Single-Sourcing for Desktop and Web Applications with EMF Parsley

Lorenzo Bettini[a]

Dipartimento Statistica, Informatica, Applicazioni, Università degli studi di Firenze, 50134 Firenze, Italy

Keywords: EMF, Eclipse, User Interface, Desktop and Web Applications, Single-Sourcing.

Abstract: While Java allows a compiled program to run on different operating systems where a Java Virtual Machine is installed, a Java desktop application cannot be directly executed as a web application and vice-versa. Additional tools and techniques must be employed to achieve a "single-sourcing" mechanism. The Eclipse project EMF Parsley, built on top of EMF, aims to simplify implementing EMF applications by hiding most EMF internal details, providing some reusable UI (User Interface) components, and providing declarative customization mechanisms through a DSL with IDE support. In this paper, we show how EMF Parsley allows the developer to achieve "single-sourcing" for desktop and web applications: the developer can implement a desktop application that can also be deployed as a web application, re-using most of the source code, including the UI code, with a minimal effort to specify a small set of specific classes to start the application for the specific running platform.

1 INTRODUCTION

Java allows a compiled program to run on different operating systems where a Java Virtual Machine is installed. However, Java does not provide mechanisms to run the same program on different platforms: running a Java desktop application as a web application is impossible, and the other way round. Java frameworks to implement Java web applications (e.g., Spring, Quarkus, and Jakarta EE) are not compliant with Java toolkits for the UI (User Interface) of desktop applications (e.g., Java Swing, JavaFX, and Eclipse SWT). Following good design practices, the developer can re-use the application's core and business logic parts; however, the rest of the application, namely the UI and its logic, must be implemented separately according to the application's platform.

In this paper, we show how the Eclipse project EMF Parsley (https://eclipse.dev/emf-parsley/) allows the developer to achieve "single-sourcing": the developer can implement a desktop application (based on RCP, the Eclipse Rich Client Platform) running on any Java Virtual Machine, that can also be deployed as a web application, re-using most of the source code, including the UI code, with a minimal effort to specify a small set of specific classes to start the application for the specific running platform.

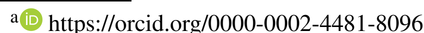
EMF Parsley, built on top of the *Eclipse Modeling Framework* (EMF) (Steinberg et al., 2008), is a framework to quickly and easily develop user interfaces based on EMF models. It provides several reusable and easily customizable UI components (trees, tables, forms, views, editors) acting on EMF models, hiding the complexity of internal details. EMF Parsley also leverages the Eclipse project, RAP (Remote Application Framework), to achieve single-sourcing for desktop and web applications (accessible with web browsers and mobile devices).

EMF Parsley and its DSL were first presented in (Bettini, 2016b). The DSL has evolved a lot since then, including many more features that we will use in this paper. Moreover, the integration of EMF Parsley with RAP has also evolved, making it straightforward to achieve "single-sourcing," as we will show in this paper.

Structure of the paper. Section 2 describes the frameworks and techniques we use. Section 3 shows the running example implemented with single-sourcing. Section 4 describes some related work. Section 5 concludes the paper by evaluating the proposed approach.

2 METHODOLOGY

In this section, we briefly describe the main features of EMF Parsley and how it allows developers



[a] https://orcid.org/0000-0002-4481-8096

Bettini, L.
Single-Sourcing for Desktop and Web Applications with EMF Parsley.
DOI: 10.5220/0012421400003645
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024)*, pages 219-226
ISBN: 978-989-758-682-8; ISSN: 2184-4348
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

to achieve single-sourcing for RCP and RAP applications. In particular, we also describe the methodology to achieve single-sourcing.

EMF Parsley is built on top of the EMF modeling framework and aims to make the development of applications based on EMF easy. EMF Parsley provides several reusable UI components (trees, tables, forms, views, editors) acting on EMF models, hiding the complexity of internal details. EMF Parsley provides Eclipse parts (i.e., editors and views) for editing EMF models. Unlike the standard EMF application development workflow, in EMF Parsley, the developer must not modify monolithic generated classes: all aspects are easily customizable and configured with Google Guice (Prasanna, 2009), a mainstream *Dependency Injection* framework.

EMF Parsley mechanisms are built on top of EMF.Edit (Steinberg et al., 2008), but it hides the internal and difficult details of EMF.Edit. EMF Parsley components can be configured and customized with Java using EMF Parsley declarative API. However, EMF Parsley also provides its DSL to simplify these operations further. Using the DSL, one can define the configuration of EMF Parsley UI components in a very compact, readable, and maintainable way. Then, the EMF Parsley DSL compiler will generate all the corresponding Java code, including the dependency injection configuration. In that respect, EMF Parsley is based on the *Generation Gap* pattern (Vlissides, 1998).

This DSL is implemented with Xtext (Bettini, 2016a), the mainstream framework for developing DSLs with a fully-fledged IDE on top of Eclipse. Thus, the EMF Parsley DSL has a rich Eclipse editor with all the typical IDE tooling (syntax highlighting, code completion, quickfixes, automated building, etc.). Moreover, the DSL relies on Xbase (Efftinge et al., 2012), a reusable Java-like expression language completely interoperable with the type system of Java: all the existing Java libraries can be used in the DSL. Java programmers will be able to learn the Xbase language easily.

An input file for the EMF Parsley DSL, i.e., a file with extension `emfparsley`, consists of a main `module` section. This corresponds to a Google Guice module in the generated Java code. Inside the `module`, one specifies customizations. Each customization has its specific sub-section. We will show several examples in Section 3.

The `module` section allows the developer to specify Eclipse parts (i.e., EMF Parsley views and editors) that are initialized, configured, and customized in the current `module`. The DSL compiler will generate the `plugin.xml` with the corresponding Eclipse *extension points*.

Thus, all the EMF Parsley UI components are specified in a compact form in a single file instead of being spread into several Java classes (like what happens when using EMF and its generated code).

Moreover, EMF Parsley provides some Eclipse project wizards to create projects configured to get started with EMF Parsley UI components and its DSL. These wizards provide initial templates for specific views (e.g., tree, tree with form, table, etc.).

EMF Parsley leverages the RAP framework to achieve single-sourcing (Lange, 2009) (before 2012, the acronym RAP was meant for “Rich Ajax Platform”). RAP provides a widget toolkit that implements the SWT widgets and other RCP concepts targeting the web platform. Since these custom implementations keep the same name as the standard SWT ones, by switching the runtime platform from the standard RCP to the RAP one, the source code using SWT can be re-used without modifications. One of RAP’s main goals is to make the web runtime environment transparent for the developer. In particular, just by using Java and no HTML or Javascript, one can create a web application. RAP does not implement all the RCP/SWT; thus, only the API available in both runtimes must be used to achieve single-sourcing. EMF Parsley uses only the common API, which is available both in RCP and RAP. This way, applications implemented with EMF Parsley will transparently be implementable with single-sourcing.

EMF Parsley provides its Eclipse features and bundles into two separate update sites: one for standard RCP desktop applications and one for web RAP applications.

The crucial thing for single-sourcing is to use two different *target platforms*, depending on the desired runtime platform. Eclipse uses a “target platform” to specify the external dependencies (called *bundles* in OSGi). The dependencies specified in the target platform are used to compile and run the code in an Eclipse workspace. EMF Parsley provides a wizard that creates a target definition file already configured with all the needed dependencies from the RAP version of EMF Parsley and additional required RAP dependencies.

Note that an Eclipse workspace can activate a single target platform at a time. For this reason, it is best to have two separate workspaces, one for RCP and one for RAP, each with the proper target platform enabled. Then, we must configure the projects of our single-sourcing application appropriately. As we will see later, we will also need two small projects (one for each runtime platform) to deal with the start of the ap-

plication. These are specific to the running platform and cannot be shared. On the contrary, the rest of the projects are meant to be shared between the two applications.

Now, we have to handle user interface dependencies for the projects meant to be shared. Those projects cannot simply depend on platform-specific bundles, e.g., `org.eclipse.ui` or `org.eclipse.rap.ui` (the RAP bundle with the implementations of the classes of the standard Eclipse former bundle) because only one can be resolved with a specific target platform. The same holds for EMF Parsley UI bundles.

We can apply two possible alternative solutions for the above problem in the `MANIFEST.MF` of the projects that are meant to be shared:

- Both the RCP and RAP bundles are specified as `Require-Bundle` as "optional".
- All the Java packages of the dependencies are specified as `Import-Package`.

The first solution is easier to implement, and it is a good starting point (it is the one implemented by the EMF Parsley project wizards). The second solution requires more work since dependencies are specified in the shape of imported Java packages, typically many in an Eclipse project. Moreover, with such a solution, one has to deal with a few Eclipse packages that are "split": A "split package" has content spread in more than one required bundle. (Concerning best practices when dealing with OSGi dependencies, we refer the interested reader to (Ochoa et al., 2018).)

We need at least one workspace project with (non-optional) `Required-Bundle` specifications for the current target platform. This way, the platform-specific optional requirements (either for RCP or for RAP) or imported packages above can be resolved. Such a project is platform-specific and is not meant to be shared. Note that such a project will ensure the build fails if none of the optionally required bundles are present in the target platform.

Of course, projects that do not depend on UI dependencies, e.g., "core" projects, can be seamlessly shared and depend on Eclipse core bundles. This requires following the best practice of keeping code independent of the user interface into separate projects concerning code that deals with the user interface.

3 THE MAIL EXAMPLE WITH EMF PARSLEY

In this section, we present our running example (The source code of the example can be found here: https://github.com/LorenzoBettini/emf-parsley-demo-rcp-rap) to demonstrate how EMF Parsley can be used to implement a desktop (RCP) application and a web (RAP) application with single-sourcing. In particular, we will show only the most interesting parts of the source code and refer to the Git repository for the whole implementation. We will re-use most of the code for both applications. We will implement the user interface of an e-mail client. The standard Eclipse RCP Mail template (Burnette, 2006) inspires our example, though it is implemented completely from scratch using EMF Parsley. Note that this is not a trivial example: the UI reflects the typical UI of e-mail clients; the fact that the source code we write is small is thanks to the features of EMF Parsley and its DSL.

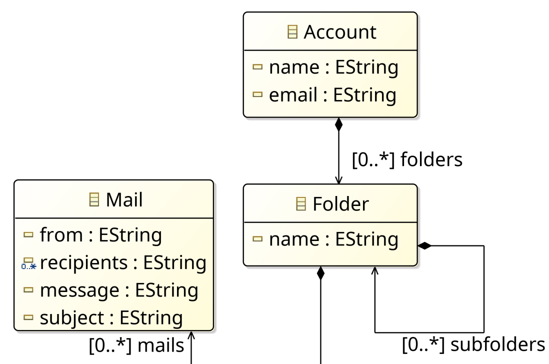


Figure 1: The metamodel of the Mail example.

The metamodel of the Mail example is shown in Figure 1. We intentionally keep the metamodel as simple as possible. Of course, it can be easily extended with additional features and classes. The Ecore model and the generated EMF Java classes are kept in a separate project since they depend not on Eclipse UI bundles but only on Eclipse and EMF core bundles. In our example, the project is called `emf.parsley.demo.mail.model`.

Note that we only deal with the model of the Mail application. In a real e-mail application, the EMF model should be synchronized with real e-mail services (e.g., an IMAP server). This requires communication with remote services, which is out of the scope of the paper. However, this task can be achieved by connecting the services and the EMF model, which can be done without touching the EMF Parsley implementations of the UI we show in this paper.

Thus, the EMF model will be kept in memory in this example. In particular, we hook into the initialization stage of EMF Parsley to fill the EMF resource with some initial contents (not shown here). We created three accounts with a few folders and emails (with some test contents).

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

Listing 1: The definition of the Accounts view.

We want to mimic a typical e-mail GUI layout (see, e.g., Thunderbird) consisting of:

- A tree view on the left with the email accounts;
- A table on the top-right showing the e-mail messages of the currently selected folder in the tree;
- A form on the bottom-right showing the body of the e-mail message of the currently selected e-mail in the table.

In this project, named `emf.parsley.demo.mail.views`, we will define all the views for the Mail user interface.

In particular, we store in the `icons` subdirectory a few image files that we will use to customize our example application's views.

For the tree with the accounts, we use a tree view from EMF Parsley, which we define and customize with the EMF Parsley DSL file shown in Listing 1.

The `parts` specification defines the extension point for Eclipse view parts. The `viewpart` specifies the ID of the standard Eclipse view extension point, and the `viewname` specifies its name (i.e., the title of the view). The `viewclass` specifies the Java type of an Eclipse view part. In this example, we use one of the standard pre-defined views of EMF Parsley, `SaveableTreeFormView`, which implements a view with a tree based on a resource that can be saved. From this specification, the EMF Parsley compiler will generate a few Java classes and the extension point in the `plugin.xml`. In particular, the extension point will be configured so that the EMF Parsley dependency injection mechanism will be used to create the runtime view object.

By default, EMF Parsley views delegate to the reflective mechanisms of EMF.Edit. For example, the content provider of a tree view will represent the elements in a tree (parent and children) by using the containment features of the Ecore metamodel. Thus, for our example, the default content provider implementation would show the account element, the folders (and, recursively, the subfolders) as its children, and the emails contained in the folder. In the tree view of our example, we do not want the accounts view to show the emails of a folder (we will show the emails in the table view). To specify this customization, we use the `viewerContentProvider`. In particular, we specify that the children of a `Folder` object should be (only) its subfolders. Note that `Folder` is a (statically-checked) type reference to the Java type `Folder` (the EMF generated Java class) and `subfolders` is a (statically-checked) reference to the `subfolders` EMF feature of `Folder`.

Finally, we customize the aspect of the tree elements with the `labelProvider` section. Like a standard JFace label provider, we can customize every element's text and image. In the EMF Parsley DSL, we do that declaratively, avoiding "instanceof" checks: we specify the Java type and the corresponding text or image (by the file name of an image in the `icons` folder). Thanks to Xbase, we can use Java-like expressions like `switch`. Indeed, we want to use special icons for special folders like "Inbox", "Sent" and "Trash". For the "text" of the label provider, we specify the feature to be used to represent the string representation of that element. As before, the standard EMF Parsley implementation of a label provider delegates to the one of EMF.Edit.

The default implementation of the EMF Parsley tree view provides editing single nodes by double-clicking: a dialog appears to edit the fields of the corresponding EMF model object. Modifications will be automatically propagated through the framework. For example, if we change the name of a mail folder, the label provider will be notified and, in case, will repaint the tree node label's text and image. EMF Parsley also implements context menus for the view's nodes, with standard menus like copy/cut and paste and undo/redo. The EMF Parsley DSL allows for the customization of context menus as well.

```
module emf.parsley.demo.mail.views.mailsview {
  parts {
    viewpart emf.parsley.demo.mail.views.MailsView {
      viewname "Mails View"
      viewclass OnSelectionTableView
    }
  }
  configurator {
    eClass {
      OnSelectionTableView -> MailPackage.Literals.MAIL
    }
  }
  featuresProvider {
    features {
      Mail -> subject, from
    }
  }
}
```

Listing 2: The definition of the Mails view.

For the table with the e-mails, we use a predefined table view from EMF Parsley, `OnSelectionTableView`. Such a view implements a table that reacts to selection change from another view in the application. In this example, we want this table to show the emails in the folder currently selected in the accounts tree view. When we select a different folder in the accounts view, our table will be automatically updated. EMF Parsley already provides this behavior in `OnSelectionTableView` (the table components provided by EMF Parsley also allow for automatic sorting according to column contents by clicking on the desired table column header).

We define and customize such a view with the EMF Parsley DSL file in Listing 2.

The `parts` specification is similar to the one we have already seen for the accounts view. The automatic behavior of the table view in EMF Parsley would be to show all the features of all the contents of the selected EMF object in the table. In this example, we only want to show the e-mail objects of a selected folder. We can specify such a custom behavior with the `configurator.eClass` specification. To fill the table view, we must specify an EMF `EClass` of the EMF objects. We use the standard EMF API to refer to such an `EClass`, i.e., by using the Java class of the `EPackage` (`MailPackage`, in this example), generated by EMF, and its static fields (`Literals.Mail`, in this example, corresponding to the `EClass` `Mail`).

By default, EMF Parsley would use all the features of the specified EMF object to create the table's columns with the corresponding values. Since the e-mail's recipient and message will be shown in the third view of our application, we specify that only the features `subject` ad `from` of `Mail` should be used to fill the table's contents, by using a `featuresProvider` specification. Note that, as before, all the references in these specifications are statically typed.

```
module emf.parsley.demo.mail.views.messageview {
  parts {
    viewpart emf.parsley.demo.mail.views.MessageView {
      viewname "Mail Message View"
      viewclass OnSelectionFormView
    }
  }
  labelProvider {
    text {
      Mail -> subject
    }
    image {
      Mail -> "email.png"
    }
  }
  featuresProvider {
    features {
      // the subject is already in the title
      Mail -> from, recipients, message
    }
  }
  featureCaptionProvider {
    text {
      Mail : recipients -> "to"
    }
  }
  formControlFactory {
    control {
      Mail : message -> {
          val t = createText("",
              SWT.MULTI, SWT.BORDER,
                SWT.WRAP, SWT.V_SCROLL
            )
          t.layoutData = new GridData(GridData.FILL_BOTH)
          return t
        }
        target observeText(SWT::Modify)
    }
  }
}
```

Listing 3: The definition of the Message view.

For the view with the e-mail message, we use a predefined form view from EMF Parsley, `OnSelectionFormView`. Such a view implements a form that reacts to selection change from another view in the application. In this example, we want this form to show the e-mail message of the e-mail that is currently selected in the e-mail table view. When we select a different e-mail in the table view, our form will be automatically updated. EMF Parsley already provides this behavior in `OnSelectionFormView`.

We define and customize such a view with the EMF Parsley DSL file in Listing 3.

We have already explained the specifications `parts`, `labelProvider`, and `featuresProvider` in the description of the previous views. Their specification for the message view should be easy to understand.

Concerning `featureCaptionProvider`, this specification allows us to customize the label that describes a feature in an EMF Parsley view. For

example, instead of the default label (which is based on the name of the feature of the corresponding value) for the `recipients` feature, we want the form to show “to”.

Finally, the specification `formControlFactory` allows the developer to fully customize the form elements corresponding to a feature of the selected EMF object. For example, for the text of the e-mail message, instead of a default single-line text box, we want to use a multi-line text box. We can use the JFace/SWT API to specify the form’s control corresponding to an EMF feature (in this case, `Mail.message`). EMF Parsley relies on EMF Databinding to connect view controls to the EMF model’s values. The EMF DSL allows to customize the EMF Databinding accordingly, using the EMF Databinding API (the `target` specification above, where `observerText` comes from the EMF Databinding API).

Now, we have to implement the code to start the application. Independently from RCP or RAP, such a code has a standard shape, relying on a few classes that an Eclipse application requires to define the shape of the application’s user interface. This code can be shared as well between the RCP and RAP applications. In our example, this corresponds to implementing `ApplicationActionBarAdvisor`, `ApplicationWorkbenchAdvisor`, `ApplicationWorkbenchWindowAdvisor`, and `Perspective`. Their implementation is out of the scope of the current paper and can be seen in the Git repository of this example. This shared code is implemented in a shared project named `emf.parsley.demo.mail.app.common`.

Finally, we implemented two projects with only the code required to start an RCP and RAP applications, named `emf.parsley.demo.mail.rcp` and `emf.parsley.demo.mail.rap`, respectively. This code is the only code we must implement differently since an RCP and a RAP application have two different ways to start. The two projects will also have a different `plugin.xml` since the extension points for RCP and RAP applications are different. Only these two projects contain code that cannot be shared, but such code is very small. We do not show them here because they are irrelevant to our example.

Note that these two platform-specific projects must specify in their `MANIFEST.MF` the (non-optional) `Required-Bundles` for the platform-specific bundles. This way, the platform-specific optional requirements (either for RCP or for RAP) or imported packages of the common projects can be resolved.

The final result can be seen in Figure 2 for the RCP (desktop) application and in Figure 3 for the RAP (web) application.

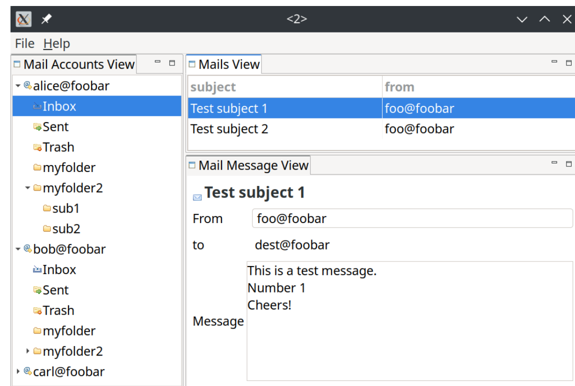


Figure 2: The Mail application runs as an RCP (desktop) application.

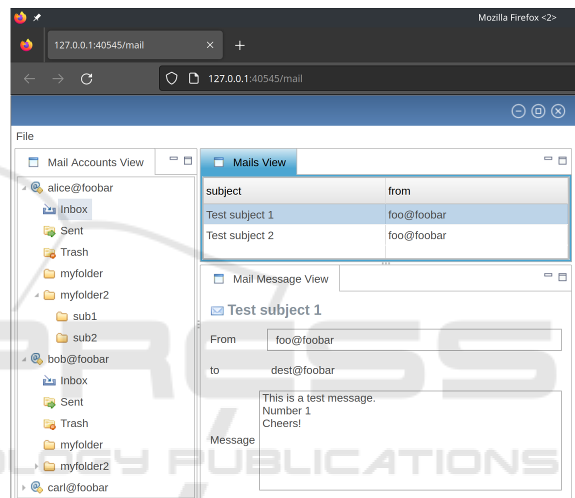


Figure 3: The Mail application runs as a RAP (web) application inside Firefox.

4 RELATED WORK

EMF Parsley can interoperate with all the existing EMF frameworks. For example, concerning EMF persistence technologies, EMF Parsley can be seamlessly used with CDO (https://projects.eclipse.org/projects/modeling.emf.cdo), a distributed shared model framework for EMF, which can save and query EMF models into and from mainstream databases.

EMF Parsley differs from many generative frameworks for EMF (https://eclipse.dev/modeling/emft/) since it does not require the programmer to modify the generated code. It also differs from the standard EMF generation mechanism: EMF generation mechanisms rely on Java annotations in the generated code. If the programmers need to modify the generated Java

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

classes, they must remember to remove such annotations; otherwise, further generations would override the custom code. This makes the application hard to maintain: the generated and custom code live together, and it is hard to easily find the customizations inside the generated Java classes. EMF Parsley injects the generated code into the framework's code. Moreover, the EMF Parsley DSL allows the developer to write and maintain the customizations easily in a compact form.

EMF Parsley and its DSL share a few characteristics with other reflective and metamodel-based frameworks, such as Magritte (Renggli et al., 2007) (e.g., specifications in one single source code file and high-level abstractions hiding internal details). However, Magritte targets a completely different programming language, Smalltalk.

Sirius (Vujovic et al., 2014b) is an Eclipse project to easily create graphical modeling workbenches (e.g., with visual editors) based on EMF. Sirius mainly targets users who need to edit an EMF model with a diagram editor instead of trees and forms (though it also supports the latter). However, EMF Parsley and Sirius are complementary. The same holds for other Eclipse frameworks for graphical modeling tools (e.g., (Gronback, 2008; Rose et al., 2012; Vujovic et al., 2014a)).

The EMF Client Platform (ECP) (https://eclipse.dev/ecp/emfforms/) and Eclipse Scout (https://eclipse.dev/scout/) share with EMF Parsley the main goals and they both support RAP. However, we believe EMF Parsley components are easier to reuse and customize thanks to the DSL.

Recently, an increasing effort has been made to move IDEs to the Web. In that respect, the Language Server Protocol (LSP) (Gunasinghe and Marcus, 2021) is a promising technology allowing the decoupling of the language implementation and the IDE support (Bünder, 2019). In particular, modeling technologies are being used on the web as well (Rodríguez-Echeverría et al., 2018; Saini et al., 2019; Bork and Langer, 2023). How EMF Parsley IDE tooling can be made available in a web-based IDE is still under investigation. In that respect, Xtext already supports the LSP, so EMF Parsley could leverage the integration of Xtext with LSP.

5 CONCLUSIONS

In this paper, we showed how to implement desktop and web applications by reusing most of the code with EMF Parsley and its support for RAP: the developer can implement a desktop application running on any Java Virtual Machine that can also be deployed as a web application, re-using most of the source code, including the UI code, with a minimal effort to specify a small set of specific classes to start the application for the specific running platform.

Table 1: Some statistics about the code of the example application.

Common code

Language	Files	Lines	Code
Java	40	3250	1405
XML	3	59	50

RCP code

Language	Files	Lines	Code
Java	1	43	36
XML	1	39	37

RAP code

Language	Files	Lines	Code
Java	1	27	18
XML	1	25	25

We conclude with a few statistics, shown in Table 1, about the source code of our application, which is divided into separate projects according to the layout shown above (the statistics have been computed with Tokei, https://github.com/XAMPPRocky/tokei). The table does not show the code written in the EMF Parsley DSL, which is about 100 lines. From this code, the EMF Parsley DSL generates all the Java code corresponding to the one shown in the table in the "Common code" section. That section also includes all the Java files generated by EMF for the EMF model code. The XML code concerns the `plugin.xml` files of the three common projects (the EMF model project needs one as well); the `plugin.xml` of the EMF Parsley views project is generated by the EMF Parsley DSL corresponding to the Eclipse parts specified in the `emfparsley` files (as already explained). In the other two sections ("RCP code" and "RAP code"), the XML code corresponds to the `plugin.xml` with the extension points of the specific platform.

The table shows we could reuse most of the code: the code specific for RCP and RAP is minimal.

Of course, we might have to write additional code specific to RCP and RAP to provide features and customizations that are only available on that specific platform. This can still be done by writing such platform-specific code in projects not meant to be reused in different platforms. The methodology shown in this paper does not prevent that: on the contrary, we showed that some code must be platform-specific.

As we said in Section 2, RAP does not implement all the RCP/SWT; thus, only the API available in both runtimes must be used to achieve single-sourcing. However, such a code could not be reused anyway and should be put in separate projects, as we have done in our example. Moreover, the code that RAP does not provide corresponding to RCP is minimal. For example, in EMF Parsley, we never encountered the need for an RCP API that was not also provided by RAP.

In summary, we reached the following achievements thanks to EMF Parsley, its DSL, and its integration with RAP:

- we define our UI by reusing the EMF Parsley views;
- we customize such views in a compact form with a DSL and have the Java code generated;
- we achieve single-sourcing by reusing most of the code.

Note that the EMF Parsley DSL Eclipse-based IDE allows developers to debug the original Parsley DSL code when running the Eclipse views (it is also possible to debug the generated Java code, in case). Of course, this can be done when running the RCP and RAP applications. All the typical debug views of Eclipse, e.g., “Variables”, “Breakpoints”, etc. are available.

ACKNOWLEDGEMENTS

This work was partially supported by the PRIN project “T-LADIES” n. 2020TL3X8X.

REFERENCES

Bettini, L. (2016a). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2nd edition.

Bettini, L. (2016b). The EMF Parsley DSL for Developing EMF Applications. In *MODELSWARD*, pages 301–308. Science and Technology Publications, Lda.

Bork, D. and Langer, P. (2023). Language Server Protocol: An Introduction to the Protocol, its Use, and Adoption for Web Modeling Tools. *EMISAJ*, 18:9–1.

Bünder, H. (2019). Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages. In *MODELSWARD*, pages 129–140. SCITEPRESS.

Burnette, E. (2006). Rich Client Tutorial Part 3. https://www.eclipse.org/articles/Article-RCP-3/tutorial3.html.

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.

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

Gunasinghe, N. and Marcus, N. (2021). *Language Server Protocol and Implementation*. Springer.

Lange, F. (2009). *Eclipse Rich Ajax Platform: Bringing Rich Client to the Web*. Apress.

Ochoa, L., Degueule, T., and Vinju, J. (2018). An Empirical Evaluation of OSGi Dependencies Best Practices in the Eclipse IDE. In *MSR*, page 170–180. ACM.

Prasanna, D. R. (2009). *Dependency Injection: Design Patterns Using Spring and Guice*. Manning, 1st edition.

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

Rodríguez-Echeverría, R., Izquierdo, J. L. C., Wimmer, M., and Cabot, J. (2018). An LSP infrastructure to build EMF language servers for web-deployable model editors. In *MDETools*, pages 326–335. ACM/IEEE.

Rose, L. M., Kolovos, D. S., and Paige, R. F. (2012). EuGENia Live: A Flexible Graphical Modelling Tool. In *XM*, page 15–20. ACM.

Saini, R., Bali, S., and Mussbacher, G. (2019). Towards Web Collaborative Modelling for the User Requirements Notation Using Eclipse Che and Theia IDE. In *MiSE*, pages 15–18. ACM/IEEE.

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

Vlissides, J. (1998). *Pattern Hatching: Design Patterns Applied*. Addison-Wesley.

Vujovic, V., Maksimovic, M., and Perisic, B. (2014a). Comparative analysis of DSM Graphical Editor frameworks: Graphiti vs. Sirius. In *ERK*, pages 7–10.

Vujovic, V., Maksimovic, M., and Perisic, B. (2014b). Sirius: A rapid development of DSM graphical editor. In *INES*, pages 233–238. IEEE.