Origo Mylyn plug-in and advanced issue management

Chair of
Software Engineering

Origo Mylyn Plug-In
and
Advanced Issue Management

Michael Stämpfli

02-910-867
Master Thesis
August 2008 - February 2009

Chair of Software Engineering
Department of Computer Science
ETH Zürich

Dr. Till G. Bay
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

The Origo issue tracker is a small but powerful tool to keep track of outstanding bugs in project.

Using the issue API, a seamlessly integrated issue editing plug-in has been implemented for the Eclipse IDE.

With this thesis, we try to improve the usability of the issue tracker itself and the Eclipse plug-in by performing an extensive re-design of the issue tracker.

Acknowledgments

I would like to thank my supervisor Till Bay for his guidance and support in design decisions. I would also like to thank Dennis Rietmann for his support in Mylyn and Origo. My thanks also belong to Dominique Schneider, who always gave me valuable information about implementation details, Patrick Ruckstuhl for support with the Origo back-end and Julian Tschannen for an introduction in Drupal.

Contents

Chapter 1

Introduction

1.1 Goals

The Origo issue tracker allows developers to keep track of outstanding issues in their projects. It is often the case, that a project has more than thousand registered issues (open and closed ones). To operate efficiently with such a large amount of data, it is very essential, that the issue tracker provides appropriate features. Due to user input and our own experience with issue trackers, we identified several flaws. The first goal of this master thesis is to correct the detected flaws.

The second part of this master thesis discusses a plug-in to Eclipse [5] called Mylyn [9]. It allows users to locally edit issues. This plug-in is closely related to the issue tracker, since it uses the Origo API to retrieve and update issues. If the issue tracker changes, then the plug-in needs to be adapted to the changes as well. The second goal of this master thesis is to make the plug-in compatible with the redesigned issue tracker.

1.2 Overview

To build some basic knowledge about Origo, Chapter 2 gives a short introduction in the Origo system. Chapter 3 presents the implemented redesign of the issue tracker. Chapter 4 discusses the integration of the Origo issue tracker with Mylyn. Some additional work on Origo is explained in Chapter 5. The paper concludes with the results and some ideas for future work in chapter 6.

Chapter 2

Origo

2.1 General

Origo [12] is a distributed development platform. It is open, modular, extensible and integrates various tools. It consists of a back-end and a front-end which are separated. The back-end is responsible for controlling and managing data, the front-end is responsible for displaying data and information retrieved from the back-end. The interaction between front-end and back-end is done using API calls which are implemented using XMLRPC [20].

An important feature of Origo is the notion of a work item. Work items can be used by any Origo user to keep track of progress in own projects as well as in bookmarked projects.

2.2 Back-end

The back-end of Origo is a middleware architecture and control infrastructure for the Origo platform. The back-end contains programmed use cases that direct the interaction with the different services. The main target was a very good scalability and extensibility. For more detailed information about the back-end cf. [14].

2.2.1 Nodes

The back-end consists of nodes of different types, each having a different role and function. Nodes work in a P2P environment which makes it easy to add and remove nodes dynamically to and from the network. The inter-node communication uses a reliable message passing layer implemented as a JXTA service [8] using VamPeer [16].

For each node type, it is possible to have multiple actual nodes running. This allows good scalability and redundancy. Nodes can be addressed directly using their name.

- Core node: Message bus and controller of the back-end. It controls the other nodes according to user defined use cases by sending control instructions.
- API node: Provides an XMLRPC interface for Origo using Goanna [6]. For each incoming XMLRPC, a specific message is constructed and sent to the Core node. An API node can be started in a normal mode and in an internal mode. The internal mode provides special services that should only be available for the front-end.
- Config node: Used to execute configuration scripts and generate configuration and access control files directly on the server. This is used e.g. for the FTP and Subversion access rights management.
- Storage node: Responsible for managing all Origo related data. Internally a MySQL database [11] is used to store data persistently.
- Mail node: Allows sending of mails to Origo users. It can use a local or remote mail server to deliver mails.
- Build node: Starts a compilation of a given project. Not used at the moment.

2.3 Front-end

The front-end of Origo is based on the Drupal content management system [4]. It provides all necessary functionality to manage a project in Origo using the back-end. Drupal has a flexible extension system to include own themes and modules. Own modules use hooks to interact with the Drupal core and internal processes (see Drupal API [3]). Each Origo project is a separate Drupal site which theoretically could have its own special modules and themes. Also each Origo project has

an own Drupal database which allows to scale. An important feature is a single sign-on mechanism which allows a user to browse every project without having to log in every time. This is necessary because each project has its own Drupal site and therefore its own database and user table. The single sign-on mechanism uses two cookies to store a Drupal session in one cookie and an Origo user name together with the encrypted password in the other cookie. The main page is Origo Home where work items of owned and bookmarked projects are displayed. This page is independent of the currently browsed project and retrieves its data directly from the back-end using API calls.

Chapter 3

Issue Tracker Redesign

3.1 Overview

The Origo issue tracker is a small but powerful mechanism for keeping track of outstanding bugs in a project. It allows users to report irregular behavior of a project. The goal of the reported issue is to provide enough information, so that programmers are able to resolve the observed problem.

The information that a reported issue provides can be categorized as follows:

- An issue has a title which tells in very few word, what the issue is all about.
- A description which should contains all necessary information to understand the observed problem. Such information could be the steps to reproduce the problem, the expected and the observed behavior and information about the environment, in which the problem occurred.
- An issue is either public or private. Private issues will only be visible to project owners and members, public issues are visible to everyone.
- An issue is either open or closed. If an issue is closed, then the reported problem has been fixed.
- An issue can be assigned to a person, who is responsible to solve the problem.
- Additional tags to categorize the issue.

- Issues can also be commented by any other user, if they for example observe similar behavior.

As soon as someone adds, edits or comments an issue, the notification mechanism will be triggered. All users, who are subscribed to the issue workitem [9], will receive an email with the changed content.

3.2 Improvements

Over time, the issue tracker revealed more and more features that were incomplete or missing. Using the input of the Origo users and our experience with other issue trackers such as Bugzilla [2] and Trac [19], we created a redesign of the Origo issue tracker. The redesign consists of three different steps, which we discuss in the following.

3.2.1 Additional attributes

Currently the state of issues can only be expressed by the status attribute, which is either "open" or "closed". This is too little information to represent all possible states. What to do about duplicates? There is no official way to mark an issue as a duplicate of another. With the redesign we introduced a new attribute called resolution. The resolution is either "Open", "Fixed", "Won't fix", "Duplicate" or "Works for me". It is internally saved as a special tag resolution::value where value is replaced by one of the resolution options.

The redesign also demands to add issue planning attributes. The first new attribute is the deadline, which defines the last possible date at which the issue has to be fixed. The second new attribute is the estimated amount of work that will be necessary to fix the issue.

3.2.2 Search and filter options

The issue tracker uses the Taxonomy Module [17] to categorize issues by tags. With Taxonomy comes a query language called Taxonomy Query Language (TQL) [18]. In general, TQL is query language to search for Drupal nodes that contain the tags entered in the query. Since issues are also Drupal nodes, the result of such a query will also contain issues.

TQL as the search engine for the issue tracker would have several disadvantages. TQL can only search for tags. It is not possible the query to title or the reporter of an issue. This missing feature is a heavy drawback. Also, TQL requires users to know its syntax to perform a query. This is not that bad, because TQL is quite easy and intuitive. But it would be more comfortable and efficient to search issues with a single mouse click instead of entering a query in a text field. Hence TQL is not well suited for an efficient issue search engine.

Instead we implemented a separate search mechanism specifically for issues. The main idea is to make users able to search issues with a few mouse clicks. We used the a plug-in for Mozilla Thunderbird called Seek Plug-in for Thunderbird [15] as a guideline. For each attribute of an issue the issue tracker search engine shows a list box. The list boxes are filled as follows:

- Status list box: possible values are “open” and “closed”
- Resolution list box: possible values are “Fixed”, “Won’t fix”, “Duplicate” and “Works for me”
- Tags list box: possible values are all tags of all project issues
- Visibility list box: possible values are “Public” and “Private”
- Reporter list box: possible values are all users who reported an issue
- Assigned to list box: possible values are all users to whom an issue is assigned to

Figure 3.1 shows how it looks like. Clicking a item of a list box will active the filter and show only issue whose attributes match the selected value. Additionally we provide a text field to search for a specific text in the title or the description of an issue.

Figure 3.1: The layout of the issue filter.

As soon as the filter changes an AJAX [1] request is sent to the Origo frontend, where the actual filtering takes place. As long as the client didn’t receive

a response from the server, it replaces the old issue table with a progress bar. The Origo front-end searches all issues, that match the selected filter values, and generates a new issue table in HTML. Then it sends the issue table back to the client, which replaces the progress bar with the new issue table. This guarantees a fast and comfortable way to filter issues even without reloading the whole page.

To improve usability even further, the issue tracker allows filters to be saved under a name. You can for example save a filter to select all open issues, that are assigned to you under the name “My issues”. A button “My issues” will then appear in the “Personal filters” section (see Figure 3.2). Clicking this button will then instantly set the filter such that it shows all open issues, that are assigned to you.

Figure 3.2: Personal filters are shown above the filter section.

3.2.3 Issue Notification

The issue notification is a tool that notifies users about changes on issues. All users who marked the issue workitem subscription will receive an email for every change. Unfortunately this leads to an unnecessary email flood, since most users would only like to be informed about few issues and not all of them. Besides it is very troublesome for the users if they receive a lot of issue notifications of which 90% is garbage.

For these reasons we refined the mechanism to provide a more fine grained subscription. Beside the issue workitem subscription, it is now possible to subscribe single issues. So users can subscribe themselves to the issue in which they are interested.

The implementation of the issue subscription is very simple. Every user, who wants to subscribe to an issue, can add the tag subscribed::user-name to the issue, where user-name will be replaced with the name of the subscribing user. The issue view provides buttons to subscribe or unsubscribe issues.

As soon as an issue changes, the back-end collects all email addresses of the users, who are subscribed to the issue workitem, and all email addresses of the

Figure 3.3: A button is shown in the issue view to subscribe or unsubscribe the current issue

users, who are subscribed to the changed issue. Since a user can be subscribed in two ways, the back-end has to guarantee that no user receives two emails about the same change. The easiest way to do this, is by computing the union of the two address sets. Then the back-end send a notification email to the addresses of the united set.

Chapter 4

Mylyn

4.1 Overview

Mylyn [9] is a task management plug-in for Eclipse [5]. The developers describe it as follows:

> Mylyn is a task-focused interface for Eclipse that reduces information overload and makes multi-tasking easy. It does this by making tasks a first class part of Eclipse, and integrating rich and offline editing for repositories such as Bugzilla, Trac, and JIRA. Once your tasks are integrated, Mylyn monitors your work activity to identify information relevant to the task-at-hand, and uses this task context to focus the Eclipse UI on the interesting information, hide the uninteresting, and automatically find what's related. This puts the information you need to get work done at your fingertips and improves productivity by reducing searching, scrolling, and navigation. By making task context explicit Mylyn also facilitates multitasking, planning, reusing past efforts, and sharing expertise.

Mylyn has a generic interface that makes it easy to integrate any repository into Eclipse. The plug-in, that connects a repository with Mylyn, is called a connector. Each repository, like Bugzilla, Trac or JIRA, has its own connector. Dennis Rietmann created in his master thesis a connector for the Origo issue tracker [13], which makes it possible to locally edit Origo issues.

The Origo connectors is based on Mylyn 2. Recently Mylyn advanced to the major version 3. The developers made the API more consistent and thus it changed

a lot. As a consequence, the existing Origo connector doesn't work with Mylyn 3. In the following, we present the Origo connector version 2 based on Mylyn 3.

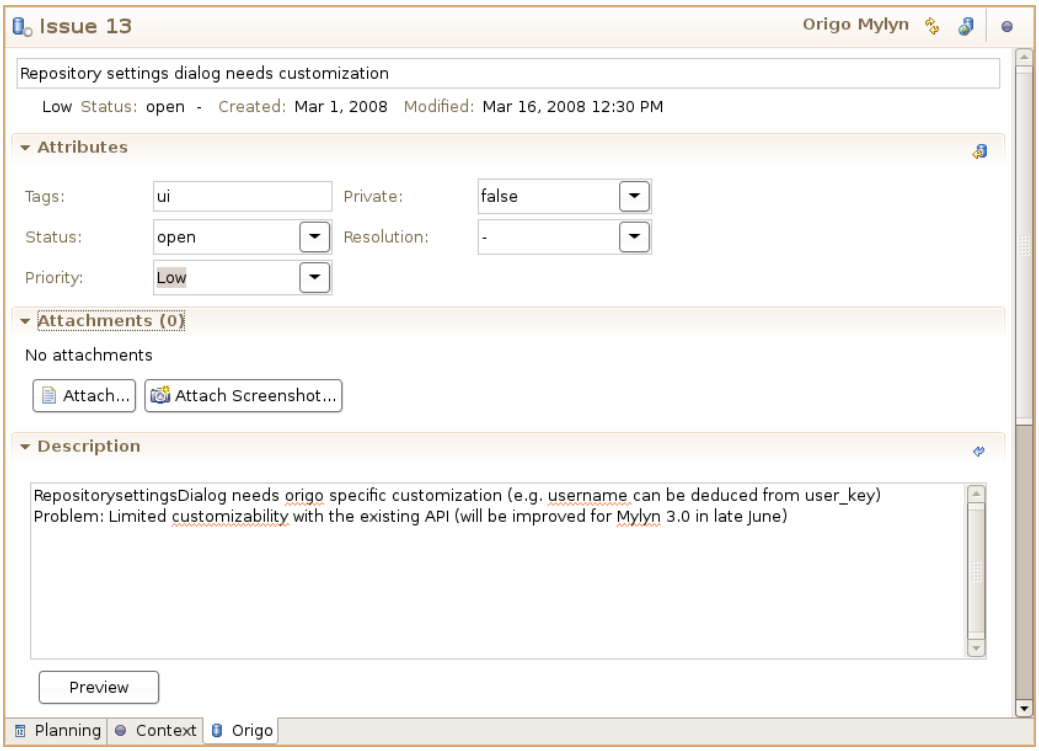

Figure 4.1: A typical overview of an issue in Mylyn

4.2 Existing Work

Documentation for the API of Mylyn 3.x barely exists. There is a porting guide [10] to adapt a connector from Mylyn 2.x to Mylyn 3.x, but it is poorly structured and incomplete. The only reliable resources are the existing connectors for Bugzilla, Trac and JIRA. In the development process we often considered the source code of those connectors.

To concepts behind the Origo connector are well explained in the master thesis of Dennis Rietmann in chapters 5-7 [13]. In the following, we will not explain these concepts again, since they didn't change, but rather show the differences between the old and the new Origo connector.

4.3 Data model

The issues are retrieved by the Origo API in a hash map based representation. This representation will then be mapped to the class *Issue*, which is the internal representation for the Origo connector. Formerly, this class had to be a subclass of *AbstractTask* for Mylyn to understand it. In Mylyn 3, *Issue* can be a non-descendant class. The class *Issue* has then be mapped to the class *TaskData*. This class is the main representation of a task in Mylyn 3. Every Origo issue or Bugzilla bug has to be mapped to *TaskData*. A task is identified by a per-repository unique id. In the case of the Origo connector, this would be the attribute "issue_id".

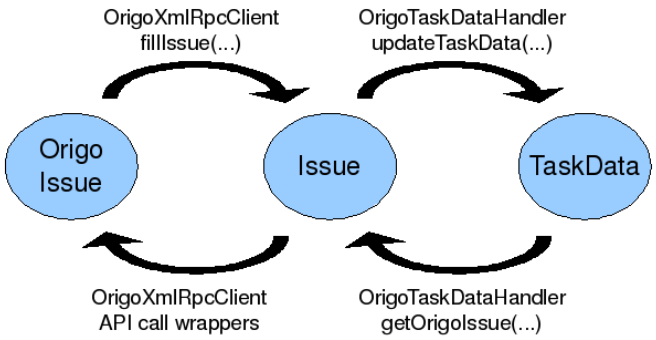

Figure 4.2: Convert issues from Origo representation to Mylyn representation and vice versa

The next important concept heavily related to tasks are task attributes. An attribute has two functions:

1. it stores the value of a field of the class *Issue* like the title or the description.
2. it stores meta data about the value, such as data type, read-only and label string, to define the presentation style in the UI plug-in.

The meta data for the attribute creation is stored in the enumeration class *OrigoAttribute*. The mapping from *Issue* to *TaskData* happens in the function *OrigoTaskDataHandler.updateTaskData*(). For each entry in *OrigoAttribute*, it creates an attribute in the *TaskData* instance and sets its meta data. Afterwards, it sets the values of the attributes with the values from *Issue*.

Instead of converting the hash map representation to the class *Issue* and the class *Issue* to the class *TaskData*, one could easily implement a direct conversion

from the hash map representation to the class *TaskData*. I decided against this solution, since the class *Issue* exactly represents an Origo issue and it adds an additional degree of structure.

4.4 Attachments

A very essential feature of the Origo connector is the attachment mechanism. Issue attachments can be very handy to provide additional information about issues like screenshots or source code files.

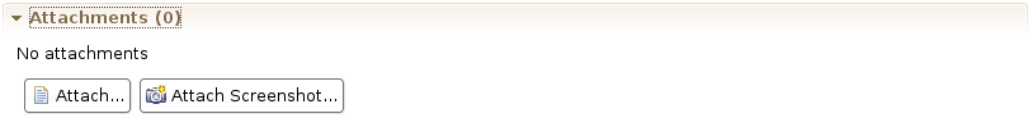

Figure 4.3: Add attachments to provide additional information about an issue

Attachment files are managed by Drupal, the Origo front-end. Each project has it own folder, in which attachments are saved. This means, that attached files must be uploaded to this very folder. Because of this, the attachment upload procedure works as follows:

1. The attachment will be uploaded via FTP to the users FTP account.
2. The Origo connector calls the functions issue.add_attachment via XML-RPC.
3. The Origo back-end will then move the file from the users FTP account to the project attachment folder and save the attachment meta data in the Origo and Drupal database.

After these three steps, the attachment is accessible from the back-end using API calls and from the front-end using the web interface.

The download of an attachment is a lot simpler. Since the attachment files are managed by the Origo front-end, we can access all of them with a HTTP request. The URL of the attachment file called “attached_file.jpg” on project “myproject” looks like this: http://myproject.origo.ethz.ch/system/files/attached_file.jpg.

Chapter 5

Other work on Origo

This chapter gives a overview over some small tasks related to this master thesis.

5.1 Project creation with project owner

The project creation consisted of two steps:

1. The project creation
2. Adding the project owner

I adapted the API of the project creation to combine those steps into a single one. The API call project.create now takes an optional parameter “owner”. If the owner is not empty, the specified user will automatically be set as the owner of the project. If the owner is empty, the project will be created without an owner.

5.2 Project removal

Since there will always be unused projects, the Origo administrators need to be able to remove projects. For this, I implemented an API call project.remove. The project removal consists of several steps:

1. The project and all its dependencies have to be removed from the Origo database. Project dependencies are issues, communities, user roles for the project, etc.

2. The subversion repository has to be deleted. A backup of the latest repository dump will be saved in a zip archive.

3. The subversion access file has to be updated.

4. All project releases have to be deleted.

5. The shared jabber rooster [7] has to be removed.

6. The project has to be removed from Drupal. This is achieved by dropping the database origo_project-name, where project-name will be replaced by the removed project name.

5.3 FTP race condition

We continuously received complaints about newly created users, who could not login to their FTP account. The source of this error lies in a race condition of the messages sent by user creation use case in the Origo back-end. It may have happened that the ftp access file has been updated before the new user was inserted in the database. This is now fixed.

Chapter 6

Conclusion

6.1 Difficulties

The biggest difficulties occurred with the porting of the Origo connector from Mylyn 2 to Mylyn 3. Documentation barely exists and it was thus very difficult to understand how Mylyn 3 works. Fortunately the source code of existing connectors for Bugzilla and Trac provided enough information make the Origo connector work.

6.2 Future Work

There are some possibilities to extend the Origo issue tracker and the Origo connector.

- Issue planning attributes like deadline and estimated amount of work have already been implemented. But in this context there is at least one attribute missing: the scheduled date, which indicates a point in time where the user plans to fix an issue. This attribute exists per issue and user, which requires an additional table in the database of the Origo back-end.
- Currently all issue data is stored double. The back-end stores it to provide the API for issues. The front-end needs to store it because of the underlying structure of web pages of Drupal. This can easily lead to inconsistencies, between the two systems. It would be more adequate to save it only in the

back-end. One would have to carefully investigate the necessary changes in the front-end.

- The configuration of the Origo connector is currently hardcoded, e.g. the attributes of an issue, the URL of the FTP upload server, the special tags and their possible values (resolution: normal, fixed, won't fix, duplicate, works for me). It would be nice, if the configuration was stored in the Origo back-end as well. As soon as the Origo connectors starts, it would then update its config with the server. This way the connector would be more flexible and we wouldn't have to release a new version of the Origo connector with every new special tag.

List of Figures

Bibliography

[1] AJAX. URL http://www.w3schools.com/Ajax/Default.Asp.

[2] Bugzilla. URL http://www.bugzilla.org/.

[3] Drupal API. URL http://api.drupal.org/.

[4] Drupal Content Management System. URL http://drupal.org/.

[5] Eclipse IDE. URL http://www.eclipse.org/.

[6] Goanna project. URL http://goanna.sourceforge.net.

[7] Jabber. URL http://www.jabber.org/.

[8] JXTA Project. URL http://www.jxta.org.

[9] Mylyn Plug-in for Eclipse. URL http://www.eclipse.org/mylyn/.

[10] Mylyn Porting Guide. URL http://wiki.eclipse.org/Mylyn/Porting_Guide/3.0.

[11] MySQL Database. URL http://www.mysql.com/.

[12] Origo. URL http://www.origo.ethz.ch.

[13] Dennis Rietmann. Origo plug-ins - extension and maintenance. Master's thesis, ETH Zürich, March 2008.

[14] Patrick Ruckstuhl. Origo core - middleware and controller for origo. Master's thesis, ETH Zürich, July 2007.

[15] Seek Plug-in for Thunderbird. URL http://simile.mit.edu/seek/.

[16] Beat Strasser. VamPeer - JXTA implementation for Eiffel. Master's thesis, ETH Zürich, March 2007. URL http://vampeer.origo.ethz.ch.

[17] Taxonomy Module. URL http://drupal.org/handbook/modules/taxonomy.

[18] TQL - Taxonomy Query Language. URL http://drupal.org/project/tql.

[19] Trac. URL http://trac.edgewall.org/.

[20] XML-RPC. URL http://www.xmlrpc.com/.