Re-engineering Eclipse RCP Applications - the RED Case Study

Maciej Kucharek

Kongens Lyngby 2013
IMM-M.Sc.-2013

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013

Summary

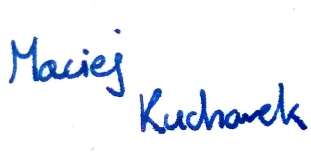
The goal of the thesis is to investigate and present a way of re-engineering *Eclipse RCP* applications. As a case study, RED - "REquirements eDitor" have been chosen. RED is a one of the major tools for the *02264 Requirements Engineering* course at DTU, and as such is being extensively used throughout the course, resulting in a number of feature requests that are currently difficult to implement due to the poor architecture. RED is also a typical example of an *Eclipse RCP* developed purely for providing a huge number of features, while neglecting the maintainability aspect, which resulted in a major roadblock in further development. The re-engineering process will cover the improvements that could be made to the build process, the high-level architecture and the actual implementation at a *plug-in* level, all of which will contribute to the overall *Eclipse RCP* maintainability.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring an M.Sc. in Computer Science and Engineering.

The thesis deals with re-engineering of Eclipse RCP applications, with the particular focus on RED - "REquirements eDitor" used as a main tool in the *02264 Requirements Engineering* course at the Technical University of Denmark.

Lyngby, 18-October-2013



Maciej Kucharek

Acknowledgements

I would like to thank my supervisor Prof. Dr. **Harald Störrle** for being there to help, and for motivating me for achieving what I thought was not possible. If it was not for your support, I would probably have given up a long time ago.

I would also like to thank my friends and family, for their patience and confidence in me. Knowing you were all there for me was the best motivation I could get.

Contents

Summary i

Preface iii

Acknowledgements v

1 Introduction 1
- 1.1 Software Re-engineering . . . 1
- 1.2 RED - "REquirements eDitor" . . . 2
  - 1.2.1 User experience shortcomings . . . 3
  - 1.2.2 Maintainability problems . . . 4
  - 1.2.3 The reason for re-engineering . . . 5
  - 1.2.4 Goals . . . 6

2 Improving the RED build process 11
- 2.1 Introducing code versioning & issue tracking . . . 11
- 2.2 Building RED . . . 14
  - 2.2.1 Eclipse Eclipse PDE build process . . . 14
  - 2.2.2 Current RED build process . . . 14
  - 2.2.3 Resolving external dependencies . . . 16
  - 2.2.4 Adding target platform definition . . . 19
- 2.3 Eclipse Tycho - a new approach to building Eclipse plug-ins . . . 21
  - 2.3.1 Improvements over Eclipse PDE build . . . 22
  - 2.3.2 Continuous integration . . . 23
- 2.4 Summary . . . 23

3 Restructuring RED 25
- 3.1 Upgrading the underlying *Eclipse* framework . . . 25
  - 3.1.1 Migrate RED plug-ins to Eclipse 4 API . . . 27

3.1.2 Using Eclipse 4 compatibility layer . . . 28
3.1.3 Keeping the original Eclipse 3.X API . . . 29
3.1.4 Decision . . . 29
3.2 Improving high-level architecture . . . 30
3.2.1 Eclipse RCP development . . . 30
3.2.2 Components . . . 32
3.2.3 Implementation issues . . . 34
3.2.4 Countering cyclic dependencies . . . 40
3.2.5 Examining dependencies between the modules . . . 46
3.2.6 Dividing Specification Elements . . . 49
3.2.7 Feature-based product . . . 51
3.3 Improving low-level implementation . . . 53
3.3.1 RED plug-ins implementation problems . . . 53
3.3.2 Restructuring plug-in implementation . . . 53
3.3.3 Removing unused source code . . . 56
3.4 Summary . . . 57

4 Addressing conceptual weaknesses 61
4.1 Fixing report generation . . . 61
4.2 Fixing model weaving . . . 63
4.3 Aligning EMF models with domain classes . . . 64
4.3.1 Handling GMF models . . . 66
4.4 Excluding SCENARIO support . . . 67
4.5 Adding horizontal scroll-bar support . . . 67
4.6 Branding . . . 69
4.6.1 Mac OS X native app packaging . . . 70

5 Evaluation 73
5.1 Final architecture . . . 73
5.2 Measurements . . . 75
5.2.1 Original RED source code . . . 76
5.2.2 Final RED source code . . . 78
5.3 Case Study: A "Test Case" Specification Element . . . 82
5.3.1 Editor design & implementation . . . 82
5.3.2 Module implementation . . . 84
5.3.3 Reporting integration . . . 87

6 Conclusion 89
6.1 Summary . . . 89
6.1.1 Build process . . . 89
6.1.2 Restructuring RED . . . 92
6.1.3 Additional fixes & improvements . . . 94
6.2 Discussion . . . 95
6.3 Future work . . . 97

Bibliography **99**

Chapter 1

Introduction

1.1 Software Re-engineering

According to [Arn93], software re-engineering means to examine and analyze the existing software and to re-implement it to achieve improvements in functionality, performance or the architecture. This is especially true for the software that has been implemented using legacy languages, frameworks or libraries, which are often not supported anymore. In such a case, maintaining the software is becoming more and more expensive, as the technologies grow old and the number of specialists that know them is decreasing. At some point, the underlying technology limitations will prevent the software from growing further, or there will simply be no developer capable of maintaining it. In order to fight it, a re-engineering process is required, which will move the software from the legacy technologies to more modern ones, which will make it possible (or cheaper) to maintain the application. It would not only make it easier to find developers familiar with newer technologies, but will also make more room for the future improvements, as the newly chosen technology will probably be supported for some time after the moving process.

However, underlying technologies are not the only reasons for software re-engineering. If, during the initial phase, the software being developed has not been properly designed for extending, it may be the case that at some point, it may not be

possible to extend a certain systems just because of the incorrect, or simply not well thought, decisions made in the past. This is especially true for plenty of start-up projects, which are mainly focusing on how the number of features offered, rather than the properly organized, extensible architecture. Such an approach is often taken to demonstrate that a certain concept for the application may work, but once someone decides to take the next step and to continue with development, a proper re-engineering of the initial concept is often a necessity.

Last, but not least, re-engineering may be applied at the implementation level. When in a hurry, developers tend to lean towards taking shortcuts, or "hacks", such as violating or simplifying the initial design to obtain a certain functionality. This results in a code that does what it is supposed to, but is often reluctant to change or to extend. Again, while valid in some cases, such an approach makes the life harder in the long run, as maintaining such a hacked code will require much more effort in the future, either because it is simply not designed to allow changes, or simply because it may be difficult to understand by anyone else except the author.

To sum up, re-engineering is an important part of software development process, which should not be neglected in any of the production software. While it may sound as a waste of both time and money, as it does not directly result in any additional features that can be noticed by the users, it may result in much swifter development in the future. Also, identifying and resolving issues that may either prevent or severely slow-down system extension before they become too expensive to fix may be a may or break in terms of the whole development process.

1.2 RED - "REquirements eDitor"

RED has been created in 2010 as a tool aiding students in *02264 Requirements Engineering* course at Technical University of Denmark. Its main purpose was to make it easier for course attendees to gather, store and elaborate on the outcomes of their work during the semester. The course covers a number of *Requirements Engineering* techniques, including *Stakeholder* and *Persona* analysis, *Goal* modeling and *Requirement* management in both textual and visual forms. Since none of the existing tools were suitable for the course needs, either because of they did not cover the whole material or used different terminology, it was decided that creating a new tool from scratch would be the best option in the long run [Fri12].

RED has been created in 2010 as a result of joint MSc-thesis project of two

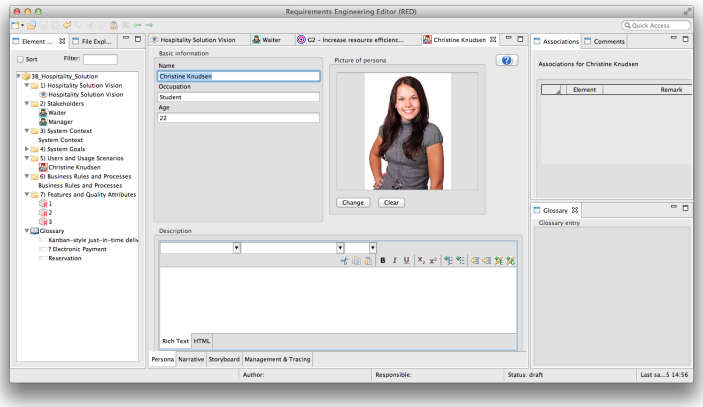


Figure 1.1: RED Workbench

students - **Anders Friis** and **Jakob Kragelund**. The tool was built on top of *Eclipse* Rich Client Platform (Eclipse RCP), which is a popular choice for building advanced, multi-platform software, that encourages modular, and therefore extensible, architecture. It was based on latest version of *Eclipse Framework* available at the time, being *3.7 Indigo*. First version of RED was released in September 2011, just in time for the autumn edition of the *02264 Requirements Engineering* course, where it was first used as a main course tool. Meanwhile, the project was taken over by **Johan Flod**, who was extending RED as his MSc-thesis project.

1.2.1 User experience shortcomings

Shortly after the initial release, a number of problems have been identified. First and foremost, while the tool was more or less stable on *MS Windows* machines, it was not the case on both *Mac OS X* and *Linux* operating systems. Since both Java and *Eclipse* are by definition designed for creating multi-platform tools, it was surprising that RED was not performing well on non-windows-based computers. This was partially because the development and initial testing has been done on *MS Windows* machines, while other environments have been neglected. Moreover, the module that allows weaving the UML models into Prolog was based on JPL (a Java interface to Prolog), that requires native libraries to operate properly and the module itself provides only *MS Windows* libraries. Both of these resulted in various errors presented to the user when accessing certain parts of RED application, making it extremely annoying to use it extensively.

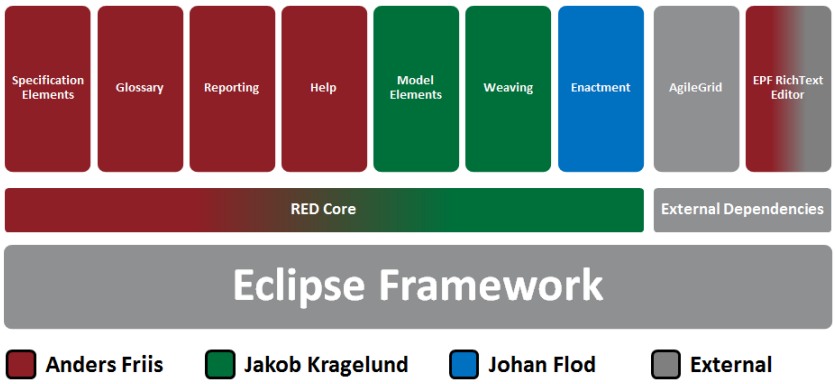


Figure 1.2: RED Overview

Another problem that made it difficult for students to work was the lack of group-work support. While it was possible in RED to open multiple projects and move or copy the contents between each other, there was no straightforward way of distributing the work between group members while keeping the results in a single project. The main problem was that a regular RED project is being stored in a single, non-human readable file, which made it impossible to make use of neither shared cloud spaces, like *DropBox*, nor various version control systems, such as Git or SVN, for handling multi-user scenarios. Since nowadays these are two most common ways of work distribution by students, it made it difficult for them to accept and use the newly introduced tool.

1.2.2 Maintainability problems

Unfortunately, usability problems are not the only one that has been troubling RED. While an enormous effort has been made to pack RED with such a number of features in such a short amount of time, it must have come with a price. Despite RED being developed for over a year, the code has not been kept in neither DTU or any third-party code repository, which results in several shortcomings that will affect any future developer. First of all, there was no remote and accessible copy of the RED source code. While each of the previous developers most likely used some sort of version control systems for their personal use, these were not available to anyone else, and the source code had been passed along as a single *zip* file. The problem with such an approach is that if such a package gets damaged, it may get difficult, or even impossible to retrieve the RED source code. This is a major problem that, under certain circumstances, may result in severely impacting, or even completely stopping the application

development. The other issue originating from not having a code repository is that the RED source code was not being versioned, which made it impossible to track changes made by individual developers. This would be extremely helpful when trying to understand how the development had evolved over time, as well as to understand some of the complex design decisions that has been made in certain parts of the code. It would also make it possible to revert some of the changes, especially the features that are either broken or not completed, and to exclude them from the stable RED release.

Also, it quickly turned out that there was no straightforward way of setting up a development environment required for building RED. As described in [Kra12], building RED required quite a number of complicated and error-prone steps. While performing these should not be a problem for a regular Java or *Eclipse* developer, it made it almost impossible for anyone else to build RED from the available source, which could be especially vital for **Harald Störrle**. Also, some of the required *plug-ins*, like AgileGrid, had already been abandoned and therefore it was difficult to find and install their binary files, and it may be even impossible in the future. Last, but not least, it is generally not a best practice to tie a build process to a certain IDE, as various developers may prefer to use different development environments.

There were also several problems with the codebase itself. At a high level, RED had been divided into number of modules, with a clear distinction between their purpose and how do they communicate between each other. Unfortunately, the implementation part of the modules has not been given enough thought, resulting in a complex and often inefficient code. Not only is the code badly organized in terms of the underlying framework specifications, but it also leaves plenty of room for improvements at the programming language level. Both of these shortcomings will be discussed in more details in the following chapters.

1.2.3 The reason for re-engineering

All of the maintainability issues mentioned above resulted in two, high-level problems. Taking them all into account, it is quite clear that both fixing the existing usability issues and extending RED with new functionality required a lot of effort. In fact, this has been tried before, both by **Johan Flod** who was trying to add enactment support, and myself, when trying to add merging capability to RED projects. Each of these attempts were actually unsuccessful, only to prove the need for a major re-factoring of RED. Since RED was meant to be extended by students as their MSc thesis projects, it must be possible to for new students to quickly set up the development environment and understand both the underlying design and the code, so that they put the maximum effort

on the thesis subject itself. That, of course, would not ultimately make every extensions attempt a success, but will would greatly increase the chances of a steady, continuous development of RED.

The other reason would be that RED has actually become quite a big project, containing a considerable number of features and that is actually advanced enough to help students thorough the course. As up to that point, the development process was mostly feature-oriented, it was probably a good time to spend some time on improving what had already been achieved, introducing necessary architecture and performance fixes, as well as enforcing the design rules that had been initially introduced for the application. That would not only contribute to the maintainability process improvement, but would also make RED a much more mature and reliable software, which is clearly what one would expect from a tool developed by future software engineering graduates.

1.2.4 Goals

Before formulating the actual goals for the thesis, it would be vital to think of what the future of RED should look like. As it is mainly a university tool, made by and for students, it would be a good idea to allow students to contribute to the project. In order to achieve that, RED should definitely be made open source, with a public access to its source code, and therefore open for improvements from both students and the open source community. As RED is quite an interesting project, it should be possible to make students interested in contributing even in a small factor, by either fixing bugs or introducing simple extensions, while still giving possibility of implementing more sophisticated functionality as MSc thesis projects.

In fact, getting developers help is not the only benefit that RED can get. While *02264 Requirements Engineering* course is mostly attended by computer science students, not all of them may be willing to contribute to the project by writing the code. Such students may be able to help in other ways, such as identifying various bugs and proposing interesting extensions. Introducing an issue tracker will make it relatively easy to report, track and comment on various shortcomings of RED, and would also serve as a way of representing progress of the development. Obtaining such a feedback from the actual users would make it much quicker to identify both bugs, as well as usability improvements. Last, but not least, some of the course attendees could help with testing the introduced improvements before releasing them to the wider public, so that if the changes make RED unstable, they do not impact the course flow.

Another important factor is the maintenance part, which would certainly be the

most challenging part of the thesis. As stated before, maintenance aspect has been severely neglected during the whole RED development process, and it has currently come to a point where it is a show-stopper. In order to counter that, it is important to improve the overall build process, starting with setting up a proper source code repository, so that both the code backup and development history is being preserved. It is also important to simplify and document the steps required to start the actual development, which would aid future student contributors. After that, there is a need of examining the code quality, so that the shortcomings can be identified and the proper high- and low-level fixes can be applied. Lastly, there are number of conceptual weaknesses that RED is suffering from, which should also be taken care of, such as problems with both reporting and weaving features, or improving certain code inconsistencies.

Summing up all of the above, we may organize all the goals in the following list:

1. Improving the RED build process
   (a) Introducing code versioning & issue tracking
   (b) Improving the current *Eclipse* PDE build process
   (c) Investigating other build tools
2. Restructuring RED
   (a) Upgrading the underlying *Eclipse* framework
   (b) Improving high-level architecture
   (c) Improving low-level implementation
3. Addressing conceptual weaknesses
   (a) Fixing report generation
   (b) Fixing model fragment weaving
   (c) Aligning EMF models with generated source code
   (d) Branding
4. Maintaining the existing features
   (a) Keeping as much of existing functionality as possible
5. Making RED more platform-independent
6. Making RED an open source project
7. Making RED usable in the classroom
8. Increasing RED maintainability

All of the above should be addressed with caution, as not only should they contribute to overall RED maintainability improvement, but also may not result in breaking hardly any of the existing functionality. This would be, in fact, one of the greatest challenge of the thesis, as it would result in a number of trade-offs required to be made. Figure 1.3 shows a goal diagram composed out of the goals listed before.

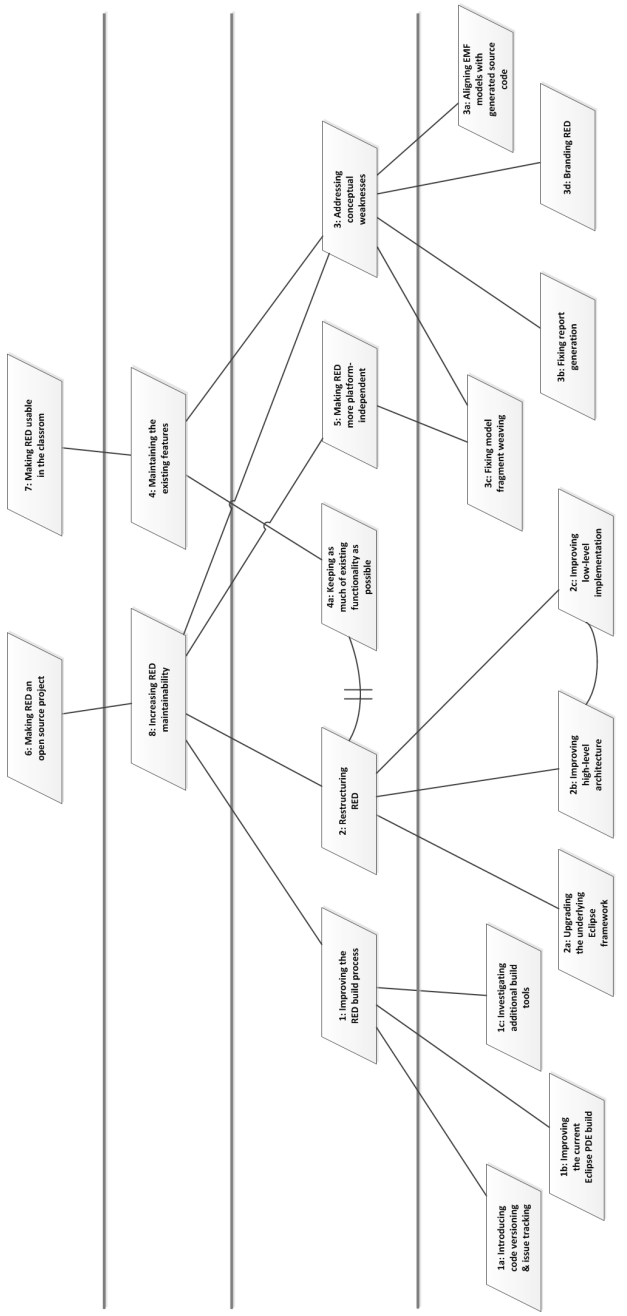


Figure 1.3: Thesis goal diagram

Chapter 2

Improving the RED build process

2.1 Introducing code versioning & issue tracking

As described in Section 1.2.2, one of the main maintainability problems identified with RED is the lack of source code versioning. Since the ultimate goal is to turn RED into an open source project, a good way to start would be to upload RED sources to a version control system, so that it can be accessible and so that any changes made to RED would be tracked. When it comes to code versioning, there are several alternatives in terms of source code management technologies and service providers.

When choosing a version control system (VCS), there are several requirements to be considered. First of all, RED is going to be developed by students, so the technology of choice should be open and also familiar to most of DTU students. The reason for that is that using a VCS should be as seamless as possible, so that MSc students that will be developing RED do not spend much time struggling with the tool. Therefore, using any proprietary technology is hardly an option.

Among of various version control systems, there are two open ones that are most commonly used - Subversion (SVN) and Git. Apache Subversion is a centralized,

open source project founded in 2000 by CollabNet, Inc., which since then have been adopted by many of open source and enterprise software projects. It is still being actively developed by the community behind Apache Software Foundation [Fou13a]. Subversion repositories are also being used at DTU, as every student is offered space on G-Bar servers for their personal use.

Git, on the other hand, is a distributed VCS that has been founded by **Linus Torvalds** in 2005. It was initially supposed to be used for *Linux* kernel designed mainly for efficiency and corruption-safety. The main difference over SVN is that each Git working directory is a separate repository with full version history, which does not depend on any external server. [Wik13c]

Since both technologies are performing equally well, choosing between them is more of a personal preference rather than a result of doing a research. It is safe to assume that DTU Compute students are familiar with both these technologies, and even if not, there are plenty of documentation and tutorials available on the web. One factor that we can compare the technologies against it the portability issue. Since at the time of writing the thesis, it has not been decided yet whether the RED source code will be kept on DTU servers or on the external ones, it would be a good idea to make sure it is relatively easy to migrate the repository used for the purpose of this project to another server if necessary. Taking that into account, Git is probably a better choice due to its distributed design, which is also my personal preference. Also, a very nice feature of Git is that it does not allow to submit any changes without a commit message. Enforcing developers to comment on their changes may be invaluable, as a proper message may let future developer browsing the change tree understand what the actual change is supposed to introduce, without the need of guessing that from the source code.

As stated before, every Git working directory is also a regular repository, so applying Git does not require an external hosting server. However, keeping the source hosted on an accessible server is definitely in scope of making RED an open source project. When choosing a hosting provider, the following are to be considered:

- The repository should be kept private before RED is officially made open source
- Hosting should be free of charge

Aside from hosting RED on DTU servers, which matches these requirements by default, there are few other options. First, there is **github.com**, a popular platform providing code management, issue tracking and code review services for both open source and private projects. It has been founded in 2008 and quickly

became one of the most popular Git hosting service. Due to its social nature, **github.com** promotes collaboration and currently hosts million of repositories developed by over four million developers [Git13]. **github.com** allows only public repositories to be hosted free-of-charge, but there is a educational plan that allows to keep up to 5 private repositories for two years.

Another popular service is Atlassian's **bitbucket.org**. It is a simplified version of a commercial tool called Stash, that Atlassian made available as a public platform. Similarly to **github.com**, **bitbucket.org** offers not only code management functionality, but also bundles a simple issue tracker and documentation facilities with every repository. However, **bitbucket.org** allows to host an unlimited amount of both public and private repositories, but it puts the limit on the number of private repository users. A regular free-of-charge plan allows up to 5 users for each private repository, which is already more than enough in terms of RED, but applying for an academic subscription takes that limit out.

One of the important features provided by both **github.com** and **bitbucket.org** is the issue tracking capability. It has already been discussed that RED could use a way of recording and keeping history of issues captured by the course students. Having a remote issue tracker can make the process both quicker and more reliable than asking students to submit the issues either verbally during the class or by email. By capturing and resolving the issues, we should be able to make RED a better tool in terms of user experience. Also, having issue tracker integrated with the repository makes it possible to quickly identify which code changes have been introduced to fix the issue, which may help the developers to understand certain parts of code.

Table 2.1 shows a comparison between available providers. Considering the initial requirements, **bitbucket.org** is a much better choice thanks to less-restrictive approach to private repositories. Since there is no limit on private repositories, it may be used by Harald to host projects other than RED, and once ready they can be quickly turned into public, open source ones.

Table 2.1: Git hosting providers comparison

	DTU	GitHub	**bitbucket.org**
Internal / External	Internal	External	External
Number of private repositories	Unlimited	5 (for 2 years)	Unlimited
Number of users	Unlimited	Unlimited	Unlimited
Bundled Issue tracker	No	Yes	Yes

To sum up, having a proper source code management tool is vital for such a large project as RED. Starting from the obvious, it servers as a backup copy, which prevents from loosing the source code in case of accidents, such as hard

drive malfunction. It also serves as a central point for every developer working on RED, letting them coordinate their work and review what has already been done. Since the tool is being developed by students that vary in terms of skills and experience, having a possibility to review certain changes made by the the previous developers, along with a proper comment, may be helpful in understanding the context in which a change has been introduced. Last, but not least, having the issue tracker bundled lets us manage not only the source code, but all the issues that have been found and fixed.

2.2 Building RED

2.2.1 Eclipse Eclipse PDE build process

Eclipse Plug-in Development Environment (Eclipse PDE) is a main tool for handling build process of Eclipse-based software. Eclipse PDE provides tools for creating, developing, testing, debugging and building Eclipse plug-ins, features and Eclipse RCP products. The build process is based on Ant, an open source scripting engine widely used for building various Java applications at the time Eclipse PDE was created. It consists of three components:

- UI
- API Tools
- Build

UI component is a set of regular Eclipse plug-ins that allows to seamlessly use Eclipse PDE from within Eclipse SDK. It contains views, editors, wizards etc. that are supposed to make the build process more user friendly. API Tools provide a set of useful analysis tools, such as compatibility analysis and various validations helping developers in finding problems in their plug-ins. Lastly, Build component provides the actual core of Eclipse PDE environment, providing a way of automatized build process. It produces Ant scripts to handle the actual build based on development-time information. [Fou13b]

2.2.2 Current RED build process

As RED is based on Eclipse Eclipse RCP framework, building it is no different than building any Eclipse application. Currently, in order to make RED one

needs to set up the Eclipse SDK, import all the RED projects and export the RED product using a proper export wizard. Since there is no explicit target platform definition, build process will scan the Eclipse SDK's plug-ins and treat them as the target platform. The problem with such approach is that in order for RED to be built successfully, all the plug-ins it depends on must be available in the Eclipse SDK. What is even more important, the plug-ins in the Eclipse SDK must match the versions required by RED, which may be even more difficult to ensure. In fact, versions constraint applies not only to the extending plug-ins, but to the Eclipse platform as well, which means that building RED that is based on a certain version of Eclipse framework requires using the same version of the Eclipse SDK. This results in a situation, where building a stable RED version require downloading Eclipse SDK 3.7 (as using the next Eclipse version, 4.2, resulted in a number of issues), then finding and installing the exact versions of plug-ins as described in [Kra12], and then finally exporting the RED product.

Apart from the tight coupling between make process and used Eclipse SDK, the current setup has one other shortcoming that makes it difficult to build RED. Most of the plug-ins required by RED are widely available from **eclipse.org** p2 repositories. However, there are some third-party plug-ins that are not available neither from official, nor any other repositories, which makes them difficult to find and install. One of such plug-ins was EPF RichText, that has been reused and extended to match RED needs. While EPF (Eclipse Process Framework) is widely available, the richtext editor that it provides is, for some reason, not accessible as a standalone package. Therefore, the decision has been made to include the required source code into RED make process, so that the required component can be obtained during the build. Surprisingly, the other not publicly accessible plug-in, being AgileGrid, has not incorporated into the build process. What is even more problematic, is that the plug-in is no longer maintained in the same form, as the latest versions introduced additional dependencies that are not suitable for RED. Therefore, it may only be a matter of time when the version required by RED will disappear from the AgileGrid website, leaving future developers confused.

In order to improve the make process, both of the previously mentioned limitations need to be addressed. First, we need to find a way of either dropping the legacy AgileGrid dependency, or keeping it in a form that will not depend on any external resources. After that, we will focus on decoupling the make process from the SDK state, which should make the build process much easier to set up.

2.2.3 Resolving external dependencies

As noticed before, dependency on AgileGrid is causing problems during the RED make process. As of now, adding the plug-in to SDK is possible, but requires some effort to do so. However, as the AgileGrid version being used is already quite outdated, it may be the case that it will disappear from the plug-in download site, and the problem will then be much more serious. In order to prevent such a situation, the AgileGrid usage need to be evaluated, and actions need to be taken that will prevent such a deadlock in the future. There are basically three options to consider:

1. Remove the dependency
2. Include the plug-in's binaries in the codebase, and attach it during the build
3. Include the plug-in's sources in the codebase, and build the plug-in along with RED

Since the choice is not straightforward, let us discuss all the possibilities.

2.2.3.1 Dropping (or substituting) the dependency

Since AgileGrid is no longer maintained in the required form, it may be a good idea to make RED independent of it. A first step towards dropping the AgileGrid dependency would be to analyze the code and find what modules/packages require it, as only then could we evaluate how much would removing AgileGrid impact the whole project. From the UI perspective, AgileGrid is being used in two use cases:

- Displaying element associations in Associations View
- Displaying "Management & Tracing" data for most of the editors

Both of them belong to the Core module, and both of them contribute to important features of RED, so it would not be possible to simply remove them from the main product. What could be done, however, is to replace the AgileGrid widgets with another implementation that would provide similar functionality. Following ... , AgileGrid is a new implementation of control-based table based on Simple Widget Toolkit (SWT), that lets developers display various data in

a table format, providing plenty of room for customization. Out-of-the-box, the columns are sortable and their width is scalable, which is not available in standard SWT. The plug-in also provides a pop-up cell editor, which could be helpful when editing cells containg complex data types.

When taking a look at RED, the pop-up cell editor feature is not used there at all. However, both sortable and scalable columns, while not crucial, are certainly nice ones to have, as they make the UI much more user friendly. Hence, when looking for AgileGrid substitute, we should keep these two features in mind.

Unfortunately, there are not many plug-ins that could be used instead of AgileGrid. There is currently no advanced table implementation in Eclipse SWT framework, and the only available alternatives, such as Ktable, NatTable or Grid Widget, are third-party implementations, much like AgileGrid. This means that even if we substitute AgileGrid with any other of available plug-ins, the problem with depending on an external library would still remain. As AgileGrid is currently sufficient enough in terms of features it provides, replacing it would not benefit RED in any way. We could also consider providing our own implementation instead of reusing third-party one. However, it would require a considerable amount of effort that could be spent on developing on more important features.

2.2.3.2 Attaching the binary to the build

A simple way of resolving the problem would be, of course, to keep the backup of the plug-in along with the source code and attach it to the SDK when necessary. This will clearly aid the developers, as they won't have to depend on an external server's state and let them easily find the required dependency. While this is a sufficient option, there is still room for improvements in this area. A much more helpful way of using the AgileGrid binaries would be to include them in the build process itself and let RED simply pick it up. That way we could not only have the copy of the plug-in along with the RED source code, but could also let developers free from the manual step of adding the plug-in to the SDK. While this may sound like a minor improvement, but it will actually contribute to simplifying the development environment setup process. The price to be paid with this approach would be an additional overhead during the make process, as the plug-in will still have to be composed from the binaries. However, the time required to perform such a operation may safely be considered negligible when taking into account the size of whole RED application.

2.2.3.3 Attaching the source code to the build

An alternative to attaching the binaries would be to to attach the source code instead. As AgileGrid is an open-source project, licensed under Eclipse Public License, we are free to use the source code in any open source projects. That way, AgileGrid would be built from scratch during every RED make process and, similarly as with the previously discussed approach, make RED independent of external, unreliable resources. Compared with attaching pre-compiled binary, it certainly gives the developers much more control over the library. This is especially important as the AgileGrid plug-in is no longer maintained by its creators, so if a need for extending the plug-in's functionality ever occurs, it would be possible to implement it right away. Also, since AgileGrid is a relatively small project, building it takes only few seconds, which is totally acceptable and is only a small factor comparing to total time of RED make process.

The only significant drawback of taking this approach would be the fact, that the RED codebase, which is already of considerable size, will grow even further. The size of the AgileGrid source plug-in is shown in Table 2.2. However, since this is a utility plug-in that would not be changed unless new functionality is requested, adding it should not over complicate the RED source code.

Table 2.2: AgileGrid source plug-in size

Metric	Value
Number of Units (classes)	59
ELOC	8849

2.2.3.4 Decision

As different options have been discussed, it is time to decide which one is the most suitable. It has been made clear that dropping AgileGrid dependency is not a possibility, as it serves for a background for two important features. Also, there is no other plug-in that would be a suitable candidate for substituting AgileGrid, as there is no first-party alternative that would provide similar functionality. Since the only reasonable option is to attach AgileGrid to RED build process, the only choice that remains is between attaching a pre-compiled binary and attaching the source code.

The Table 2.3 shows a summary of comparison between two approaches.

It is not surprising that building a binary plug-in is considerably faster, as it does not require compiling the sources. However, the difference of around 2

Table 2.3: Comparison between attaching binary vs attaching source code of AgileGrid

Comparison Metric	Binary plug-in	Source code plug-in
Size (KB)	700	729
Time to build (seconds)	~ 1	~ 2.5

seconds is not much compared to the total build time of RED, so the build time is hardly a convincing benefit over building a source plug-in. What we get from the other approach is the option of customizing the code to our needs. It is hard to tell though when and if such a need will occur, but having more control over the code in use is always a good thing.

That being said, building AgileGrid from sources seems to be the best solution for our needs. Having a copy of the deprecated plug-in, along with a way of customizing it to fit our needs is precisely the combination that RED can benefit from.

2.2.4 Adding target platform definition

Another problem that has been identified is the need of importing all the plug-ins required by RED into the Eclipse SDK that is being used for development. This means that every time a new development environment needs to be set up, either because of using a new computer or because a new developer joined, the whole IDE set up process needs to be repeated.

A good way of resolving this problem would be to provide a way for RED to fetch the required dependencies from an external resource, rather than the IDE in use. Fortunately, Eclipse Eclipse PDE provides a way of doing so by letting developers specify a target platform. By definition, a target platform is a set of plug-ins that the Eclipse RCP will be built and run against. The plug-ins from the target platform are being used during code compilation, as well as during the launch of the product. They are also being used during development, as Eclipse SDK will scan and offer them as candidates for possible dependencies, making it easier for developers to find certain packages in the plug-ins and making sure the target platform contains all the plug-ins required by the Eclipse RCP. [Fou13c]

By default, the target platform contains all the plug-ins that are currently there in the used Eclipse SDK. This is a reasonable default, as starting Eclipse plug-in development does not require any explicit configuration. Instead, the plug-ins used by the IDE are being propagated to the developed plug-in / RCP and one

can start the development process right away. However, when the development reaches a more mature state, specifying an external target platform definition, that is independent of the IDE in use, makes the project much more portable. Also, when defined properly, target platform will contain only a minimal set of plug-ins required to run the desired code, which results in a smaller size of the final binary, as the unnecessary plug-ins are simply not included. Last, but not least, some of the required plug-ins may be platform specific, and therefore it could not be possible to include them in the currently used IDE. This would make it impossible to successfully export the RCP.

A target platform wizard shows that there are a number of plug-in sources that can be used for the platform definition:

- Directory - a directory in the file system that contains a number of Eclipse plug-ins.
- Installation - an Eclipse installation, plug-ins of which will be added to the platform.
- Features - A subset of plug-ins defined by a number of features from the current installation.
- Software site - Plug-ins from either remote or local plug-in repository or update site.

All of these give quite a flexible way of defining the target platform, allowing to use both local and remote plug-in sources. In our case, however, using the locally-defined plug-ins is not really an option, as the developers will be changing every few months, and so will the machines used for building. Therefore, using the remote sites that would allow to fetch the plug-ins whenever necessary is the only reasonable option.

Taking into account RED dependency requirements listed by **Jakob Kragelund** in [Kra12], a list of plug-in sources has been determined and included in RED's target platform. The final platform definition has been depicted in 2.1. It uses the latest official Eclipse 4.3 Kepler download site, which is very likely to stay available in the future, and then uses concrete features to narrow down the platform to the absolute minimum required for building RED successfully. In order for the target platform to be used by Eclipse SDK, one needs to explicitly activate using the target platform editor.

Not only did it allow to detach the build process from the IDE in use, but also contributed significantly in REDucing the size of the final RED binary. Each of RED plug-ins has been examined carefully for the unnecessary dependency

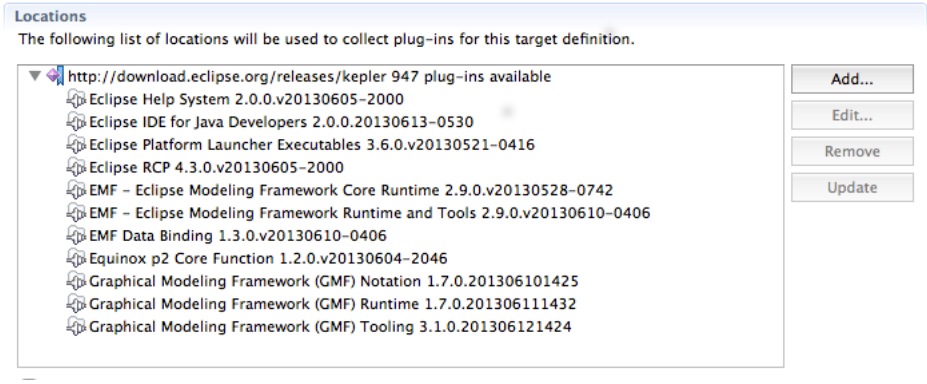


Figure 2.1: RED's external target platform declaration

declarations, that have been either added by mistake or are simply not being used anymore, and then the target platform has been REDuced to contain only what is truly necessary. That allowed to REDuce the binary size from initial 120MB to around 55MB, which is roughly 50% an improvement.

2.3 Eclipse Tycho - a new approach to building Eclipse plug-ins

As regular build process of Eclipse applications has its drawbacks, many attempts have been taken to improve it. Perhaps the most popular one, which has also been adopted by the Eclipse Foundation, is a Apache Maven-based solution named Eclipse Tycho. Eclipse Tycho is being actively developed since 2009 and it has been adopted as the default build tool for all the eclipse core plug-ins. The most recent version is 0.18.1, released on 15 July 2013.

Eclipse Tycho is a set of plug-ins and extensions that provide a way of building Eclipse-based software using a popular Java build tool - Apache Maven. Apache Maven is a command-line based make tool for Java applications, that has been adopted by many Java developers around the world. It offers a unified approach for building various types of applications, ranging from standalone, desktop applications to Java Enterprise bundles, and features a plug-in-based structure that allows extensions. Plenty of plug-ins have been written to extend core Apache Maven functionality, such as other JVM-based language support, advanced dependency-management, or final package structure.

2.3.1 Improvements over Eclipse PDE build

Setting up Eclipse Tycho build to an Eclipse RCP application provides a number of benefits. First of all, it allows to build the resulting product using command-line only, with no Eclipse development environment required. This makes it much easier for a non-developer to build a product from the latest sources, which is certainly an asset for Harald Störrle, who will be building RED from time to time. As Harald is not a Java/Eclipse developer, it will be much easier for him to execute a simple command in the terminal, rather than setting up Eclipse SDK and following the regular Eclipse PDE product export. Also, Eclipse Tycho makes it easy to issue the build for all the supported platforms. By passing a single parameter to the build command, one is able to produce a zip-packaged RED binaries for Windows, Linux and Mac OS, for both 32- and 64-bit architectures. Therefore, thanks to Eclipse Tycho build, one obtains a ready-to-distribute versions of RED for all the possible platforms, that can be simply uploaded and passed to the students.

Another useful feature provided by Eclipse Tycho is automated test execution. As in a regular Apache Maven build, all the tests are being executed during every regular Eclipse Tycho build. If the unit tests are maintained correctly, this gives the developer an instant information about the condition of RED and may help in determining whether or not latest changes introduced any regressions. Tests can also be run outside of the build process, using a separate Apache Maven command. One potential drawback here is that executing unit tests on every build make the build process time consuming. There are two different types of Eclipse-plug-in unit tests - ones that require the UI to be initialized and the ones that does not (called "headless"). Since initializing the UI thread is done separately for every UI test, depending on the number of such tests the make process can take a considerable amount of time. However, this is not much of a problem, as Eclipse Tycho build do not need to be executed during an actual development process. When working on a feature, a developer would normally use the Eclipse-provided Eclipse PDE build to verify his changes on-the-fly, without the need of recompiling the whole source. Only when the work is done, a Eclipse Tycho build should be made, so that one can verify none of the tests are failing, and that the resulting product is working as expected. In such a case, a longer time required to build should not be a nuisance. If, however, a need of excluding test execution on Eclipse Tycho build arises, one could easily do that by passing a "skip tests" parameter to the build command.

What is also important is that Eclipse Tycho build does not replace the Eclipse PDE build, but extends it instead. Therefore, if a for some reason someone ever decides to drop Eclipse Tycho support, it would not affect the possibility of building RED in a regular, Eclipse Eclipse PDE way. While certainly not

advised, this may become relevant when a developer that will be working on RED would find it somehow difficult to attach his contribution to the Eclipse Tycho build process. Doing that is not difficult, but taking into account that Eclipse Tycho is still in its early stages, it may be troublesome for developers not experienced in Apache Maven.

Tycho also works very well with the previously defined target platform. During every build, Eclipse Tycho contacts the defined plug-in sources (Eclipse software site, in our case), downloads the necessary plug-ins and check if there are any new versions of the already downloaded ones. Thanks to Apache Maven caching capability, once a plug-in is downloaded, it is being put in so-called local repository, so that it does not have to be downloaded again. Eclipse Tycho built can be run in an offline mode, so that it does not consult the remote site, but such a build will fail if all the required plug-ins are not cached in the local repository.

2.3.2 Continuous integration

Lastly, while not relevant as of now, Eclipse Tycho build allows using third-party tools to execute the make process. This includes popular continuous integration applications, such as Hudson or Jenkins, which helps in monitoring the state of the code in the repository in a managed, periodical way. Continuous integration tools kick off the build periodically, preferably every 24 hours, and produce a report on whether or not the code from the repository has been successfully compiled and packaged, as well as if the current unit tests are passing. It is especially vital when there are number of developers working simultaneously, as having the external builds being done every day ensures that the code on the repository is stable and that the developers did not introduce any conflicts. While it takes an initial effort to set up such a tool, requiring time, server space, and a bit of knowledge, it provides a great way of monitoring the project stability, which, in the long run, significantly contributes to the project's maintainability. However, as RED is usually developed by a single developer at a time, there is no need to set it up just yet, but it certainly should be taken into account in the future perspectives.

2.4 Summary

To sum up, so far two important maintainability issues have been addressed. First of all, a proper versioning control system has been set up, with a remote repository. Discussing pros and cons, it was decided to use Git repository hosted

on **bitbucket.org**, which supports private repositories at no charge, and comes with a number of additional features, most important of which being a bundled issue tracker. Introducing both source code and issue management is the first important step of increasing RED maintainability and, in the long run, making it an open source project.

The other task was to analyzing and improve the RED make process. By attaching AgileGrid, a required third-party bundle, into the RED source code, we got rid of the need of manually resolving the dependency on each new development environment. Taking into account that AgileGrid is no longer supported, it was of great importance to make AgileGrid available to RED build process without the need of downloading any extra files from AgileGrid website, as it may simply become unavailable at some point. Also, not having to manually download the additional plug-in is certainly at future developers convenience, making the development environment setup process a bit easier.

Another important improvement was defining a target platform. Before, RED make process was tightly coupled with the state of Eclipse SDK in use, and all the plug-ins required by RED had to be manually installed in the SDK first. By adding an external target platform definition, the dependencies are now declared in a separate file and they can be downloaded automatically without much effort required from the developer. It also provides a great way of managing the required plug-in, as it ensures each developer uses the same versions of the plug-ins, minimizing the risk of any incompatibilities resulting from using an outdated, or incompatible version of a certain plug-in from target platform.

As it turned out, using Eclipse Tycho as an extension of a regular Eclipse PDE built greatly extended the build process capabilities. Although it required a bit of effort to set up, it allowed us to build RED without the need of having Eclipse SDK, using a command-line. Also, it makes it possible to execute all the unit tests during the build, which is especially important for checking against any regressions that may occur during development. Eclipse Tycho is based on Apache Maven, an open and popular Java make tool, which means that any developer that is familiar with Apache Maven should not have any troubles with understanding how Eclipse Tycho works. What is more, thanks to Apache Maven we can easily integrate the RED build process with other third-party tools, such as continuous integration systems, with a little effort.

Chapter 3

Restructuring RED

3.1 Upgrading the underlying *Eclipse* framework

When it comes to renovating RED architecture, it would be wise to start with taking a look at the underlying Eclipse framework. RED has been created in late 2010, and the latest available version of Eclipse available at that time was 3.7 Indigo. Until then, there were several updates made to the platform. As the updates contain a number of fixes and new features, updating the framework could benefit RED both in terms of stability and latest upstream features.

While the regular Eclipse updates are mostly about bug-fixing and bringing features, Eclipse 4 is a new generation of Eclipse Platform. The platform has been re-design from the very bottom and is supposed to make it much easier for developers to design and build Eclipse RCPs. Figure 3.1 shows one of such improvements - a GUI editor for the Eclipse Application, which makes it much simpler to design an RCP as when using Eclipse 3.x API. Eclipse 4 was based on a flexible programming model, trying to keep and improve the best features provided by the previous, Eclipse 3.x, platform, while also fixing most of the heaviest shortcomings. It also introduces a new, more straightforward API for RCP development that takes the advantage of dependency injection and object composition rather than object inheritance. With introduction of Eclipse 4.3 Kepler, Eclipse 4 platform has officially replaced the now-legacy Eclipse 3

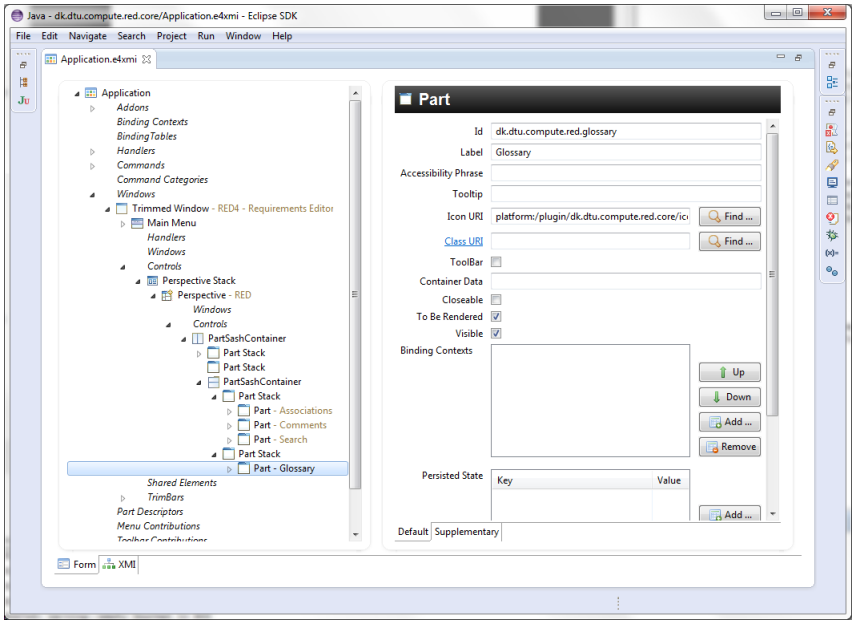


Figure 3.1: Eclipse 4 Application Editor

[Vog12a].

In order to keep the backwards compatibility, a so-called compatibility layer has been developed, which makes it possible to run the plug-ins coded against Eclipse 3 API on the new platform. This is extremely important as the Eclipse project itself is so big it cannot simply be re-engineered to use the new API. Thanks to the compatibility layer, one can run most of the legacy plug-ins and still benefit from most of the features introduced by the new platform. As of now, the only serious limitation of the new platform is that it is not possible to mix legacy plug-ins (coded against 3.x API) and the plug-ins based on the new platform. This issue is supposed to be addressed in Eclipse 4.4, scheduled for June 2014, which should allow developers to include both 4 and 3.x API components in the same RCP. [Vog12a] [Vog12b]

Taking into account all the benefits coming from both major and less-major Eclipse versions, there are several approaches that can be taken:

- Rewrite the existing source code to leverage latest Eclipse 4 API
- Use the latest Eclipse 4.3 with compatibility layer

- Keep the Eclipse 3.7 API compatibility and do not upgrade the framework

Each of the approaches have their pros and cons, let's discuss them in more details.

3.1.1 Migrate RED plug-ins to Eclipse 4 API

Migrating the existing RED code-base may be a good decision in the long run, as it will re-base RED on the latest Eclipse 4 API, which will remain official for the next couple of years. This would give us a full access to all the latest Eclipse features, and would also let us benefit from the support of the Eclipse community. However, RED is already a big project, so migrating the whole source code would certainly be time consuming. Taking into account the size of RED, it is quite clear that such a deep re-engineering would take a huge amount of effort, and it would be difficult to do it in the scope of this thesis.

Another benefit we would get from migrating RED would be that it would force the major architectural re-engineering. Since it would be necessary to rewrite the plug-ins to match the new API, it would be much easier to introduce vital architectural changes during the process. The problem with such approach would be that it may result in breaking some of RED features, during either migration or architecture restructuring process. Fixing the introduced issues would have to be considered when estimating the time required for RED migration, but hopefully making fixes using a new API should not be difficult.

One of the major drawbacks, however, is that while it would be possible to migrate (rewrite) RED plug-ins to the new API, it is not possible to do the same with all the third-party dependencies. This means that even though we can migrate the RED code-base, we would have to do the same with both AgileGrid and RichText plug-ins, which requires additional effort. Also, RED depends on other frameworks, such as Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF), which are still based on 3.X API and there are no plans of migrating them in the near future. Since both of them provide one of the most important features of RED, being fragment modeling, there is no way of not using them. The need of keeping these dependencies may be a major roadblock when it comes to migrating RED, at least until Eclipse 4.4 provides the possibility of mixing both "old" and "new" plug-ins in the same application.

A migration process have been briefly tested as a part of the thesis. Figure 3.2 shows an early version of a RED4 project, an attempt to move RED to Eclipse

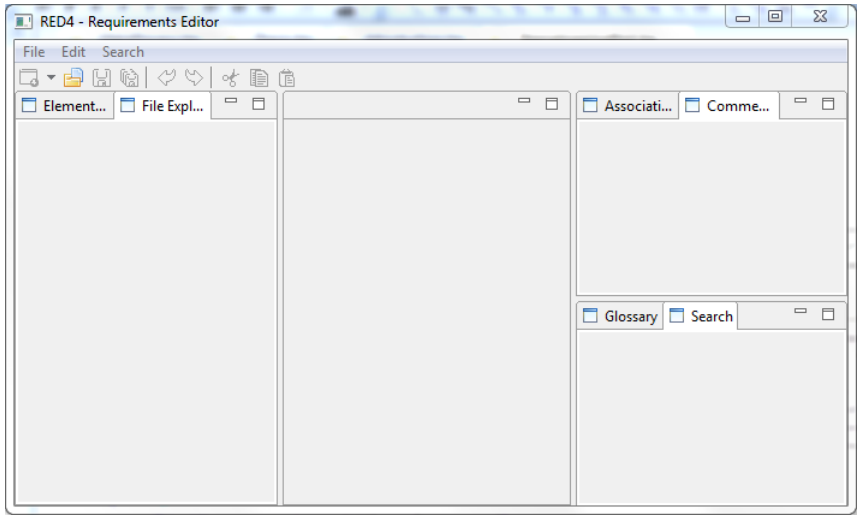


Figure 3.2: RED4 Proof of Concept

4 platform that has been developed as a proof of concept. What is most interesting, not a single line of code has been written to achieve the presented result, which only proves how powerful and flexible Eclipse 4 platform is. Unfortunately, until the release of Eclipse 4.4, further development in this direction is simply not possible due to the previously mentioned limitations.

3.1.2 Using Eclipse 4 compatibility layer

An alternative to a full-scaled migration would be to take advantage of Eclipse compatibility layer. That way, there would be no need of introducing any code changes to the existing plug-ins, we would only need to ensure that the compatibility layer is referenced in the RCP product definition. This would allow RED to benefit from most of the upstream Eclipse fixes and features with almost negligible effort. Also, since the whole Eclipse project is taking advantage of the compatibility layer, as it will not be migrated directly to Eclipse 4, we may safely assume that compatibility layer will be supported in the future by Eclipse Foundation.

As mentioned before, taking this approach would unfortunately prevent us from leveraging the new Eclipse 4 API, as mixing plug-ins is not currently supported. However, with the release of Eclipse 4.4 Luna, such an option should be introduced, hence letting the future extensions be written on top of the latest APIs.

This may be considered as a balanced trade-off, as we would be able to keep the existing code-base without the need of migration, and yet extend RED without having to stick to the legacy APIs.

One problem that has been identified when it comes to using compatibility layer is that it does not always make the legacy code work the same as when using the Eclipse 3.X API directly. A good example of a functionality broken by the compatibility layer was the RED top toolbar, which for some reason has been missing when RED was based on Eclipse 4.2 Juno. A quick research has shown that the toolbar was for some reason not converted into the resulting Eclipse 4 application model, and hence it was not visible to the user. Another issue was that the custom "Save" and "Save As" handlers stopped working, making it difficult for RED users to actually make any changes. Thankfully, with a recent release of Eclipse 4.3 Kepler, both these shortcomings have been fixed.

3.1.3 Keeping the original Eclipse 3.X API

The last option to take in terms of the underlying framework would be to simply keep using the same version that RED was designed against. A clear advantage of this approach is that RED is actually performing quite well when run on Eclipse 3.7. Therefore, there would be no risks on introducing any issues either by the migration process or by using the compatibility layer. However, in this case we would be using a version that is not being developed for more than three years, with no access to the latest upstream fixes. It would also force the developers to use the legacy, outdated API for introducing future improvements, while the support for the old API will be slowly fading away, which could severely impact the development a year or two from now.

3.1.4 Decision

Having evaluated all of the options above, it is time to choose the approach. In terms of the required effort, migrating RED is definitely the most demanding one. Re-writing the whole source code to match the current API would alone be a time-consuming process, let alone migrating the dependencies. Also, even after successfully migrating RED, we would still need a possibility of running the legacy plug-ins along the migrated ones, which will not be (officially) possible before June 2014. Finally, if this option is soon to be provided, the whole migration process would be highly inefficient, as we would be able to run RED as is using the compatibility layer and still develop future extensions using the

new API. Taking that into account, the code migration is certainly not the best approach to take.

Let's consider keeping the legacy API and not upgrading the Eclipse framework. If RED was stable enough, and there were no further development plans, not making any error-prone changes would certainly be a good idea. However, in the current state, RED is neither fully stable nor its development has been finished, so making use of the upstream updates may be beneficial. Also, with the support for Eclipse 3.X API fading away, sticking to the outdated framework may slow the development down in the future.

Therefore, the only reasonable solution is to use the compatibility layer for now, and use the new API for the future extensions as soon as it becomes possible. When that happens, one may actually consider migrating the existing source code one module at a time, but currently it is not an option. Also, as there is practically no effort involved in using the compatibility layer, there is more time to concentrate on the actual RED re-engineering.

3.2 Improving high-level architecture

3.2.1 Eclipse RCP development

Since from the very beginning RED has been developed as an Eclipse RCP, it would be a good idea to analyze RED in terms of how does it implement the actual Eclipse framework. In order to discuss it, let's start with going through the main concepts in the Eclipse RCP development, covering the basics of Eclipse platform and the building blocks of every RCP application.

3.2.1.1 OSGi

First of all, Eclipse platform is based on OSGi framework, standing for *Open Service Gateway Interface*, which is a specification that introduces a service and component based infrastructure on top of Java programming language. Every component within the OSGi environment, called *bundle*, consist of a portion of code and resources organized in a set of java packages. Each bundle may also declare what java packages (or other bundles) it requires to be able to run, as well as the java packages it exposes for other bundles to depend on. OSGi run-time is dynamic, which means the bundles may be deployed and deleted dynamically, following the life cycle shown in Figure 3.3. [Fou]

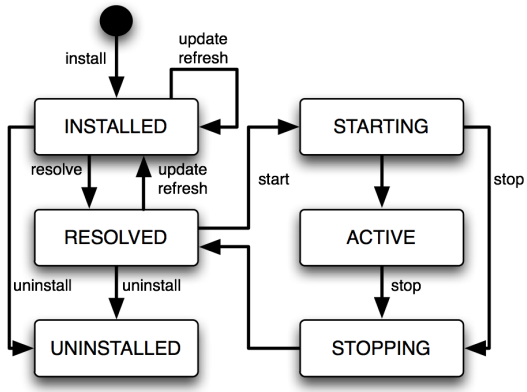


Figure 3.3: OSGi bundle lifecycle diagram

The main idea behind OSGi is letting the developers to create their programs using a small "building blocks" that communicate with each other using fixed contracts. Such an approach is especially useful when there are a number of programmers working on a certain application, as it allows to distribute the work efficiently. A general design pattern is for each bundle to provide a small piece of functionality, which combined with other plug-ins delivers a higher-level feature. Therefore, when developing a bundle, one should design it to be responsible for one thing, and make sure that it does this thing right.

3.2.1.2 Eclipse platform

Eclipse platform is implemented directly on top of one of the OSGi implementations, being Equinox framework, and as such follows the general idea described in the previous section. However, Eclipse introduces its own set of concepts, which heavily extend a regular OSGi specification.

A smallest, deploy-able unit in the Eclipse platform is a *plug-in*, which is essentially an OSGi bundle. Each *plug-in* consists of Java code and resources, and also defines its dependencies and contract. The same as *bundle*, a *plug-in* can also be dynamically deployed in an Eclipse program and therefore contribute to its overall functionality. Additionally, Eclipse allows *plug-ins* to declare so called "extension points", which is basically another way for *plug-ins* to communicate with each other. Extension points can be defined to let other *plug-ins* add functionality to the declaring *plug-in*, which is extensively used within the Eclipse platform.

Another important concept defined by Eclipse framework is a *feature*. Eclipse *feature* do not provide any functionality on their own, but they are simply containers composed by a number of plug-ins. *Features* basically define a higher abstraction level in Eclipse RCP development, letting developers group low-level functionality provided by separate *plug-ins* (or other *features*) into higher-level functionality. It also lets developers define contracts on a higher abstraction levels, which may become important as the RCP application grows.

The final concept that will be described is a *product*. A single *product* defines a stand-alone Eclipse program, that includes all the code and plug-ins required to run it, including Java Run-time Environment and the required Eclipse platform code. Each *product* defines what *plug-ins* and/or *features* it requires, and also contains a custom branding information and resources.

To sum up, Eclipse platform defines a number of "building blocks" that the developers should utilize in order to build their RCP applications. Each of the discussed concepts serves an important purpose in the overall development, defining different abstraction levels and therefore letting developers design their applications in an efficient, maintainable way.

3.2.2 Components

From its very beginnings in 2010, RED was designed to be a modular application, composed of a number of separate, high level components. Figure 3.4 shows a slightly modified version of a high-level design of RED modules and their relations to each other by **Anders Friis** [Fri12], so that it also shows all the changes made **Jakob Kragelund** and **Johan Flod**.

Let us describe the purpose of each component first. At the very base of RED, there is a CORE module. As the name implies, it is the most important module of RED, without which it is simply not possible to run the application. As such, it is responsible for a number of high-level tasks:

- Defining the RED application "look and feel"
- Declaring the basic meta-model
- Declaring the API to be called by other RED components
- Implementing the generic functionality, commonly used by other components

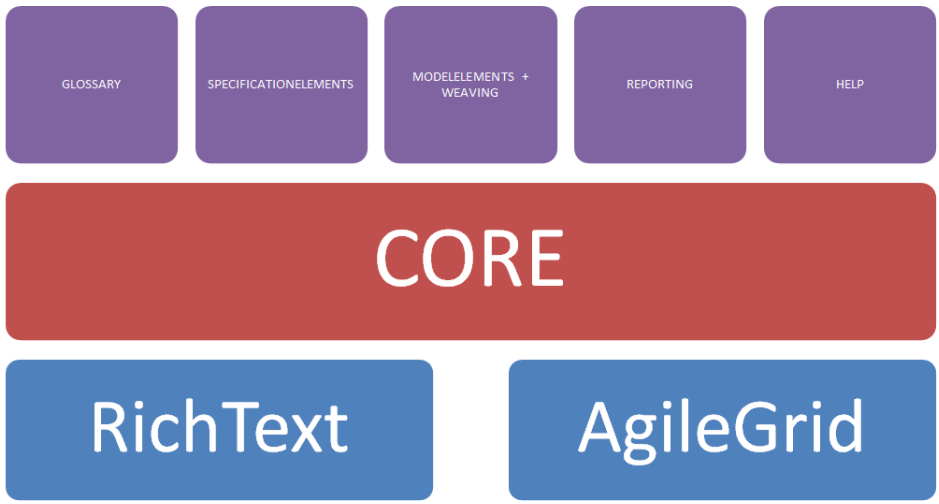


Figure 3.4: RED as designed in [Fri12]

Core makes the extensive use of the API declared by the Eclipse platform, implementing a set of basic UI components for the other RED modules to reuse or extend. In order to implement some of these components, Core requires a set of third-party dependencies, such as EPF RichText editor and AgileGrid, which have been used to implement certain high-level features desired in RED. Although Core module is the only one required to run RED, it does not make RED usable by the final users. All it does is providing a firm, reusable base for the other RED components to introduce new, more user-oriented functionality. On its own, Core provides a way of creating and modifying *.red* files, as well as editing each generic *element's* meta-data.

A first "feature" component of RED is Glossary. At a high level, it gives RED users a way of managing their glossary *terms* they use through their requirement specification in a convenient way. Users are able to create a number of separate glossaries, which in turn may contain a number of terms along with their definitions. Glossary also makes it possible to quickly check a definition of a term used for describing any other *element* in the application.

Next, there is a component named SpecificationElements, which greatly extends RED's requirement editing capabilities. The module defines a great number of RE notions, such as *Personas*, *Goals*, *Stakeholders* etc. SpecificationElements allows to create, modify and link between the declared specification elements, providing a customized visual editors for each of them. Essentially, SpecificationElements is what makes RED a true requirements editor, allowing RED users to formulate their findings using the same concepts

that are being taught in the Requirements Engineering course.

ModelElements is the component providing truly advanced capabilities to RED. While SpecificationElements allows to describe *Requirements* in a basic way, ModelElements lets user to express them as model fragments. The module introduces a full-featured graphical editor for creating UML-like models, that attached to various *Requirements* make them a bit more formal and compliment a regular textual representation with a visual one. Also, included in ModelElements is a Weaving module, that lets users merge their model fragments and then convert them into a Prolog representation, which may then be used to analyze, validate or transform the resulting model using Prolog programming language.

One of the great assets of RED is the Reporting component. It's main purpose is to let the users convert their RED project into a regular report that then will serve as a base for the final deliverable of the *02264 Requirements Engineering* course. The module reads the desired project and converts its contents one by one into an HTML report, that can then be edited either as is, or imported into a MS Word (or alike). This feature makes it much easier for the course attendees to present their findings, and lets them concentrate more on the course material and to minimize the effort required to produce the final report.

Last, there is a Help module allowing the users to access RED user guide within the application. Although RED has been designed to be straightforward to use, having an integrated help systems, describing various features and concepts greatly contributes to the tool usability.

3.2.3 Implementation issues

Having understood the design, let's take a look at how it was actually implemented. The first main problem of RED is that it does not take the advantage of all the Eclipse RCP development concepts, being *features*. Each of the RED modules have been implemented as either a single, or a group of *plug-ins*, with no higher-level entities grouping them together. Hence, the implemented architecture lacked the high-level perspective, and the architecture could be represented only at a *plug-in* level, as depicted in figure 3.5.

As describing the architecture at the *plug-in* level would be too complicated, it was necessary to try matching them into the respective RED modules. Analyzing the *plug-ins* naming revealed that each of the *plug-in* name starts with `dk.dtu.imm.red` prefix, followed by a string that closely matches the RED module names. A similar pattern have been found at the package level in-

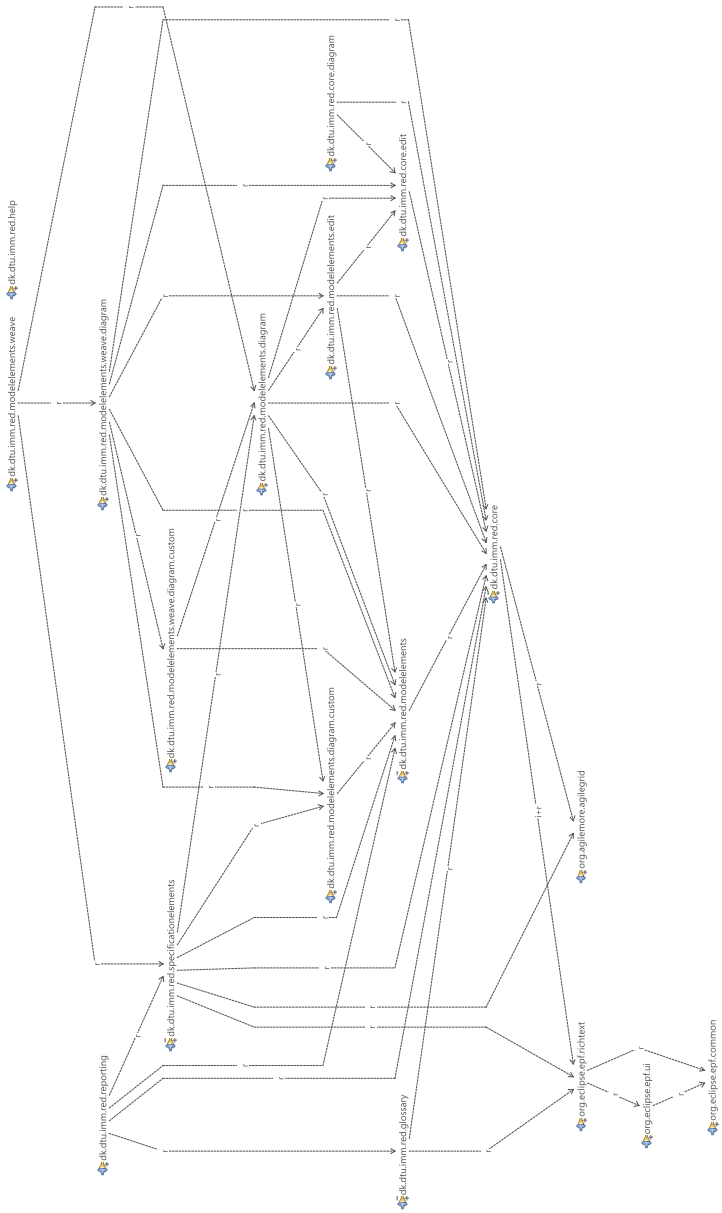


Figure 3.5: RED design as found when starting the thesis

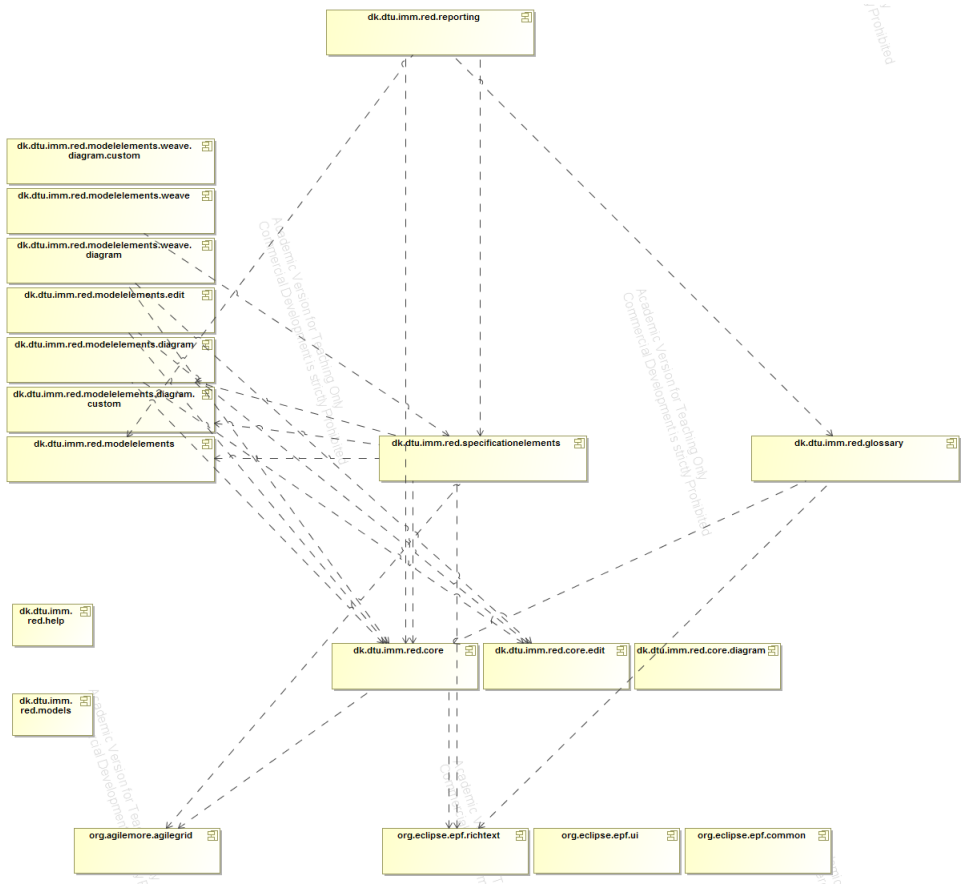


Figure 3.6: Plug-in dependencies after grouping the alike plug-ins together

side various *plug-ins*, as the packages naming conventions also matched the `dk.dtu.imm.red.`*`module_name`* pattern. Thanks to that analysis, it was possible to group the *plug-ins* into the modules and come up with a diagram that shows a better high-level picture (see figure 3.6). While it is possible to see the outline of a higher abstraction layer, without wrapping the *plug-ins* into higher-level entities, it is still difficult to tell the high-level RED module dependencies.

3.2.3.1 Dependency re-export

As the dependency diagram is cluttered a great number of dependencies, we may want to simplify it by analyzing the excessive direct dependencies between

various *plug-ins* and to make some of them indirect instead. Eclipse provides a way for a *plug-in* to re-export its dependency for the other *plug-ins* to use, which would allow to drop the direct dependencies to every *plug-in* that is exported by any other required one. In RED case, a great number of *plug-ins* depend on `dk.dtu.imm.red.core`, which directly depends on both `org.agilemore.agilegrid` and `org.eclipse.epf.richtext`. Since a number of other RED *plug-ins* depend on at least one of them as well, it would be wise to re-export the dependencies on them from the `dk.dtu.imm.red.core` *plug-in*. That way, the resulting dependency diagram will be much less cluttered, as the previously direct dependencies would be converted to indirect ones, through the dependencies on other *plug-ins* that re-export the required functionality.

3.2.3.2 Grouping in Eclipse features

Having the modules implementation separated into a number of plug-ins, and having been able to group them in term of their purpose, we should now come up with a way of grouping them by high-level functionality. The best way to do so would be to utilize the Eclipse platform, which provides a concept just for doing so. *Eclipse features* are a great way of creating a high-level feature units from a number of *plug-ins*, without directly affecting them. The main benefit of doing so is that by reusing that concept, we may implement the desired architecture in a seamless way, as Eclipse provide a simple way of monitoring each *feature's* dependencies. That makes it easy to track all the extraordinary dependencies and countering them whenever necessary.

Another important profit we get by is the actual Eclipse platform compliance, which means that by introducing features, we may reuse other important Eclipse concepts, such as, for instance, update manager. Without *features* specified, it is very difficult to handle the update process using Eclipse Platform manager, as the users would be notified with every single plug-in update available. By introducing features, we may easily release an update for the high-level component, which makes it both convenient for the user and easier to handle for the developers.

Figure 3.7 shows a diagram of RED modules as grouped into respective modules, along with the high-level dependencies between them. Also, thanks to Eclipse *features*, it is now possible to define a clear mapping between RED modules and the RED source code. The mapping is as following:

1. AgileGrid - `dk.dtu.imm.red.dependencies.agilegrid.feature`
2. RichText - `dk.dtu.imm.red.dependencies.richtext.feature`

3. CORE - `dk.dtu.imm.red.core.feature`
4. GLOSSARY - `dk.dtu.imm.red.glossary.feature`
5. MODELELEMENTS + WEAVING - `dk.dtu.imm.red.modelelements.feature`
6. SPECIFICATIONELEMENTS - `dk.dtu.imm.red.specificationelements.feature`
7. REPORTING - `dk.dtu.imm.red.reporting.feature`
8. HELP - `dk.dtu.imm.red.help.feature`

However, as one may notice, even though we were able to successfully map the *plug-in* abstraction level into *feature* (module) level, there is a number of differences between the initial component dependency design and its implementation.

3.2.3.3 Excessive high-level dependencies

First of all, as per Figure 3.4, the modules were supposed to communicate only via the base CORE component. Such an idea favored modularity of RED, as it would then be possible to include or exclude certain modules without the risk of breaking other parts of the application. However, the current implementation clearly violates that principle, introducing a number of inter-dependencies between the modules that suppress RED modularity and make it more "all or nothing" kind of program. It makes RED quite difficult to maintain, as making changes to one module may break other ones, which in turn would make it difficult to coordinate the work between more than one developers. Also, as Eclipse platform is all about modularity, creating such a tightly coupled structure makes it really difficult to utilize some of Eclipse features, such as the Update Manager. Normally, update manager is a convenient way of providing RCP users with the latest software updates, allowing the developers to update certain parts of the application independently. However, if the respective modules are tight together, it makes it very difficult to handle the update process without breaking backwards compatibility.

A good example of introducing extensive dependencies between the modules is REPORTING. Essentially, REPORTING depends on every other RED component, which means that if any of these module is missing, it would not be possible to generate a report. As the ability to generate a report is one of the most important features of RED, it is therefore not possible to exclude `any` of the other components in order to keep it.

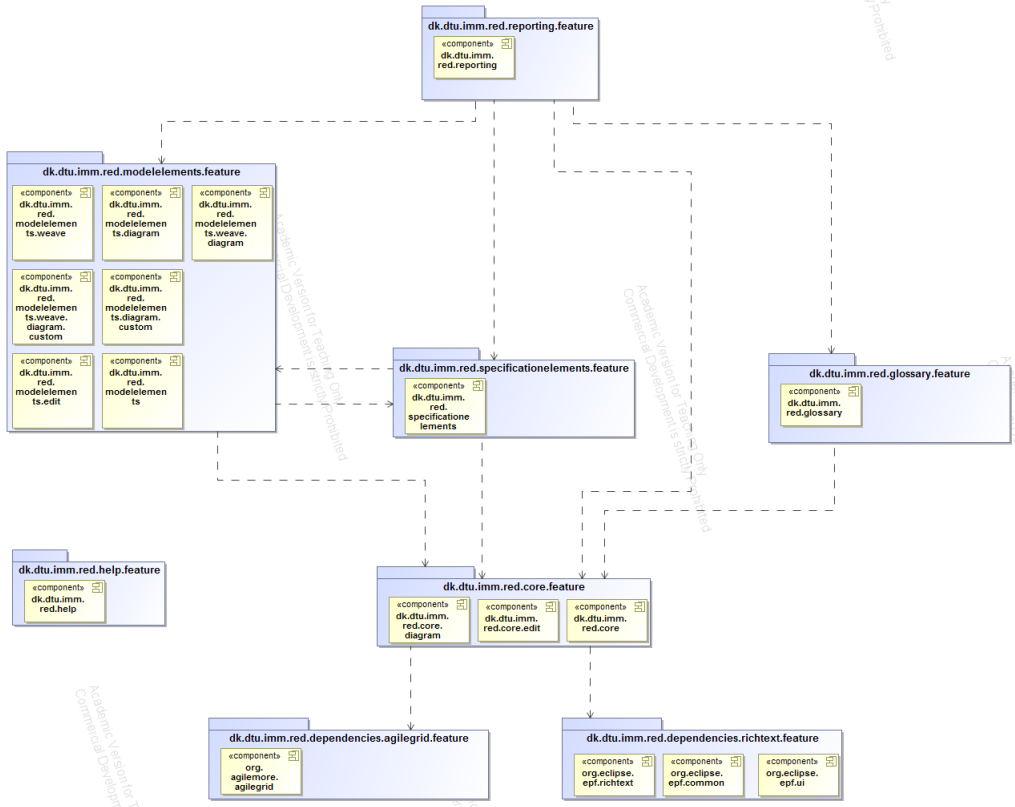


Figure 3.7: RED module dependencies after grouping the *plug-ins* with high-level *features*

3.2.3.4 Circular dependencies

Another problem that RED severely suffers from is that there are modules that depend on each other at the same time. Having such a circular dependencies makes it simply impossible to deploy one of such modules without the other, making it pointless to distinguish between the modules as they cannot function as separate entities. This creates an illusion of modularity, that attempts to hide incorrect design or implementation decisions. It also makes the application difficult to maintain, as if one of the modules gets broken (as a result of the upstream Eclipse updates, for instance), the other module have to be ultimately removed as well. By taking a look at RED modules, one can clearly notice that both SpecificationElements and ModelElements components depend on each other, which means that it is not possible to exclude the fragment modeling capabilities without removing all the specification elements from the RED program. Taking into account that Weaving module (contained by ModelElements) introduces a number of serious problems on non-Windows based machines, not being able to remove it is certainly an issue.

Also, examining Weaving module implementation quickly shows that it is circularly dependent on both ModelElements and SpecificationElements modules, making the overall architecture even more complex. Due to explicit dependencies on the native Java-To-Prolog (JPL) libraries, Weaving module is currently only supported on *MS Windows* machines, but due to the poor architecture, it cannot be simply disabled in both *Mac OS X* and *Linux* versions. Unfortunately, this creates a direct impact on RED users using these operating systems, as trying to weave a model on their machines either throws a set of exceptions or crashes the whole program, confusing them and leaving under the impression that RED is not stable enough for daily usage.

3.2.4 Countering cyclic dependencies

As identified before, circular dependencies are the most serious problems when it comes to RED architecture. In order to resolve them, a number of steps has been taken, with the overall process depicted in figure 3.10. As the first step of fixing the issue, Weaving module has been taken out of ModelElements module, so that it would be possible to analyze the dependencies at a deeper level. It was quickly found out that a number of *plug-ins* from the original ModelElements module are purely responsible for the model weaving functionality, so they could have been extracted and wrapped into a separate, `dk.dtu.imm.red.weaving.feature`, representing Weaving module.

Once extracted, it is now possible to examine how the high-level dependencies are organized between SpecificationElements, ModelElements and the newly created Weaving modules. Clearly, it is the Weaving module that creates problems here. Initially, Weaving has been implemented as a part of ModelElements module, which single purpose was to provide a way of converting a set of model fragments to Prolog. As such, implementing the weaving functionality inside the ModelElements module was not a bad idea, but making it an integral part of the module certainly was. The main reason behind is that, looking from a high-level perspective, the ability to weave a model should not be essential to the ability of creating the model fragment. At the same time, weaving functionality should not be required to manage the specification elements, which is what is currently happening. As mentioned before, model weaving is currently supported only on Windows computers, due to the underlying technology limitation, which makes it useless on any other RED versions. Therefore, it would be more than vital to simply disable it on the unsupported platforms, but since two other important modules strongly depend on its existence in the run-time environment, it is currently not possible.

Source code examination reveals that ModelElements and Weaving modules share a common domain declaration. As many Eclipse RCPs, RED's domain classes have been generated using Eclipse Modeling Framework (EMF). EMF is a powerful tool, that allows us to design a domain model using a user-friendly graphical UI, store its declaration in so-called Ecore model and then to generate Java classes out of it. While very convenient, in order to map both classes and packages, the generated model is quite complicated and difficult to understand. The reason for that is that EMF generates all the code-base required for Eclipse platform integration, not to mention a number of utility and factory methods. What is most important, is that even though ModelElements domain does not directly depend on Weaving models, the generated framework-related code introduces such a dependency, therefore causing a cyclic dependency between the modules.

The other problem is the cyclic dependency between Weaving and SpecificationElements modules. This is actually a surprising one, as from the architectural point of view, there should be no need for having such a dependency in neither direction. As weaving process is being done on the model elements only, the dependency on any specification element should not be necessary. Fortunately, that problem turned out to be very simple to solve, as the dependency was required only by an unused class import. Removing that import allowed us to remove the high-level dependency between the modules, which was the first step towards dropping cyclic dependencies.

Conversely, having SpecificationElements depending on Weaving is even more surprising, as it should be able to work as expected without even being

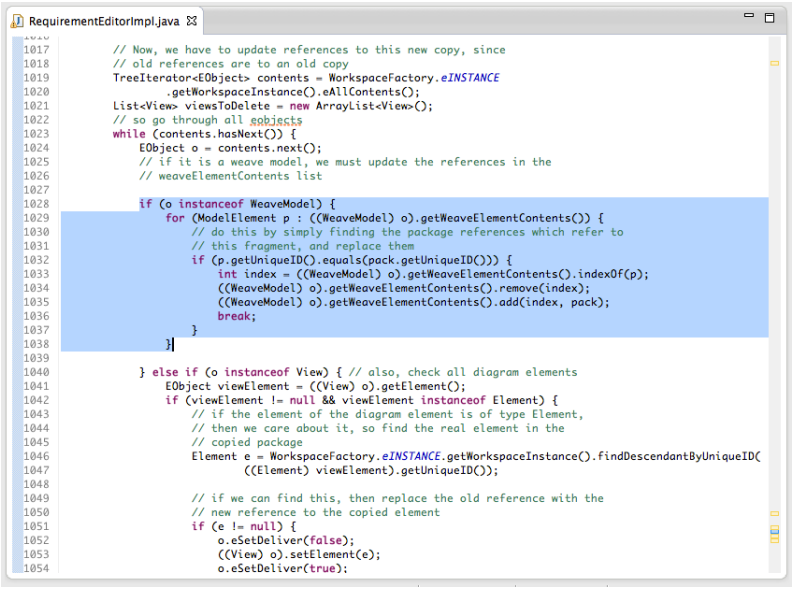


Figure 3.8: Specification Elements dependency on Weaving

aware if weaving capability is there in the RED run-time or not. Unfortunately, this dependency was much more complicated to drop. Recalling **Jakob Kragelunds** research in [Kra12], there are currently two distinct model editors in RED - a *Requirement's* model fragment editor, representing one more way of specifying a *Requirement*, and a so-called weaving model, which is basically a result of merging several model fragments. The problem is that weaving model, instead of being a snapshot of the merged model fragments, contains references to the same model elements as the model fragments that have been merged. Having said that, whenever a change is made to one of the *requirement*, RED needs to check if the *Requirement's* model fragment is a part of any weaving model, and update that model accordingly by updating the element indices. This code responsible for updating the weaving models is shown on figure 3.8, lines 1028-1038.

In order to fix that, let's take a look at the WEAVING module in more details. Figure 3.9 shows that the `dk.dtu.imm.red.modelelements.weave` package contains classes that have two distinct responsibilities. First, there are classes such as `WeaveModel` or `WeaveAnnotation` which are being used to compose and display the actual weave models. However, there are also classes here that are purely responsible for Java-to-Prolog conversion (weaving). While the the first group of classes is perfectly valid to have in the domain layer, all the Prolog conversion ones are not. The reason for that is that they are not used to store

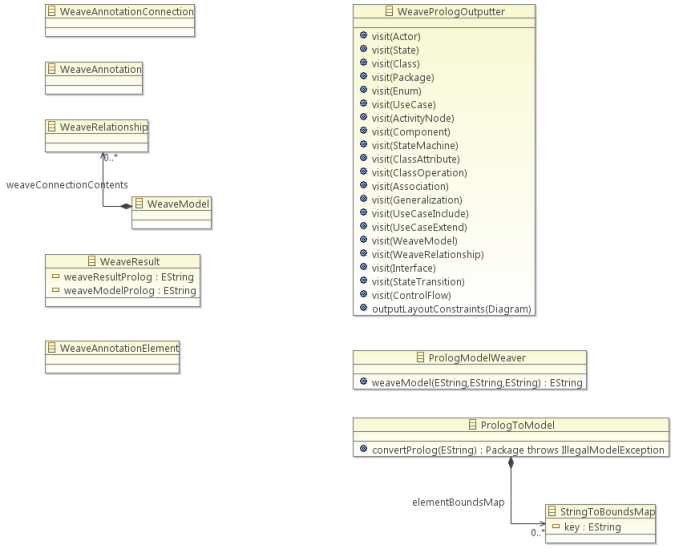


Figure 3.9: *dk.dtu.imm.red.modelelements.weave* package class diagram

any state in the system, but are just being used to handle the conversion process. Hence, it is far more reasonable to have them in a service, business logic layer, as this is exactly what they do.

Having that said, it would be a good idea to re-define the boundary between ModelElements and Weaving modules. Instead of extracting the whole "weave" package out of ModelElements module, we may consider keeping the weave model related classes, and extracting only the Prolog weaving ones. That way, we would keep the generic weave modeling capabilities inside the ModelElements package, while moving the specific Prolog implementation to another module. In order to do that, however, we will need to first split the EMF generated model which, as described few paragraphs above, ties the whole domain code together. We could try editing the source and trying to remove the dependencies manually. However, as every generated source code, the generated EMF model is difficult to edit, and it is generally not a recommended approach. What we could do instead, is to split the actual EMF Ecore model and generate the source code afterward. EMF models are stored in *.ecore* XMI files, which can be edited using EMF editor. In order to successfully split the Prolog weaving implementation from ModelElements one, the following steps had to be completed:

1. Backup the originally generated source code. As some of the domain

classes may have been edited manually after being generated, we do not want to loose the customizations

2. Remove the originally generated source code from the `dk.dtu.imm.red.modelelements` plug-in.

3. Create a new, empty EMF Ecore file. This file will contain the extracted prolog weaving domain.

4. Move the classes responsible for prolog weaving from the initial Ecore model to the newly created one.

5. Copy all the Ecore file properties. Every Ecore model contains a number of settings that will be used for generating the Java model. Since we want the newly generated model to perform the same as before the splitting operation, we should keep the same settings as in the original Ecore file.

6. Generate the new, splitted Java model from both files.

7. Append all the customizations on the new model. In order to do that, we need to *diff* the new model against the backup we've made in step 1. While time consuming, this is the most essential part of the domain splitting process, as if we do not apply the customizations, the domain will not operate properly.

8. Validate the changes. We need to *diff* again, and carefully examine the changes that have been introduced, and make sure non of the model fields or operations are lost. Also, we should make sure that the only changes between the original and the modified models are related to splitting package dependencies.

Having completed these steps, we obtained two sets of generated classes, but in contrast to the original model, the prolog weaving classes are no longer required by the ModelElements module domain, which breaks the cyclic dependency. By keeping the weaving model classes inside ModelElements module, we have also made broken the SpecificationElements dependency on Weaving, as *Requirement* element depends only on the generic weaving model classes, and not on the Prologimplementation. Figure 3.10 shows a complete process of fixing the circular dependencies. While there are still dependencies before the modules that do not comply with the initial design, breaking the cyclic dependencies is the first important step in improving the general RED architecture.

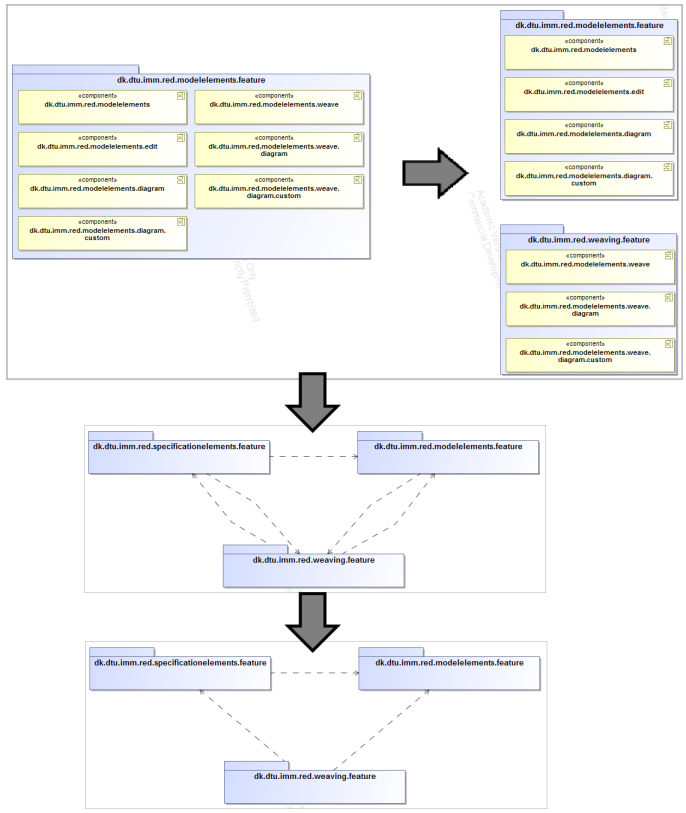


Figure 3.10: RED SpecificationElements and ModelElements circular dependency fixing process.

3.2.5 Examining dependencies between the modules

After fixing the cyclic dependency problem, we may now concentrate on the general problem of excessive dependencies between various modules. Even after resolving the problem with circular dependencies, recalling figure 3.7, there are still several inter-module dependencies that should be removed according to the design. These are:

- Reporting depends on SpecificationElements, ModelElements and Glossary
- SpecificationElements depends on ModelElements
- Weaving depends on SpecificationElements and ModelElements

3.2.5.1 Reporting

A Reporting module is considered to be one of the most important features of RED, hence it has to be approached carefully. Basically, generating a report requires to read every single element of the RED project and convert it into a report paragraph. Currently, report generation is purely centralized, which means that it is the Reporting modules that obtains a reference to a *Project* element, scans its contents looking for the types it can process, and converts these types to desired paragraphs. In such a case, having the dependencies on the other modules, such as SpecificationElements, ModelElements or Glossary is necessary, as Reporting has to be aware of the types a regular RED project may contain.

One way of changing that would be to make the Reporting module more distributed. Since RED meta-model ensures that every single domain class extends from an *Element* super-class, we may simply add an abstract method, like *toReport()*, which could then be accessed by the report-generating code. That way, Reporting component would no longer need to be aware of the particular types, but will operate on a generic *Element* type instead. However, it may be difficult to design that method so that it return a data-type suitable for every single *Element* subtype. For instance, *Persona* element consists not only of textual data, but contains an image as well, and *Requirement* has a model fragment attached to it. Covering such cases would make it difficult to make reporting a distributed process, while keeping the current reporting functionality. Also, reporting is actually working good as of now, and there are other, more important issues with RED that should be handled in the first place.

3.2.5.2 Weaving

Even though the cyclic dependencies have been handled, the WEAVING component still depends directly on both SPECIFICATIONELEMENTS and MODELELEMENTS modules. As per [Kra12], weaving process converts a number of requirements, along with their model fragments, into prolog scripts. Hence, WEAVING module needs to be aware of both *Requirement* element, as well as all the *Model Elements* from the MODELELEMENTS module. As with the REPORTING module case, we could try making WEAVING module distributed rather than centralized, but in this case such an approach is definitely not recommended. The reason is that the actual logic for prolog conversion is already complicated, and making it distributed would make it complicated even more. Also, in order to work properly, weaving requires native JPL libraries, and by distributing the conversion process, we would make both SPECIFICATIONELEMENTS and MODELELEMENTS dependent on their existence, which is simply not acceptable.

3.2.5.3 Tool - a new module type

Since there is no easy way of making making the previously mentioned modules comply with the design, let's examine the design itself. Currently, the design forbids the inter-module dependencies, unless they communicate via the base CORE module. As may have been noticed, while it makes the architecture design structured and easy to understand, it is often difficult to implement all the desired functionality in such a way. In fact, all the functionality that need to process the RED model are by definition violating the design, including both REPORTING and WEAVING modules. Hence, it may be a good decision to redefine the design a bit.

As not depending on other modules is not always possible to achieve, perhaps there should be a way for a module to both keep compliant with the design and still be able to depend on another module other than CORE. However, letting developers add such modules (let's call them **Tools**) without any constraints would quickly make RED architecture deteriorate and therefore even more difficult to maintain as it is now. Hence, if we are to allow **Tools** into the architectural design, they would have to comply with the following rules:

- A module must not be referenced by any other modules, so that it may be removed without affecting other RED functionality
- After successfully implementing a **Tool**, one needs to evaluate whether it is possible to make it independent on other modules, and if so, re-factor the module to comply with the initial design.

By following these simple principles, we make sure that introducing such a possibility is, to some extent, safe. If the **Tools** are not being referenced by other **Tools**, we will retain the possibility of removing them if necessary without affecting other parts of RED. The second rule should encourage the developers to still try to comply with the initial RED architecture design, at least making them think if it is possible (or efficient) to implement the functionality they are working on in a different way.

In fact, introducing such a possibility would provide a number of benefits. First of all, we would have a proper "bucket" for both REPORTING and WEAVING modules. As discussed before, making them compliant with the initial design will be either inefficient or simply impossible, and yet we'd like to keep the functionality they offer in RED. A second plus coming from introducing **Tools** module type is that it provides a great entry point for new functionality in RED. Whenever a developer comes in and starts implementing a feature, such a detachable module makes for a great "sandbox" for the implementation without affecting the other modules. Once mature enough, such a **Tool** may be converted to a proper RED module, becoming a perfectly valid part of RED code-base. This is especially useful as RED is being developed by students, which greatly differ in both skills and programming experience. If an unskilled developer start modifying the base source code, chances for breaking existing features are quite high. However, if such a developer may start by simply creating a module that contributes to RED but is not required by it, this should make RED development much safer.

3.2.5.4 Specification and Model elements dependency

Let's start with SPECIFICATIONELEMENTS and MODELELEMENTS modules relationship. When introducing fragment modeling feature, it was decided that model fragments will be implemented as part of each *requirement* specification element. Therefore, the SPECIFICATIONELEMENTS module dependency on MODELELEMENTS originates from a *requirement* having a direct containment on a `ModelFragment` class, which is a difficult one to break. Figure 3.11 shows the current *Requirement* element editor, and how the model fragments are incorporated into it.

One way of resolving that problem would be to make *Model Fragments* separate from *Requirements*, but since *Model Fragment* is considered as one of *Requirement's* representations, dropping this relationship is probably not a good option. Keeping these two elements entirely separate would make it difficult to see their relation and would therefore impact RED usability.

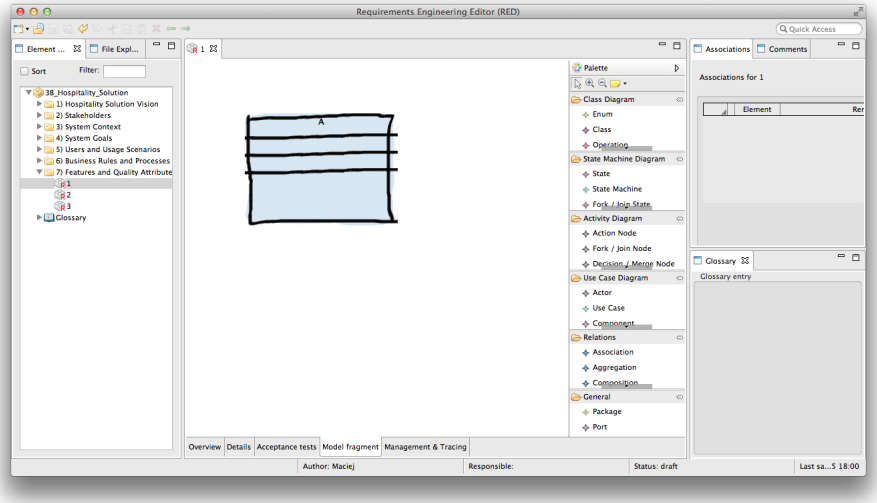


Figure 3.11: *Requirement* element visual editor

Another way of handling the problem would be to remove the containment dependency, and representing the relationship using RED associations model. That way, it would still be possible to reference a *Model Fragment* from a certain *Requirement*, but it would require to drop the convenient "Model Fragment" tab from the *Requirement* editor. However, it would probably be possible to implement the missing tab using a Tool module type, hence keeping the current functionality, but dropping the direct dependency between the two components.

For now, it has been decided to keep the dependency as is. The main reason for that is that ModelElements module has been identified to cause a number of issues, such as model elements not being properly saved or breaking the reporting module. As the issues are serious, it is probably not a good idea to introduce more changes that may break the functionality even further.

3.2.6 Dividing Specification Elements

Having discussed the high-level dependencies between various modules, let us look with more details at SpecificationElements module. Basically, the module itself introduces a number of Requirements Engineering concepts into RED, making it such a useful tool in *02264 Requirements Engineering* course. However, taking into account that each of the introduced concepts is essentially

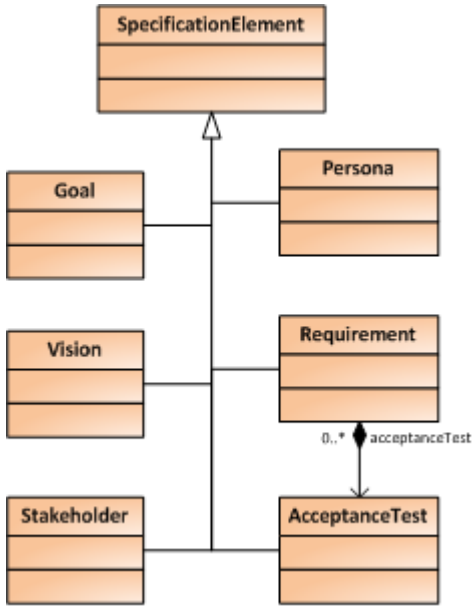


Figure 3.12: Supported *Specification Elements* [Fri12]

a separate feature, it may be vital to discuss whether or not it is a good idea to make them separate (sub)modules.

Figure 3.12 shows a simple diagram of all the *Specification Elements* provided by SpecificationElements module. Each of these elements represent a distinct addition to the overall system, most of which are not directly related to each other. In fact, it should be possible to model them as distributed, plug-gable modules, instead of a single, large one.

The main problem with the current module structure is that it is "all or nothing" package. One cannot simply take out one of RE concept out of the package, but have to include all of them instead. This has already proven to be a problem with the *Scenario* case, which resulted to be non fully functional when delivered. However, as it has been incorporated into a large SpecificationElements module, and not as a separate entity, it was not possible to easily remove the not working part. The size and complexity of the SpecificationElements module is also a problem. As all the specification elements are implemented as a single module, it contains a lot of source code, which is quite difficult for one to understand without spending some time analyzing it. This increases the time required for the new developers to start the actual development. The last problem is the fact that in the current setup there is no possibility of enforcing

the RE concepts independence, which, as in case of *Scenarios*, may make it possible for developers to introduce unnecessary dependencies between them.

That being said, splitting the SpecificationElements module into a number of sub-modules makes perfect sense, but it will require a certain amount of effort. Since the module has been treated as one so far, splitting it will require to identify and drop the unnecessary dependencies. However, doing so will greatly contribute to the general architecture improvement, as we will end up with a number of small, plug-able and independent units, rather than a huge module. The separating process required to manually divide the `dk.dtu.imm.red.specificationelements` *plug-in* source code between a number of smaller *plug-ins*, which could then be wrapped into the *features* corresponding to each of the *Specification Elements*. Also, since the *Acceptance Test* is not being used as a separate element, but it is an integral part of *Requirement* element, it has been included into the `dk.dtu.imm.red.specificationelements.requirement.feature`.

The result of separation is shown in figure 3.13. The `dk.dtu.imm.red.specificationelements.feature` has been reduced to the absolute minimum, containing a simple wizard category definition, and few classes for the other modules to reuse. It is important to mention that due to the SpecificationElements module splitting, it was possible to reduce the initial SpecificationElements module dependency on ModelElements to Requirement sub-module only, as this is the only *Specification Element* that depends on ModelElements.

3.2.7 Feature-based product

Last, but not least, we should introduce a simple change to the RED *product* packaging. Since we have already specified a number of *features* in RED, we can now compose the resulting RED RCP on them. A clear benefit for that is we do not need to update the *product* file with addition of a new *plug-in*, as long as it is covered in an included *feature*. Also, having the *product* specified on the *features* makes it much easier to understand on what high-level features it is based on.

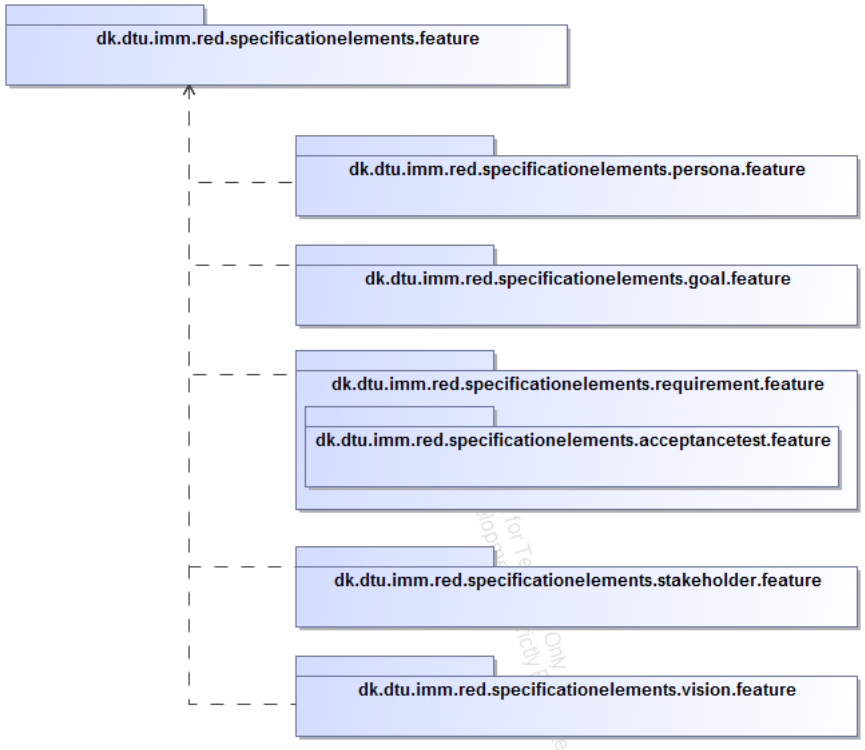


Figure 3.13: The result of separating the SpecificationElements module into a number of sub-modules

3.3 Improving low-level implementation

3.3.1 RED plug-ins implementation problems

Having gone through the basic idea behind RCP development, let's take a look at RED as an Eclipse RCP application. From the very beginning, RED has been designed in a modular way, consisting of number of components that serve different purpose. While such an approach was definitely a correct one to take, as RED provides great high-level functionality, the way RED modules have been implemented in the Eclipse platform results in a number of problems.

First of all, as described in [Fri12], most of RED components have been implemented as single *plug-ins*. Taking into account the fact that each component provide a great number of features, such a design decision does certainly not comply with the Eclipse RCP programming best practices. As a result, instead of having each component represented by a number of small, efficient units that all contribute to the overall component functionality, modules such as CORE and SPECIFICATIONELEMENTS are each implemented by a single plug-in attempting to provide it all. This makes RED low-level implementation complicated, and the actual source code units over-sized, which makes it difficult to maintain RED in the long run. Also, because of violating the general RCP development principles, it is difficult for new developers, especially ones familiar with Eclipse programming, to start the development quickly. This is because each of the plug-ins is responsible for multiple operations, which is rarely expected in Eclipse programming, and it is therefore difficult to find the code responsible for certain behavior.

However, what is the most problematic with the current approach, is that having such oversized plug-ins favors inefficient Java code implementation. The code is often unnecessarily complex, with a number of circular dependencies between both packages and classes, which makes it very difficult to modify. That being said, adjusting the source code to match the high-level design discussed in section 3.2 will be a time-consuming process, but it will surely pay off in the maintainability point of view.

3.3.2 Restructuring plug-in implementation

As discussed before, introducing Eclipse programming best practices would contribute to the restructuring process. Splitting the modules is the most time-consuming, but the most important part. The plug-ins should be investigated

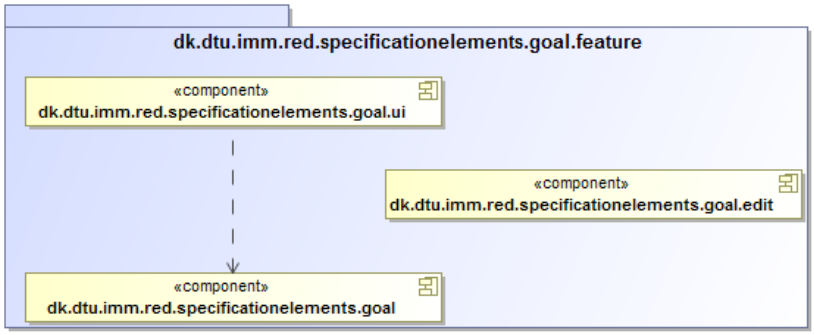


Figure 3.14: GOAL module *plug-ins*

carefully, and they should be separated in such a way that they do not depend on each other in a circular way. Also, an attention should be put to separate the generated code from the regular one. First of all, generated code is often verbose, and is often placed in a large number of classes and packages, which makes it difficult to understand. Also, very often the generated code should not be modified at all, or the customizations should be reduced to minimum, so that it may be re-generated if necessary. Therefore, mixing the custom code with the generated one makes it difficult to distinguish between them, and makes such plug-ins really difficult to maintain.

3.3.2.1 Introducing layer separation

That being said, we can separate the outstanding plug-ins into a number of smaller ones, that provide different functionality. A natural division would be to implement the domain-related code in one *plug-in*, and the user interface code in another. Also, as with every EMF model, we would need a separate *plug-in* containing the model editing domain. That way, we would not only split the plug-ins in the different functionality layers, but would also separate the auto-generated code from the custom one. As an example, figure 3.14 shows how the GOAL module has been divided into a set of plug-ins using the described scheme. It is important to notice that such a configuration allows a clear dependency structure, as the UI *plug-in* depends on the domain *plug-in*, but not the other way around. That way, we make sure that no domain code is independent of the presentation layer, which is a good architectural practice.

When moving the UI-related code into the, there is another improvement that could be made. Basically, all the outstanding plug-ins contain a lot of Java

packages, but the code is not organized into them very well. Basically, the same packages are being used to hold the classes serving different functionality. Therefore, it would be wise to introduce a certain package structure, that would separate different Eclipse UI concepts, such as editors, commands and wizards. As a result, each UI *plug-in* should be organized into the following Java packages (some may not be required for certain *plug-ins*):

- `*.ui.actions`
- `*.ui.editors`
- `*.ui.extensions`
- `*.ui.handlers`
- `*.ui.operations`
- `*.ui.views`
- `*.ui.wizards`

Organizing the code in such a way makes it possible quickly distinguish between the various UI elements, and should also prevent from circular dependencies between the packages.

3.3.2.2 Separating `dk.dtu.imm.red.core` plug-in

While layer approach was enough for most of the modules, the CORE module required a bit more attention. As it is the base module of RED, it's got a few more responsibilities than simple *domain* and *UI* implementations, such as:

- RED Application code
- Reusable API components (generic views, editors, and wizards)
- Custom UI elements

Therefore, CORE module should probably be divided in a greater number of plug-ins. However, as it is one of the most important modules, modifying it needs to be approach with great care and attention. As it turned out, both the reusable UI components and the domain ones are unfortunately tightly coupled. Because of that, splitting them into separate plug-ins, while not introducing a

cyclic dependency, is both difficult and time-consuming. As despite the effort spent on that task, it was not possible to separate it successfully while keeping the initial functionality. It was, however, possible to extract the application-related code, which is an important step if it was ever decided to distribute RED not only as Eclipse RCP, but also as a downloadable extension to Eclipse IDE.

3.3.3 Removing unused source code

One of the consequences of the oversized *plug-ins* in RED was the fact that in between of the huge number of packages, there were certain pieces of code that did not serve any purpose whatsoever. While it was difficult to notice them at the beginning, thanks to the *plug-in* restructuring process, it became obvious that some parts of the code are not responsible for any functionality. There were basically two such pieces of code found:

1. A *Specification Element* wizard
2. Auto-generated *Scenario* editor

As for the *Specification Element* wizard, it seems like one of the initial developers of RED attempted to create a generic wizard for *Specification Elements*, that would let the user create a new concrete type of *Specification Element* from a single wizard. However, probably at some point they have realized that this is not a correct approach, as various *Specification ElementS* should be created using separate wizards, and therefore this functionality has been either deactivated or simply not finished. Either way, as there is no need for having such a "generic" wizard, it was decided that this part of code should be totally removed. However, if anyone wishes to recover that code, he may revert to *7a0e64a21* revision.

The second unused part of the code comes from the SCENARIO module, which contains a two distinct *Scenario* editors. One of them is the a custom one, written by **Johan Flod**, and is actually being used as a default *Scenario* editor. The other one is the auto-generated editors created using EMF framework, perhaps as a side effect of generating *Scenario* model classes. As the auto-generated editor is not suitable for any user-friendly usage, and (if really necessary) it can be re-generated from the EMF Ecore model, it was decided to remove it as well.

As a result, we have removed two considerable pieces of unused source code, decreasing the overall code-base and making it less troublesome for the future developers to browse through and to understand it.

3.4 Summary

First of all, we have evaluated various possibilities regarding the underlying Eclipse framework, and updated it to the latest stable version. As a consequence, we've got access to most of the latest features provided by Eclipse platform, while not having to update the actual source code by using the compatibility layer. Also, since the compatibility layer's capabilities are being extended with every new Eclipse framework release, we made sure the approach we took is not only the best one to take in the current situation, but is also a good one in term of the future of the project.

Apart from that, a number of improvements has been made from the high level perspective:

- All the modules have been implemented as Eclipse features
- The circular dependencies have been resolved
- A new module type has been introduced
- All the dependencies between the modules have been elaborated on
- The SpecificationElements module has been restructured

As a result, the RED architecture has changed quite a lot, making it more distributed on one hand, but more flexible on the other. Due to the changes, some of the modules can now be excluded from the application without breaking other features, which is a great improvement in the project's maintainability. The RED modules have been reimplemented on the Eclipse platform, as the *plug-ins* have been grouped by their high-level functionality using *features*. Also, by defining a new **Tool** module type, we have provided a great entry point for new developers, that would let them extend RED without impacting its "core" functionality. Last, but not least, it specified a number of high-level contracts, enforcing which should make the low-level implementation part much less complicated, and therefore easier to understand.

Figure 3.15 shows a new, improved RED architecture design.

As of the implementation part, the *plug-ins* in the RED modules have been analyzed and separated into different functionality layers. Introducing a clear dependency structure between the introduced *plug-ins*, preventing unnecessary complexity. Moreover, the UI *plug-ins* packages have been restructured, so that the various UI elements are stored efficiently. Also, a number of unused code

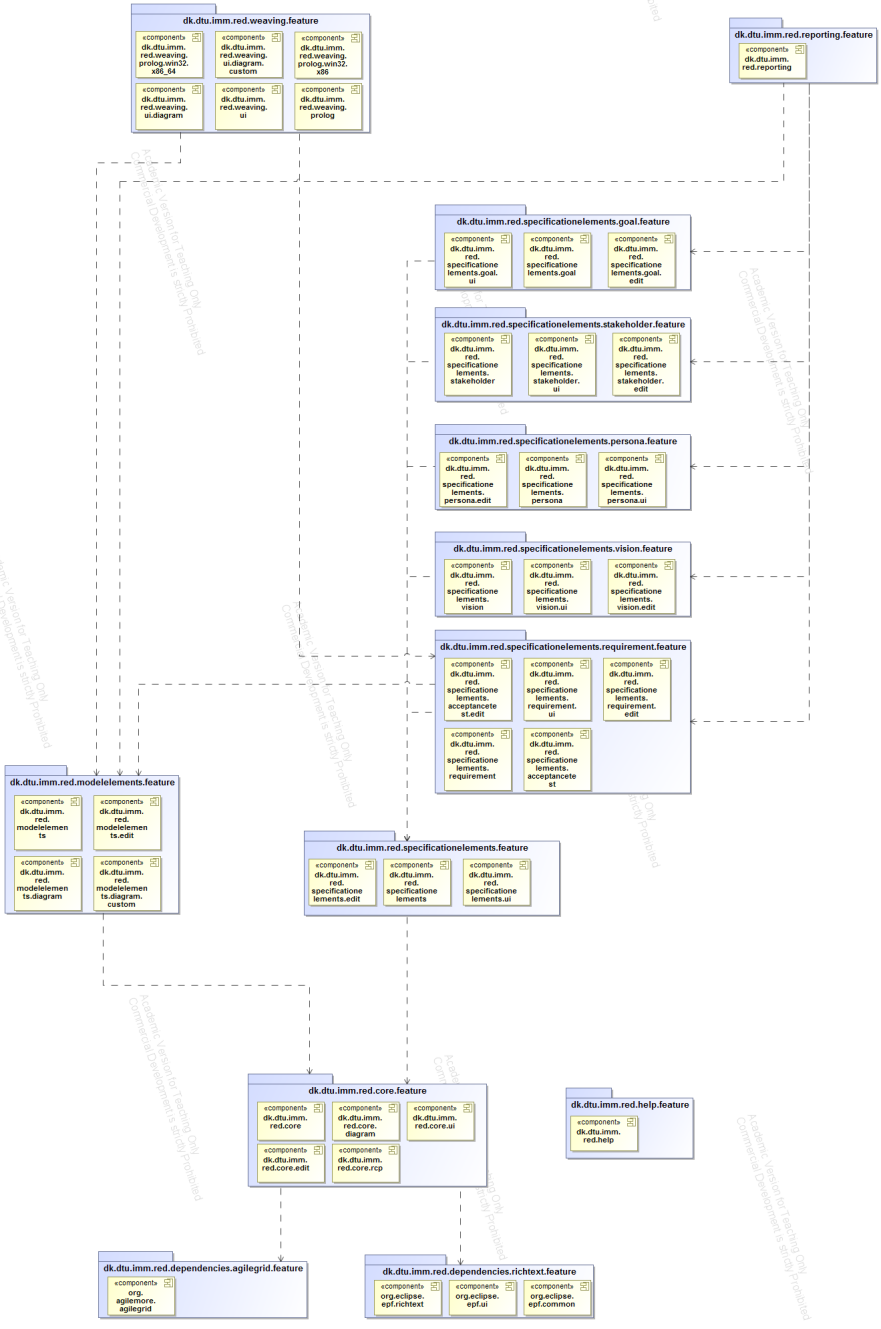


Figure 3.15: RED restructured architecture

fragments have been identified and removed from the code-base. Last, but not least, the SCENARIO module has been deactivated, as it was not working properly in the first place, and it's code was too complex to be split into separate domain and UI *plug-ins*.

It is important to mention that **Anders Friis** in [Fri12] had been wondering whether to organize the source code using functionality layers or by high-level features, and finally decided to take the layer approach. What he actually missed, is that thanks to Eclipse platform it is possible to do both at the same time, therefore getting the benefits of the two approaches.

Chapter 4

Addressing conceptual weaknesses

4.1 Fixing report generation

While Reporting module is a great feature, there is one considerable problem with it. Whenever generating a report, if a RED project contained an *Requirement* that has an *Model Fragment* attached to it, the result was either an exception thrown or the corrupted *Model Fragments* in the exported report. The issue affected all the supported operating systems, and was a major drawback for every student that wanted to export his findings as an HTML report.

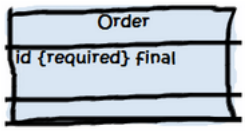
During the examination of the problem, it was found out that it originates from an incorrect initialization of the ModelElements module. Due to performance reasons, the *Model Fragments* attached to the requirements are being initialized on *Requirement* initialization. However, if no *Model Fragment* has been initialized, and a user wanted to generate a report, an `NullPointerException` was being thrown on a piece of code trying to access the fragment. What is even more interesting, if the first *Model Fragment* has been initialized, it was possible to generate the report, but all the *Model Fragments* were swapped with the first initialized fragment. Both these shortcomings made it really annoying to use the Reporting module, as it was either not working, or providing incorrect data (see figure 4.1).

Requirement - null

ID: BLC8, Type: Feature, Level:
Description:

The users that is part of the booking is the guest who created the booking and any employee in the restaurant

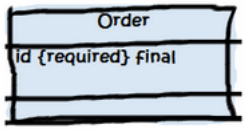
Requirement model:



Requirement - null

ID: BLC9, Type: Feature, Level:

Requirement model:



Requirement - null

ID: BLC10, Type: Feature, Level:
Description: Refer to business rules for sanctions upon not fulfilling a booking

Requirement model:

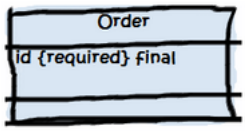


Figure 4.1: An example of a corrupted RED HTML report

As the problem turned out to be non-trivial and various attempts to fix it were not successful, it was decided to temporarily remove the *Model Fragments* from the resulting report. While it fixes the `NullPointerException` problem, the users are still required to manually attach the *Model Fragments* to the report, which is hardly convenient. That is why fixing the *Model Fragments* corruption is one of the most important tasks to be taken outside of the scope of that thesis.

4.2 Fixing model weaving

Fixing model weaving turned out to be a problematic process. First of all, WEAVING module weaves the RED *Model Fragments* into Prolog representation, which requires an external JPL library. JPL is a Java-to-Prolog bridge, implemented using a simple Java wrapper and a C-code compiled into native libraries. This makes it difficult to run from such a thick client as Eclipse Framework, as the required native libraries need to be accessible from the Eclipse run-time. As of now, JPL supports only *MS Windows* and *Linux* machines, but the weaving support has been tested on *MS Windows* only, and is not working on any other platform. What is also disturbing, is the fact that even on *MS Windows* machines, making WEAVING module work is a tedious and time-consuming process. Hence, the initial fixing process will include the following steps:

1. Make WEAVING module working on *MS Windows* computers
2. Disable WEAVING module on any other OS.

When it comes to making it work on *MS Windows* machines, I've been able to run it successfully on my personal machine, but had difficulties in replicating the process. Basically, it turned out that in order for WEAVING module to work, an SWI-Prolog has to be installed, as well as its *bin* folder has to be placed in the PATH environmental variable. While these steps were enough on my local machine, they were not sufficient for making WEAVING work on any other machine. Therefore, finding a proper install or configuration guide for making model weaving operational will be one of the first thing to do in the future.

Thanks to the fact that the modules are now implemented as Eclipse *features*, it is possible to exclude certain *plug-ins* based on the operating system the build is being prepared for. That being said, disabling the module on both *Mac OS X* and *Linux* machines was as easy as setting an option in the `dk.dtu.imm.red.weaving` feature editor, which made it possible to deploy the WEAVING module on the *MS Windows* machine only.

One of the actual improvements made to WEAVING module was fixing an issue when weaving *Requirements* having a "." (dot) in the *Id* field resulted in weaving errors. This was a result of a poor prolog weaving service implementation, which did not properly handled the dot characters. Making necessary fixes to the code successfully resolved that problem.

4.3 Aligning EMF models with domain classes

As many RCP applications, RED is taking advantage of Eclipse EMF framework. In short, EMF lets developers design the desired model using a UML-like visual editor, storing it in an *Ecore* model file and then generate the Java source code out of it. While the generated code exactly matches what was designed in the editor, it is up to the developer to implement the actual methods. EMF also distinguishes between the generated and the custom code, which makes it possible to introduce some changes to the model and then re-generate it without using the custom method implementations.

Originally, RED *Ecore* models have been stored in an external `dk.dtu.imm.red.models` plug-in, from which they could have been accessed when necessary (see figure 4.2). Unfortunately, over time the models have not been updated, yet the generated source code was altered, which resulted in an incompatibility between them. This leads to an issue where if someone tries to re-generate one of the RED modules model code, it resulted in a massive domain corruption. The other problem was that the *Ecore* model was actually separated with its generated code (by being placed in a separate module), which may confuse the developers inexperienced with code generation frameworks.

In order to compensate these problems, the first step to take was to decompose the `dk.dtu.imm.red.models` plug-in and to move the *Ecore* models into the actual plug-ins that contain the generated model source code. That way, in a single plug-in there is a Ecore model declaring the classes and the source code generated from it (which can then be customized). The second one though, was to re-engineer the manual changes made to the generated Java classes and incorporate them into the actual EMF models.

Aligning the models and the source code required the following approach:

1. Examine the generated source code and identify the fields and methods that are not declared in the *Ecore* model
2. Add all the custom fields and methods to the *Ecore* model

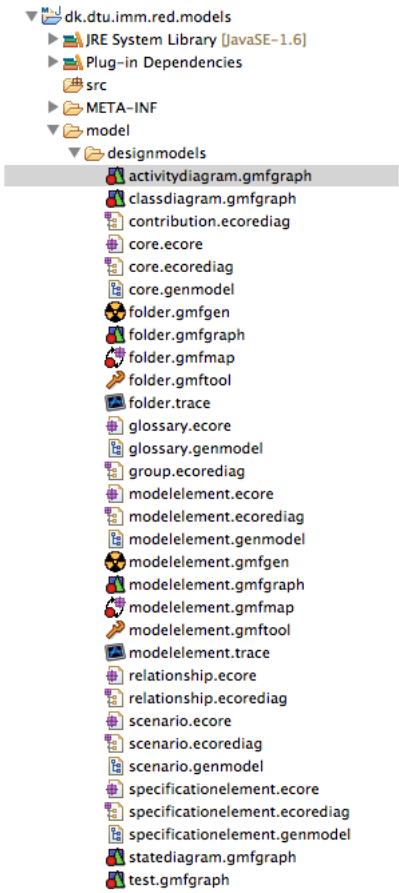


Figure 4.2: Original `dk.dtu.imm.red.models` plug-in contents

3. Annotate the methods implementing custom code with `@generated NOT` annotation
4. Backup the generated source code
5. Re-generate the source code from the adjusted *Ecore* model and `diff` it against the backup copy
6. Make sure there are no differences in class contracts or method logic; if there are any, repeat the process until there are none

While being a tricky and time-consuming process, aligning the EMF models with the generated source code allowed the developers to once again use the EMF framework to extend the model, without the risk of corrupting the RED domain. It may become useful when adding additional features to the existing RED modules, especially that manually modifying the generated source code is not a preferred approach. The *Ecore* models also serves as a good overview of the domain fragment, and having them up-to-date may prevent confusion in the future.

4.3.1 Handling GMF models

As one may notice in figure 4.2, apart from the *Ecore* models, there are also GMF framework models. The GMF models are also used for generating the Java classes, but instead of generating the RED domain, they generate all the code responsible for displaying a visual editors for *Model Fragments* and folder diagrams. As the `dk.dtu.imm.red.models` plug-in was decided to be decomposed, it was important not to loose the GMF models during the process.

The code generated by the GMF models is usually kept in the plug-ins with `*.diagram` suffix, and in RED there are three such plug-ins:

1. `dk.dtu.imm.red.code.diagram`
2. `dk.dtu.imm.red.modelelements.diagram`
3. `dk.dtu.imm.red.weaving.ui.diagram`

Apart from them, some of the GMF models also belong to the SCENARIO module. Each of the respective GMF models has been moved into the proper plug-in. However, due to the generated source code complexity and to the lack of time, the alignment has not been done, and re-generating the GMF models should be handled with the extreme care.

4.4 Excluding SCENARIO support

As a result of both high-level and low-level changes introduced, it was necessary to alter each of the modules implementation so that they correspond to the architecture. However, while it was possible to do in most of the cases, adapting SCENARIO module to the new standards turned out to be too problematic. First of all, the SCENARIO module did not work properly in the original RED version, which was already a problem from the user perspective. Having such a unstable feature in the system may impact its stability, and therefore it should not be shipped with the resulting RED product. Also, as the module does not work properly, and does not provide a way of managing *Scenarios*, it does not provide any benefit to RED users.

What is more, SCENARIO module could not have been separated into different layers as described in section 3.3.2.1. The reason for that is that the domain code is tightly coupled to the presentation layer, which makes it impossible to separate the code into separate domain and UI *plug-ins* without introducing a circular dependency. As we want to avoid introducing such dependencies, it is impossible to make SCENARIO module comply with the new design decisions.

Taking all of the above into account, it has been decided to deactivate the SCENARIO module, but to keep the source code. The module can be activated by including it in the build process.

4.5 Adding horizontal scroll-bar support

While RED provides a great number of editors for its *elements*, they all seem to lack the ability to scroll them horizontally. This is especially a problem for all the RED users working on a low-resolution monitors, as they are not able to edit the contents of some of the fields (see figure 4.3). In order to provide the users a seamless access to the whole content of the editors, it was decided to add the horizontal scrolling support to all the editors. While this is a very simple fix, it will surely be a great usability improvement (see figure 4.4).

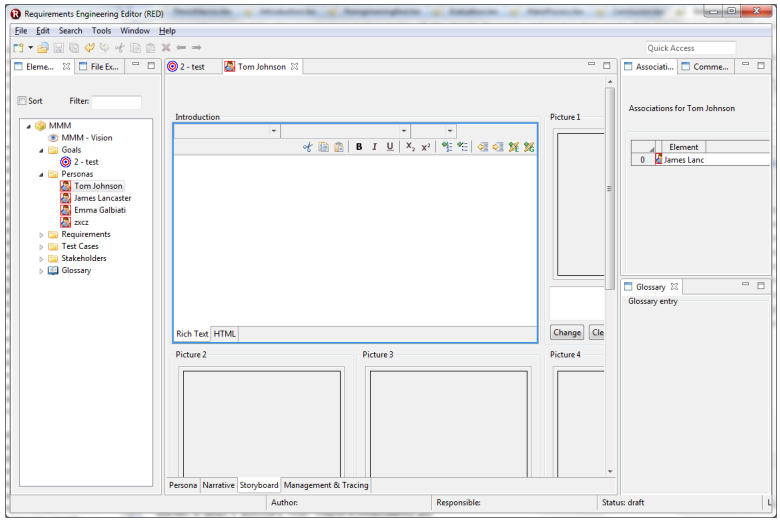


Figure 4.3: An example of a missing horizontal scroll-bar

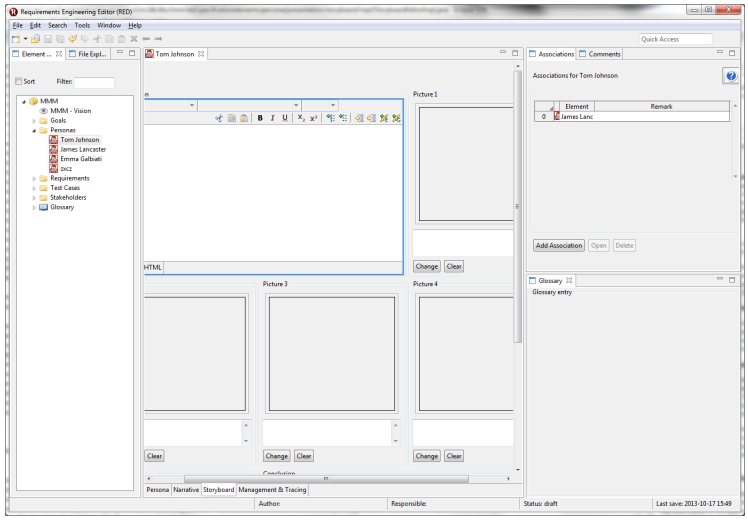


Figure 4.4: Fixed horizontal scroll-bar issue

4.6 Branding

In the last two years RED has become an important tool, providing a lot of features and a look and feel of a modern software, but it certainly lacks a certain amount of branding. While it is not exactly a conceptual weakness, having a bit of personal touch would certainly be beneficial for RED. As of now, RED does not provide any kind of custom icons or splash screen, which makes it look a lot like "yet another" Eclipse customization, rather than a full-fledged piece of software. Adding a bit more of personal touch to RED would certainly make the tool appear more mature, and would certainly make it more pleasant to use.

Thankfully, Eclipse platform developers realized that branding may be important for Eclipse RCP applications, and provided an easy way of doing so. There are several ways for the RCP developers to add custom branding to their products, including:

- Launcher name and icon
- Window name and icon
- Splash screen
- "About" dialog
- Welcome page

Specifying custom launcher name and icon is certainly a good start. By now, RED binary builds used the default launcher settings provided by Eclipse. On *MS Windows*, in order to run RED one would have to navigate to the folder and run *eclipse.exe* file, having a default Eclipse icon that is shown in figure 4.5. While it is not a deal breaker, it must be surprising to some of the users that in order to run a tool that is advertised as being designed and built for the course needs, they should run a program with some other name. Changing the launcher name to a more appropriate *red.exe*, and using a a new, entirely custom icon designed by me (see figure 4.6), makes launching RED a but more intuitive. Also, launcher icons are often used by the operating systems to render the program menu entries.

Similar problem is with the RED window. While the developers have already named it appropriately, the default Eclipse icon could be replaced by a more custom one. Also, the window icons are being used by the operating systems when displaying the currently running programs (e.g. *MS Windows's* task bar), rendering the icons to make the running program appear more convenient to the

Figure 4.5: Default Eclipse Juno icon [Bul12]

Figure 4.6: New RED icon

user. Hence, setting up a proper window icon gives even more usability benefits that the launcher icon.

Customizing the splash screen is another interesting feature. By default, RED greets us with a few-seconds-long splash screen providing simple information about the Eclipse platform used. However, it would be much more informative to provide some details about RED instead. Figure 4.7 shows a new RED splash screen, designed by **Harald Störrle**, showing general information about RED, such as the authors, version number etc. As for the "About" dialog, it provides similar, high-level information the same as the splash screen, but may be accessed "on demand" instead.

The welcome page is an optional feature, and makes it possible to display a custom page whenever opening new Eclipse workspace. As RED does not make use of the workspace concept, there is no need nor reason to specify one.

4.6.1 Mac OS X native app packaging

One interesting feature that is provided by the new build tool, Eclipse Tycho, is the ability to package the Mac OS X version of Eclipse RCPs into a native format. Normally, Eclipse-based applications are delivered with a regular Unix script, that needs to be executed in order to run the program. However, such an approach does not integrate well with the OS, as it prevents the user from

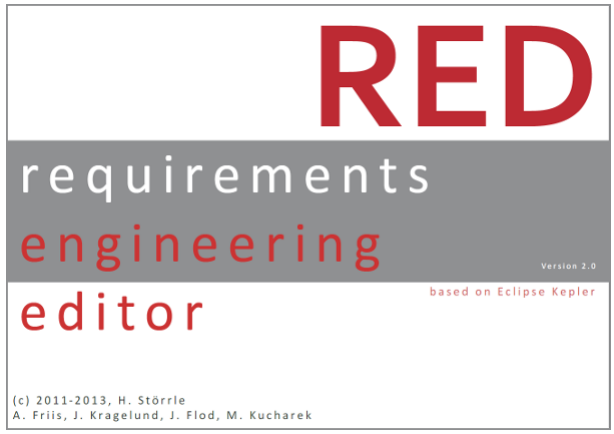


Figure 4.7: New RED splash screen

running the RCP in the same way as other, native, applications (e.g. using the Launchpad or Spotlight). By utilizing the native *.app* Mac OS packaging, the resulting RCP application is bundled into the same format as the regular Mac OS applications, letting the Mac users to place RCP in the dock, select it from Launchpad and run it quickly using Spotlight toolbar.

Setting it up in case of RED is truly straightforward, as it only a matter of configuring the Tycho bundling plug-in. As a result, we obtained a nicely packaged *RED.app*, as shown on figure 4.8.

Figure 4.8: RED Mac OS X native packaging

Chapter 5

Evaluation

5.1 Final architecture

In order to evaluate the architectural changes, let's take a look at the final RED architecture and compare it with the previous state. Figure 5.1 shows a high-level overview of the architecture, that is a result of the re-engineering process.

The first major improvement over the initial architecture is that all of the RED modules have now been implemented as *Eclipse features*. Proper use of the underlying framework principles is the first step towards simplifying the architecture, as it makes it compliant with the general best practices of RCP programming, and therefore makes it easier to understand for anyone familiar with them. As *features* group a number of *plug-ins* into a single entity, they provide an additional abstraction level to operate on. Thanks to them, we can specify and enforce high-level dependencies between the modules.

Another important improvement was dropping the circular dependencies both between high-level modules and individual *plug-ins*. Doing so made the modules more independent, as the modules no longer depend on each other and therefore making changes to one of them does not necessarily affect the other. Also, circular dependencies between modules imply that it is not possible to

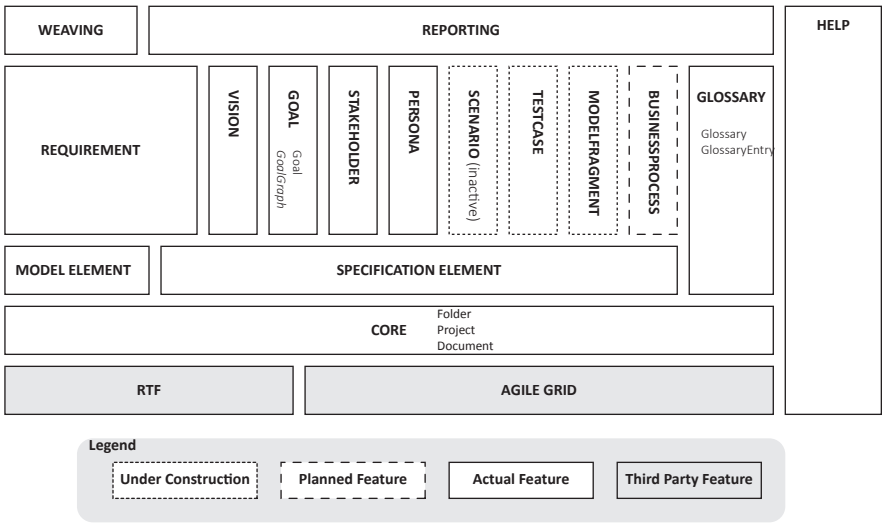


Figure 5.1: RED final architecture overview

deploy only one module of such a dependency in the system, which in our case meant that each SpecificationElements, ModelElements and Weaving modules needed to be deployed either all together or none at all. Splitting that dependency allowed us to break this requirement, and let us remove the Weaving module for all non-Windows-based machines. The same benefits apply at the plug-in-level as well. As each of the RED modules is implemented as a number of plug-ins providing distinct functionality, breaking the circular dependencies between them makes the plug-ins more independent and prevents the situation when making a small change in one plug-in affects the others.

What is more, the new architecture separated the huge SpecificationElements module into a number of small, separate sub-modules. Instead of providing all of the RE concepts together, SpecificationElements now contains only the generic code for all the *Specification Elements* declared in the sub-modules. Introducing such a hierarchy allowed us not only to detach various *Specification Elements* from the final build, but also reduce the dependency on ModelElements module to the Requirement module only. When the ModelFragment module is finally implemented, the ModelElements will be merged into it and the architecture will be even simpler. Moreover, implementing new *Specification Elements* will now be straightforward to do, and will let us control the source code dependencies much more efficiently than it was when having a single SpecificationElements module.

Adding a new **Tool** module type and converting existing Reporting and Weav-

ING modules into it gave the new developers a certain amount of flexibility, while still keeping the architecture structured and extensible. Since no module can depend on a **Tool** module, the developers will be free to implement their features in a way that should not make the architecture too complicated.

Last, but definitely not least, for the first time since RED development started, its architecture has been depicted on a diagram that actually reflects the state of the source code. Until now, the architecture had to be either guessed by the plug-in naming convention, or by the general design diagram shown in [Fri12] that had little to do with what was there in the code. As of now, the architecture has not only been revamped, but has also been presented in form of a less (see figure5.1) and more (see figure 3.15 detailed diagrams, that will serve as a reference for the future developers. This will make it easier for them to introduce new features to the system, as understanding the RED architecture has never been so easy.

5.2 Measurements

After evaluating the high-level architecture overview, let's try to evaluate the low-level fixes. Thanks to the high-level architectural changes, we ensured that there is no circular dependencies between the RED modules, however, it would be vital to evaluate each of the respective modules individually. A good way of doing so would be to measure the code complexity and compare the results before and after introducing the changes. However, due to the architectural changes, like separating certain modules or introducing new plug-ins, it may not always be possible to compare each of the modules directly. However, in general we expect the resulting code quality to be better than before.

There are several metrics that can help in measuring the code quality. First, there are various size metrics, which can inform us how big the code-base is and how it is distributed into classes and packages. Basically, one of the most popular size metric is the number of code lines, often referred to LOC ("Lines Of Code"), or ELOC ("E" standing for "Estimated"). We can also count the number of top-level classes (units), the total number of classes, and an average distribution of fields and methods inside them.

When it comes to measuring the code complexity, there is a number of methods to do so. There is *Cyclomatic Complexity* (CC), developed by **Thomas J. McCabe, Sr.** in 1976, which measures the number of linearly independent execution paths, and hence the potential complexity of the program [Wik13b]. As per [McC76], it is suggested to measure the CC factor during the development for certain modules, and re-factor whenever exceeding the factor of 10. Another

Plugin	Units (Top level classes)	Classes / Class	Methods / Class	Fields / Class	ELOC	ELOC / Unit	Average Cyclomatic Complexity	Fat	Tangled
dk.dtu.imm.red.core	292	0.04	5.43	3.68	17452	59.76712329	1.73	110	28.42%
dk.dtu.imm.red.core.diagram	69	0.15	5.92	1.68	5740	83.1884058	2.18	12	20.67%
dk.dtu.imm.red.core.edit	50	0.02	9.94	1.45	3428	68.56	1.73	14	0.00%
dk.dtu.imm.red.glossary	49	0.03	4.31	2.36	2397	48.91836735	1.37	6	0.85%
dk.dtu.imm.red.modelelements	84	0.03	8.52	12.75	6762	80.5	1.8	6	1.57%
dk.dtu.imm.red.modelelements.diagram	390	0.11	8.51	1.31	196948	504.9948718	6.74	1	0.00%
dk.dtu.imm.red.modelelements.diagram.custom	29	0	2.06	1.94	523	18.03448276	1.22	1	0.00%
dk.dtu.imm.red.modelelements.edit	38	0.03	9	1.05	2735	71.97368421	1.95	1	0.00%
dk.dtu.imm.red.modelelements.weave	13	0	3.48	1.35	985	75.76923077	2.28	3	0.00%
dk.dtu.imm.red.modelelements.weave.diagram	398	0.11	8.97	1.26	276872	695.6582915	8.42	16	0.56%
dk.dtu.imm.red.modelelements.weave.diagram.custom	4	0	3	0.5	74	18.5	1.25	0	0.00%
dk.dtu.imm.red.reporting	19	0	2.65	6.48	1414	74.42105263	2.59	3	0.00%
dk.dtu.imm.red.specificationelements	278	0.1	5.54	2.54	29230	105.1438849	1.78	82	25.49%
org.eclipse.epf.common	38	0.05	9.24	3.78	4394	115.6315789	2.98	16	16.67%
org.eclipse.epf.richtext	67	0.02	4.98	4.93	5212	77.79104478	1.89	9	20.99%
org.eclipse.epf.ui	17	0	8.68	1.86	980	57.64705882	1.14	4	0.00%
				SUM:	555146				

Figure 5.2: Code quality measurements performed on the initial RED source code

metric we could use is *fat*, which tells us how "big" in terms of packages, classes and class members the certain part of the code is. Lastly, we have a *tangled* factor, which helps us follow the Acyclic Dependencies Principle [Wik13a] and provides us with a factor informing us whether or not our source code contains a circular dependencies between classes and/or packages.

All the measurements have been done using STAN4J tool, available under `http://stan4j.com`.

5.2.1 Original RED source code

Figure 5.2 contains the measurement summary that has been done on the initial version of RED that has been provided by **Johan Flod**. Using these data, it is possible to notice which plug-ins contains the most number of classes, as well as which of them suffer from circular dependencies (tangling). The chart shown in figure 5.3 shows the plug-ins ordered by their size, as well as showing how much they are tangled.

As we can see, both `dk.dtu.imm.red.modelelements.weave.diagram` and `dk.dtu.imm.red.modelelements.diagram` greatly oversize the rest of the plug-ins. However, as these mostly contain the auto-generated code, and therefore is virtually no tangling present, we may exclude them to get the better picture of the more interesting plug-ins. Figure 5.4 shows the same chart as figure 5.3, with the two outstanding plug-ins excluded. From there, one can clearly notice that the two most problematic plug-ins are `dk.dtu.imm.red.specificationelements` and `dk.dtu.imm.red.core`, which are not only the largest ones in terms of esti-

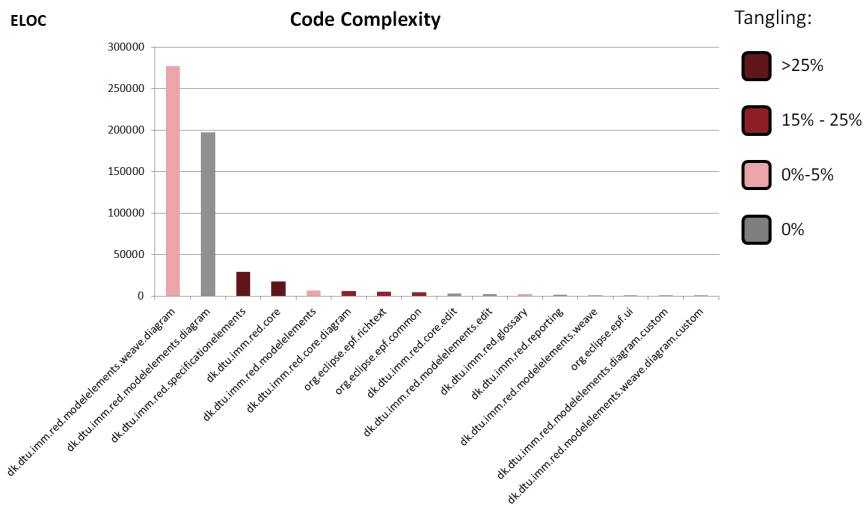


Figure 5.3: Original RED plug-ins size and complexity

mated lines of code (ELOC), but are also the most heavily tangled. Also, their fat factor is disturbing, as having 110 and 82 average method size, respectively, suggests that they are filled with sub-packages and classes, that probably can be simplified. Improving the quality of the code inside these two plug-ins is definitely the priority.

The size of the rest of the plug-ins is comparable, and oscillates around ∼5000 ELOC, which is a reasonable quantity. However, they do differ in terms of tangling. The most tangled plug-ins, aside from the previously mentioned ones, are:

1. `dk.dtu.imm.red.code.diagram`
2. `org.eclipse.epf.richtext`
3. `org.eclipse.epf.common`

The first plug-in contains the auto-generated code for the folder diagram feature of RED. However, judging by its tangling factor, it had to be manually modified, and the custom code introduces a lot of the circular dependencies. Both second and third plug-ins contain the EPF implementation of the RichText editor, which either means that the initial EPF RichText implementation has suffered from tangling problem, or it was due to the **Anders Friis** customizations that the tangling occurred. The deeper examination of all the EPF RichText plug-ins

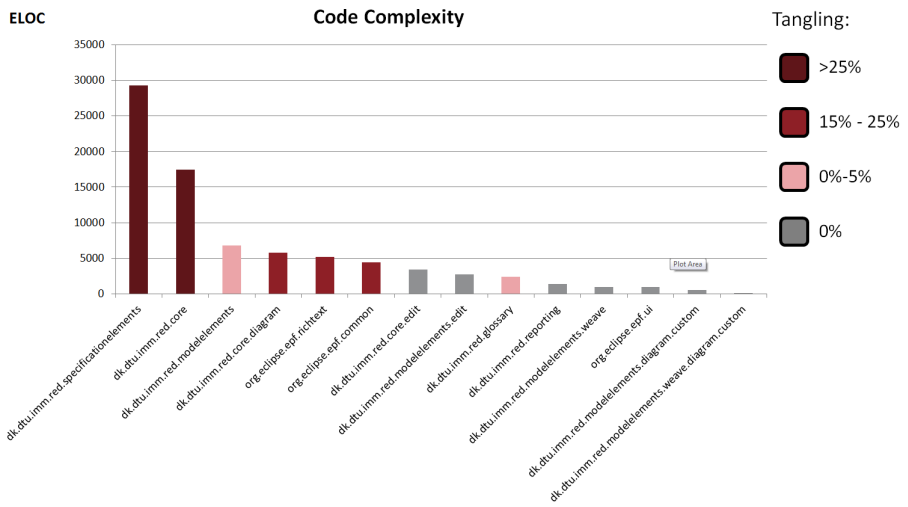


Figure 5.4: Original RED plug-ins size and complexity, excluding the two outstanding plug-ins

revealed that both `org.eclipse.epf.richtext` and `org.eclipse.epf.common` have been manually modified to fit the RED needs. However, `org.eclipse.epf.common` contains only a minor change, which could not have caused such an amount of circular dependencies on its own.

There are two plug-ins that are tangled up to 5%, being `dk.dtu.imm.red.modelelements` and `dk.dtu.imm.red.glossary`. Considering the fact that this is close to negligible, and the fact that there are other plug-ins that suffer much more from the tangling problem, these plug-ins are not considered a priority right now.

When it comes to CC metric, for every of the original *plug-ins*, the average CC was always below 10, which is already fine. The only *plug-ins* that were relatively close to that border are the ones containing mostly the auto-generated code, which we cannot do much about.

5.2.2 Final RED source code

Figure 5.5 shows the same set of measurements done on the final version of the code, done at the Eclipse feature level. Figures 5.6 and 5.7 shows the code complexity charts based on the updated data.

Feature	Units (Top level classes)	Classes / Class	Methods / Class	Fields / Class	ELOC	ELOC / Unit	Average Cyclomatic Complexity	Fat	Tangled
dk.dtu.imm.red.core.feature	414	0.07	5.93	3.09	26590	64.23	1.82	117	18.46%
dk.dtu.imm.red.dependencies.agilegrid.feature	59	0.08	8.24	2.81	8849	149.98	2.33	4	5.99%
dk.dtu.imm.red.dependencies.richtexteditor.feature	122	0.02	6.41	4.29	10586	86.77	2.12	3	0.00%
dk.dtu.imm.red.glossary.feature	47	0.03	4.31	2.44	2221	47.26	1.39	4	1.23%
dk.dtu.imm.red.modelelements.feature	531	0.1	8.23	2.45	205121	386.29	6.21	10	0.96%
dk.dtu.imm.red.reporting.feature	19	0	2.65	6.48	1417	74.58	2.61	3	0.00%
dk.dtu.imm.red.specificationelements.feature	12	0.13	4.53	2.93	407	33.92	1.46	5	6.90%
dk.dtu.imm.red.specificationelements.goal.feature	25	0.12	6.25	3.84	1442	57.68	1.48	6	2.20%
dk.dtu.imm.red.specificationelements.persona.feature	43	0.08	6.46	3.08	2989	69.51	1.65	13	5.44%
dk.dtu.imm.red.specificationelements.requirement.feature	47	0.11	6.17	3.83	3593	76.45	1.72	10	1.79%
dk.dtu.imm.red.specificationelements.stakeholder.feature	25	0.11	6.64	3.64	1840	73.60	1.57	6	2.15%
dk.dtu.imm.red.specificationelements.testcase.feature	26	0.11	7.97	4.47	2164	83.23	1.56	6	1.43%
dk.dtu.imm.red.specificationelements.vision.feature	24	0.13	4.52	2.39	954	39.75	1.48	6	3.85%
dk.dtu.imm.red.weaving.feature	430	0.11	8.8	1.35	280056	651.29	8.2	3	8.50%
				SUM:	548229				

Figure 5.5: Code quality measurements performed on the final RED features

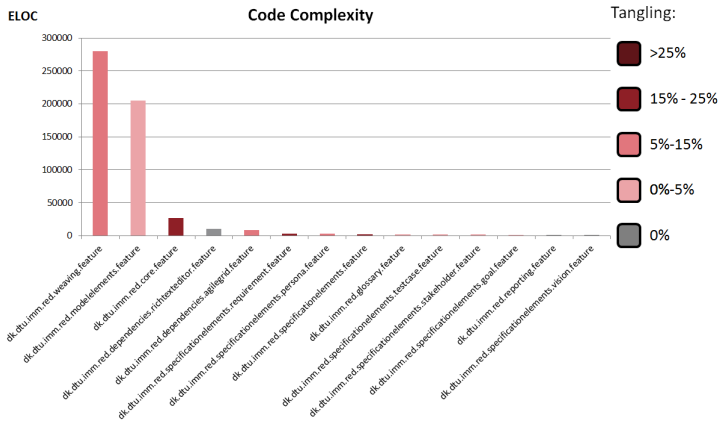


Figure 5.6: Final RED plug-ins size and complexity

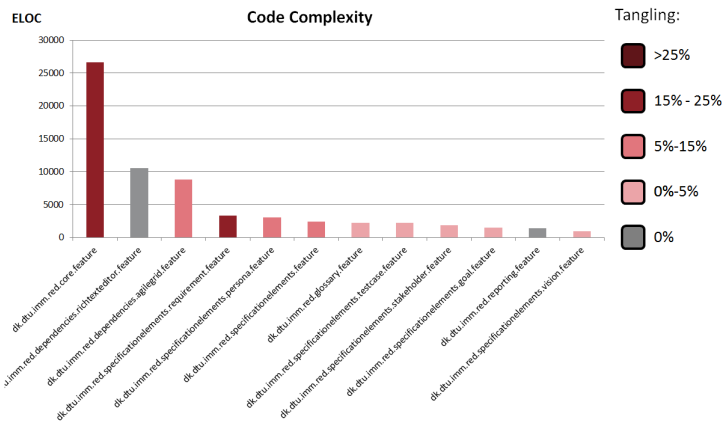


Figure 5.7: Final RED plug-ins size and complexity, excluding the two outstanding plug-ins

As one may clearly notice, thanks to the restructuring process we have been able to reduce the tangling factor for almost all of the RED modules. The greatest improvement has been made to the SpecificationElements module, which, after a successful separation into smaller sub-modules, does not suffer from the great tangling factor anymore. While the Core module's tangling factor is still quite high, it has been reduced below 20%, which is still an improvement. Also, it is worth mentioning that the tangling of SpecificationElements module and each of its sub-modules (Persona, Goal, Vision, Requirement, Stakeholder and TestCase) has been introduced by the EMF generated source code, which prevents us from reducing it any further.

One may also notice that after the restructuring, the code-base contains about ~5000 lines of code less. What is more, this also contains all the code introduced into RED as a part of the thesis, including AgileGrid sources and the new *Test Case* element, discussed in the following section. Having them excluded, the difference between the original and final code-base is around ~15000 ELOCs.

Figure 5.8 shows the same metrics performed at the resulting *plug-ins* level. Thanks to that, one may notice how the code has been distributed into various *plug-ins*. Also, it is important to see that the fat factor has been greatly reduced in all the SpecificationElements module and sub-modules *plug-ins*. The `dk.dtu.imm.red.core` *plug-in* still suffers from quite high fat factor, but as it contains mostly auto-generated code, there is not much we can do about it except modifying the domain model itself.

Plugin	Contains auto-generated code?	Units (Top level classes)	Classes / Class	Methods / Class	Fields / Class	ELOC	ELOC / Unit	Average Cyclomatic Complexity	Fat	Tangled
dk.dtu.imm.red.core	Y	157	0.08	7.56	5.83	10555	67.23	1.74	82	39.79%
dk.dtu.imm.red.core.diagram	Y	69	0.15	5.92	1.68	5740	83.19	2.18	12	20.67%
dk.dtu.imm.red.core.edit	Y	50	0.02	9.94	1.45	3428	68.56	1.73	14	0.00%
dk.dtu.imm.red.core.rcp	N	4	0	4.6	3.8	219	54.75	1.61	3	0.00%
dk.dtu.imm.red.core.ui	N	133	0.02	3.6	1.84	6628	49.83	1.71	2	11.54%
dk.dtu.imm.red.glossary	Y	12	0.07	7.86	4.43	659	54.92	1.48	3	3.17%
dk.dtu.imm.red.glossary.ui	N	35	0.02	3.43	1.95	1562	44.63	1.34	10	5.56%
dk.dtu.imm.red.modelelements	Y	77	0.02	8.01	13.23	5070	65.84	1.56	6	2.15%
dk.dtu.imm.red.modelelements.diagram	Y	390	0.11	8.51	1.31	196948	504.99	6.74	1	0.00%
dk.dtu.imm.red.modelelements.diagram.custom	N	29	0	2.06	1.94	523	18.03	1.22	1	0.00%
dk.dtu.imm.red.modelelements.edit	Y	38	0.03	9	1.05	2735	71.97	1.95	1	0.00%
dk.dtu.imm.red.reporting	N	19	0	2.65	6.48	1414	74.42	2.59	3	0.00%
dk.dtu.imm.red.specificationelements	Y	8	0.1	3.9	3.9	222	27.75	1.33	3	8.70%
dk.dtu.imm.red.specificationelements.edit	Y	3	0.25	5.75	1.25	148	49.33	1.57	1	0.00%
dk.dtu.imm.red.specificationelements.goal	Y	9	0.09	9.36	6.55	690	76.67	1.55	3	5.00%
dk.dtu.imm.red.specificationelements.goal.edit	Y	3	0.5	5.83	1.33	236	78.67	1.66	3	0.00%
dk.dtu.imm.red.specificationelements.goal.ui	N	13	0	4.13	2.87	516	39.69	1.29	4	6.45%
dk.dtu.imm.red.specificationelements.persona	Y	18	0.09	9	4.86	1443	80.17	1.84	7	14.29%
dk.dtu.imm.red.specificationelements.persona.edit	Y	6	0.33	7.67	1.44	484	80.67	1.77	1	0.00%
dk.dtu.imm.red.specificationelements.persona.ui	N	19	0	4.23	2.27	1062	55.89	1.31	5	4.26%
dk.dtu.imm.red.specificationelements.requirement	Y	9	0.09	12.45	8.09	1069	118.78	1.96	3	6.90%
dk.dtu.imm.red.specificationelements.requirement.edit	Y	3	0.5	7	1.33	302	100.67	1.62	3	0.00%
dk.dtu.imm.red.specificationelements.requirement.ui	N	24	0	3.97	2.95	1540	64.17	1.68	1	0.00%
dk.dtu.imm.red.specificationelements.stakeholder	Y	9	0.09	11.27	7.27	898	99.78	1.76	3	5.13%
dk.dtu.imm.red.specificationelements.stakeholder.edit	Y	3	0.5	6.33	1.33	268	89.33	1.66	3	0.00%
dk.dtu.imm.red.specificationelements.stakeholder.ui	N	13	0	4.05	2.26	674	51.85	1.22	4	6.25%
dk.dtu.imm.red.specificationelements.testcase	Y	11	0.08	11.62	7.46	997	90.64	1.56	3	4.17%
dk.dtu.imm.red.specificationelements.testcase.edit	Y	4	0.43	7	1.29	346	86.50	1.65	5	0.00%
dk.dtu.imm.red.specificationelements.testcase.ui	N	11	0	5.44	3.44	821	74.64	1.53	4	0.00%
dk.dtu.imm.red.specificationelements.ui	N	1	0	6	0	37	37.00	1.83	0	0.00%
dk.dtu.imm.red.specificationelements.vision	Y	8	0.1	5.9	4.2	366	45.75	1.54	3	8.00%
dk.dtu.imm.red.specificationelements.vision.edit	Y	3	0.5	5.17	1.33	193	64.33	1.58	3	0.00%
dk.dtu.imm.red.specificationelements.vision.ui	N	13	0	3.33	1.6	395	30.38	1.34	4	7.41%
dk.dtu.imm.red.weaving.prolog	N	15	0.11	9.33	5.17	2125	141.67	2.52	3	5.13%
dk.dtu.imm.red.weaving.ui	N	13	0	3.48	1.35	985	75.77	2.28	3	0.00%
dk.dtu.imm.red.weaving.ui.diagram	Y	398	0.11	8.97	1.26	276872	695.66	8.42	16	0.56%
dk.dtu.imm.red.weaving.ui.diagram.custom	N	4	0	3	0.5	74	18.50	1.25	0	0.00%
org.agilegrid.agilemore	N	59	0.08	8.24	2.81	8849	149.98	2.33	4	5.99%
org.eclipse.epf.common	N	38	0.05	9.24	3.78	4394	115.63	2.98	16	16.67%
org.eclipse.epf.richtext	N	67	0.02	4.98	4.93	5212	77.79	1.89	9	20.99%
org.eclipse.epf.ui	N	17	0	8.68	1.86	980	57.65	1.14	4	0.00%

Figure 5.8: Code quality measurements performed on the final RED plug-ins

5.3 Case Study: A "Test Case" Specification Element

In order to provide another proof that the new architecture features more flexibility, a simple case study has been developed, trying to extend current RED domain with a new RE concept - a *Test Case*. As of now, RED supports only simple cases of adding tests, by providing an ability to add an *Acceptance Test* to an existing *Requirement* element. While this is an important feature, the solution is not flexible enough as RED users may want to:

1. add other type of a *Test Case*
2. add a *Test Case* to an other *Specification Element* than *Requirement*

To make such use cases possible, it was decided to extend the add a new TestCase module, which would provide an entirely separate *Specification Element*. That way, it will be possible to associate every *Test Case* with any other *Specification Element* using the RED association mechanism. It would not only allow to specify a *Test Case* type (acceptance, functional etc.), but also to link such a test to multiple other elements.

5.3.1 Editor design & implementation

Figure 5.9 shows an initial sketch of the *Test Case* editor. In order to describe a complete test case, one should provide the following:

1. ID - a unique identifier
2. Test case name
3. Test type - chosen from a predefined list
4. A general test description
5. A set of pre-conditions
6. Input parameters
7. A list of step-by-step actions required to take
8. A set of post-conditions

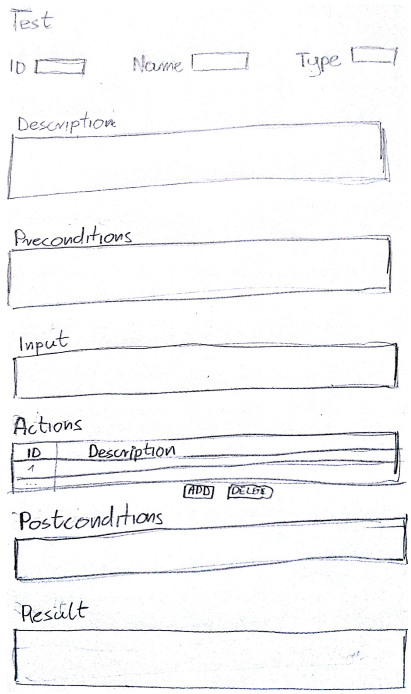


Figure 5.9: An initial sketch of *Test Case* editor

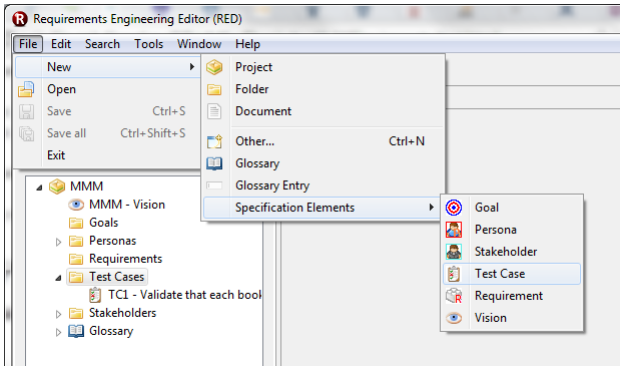


Figure 5.10: One of the possibilities of creating a new *Test Case*

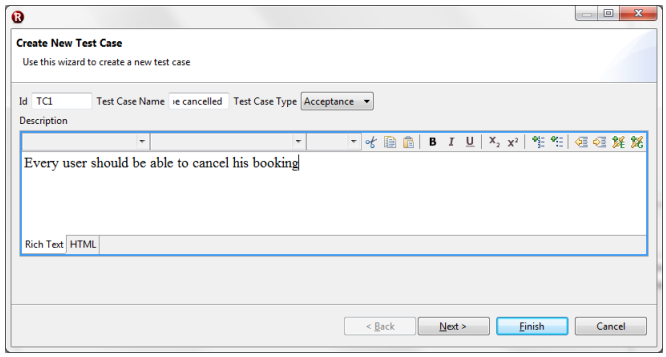


Figure 5.11: *Test Case* wizard dialog

9. A description of the desired result

Such a design allows to fill all the test data using a simple, yet easy to use layout. Figures 5.10, 5.11 and 5.12 show the user interface implemented so that it is possible to efficiently create and edit the *Test Cases*.

5.3.2 Module implementation

Although TestCase module provides an important functionality, one of its main purposes was to show the extensibility of the new architecture. As per figure 5.1, TestCase has been implemented as a separate RED module on top of existing SpecificationElements component. That way, it follows the existing architectural pattern for all the *Specification Elements*, which ensures that each

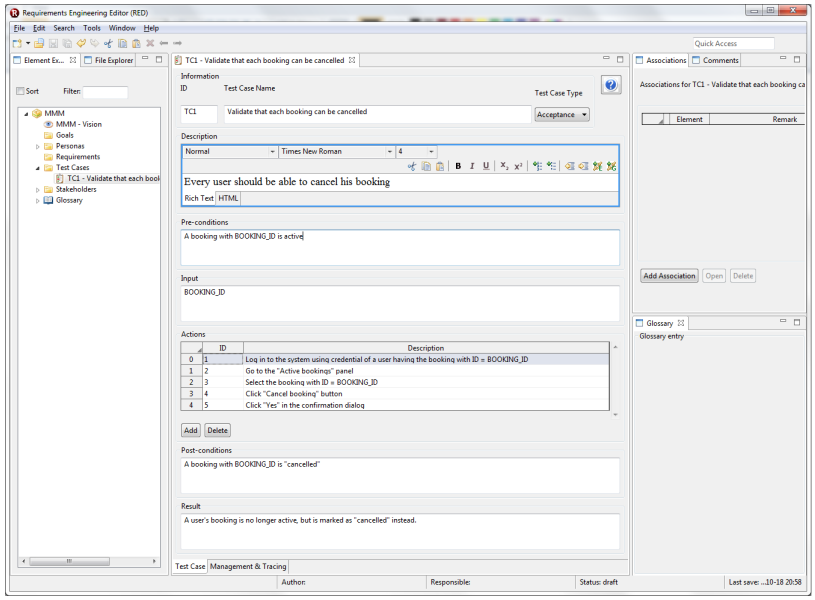


Figure 5.12: *Test Case* visual editor

Specification Element is independent and therefore can be easily detached from RED if necessary.

Figure 5.13 shows the EMF domain model diagram that has been used to generate the model classes. Figure 5.14 describes how the TestCase module have been implemented in the whole platform. Basically, an additional feature has been added, containing three plug-ins, each having different responsibility:

- `dk.dtu.imm.red.specificationelements.testcase` - model declaration (auto-generated)
- `dk.dtu.imm.red.specificationelements.testcase.edit` - EMF editing domain (auto-generated)
- `dk.dtu.imm.red.specificationelements.testcase.ui` - user interface

Such a plug-in layout is also following the general pattern for the existing SpecificationElements-based modules, which allows a straightforward, unidirectionally coupled low-level architecture.

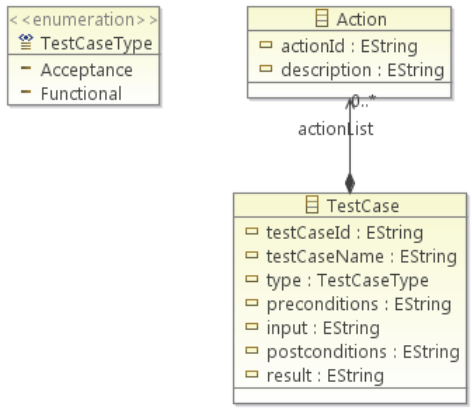


Figure 5.13: TestCase domain model diagram

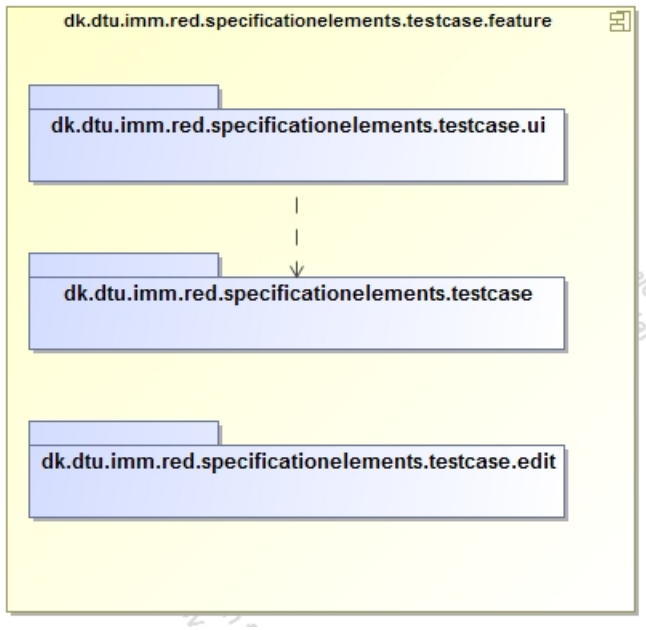


Figure 5.14: TestCase module architecture diagram

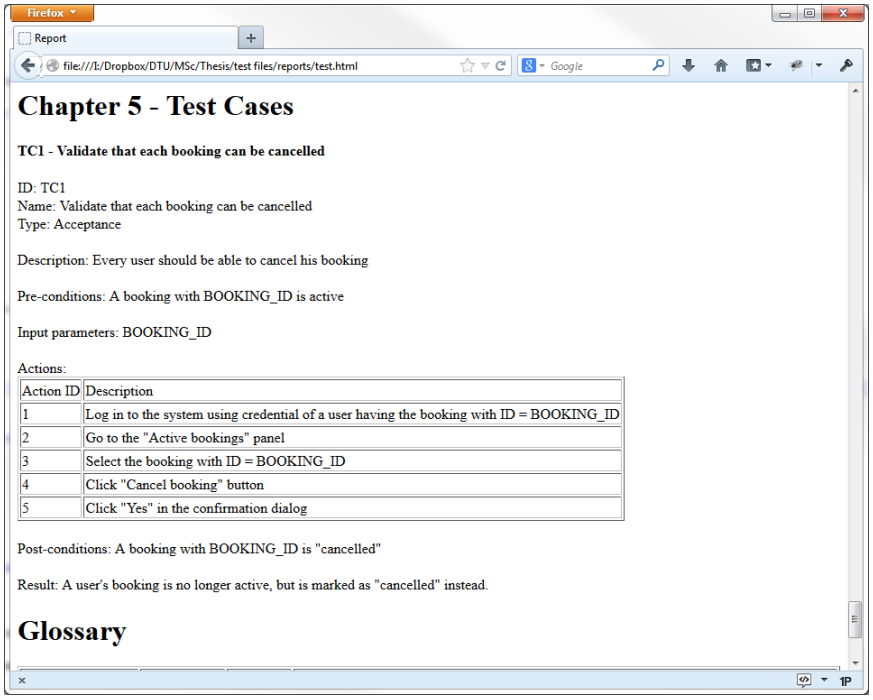


Figure 5.15: A RED report fragment containing *Test Case* info

5.3.3 Reporting integration

As of now, reporting functionality is implemented in a centralized REPORTING module, which requires an explicit dependency for every module it needs to access the data from. Therefore, in order to make the REPORTING module include the newly introduced *Test Cases*, we need to declare a dependency between REPORTING and TESTCASE modules, and implement the required logic for displaying the *Test Case* information in form of a report. However, introducing such a dependency will make it impossible to remove the TESTCASE module without any impact on RED, so it may be wise to test the newly introduced module first, and declare dependencies on it only once proven stable.

However, a simple proof of concept has been developed showing a simple *Test Case* representation in the generated RED report (see figure 5.15).

Chapter 6

Conclusion

6.1 Summary

In order to summarize what has been done, let's take a look at the initial goal diagram and discuss what has been accomplished (see figure 6.1).

6.1.1 Build process

Starting from the build process improvements, we have set up the code repository available under `https://bitbucket.org/mkucharek/red2`. It is hosted on Atlassian BitBucket service and aside of code versioning, it also provides a Atlassian JIRA-based issue tracker, a simple wiki pages for storing the project's documentation and a download section for distributing the binaries. The project is currently a private one, but shortly after the thesis submission it will be made accessible to everyone. Doing so will make it possible for the 02264 course students to contribute to the project either by submitting bug reports and feature requests using the issue tracker, or by actually joining the development team. By preparing a proper documentation in the wiki section, describing how to start the development and explaining some of the concepts and best practices, we want to make sure that it will be easier for future developers to understand

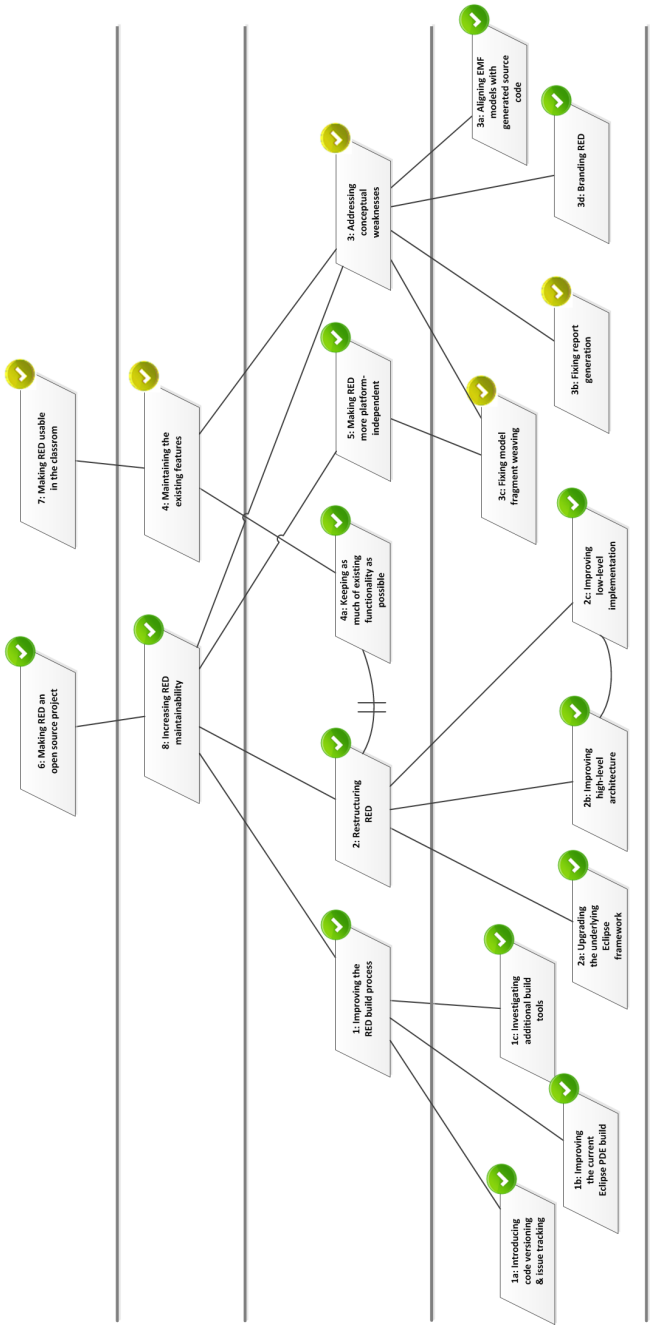


Figure 6.1: Thesis goal diagram summary

the RED development process. Lastly, having the download section will make it easy to provide the latest versions of RED to the public.

The next step was examining the default Eclipse PDE process. Originally, the actual build process was tightly coupled with the development environment (IDE) in use. In order to build RED, one needed to download Eclipse 3.7 SDK and to install a number of required plug-ins, including third-party ones, like AgileGrid, that required a manual download. Such a process was both complicated and difficult to replicate every time a new development environment needs to be set up, so it had to be revamped. First of all, the external dependencies had to be resolved. As RED is dependent on AgileGrid, a third-party library that is no longer supported, it was necessary to drop the need of downloading the AgileGrid binaries whenever setting up the development environment, as the binary may simply become unavailable at some point. A various approaches have been discussed, and it was finally decided to include the AgileGrid sources in RED source code. That way, the required plug-in was being compiled during every RED build, and we still maintained a possibility of making changes to AgileGrid in the future.

Another important build process improvement was defining an external target platform. A target platform is a set of Eclipse plug-ins, that the Eclipse RCPs (such as RED) are being built on top of. Without an external target platform declaration, the plug-ins contained in the SDK in use are being used to fulfill the RCP dependencies. This is exactly why it was necessary to fill the Eclipse SDK with the plug-ins required by RED in order to be able to build RED. However, it is possible to define a set plug-in repositories, from which the required *plug-ins* will be downloaded during every build. By specifying the required plug-ins in the target platform definition, we not only made the SDK totally independent on the build process, but also decreased the size of the final binary by around 50%, as RED does no longer contain all the plug-ins from the used SDK.

The final improvement was extending the regular PDE build process with Eclipse Tycho tool. Tycho is an Apache Maven extension that allows building Eclipse RCP projects using this popular Java build solution. Tycho provides an easy way of building RED using a command-line only, with no Eclipse SDK required, and is fairly flexible, letting developers to:

- Run unit-tests with every build
- Toggling between building RED for all the supported platforms (*Mac OS X*, *MS Windows* and *Linux*), or the current one only
- Allowing to package the resulting products in *zip* format

Providing a way of command-line building is a great advantage for **Harald Störrle**, as he will not require to set up the whole development environment to prepare a binary for the students, but will also make it possible for anyone to try the most recent version of RED without the need for it to be released. Using Tycho/Maven will also make it easy to use a continuous integration solutions, such as Jenkins or Hudson, which may be beneficial in the future.

6.1.2 Restructuring RED

After finishing the build process improvements, a decision has been made to upgrade the underlying Eclipse framework to the latest possible version. The clear benefit of that is the access to the latest API, as well as taking the advantage of all the features coming from the Eclipse developers. Last, but not least, with the introduction of Eclipse 4.4 Luna, it should be possible to run the *plug-ins* developer against the new Eclipse 4.X platform along the ones developed using the previous, 3.X, which should make extending RED a much easier process.

As a next step, we've started by looking at the high-level architecture, and comparing the initial RED architecture design with what has been found in the actual source code. As it quickly turned out, RED suffered from major architecture weaknesses, which resulted from poor utilizing of Eclipse RCP programming concepts and general negligence when introducing excessive dependencies between various modules. First of all, the modules has been initially implemented as either a single, or a group of plug-ins, which made it difficult for handling the relations between them. As plug-ins are the most low-level units in Eclipse RCP programming, implementing the high-level modules on that level made it difficult to understand the actual concept, and introduced the unnecessary confusion. In order to provide a new abstraction level, Eclipse features have been used to group all the plug-ins that implement a certain RED module and allow to treat them as a single unit. Such an implementation change made it possible to formulate the high-level dependencies between the modules, which made the underlying plug-ins much more subject to change without violating the high-level contracts. Also, by using features, we made it much easier to control the resulting RED product, as we may now freely include or exclude the desired modules without having to think which plug-ins it is composed of, as well as allowed to utilize additional Eclipse-framework functionality, such as *Update Manager*, which may be used in the future for distributing RED updates.

We've also taken a look at the module dependencies that clearly violated the initial RED design. The major problem when it comes to dependencies was the circular dependency between ModelElements and SpecificationElements modules, as it prevented us from understanding the contract between these

modules, as well as made it impossible to exclude one of them without having to exclude the other. Having such a tight coupling between the high-level modules also resulted in the problem when making change to one of the modules may result in the errors in the other one. As it turned out, the reason for the circular dependency was the WEAVING module being initially included in the MODELELEMENTS module, which had to be extracted and carefully analyzed in order to drop the unwanted dependencies.

One of the next high-level improvements was dividing the oversized SPECIFICATIONELEMENTS module. As SPECIFICATIONELEMENTS provides all the *Requirements Engineering* concept capabilities to RED, it was one of the most important modules of RED from the user perspective. However, as all the concepts were placed in a single module, we had to either support all of them, or none at all, as it was not possible to exclude a single concept from RED. It turned out to be a problem when trying to deactivate the *Scenarios*, which were not working properly, as it required to manually remove (or commenting out) the source code to achieve. Also, maintaining such a complex plug-in was more than difficult, as such plug-ins tend to serve as a way of hiding the code complexity, as there is almost no way of enforcing a clear dependency structure. Therefore, it was decided that SPECIFICATIONELEMENTS module should be divided into smaller ones, that each provide the functionality of a single *Requirements Engineering* concept. Having that accomplished, we made clear distinction between each of the concepts, making all of them independent of each other, as well as made the source code look much more clear and easier to understand.

The next step was to investigate the excessive dependencies, mostly coming from WEAVING and REPORTING modules, that tended to depend on a number of other modules. As they both provided important features to RED, and they both operated in a centralized behavior that required access to the data coming from other modules, it was not straightforward to drop the dependencies. Therefore, it was decided to introduce a new module type, called **Tool**, which is basically a module that can depend on other RED modules, but cannot be dependent on. Having the RED architecture extended by such a concept allowed developers to officially include various extensions to RED, providing an access to most of the RED modules data, while keeping the architecture safe from both circular and complex dependencies.

Having taken care of the high-level architecture, it was time to introduce the fixes at the low-level scope. Each module implementation varied a bit, as some of the modules have been implemented as single plug-ins, and others have been divided into a number of them. Since the individual plug-ins are supposed to be responsible for one thing only, but to do it well, splitting any outstanding plug-ins into a number of smaller ones was the first task to do. It was mostly

important to separate the source code written by hand from the auto-generated one, that has been created by utilizing various frameworks, such as EMF or GMF, as having them in the same plug-ins made it really difficult to maintain and understand the code. Afterwards, as with the high-level scope, it was necessary to drop all the circular dependencies between different plug-ins. Lastly, the actual code implementation had to be improved, mostly by removing the unnecessary parts of code and by implementing certain feature in a much more reusable, and therefore efficient, way.

6.1.3 Additional fixes & improvements

The very last part was addressing the conceptual weaknesses of RED, as well as extending it with the new features. First of all, all the EMF models have been aligned with their generated source code, which made it possible to use the EMF visual editors to re-generate the domain classes. It was extremely important, as the generated source code is difficult to edit manually, hence introducing any model changes in the EMF visual editor and then re-generating the source code is a much more efficient way of introducing the changes, as it limits the possibility of breaking the RED domain layer.

After that, a number of problems with both REPORTING and WEAVING modules have been identified, and attempted to be fixed. As for REPORTING, there was a problem with the exported *Model Fragments*, as they were incorrectly placed in the resulting report, or even resulted in a number of exceptions being thrown and crashing the whole application. Although the complete fix could not have been found, the *Model Fragments* have been temporarily removed from the resulting report, and will be enabled back once the *Model Fragment* rendering problem will be resolved.

The problem with WEAVING module, on the other hand, originates from the fact that it in order to successfully weave a model, the WEAVING module requires a JPL library, which is currently available only on Windows machines. Since no substitution was found for other platforms, it was decided to remove the weaving functionality from both Mac OS and Linux builds, as it is not able to function properly there. However, even on Windows-based machines, WEAVING is not working seamlessly as it is not possible to include the whole JPL library in a RED plug-in. Also, a number of problems have been identified in the code responsible for converting RED models to Prolog, which have been fixed.

The next improvement was implementing the horizontal scroll-bar support for all supported RED editors. Before doing that, some of the editors had proven themselves to be difficult to use on the low resolution screens, as they were

simply too wide to fit in the RED workbench. Adding the scroll-bars made it much easier for user using such screens to use RED efficiently.

In order to maintain RED stability, it was also necessary to disable the SCENARIO module. Even in the original source code, *Scenario* support was not working as expected, and therefore making it impossible to manage them. Another issue was that due to a poor implementation, SCENARIO module could not have been aligned with the new architectural decisions, which implies a need of totally rewriting it. Re-implementing the module would not only make it possible to resolve the design problems, but would also allow reconsidering the high-level *feature* it needs to provide.

One of the last major improvements was making branding RED. Until now, RED has been a casual Eclipse RCP application, having the default naming, icons and splash screen. In order to make RED look more professional and recognizable, it has been decided to design and apply a custom RED icon, as well as to include a custom splash screen and "About" page, so that RED feels more like a proper application rather than a simple Eclipse extension. Also, for the Mac users, thanks to Eclipse Tycho a Mac version of RED is being packaged as a native Mac application, so that RED may be installed and handled as any other Mac application on the Apple's computers.

6.2 Discussion

The main reason for focusing that thesis on re-engineering RED was that I've initially tried to extend it with a new feature and failed miserably. In February 2013, I've started a thesis aimed for providing merging functionality into RED, that would allow to coordinate the student's work within the project groups in 02264 course. However, by the time I was able to actually start working on the actual thesis subject, I had to struggle with a number of problems that were outside of its scope. First of all, it was very difficult to get the stable version of the source code, as the RED source files shipped by **Jakob Kragelund** were corrupted. At the same time, the source code obtained from **Johan Flod** turned out to contain a number of issues that were not there in the original RED version, and as there was no development history preserved, it was not possible to track down where do these originate from. Another problem was the complexity of development environment setup process, as it took me a considerable amount of time to properly set it up, and to finally compile the source code. Having to struggle with these issues made me realize that each and every future RED developer, most probably a MSc. thesis student, would have to go through these steps, which is basically a waste of time compared to the actual thesis

subjects. Hence, it became clear that a solution preventing from such situations happening in the future is required and should be found as soon as possible.

Another problem I've quickly encountered was the RED source code complexity, which made it truly difficult to understand which parts of code are responsible for which functionality. Initially, RED has been organized in a number of over-grown, mutually dependent plug-ins, with almost none high-level contracts specified. Also, most of the plug-ins contained a great number of packages and classes, mixing both generated and custom code, which made it even more difficult to understand the code execution flow. As a result, introducing changes to such a piece of software was extremely difficult, as making a, potentially safe, change in one part of the code immediately resulted in compiler errors in either other packages, or even different plug-ins. Because of all these issues, the learning curve for RED development was so steep that it was hardly possible to introduce any significant changes in the regular development cycle, being a few months long thesis project.

Taking all of the above into the account, and the fact that **Harald Störrle's** aspirations for RED were for it to be a professional tool aiding 02264 students in their course projects, I've found it unacceptable that from the maintainability perspective RED was a total disaster. Being developed by software engineering students, one would assume that RED would be a state-of-the-art application, starting from the project management, through the well-thought architecture and implemented as a high quality code. Unfortunately, at all of these steps, there were a number of gaps that, while not initially problematic, turned out to be the major roadblock in RED development.

Summing up, the RED re-engineering process started by that thesis has certainly ended up with a number of maintainability improvements that will benefit the future developers of RED. While there is not much outcome when it comes to new features, and most of the "under-the-hood" work that has been done will probably not even be noticed by the casual RED users, it will benefit them as well in the long run. By making RED open source, we invite the students to contribute, either by developing new features or by filling bug reports and feature requests, which should increase the development pace, resulting in much more stable and feature-packed RED in the close future. Also, by simplifying and documenting the steps required to set up the development environment, as well as restructuring the RED architecture, it is now much easier for new developers to start working on RED than it was for myself or **Johan Flod**. Performing the re-engineering was definitely a complex and time consuming process, and taking into the fact that one of the most important goal of the thesis was to maintain as many features as possible, there was hardly any room for mistakes. The process was actually similar to performing a surgery, or walking over a minefield, where a tiny mistake may result in great consequences. I'd argue

that this was the greatest challenge of the thesis, as there were a number of trade-offs to take between introducing certain improvements and making sure that none existing functionality gets lost during the process. Also, while there is still a lot to do when it comes to RED, I believe the outcomes of that thesis will be highly appreciated by anyone who will work with RED in the future.

6.3 Future work

In terms of what is left to be done, there are still areas for improvement when it comes to RED. First of all, while a lot of both high-level and low-level restructuring has been done, some of the plug-ins may require more re-factoring. As the main focus went for re-engineering RED architecture from high-level features (or modules) down to low-level plug-ins, but there was not enough time to examine each of the plug-ins separately and therefore to improve the code quality within them. The reason for starting from a high-level approach was that restructuring RED at the module level would greatly contribute to improving the general architecture, as it would specify and enforce the contracts between various modules. Should we've started with improving the code quality in the individual plug-ins first, we would most likely have run out of time to introduce the necessary high-level improvements, and therefore not satisfying most of the thesis goals.

Another improvement that could be made is a bit more thorough restructuring of the CORE module. While the CORE module implementation has been divided into a number of plug-ins, it may benefit from further restructuring some of the contracts it introduces for the other modules to depend on. As CORE module serves as the base of almost any other RED module, by making proper changes we may simplify the other modules, which will greatly affect the overall code quality. However, applying any fixes and/or extensions to the CORE module needs to be done with extreme care, as any change may unfortunately break the existing functionality. One potential improvement that could be made is to simplify the `BaseEditor` class, that is currently tightly coupled to an entirely custom `BasePresenter` class, which, unlike the editors or views, is not an Eclipse concept, and is therefore difficult to understand. As such implementation enforces the developers to use a new, undocumented API, on top of an already complex Eclipse API, restructuring that will definitely make it easier to develop future extensions.

There are also problems related to the platform-independence of RED that has not been resolved as a part of the thesis. As pointed in [Fri12], when discussing various RichText editor implementations, an EPF RichText has been found the

most suitable and therefore used in RED. However, as it turned out during the restructuring process, EPF RichText is a platform-dependent solution, which does not work well on all the machines running either Linux or Mac OS. This is a clear example of a bad design decision made at the beginning of the development process, which resulted in severe consequences. As the support of rich text is one of RED most important features, it is not possible to simply remove it. Also, as EPF RichText editor is being used by almost any of the editors in RED, substituting it will be a major effort, not to mention the fact that there are not many other implementations that could be used. Either way, resolving that issue is of great importance if we want to maintain RED on non Windows-based platforms.

When it comes to additional functionality, it would be vital to implement a new *Specification Element* that would hold the model fragments and make it possible to associate them with other elements. However, in order to do that, a necessary improvement needs to be done in the ModelElements module, as it has proven itself to be unstable during certain circumstances. Fixing ModelElements would not only make it possible to convert it into another *Specification Element*, but would also contribute to fixing the issues in both Reporting and Weaving modules.

Last, but not least, with introduction of **Tool** module type, we have opened an entirely new approach of integrating various *Specification Elements* together. As for now, we do not allow various *Specification Elements* to communicate with each other directly, but only via either SpecificationElements or Core modules. However, by allowing such **Tool** modules, we provide an entirely new way of integrating the *Specification Elements*, by simply extending them with new functionality, such as an additional editor page, making it possible to associate various elements in a much more specialized way than simply using the generic RED association feature.

Bibliography

[Arn93] Robert S Arnold. *Software reengineering*. IEEE Computer Society Press, 1993.

[Bul12] Ian Bull. Eclipse juno milestone 7, available for download, 2012. [Online; accessed 2013-10-05]. URL: http://eclipsesource.com/blogs/2012/05/05/eclipse-juno-milestone-7-available-for-download/.

[Fou] Eclipse Foundation. Osgi concepts. [Online; accessed 2013-10-03]. URL: http://www.eclipse.org/virgo/documentation/virgo-documentation-3.6.2.RELEASE/docs/virgo-user-guide/html/ch02s02.html.

[Fou13a] Apache Foundation. About subversion, 2013. [Online; accessed 2013-10-01]. URL: http://subversion.apache.org/.

[Fou13b] Eclipse Foundation. Pde overview - eclipse plug-in development environment guide, 2013. [Online; accessed 2013-09-30]. URL: http://help.eclipse.org/indigo/topic/org.eclipse.pde.doc.user/guide/intro/pde_overview.htm.

[Fou13c] Eclipse Foundation. Target platform - eclipse plug-in development environment guide, 2013. [Online; accessed 2013-09-30]. URL: http://help.eclipse.org/kepler/nav/4.

[Fri12] Anders Friis. Basic tool support for requirements engineering. Master's thesis, Technical University of Denmark, 2012.

[Git13] GitHub. About github, 2013. [Online; accessed 2013-10-02]. URL: https://github.com/about.

[Kra12] Jakob Kragelund. Advanced tool support for requirements engineering. Master's thesis, Technical University of Denmark, 2012.

[McC76] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, 1976.

[Vog12a] Lars Vogel. *Eclipse 4 application development : Eclipse RCP based on Eclipse 4.2 and e4*. Vogella, S.l, 2012.

[Vog12b] Lars Vogel. Eclipse 4 ide not extendable via fragments or processors - bug report, 2012.

[Wik13a] Wikipedia. Acyclic dependencies principle, 2013. [Online; accessed 2013-10-05]. URL: `http://en.wikipedia.org/wiki/Acyclic_dependencies_principle`.

[Wik13b] Wikipedia. Cyclomatic complexity, 2013. [Online; accessed 2013-10-05]. URL: `http://en.wikipedia.org/wiki/Cyclomatic_complexity`.

[Wik13c] Wikipedia. Git (software) — Wikipedia, the free encyclopedia, 2013. [Online; accessed 2013-10-02]. URL: `http://en.wikipedia.org/wiki/Git_(software)`.