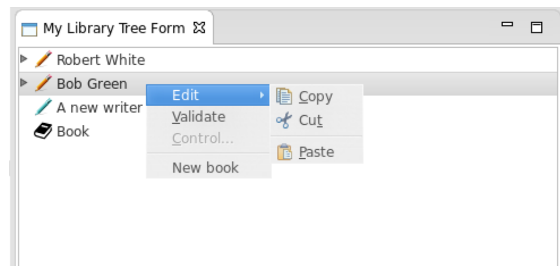

Figure 6: The custom context menu for writers.

Again, implementing all these features manually, without EMF Parsley and its DSL, would require much more work. Not to mention that Undo/Redo functionality should be implemented as well. With that respect, Undo/Redo really needs to be implemented correctly: all actions performed by a menu entry concerning the EMF model have to be undone/redone completely, otherwise the model could be left in an inconsistent way, with dangling references. This does not happen when using our `menuBuilder` and everything is done automatically by the framework.

4 RELATED WORK

EMF Parsley is the evolution of EMF Components (Bettini, 2012; Bettini, 2013b). EMF Components was a first experiment with building applications based on EMF models. EMF Parsley is a huge evolution, since it provides many more features and customizations, not to mention that all these aspects are easy to setup and maintain for the developers. In particular the EMF Parsley DSL helps a lot in this respect.

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 require the programmer to modify the generated code.

In EMF Parsley the generated code is injected in the code of the framework. The EMF Parsley DSL generates Java code that is ready to be injected. Moreover, the DSL allows the developer to easily write and maintain the customizations in a compact form.

EMF Parsley and its DSL share many design and goals with other reflective and meta-model based frameworks such as, for instance, Magritte (Renggli et al., 2007), i.e., customizations in one single place in the source-code (in our case, in a DSL module) and easy abstractions without having to know internal details. However, it is hard to further relate with Magritte since it targets a completely different programming language, Smalltalk.

Sirius (Eclipse Foundation, 2015c) is an Eclipse project to easily create graphical modeling workbenches by leveraging EMF.

Sirius targets users that need to edit an EMF model with a diagram editor, instead of trees and forms. Thus, EMF Parsley and Sirius are somehow complementary. The same holds for other Eclipse frameworks for creating diagram editors, such as, e.g., GMF and Graphiti, which are both part of the Graphical Modeling Project (Gronback, 2008; Eclipse Foundation, 2015a). The EMF Client Platform (ECP) (EMF Client Platform, 2014) is a framework to build EMF-based client applications, thus, it shares with EMF Parsley the main goals. The main difference is that EMF Parsley aims at providing many small blocks to build an application based on EMF, while ECP already provides an application ready to use. Moreover, we believe our components are easier to reuse since they are smaller and the customization is easier thanks to the dependency injection and to our declarative style (polymorphic dispatch).

At the time of writing, we are not aware of any other approaches based on DSLs to specify the structure and behavior of an EMF based Eclipse application.

There are other tools for implementing DSLs and IDE tooling (we refer to (Voelter et al., 2013; Pfeiffer and Pichler, 2008)). We chose Xtext since it is the defacto standard framework for implementing DSLs in the Eclipse ecosystem, it is continuously supported, and it has a wide community. Moreover, Xtext is continuously evolving, and the new 2.9.0 provides the integration in IntelliJ, and the support for programming on the Web. Finally, Xtext provides complete support for Maven and Gradle.

5 CONCLUSIONS

EMF Parsley does not introduce any significant overhead since it reuses all existing EMF technologies and it just adds a thin lightweight layer on top of them. Also dependency injection is not relevant with respect to performance. The DSL is statically typed, thus it rules out possible problems at compile time. This is a big advantage with respect to the reflective approaches described in Section 4, where problems are visible only at run-time.

EMF Parsley is able to interoperate with all the existing EMF frameworks. For example, all the EMF persistence technologies can be seamlessly used with EMF Parsley, such as, Teneo and CDO.

EMF Parsley supports RAP (Remote Application Framework) (Eclipse Foundation, 2015b), which allows developers to seamlessly port an existing Eclipse application to the web. Figure 7 shows the same view we developed in the previous section running as a web application in a browser, using RAP. No modification to the sources are required, we just need to use the RAP version of EMF Parsley bundles and recompile the sources.

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).

In Figure 8 we show a debug session of the view we implemented in the previous section when running as a RAP web application (note also the file name in the thread stack, the "Breakpoint" view and the "Variables" view).

EMF Parsley can be used in existing applications, and it does not require to be used from the very beginning. Thus, an existing application can be ported

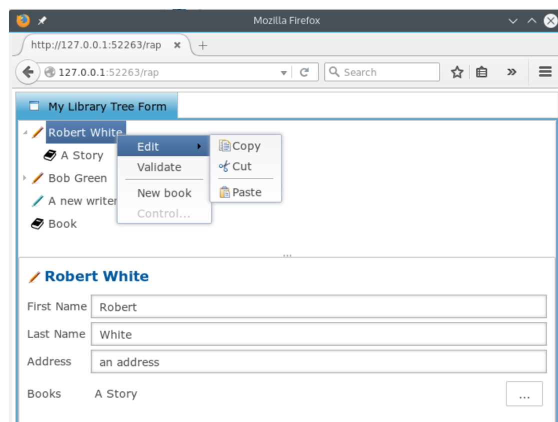

Figure 7: The view implemented with EMF Parsley in the previous section ported to the web using RAP without any source modification.

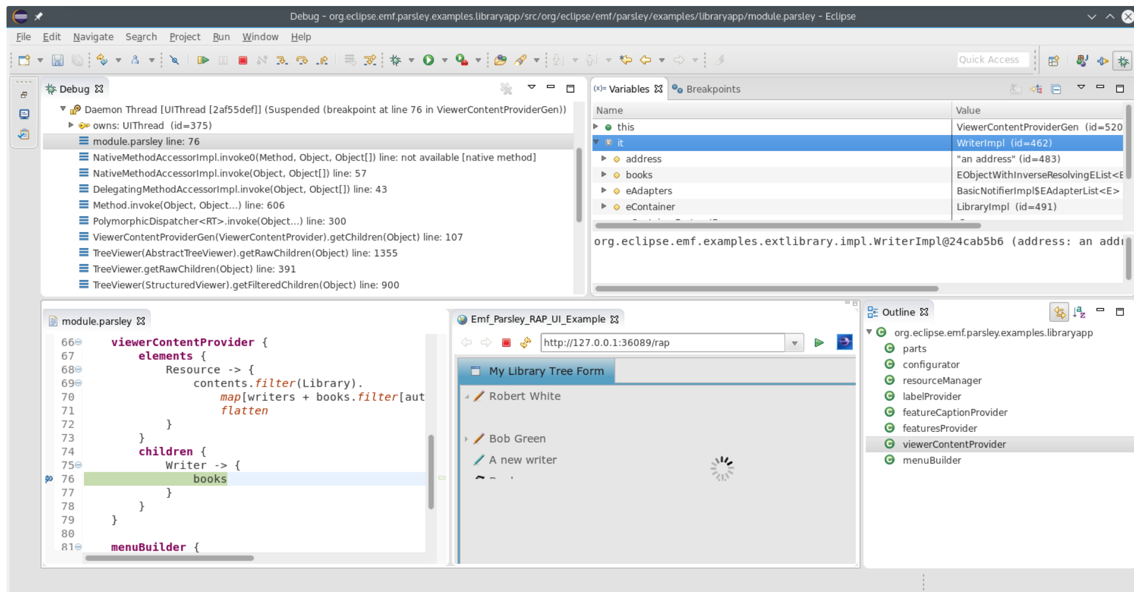

Figure 8: The view implemented with EMF Parsley in the previous section ported to the web using RAP without any source modification.

to EMF Parsley in an incremental way. Moreover, the EMF Parsley DSL is completely interoperable with Java. Thus, a DSL specification and Java code can seamlessly coexist in the same application.

We are working on the integration of other EMF technologies in our framework, like queries, transactions and advanced validation mechanisms (the standard EMF validation mechanisms are already handled inside EMF Parsley). We are also working on the integration with other Web and mobile technologies. With that respect, we have prototype implementations of the integration with JSON frameworks, AngularJS and THyM (The Hybrid Mobile) that enables cross platform mobile development with Eclipse IDE.

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. (2014). Developing User Interfaces with EMF Parsley. In *ICSOFT*, pages 58 – 66. SciTePress.

Eclipse Foundation (2015a). Graphical Modeling Project. http://www.eclipse.org/modeling/gmp.

Eclipse Foundation (2015b). RAP, Remote Application Platform. http://eclipse.org/rap.

Eclipse Foundation (2015c). Sirius. http://eclipse.org/sirius.

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.

Gronback, R. (2008). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley.

Pfeiffer, M. and Pichler, J. (2008). A comparison of tool support for textual domain-specific languages. In *Proc. DSM*, pages 1–7.

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.

Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., Visser, E., and Wachsmuth, G. (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*.