A DESKTOP CLIENT INTERFACE FOR JIRA

A Thesis
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Sreejith Gopinath
Fall 2010
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Sreejith Gopinath:

A Desktop Client Interface for JIRA

Joseph Lewis, Chair
Department of Computer Science

Carl F. Eckberg
Department of Computer Science

Janet Bowers
Department of Mathematics and Statistics

10-28-10
Approval Date
Copyright © 2010

by

Sreejith Gopinath

All Rights Reserved
DEDICATION

My journey to an advanced degree in Computer Science was by no means an easy one. It was a journey fraught with struggles, frustrations, joys and sorrows and will not have been possible without the explicit support of so many individuals and the silent support and encouragement of many more. This thesis was a culmination of that long and arduous journey, and it is fitting that I dedicate this work to each person who has been a pillar of support during this time. I dedicate this work, with lots of love and respect, to my family, friends, colleagues and teachers, all of whom always have, are and always will rejoice in my success. A special dedication goes out to my sweet little niece, Medha, who eased those tiring days with her joyous laughter.
“The larger the island of knowledge, the longer the shoreline of mystery.”

-Anonymous
ABSTRACT OF THE THESIS

A Desktop Client Interface for JIRA
by
Sreejith Gopinath
Master of Science in Computer Science
San Diego State University, 2010

JIRA is a bug, issue and project tracking software developed by Atlassian Software Systems. JIRA is very popular, being used in over 11,500 software development and research organizations worldwide. JIRA is used by software developers and managers to track tasks and bug fixes. A JIRA installation provides a web-based system where users may log on to create tasks, assign tasks, log work and comment on tasks, vote for new features and so on. However a perceived limitation is that only one issue can be updated at a time. The user also has to visit a separate web page to perform each function like editing the task, progressing the task along pre-defined workflows, editing the task, etc. While this may be acceptable for junior developers who may work on just one or two tasks at a time, the complexity fairly explodes in the case of project managers and other administrators who may be tracking scores or hundreds of tasks at a time. The time spent in visiting many web pages to update a single task, and then repeating the process for every task that needs attention, simply becomes too much of a hassle.

To address this perceived shortcoming, we propose a desktop client to JIRA that allows concurrent access to many tasks at the same time. A two-dimensional grid view allows simultaneous access to all the desired attributes of a group of tasks. The time saved and the ease of updating many tasks at one go improves the productivity of each user and by enabling comparative editing, reduces the error rate. All this translates into huge savings for the enterprise. The design is kept as intuitive and close to the original JIRA design as possible, in order to flatten out the learning curve for users.
# TABLE OF CONTENTS

**PAGE**

ABSTRACT ............................................................................................................................. vi
LIST OF TABLES ................................................................................................................... ix
LIST OF FIGURES ................................................................................................................ x
ACKNOWLEDGEMENTS .................................................................................................... xii

## CHAPTER

1 INTRODUCTION .........................................................................................................1

2 REVIEW OF THE LITERATURE ...............................................................................3
   2.1 JIRA as a Project Management Tool .................................................................3
   2.2 The Need for a Client Tool ..............................................................................11
   2.3 Candidate Technologies ...................................................................................17

3 SYSTEM DESIGN OVERVIEW ................................................................................19
   3.1 Design Philosophy ...........................................................................................19
   3.2 Features Required in the Tool............................................................................19
   3.3 High Level Design ...........................................................................................20
   3.4 Class – Level Design .......................................................................................21
      3.4.1 The Presentation Layer .......................................................................... 21
         3.4.1.1 MainWindow ................................................................................ 21
         3.4.1.2 SearchScreen ................................................................................. 23
         3.4.1.3 CreateIssueScreen ......................................................................... 23
         3.4.1.4 EditIssueScreen ............................................................................. 23
         3.4.1.5 WorklogScreen ............................................................................. 23
         3.4.1.6 CommentScreen ............................................................................ 23
      3.4.2 The Data Interaction Layer .................................................................... 23
         3.4.2.1 SessionManager ............................................................................ 24
         3.4.2.2 ClientManager ............................................................................... 24
      3.4.3 The Local Cache Layer .......................................................................... 25
         3.4.3.1 TaskScheduler ............................................................................... 25
3.4.3.2 DataCache ................................................................. 26
3.4.4 JIRA Installation............................................................ 26

4 SYSTEM IMPLEMENTATION OVERVIEW..................................................27
  4.1 Development Tools Used ....................................................... 27
  4.2 The Presentation Layer ......................................................... 27
  4.3 The Data Interaction Layer .................................................... 32
  4.4 The Local Cache Layer ......................................................... 32
  4.5 The JIRA Layer ................................................................... 39

5 THE TEST FRAMEWORK ..............................................................43
  5.1 Rationale for Developing a Test Framework ......................... 43
  5.2 Design Methodologies .......................................................... 44
  5.3 Test Framework Technologies .............................................. 44
  5.4 Test Framework Implementation .......................................... 45

6 SUMMARY, CONCLUSIONS AND FUTURE WORK ................. 49
  6.1 Summary of Accomplishments ............................................. 49
  6.2 Conclusions ....................................................................... 50
  6.3 Future Work ...................................................................... 50

BIBLIOGRAPHY ........................................................................ 52

APPENDIX
USER MANUAL FOR JIRA DESKTOP CLIENT .............................. 54
LIST OF TABLES

Table 2.1. Ahmad and Laplante's Software Project Management Tool Selection Criteria ...........................................................................................................................8
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A low-end project management tool</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>The Primavera Project Planner</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>JIRA local installation home page</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Issue home page in JIRA</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Architecture diagram of the client</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Class diagram of the presentation layer</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Class diagram of the data interaction layer</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Class diagram of the local cache layer</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>The main window</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>The menubar</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>The Start menu</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>The Login dialog</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>The Tools menu</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>Displaying unpublished worklogs</td>
<td>30</td>
</tr>
<tr>
<td>4.7</td>
<td>Selecting attributes to be viewed</td>
<td>30</td>
</tr>
<tr>
<td>4.8</td>
<td>Setting the refresh rate</td>
<td>30</td>
</tr>
<tr>
<td>4.9</td>
<td>The Actions menu</td>
<td>31</td>
</tr>
<tr>
<td>4.10</td>
<td>The JQL Search window</td>
<td>31</td>
</tr>
<tr>
<td>4.11</td>
<td>The Create Issue screen</td>
<td>31</td>
</tr>
<tr>
<td>4.12</td>
<td>Outline of the SessionManager class</td>
<td>33</td>
</tr>
<tr>
<td>4.13</td>
<td>Outline of the ClientManager class</td>
<td>34</td>
</tr>
<tr>
<td>4.14</td>
<td>Outline of the CacheManager class</td>
<td>38</td>
</tr>
<tr>
<td>4.15</td>
<td>Outline of the JobScheduler class</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>Outline of unit test for login functionality</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Outline of unit test for login screen</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Output of the code coverage tool</td>
<td>48</td>
</tr>
<tr>
<td>A.1</td>
<td>Setting the refresh rate</td>
<td>55</td>
</tr>
</tbody>
</table>
Figure A.2. Select attributes to be displayed.................................................................56
Figure A.3. Login to JIRA..............................................................................................56
Figure A.4. On successful login....................................................................................57
Figure A.5. Viewing tasks by project.............................................................................57
Figure A.6. Viewing tasks by filter................................................................................58
Figure A.7. Searching for issues by JQL Query..............................................................58
Figure A.8. Viewing sub-tasks. ....................................................................................59
Figure A.9. Creating an issue.......................................................................................59
Figure A.10. Editing an issue.......................................................................................60
Figure A.11. Editing an issue in-place..........................................................................61
Figure A.12. Deleting an issue.....................................................................................61
Figure A.13. Logging work on issues............................................................................62
Figure A.14. Commenting on an issue.........................................................................63
Figure A.15. Starting time tracking...............................................................................64
Figure A.16. Stopping time tracking............................................................................64
Figure A.17. Viewing unpublished worklogs...............................................................65
ACKNOWLEDGEMENTS

Working on this project was a special challenge that would have remained unconquered but for the motivation and support of several of my colleagues, teachers and friends. My gratitude goes out to my colleagues at Teradata Corporation in San Diego, for giving me so many wonderful ideas that inspired this work. Gratitude and respect in no less measure also go out to the chair of my thesis committee, Dr. Joseph Lewis, for spending time with me and guiding my work in the midst of his demanding work schedules. Dr. Carl Eckberg has been a pillar of strength and support during this time. Dr. Janet Bowers has been extremely helpful and the strength of her commitment to her position on my thesis committee is something that I can only hope to emulate in my life. Family and friends who were instrumental in the presentation of this thesis are also gratefully acknowledged.
CHAPTER 1

INTRODUCTION

JIRA® is a proprietary project management tool that is developed by Atlassian Software Systems. The development rationale for JIRA is being able to use a single application to track bugs, issues and manage the software development process. JIRA is being used in upwards of 11,500 software development and research organizations worldwide, and with increasing popularity, that number is growing. Several noted developer groups have integrated JIRA into their development processes, and this is attesting to the trust and reliability that developers are placing on JIRA.

JIRA is developed in Java, and consequently, is a cross-platform product. JIRA can be integrated with several popular version-control applications like ClearCase, SubVersion and CVS. JIRA supports remote procedure calls over the SOAP and XML-RPC protocols; to facilitate this, JIRA exposes a Java API.

At the core of the JIRA methodology is the concept of an issue. Every new feature request, bug report, support question, assigned task and so on, is called an issue in JIRA. An issue could have several qualifying attributes such as an issue key, the project to which it belongs, names of the reporter and assignee, release versions that could be affected by the fix and so on. Issues may also have several sub-tasks under them. As such, JIRA becomes a versatile tool that can be used by software users, software developers and software development managers for their own uses. Software users may use the JIRA system to report a bug that they encountered in their day-to-day usage of the software, to request a new feature that they would like to see in the software system or to raise a support question. Software development managers may use the JIRA system to assign tasks to project leads/developers who work under them. Software developers may use the JIRA system to create individual sub-tasks under tasks that have been assigned to them. JIRA also allows users to log time on issues; in this way, leads and managers are able to assign and track development objectives while also maintaining a close watch on resource utilization.
Software developers are able to monitor their workload at any given time. By virtue of the
time logging feature, JIRA enables tracking and accountability.

The current usage pattern of JIRA involves the home page of the JIRA installation.
The user is able to log on to the JIRA system from the home page. On the home page is show
a subset of the issues currently assigned to the user. The user may also perform advanced
searches or write a JQL (JIRA Query Language) query to query the system for any issues
(that may or may not be assigned to the current user). The issues are displayed as hyperlinks.
Clicking on any of these links leads the user to the homepage of that issue, where the user
may view the attributes of that issue. On this page are provided links that enable the user to
edit this issue (based on permissions), log work on the issue, add a comment to the issue, and
so on. Each of these operations is carried out in a separate web page.

One issue with this usage paradigm is that users have to navigate to different web
pages to accomplish a set of tasks on a single issue. Depending on the capacity of the JIRA
installation and various factors like network latency and server load, each operation may take
up to several seconds. This may be acceptable for developers who may have just a handful of
issues at any given time, but the problem is more pronounced in the case of project leads and
managers who may have scores or hundreds of issues to attend to within a short span of time.
Another issue is that users, for whatever reason, may want to update several issues in a
comparative manner. This is possible only if all the attributes of a group of issues are visible
all at the same time. This of course, is not possible in the case of JIRA, where each issue is
given its own web page.

In this study, we intend to investigate whether it is feasible to develop a desktop client
that could interface with JIRA and display groups of issues in a grid such that the user is able
to select which attributes to view, view all those attributes for all the issues in a group, and
make changes to those attributes right from that grid. It must also be possible to add
worklogs and comments without disturbing the grid display. All the issue search
functionalities available on the web version of JIRA must be available on this client too. In
addition, a time-tracking feature that automatically generates worklogs would be nice to
have. We intend to make use of the Java SOAP API exposed by JIRA, to develop this client.
See Appendix for a user manual for JIRA desktop client that resulted from this study.
CHAPTER 2

REVIEW OF THE LITERATURE

In this section, we present a systematic review of relevant literature that has appeared in several prestigious conferences over the past few years. The review has been organized into three sections – the first section investigates the features and capabilities that users demand from software applications that perform issue, task and bug tracking, provide metrics for continuous or historical project planning purposes, supports requirements elicitation and communication among stakeholders and how JIRA matches up against these expectations. The second section investigates the challenges faced by web applications. This gives an insight about the factors that challenge the adoption of JIRA across software development and research organizations and how those shortcomings may be remedied. The third section investigates candidate technologies that may be used to develop a client tool to interface with JIRA. The final section leads up to a conclusion on the features that are to be built into the client tool and the technologies that will be used to build them.

2.1 JIRA AS A PROJECT MANAGEMENT TOOL

Managing large software projects is a complex process and requires adherence to certain discipline, best practices and above all, the support of tools [1]. Previously, project management tools and their usage were regarded as the domain of professional managers, but advent in technology, practices and innovations have changed that view. Current trends in project management software have leveled the playing field in such a way that a professional manager boasts no edge over a technical manager. Over the years, several studies have attempted to categorize project management software. According to a recent study by Fabac et al. [2], project management tools may be classified into three major classes – the low-end tools that have good scheduling but little or no tracking support, the high-end tools that have good multi-project handling capabilities, and a mid-range class of products that lie between the two. Low-end tools are not very expensive, the most popular examples being MS-Project from the Microsoft stables and SureTrak from Primavera. MS-Project allows project
managers to develop workplans and budget, allocate resources and track progress on work in progress. A screen shot of an MS-Project window is shown in Figure 2.1.

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Duration</th>
<th>Start</th>
<th>Finish</th>
<th>Work</th>
<th>Resource Names</th>
<th>Cost</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training of Software</td>
<td>19.5 days</td>
<td>Tuesday 10/15</td>
<td>Friday 10/25</td>
<td>5 days</td>
<td>2 man-months</td>
<td>$20,000</td>
<td></td>
</tr>
<tr>
<td>Training of Technical Skills</td>
<td>19.5 days</td>
<td>Tuesday 10/15</td>
<td>Friday 10/25</td>
<td>5 days</td>
<td>2 man-months</td>
<td>$20,000</td>
<td></td>
</tr>
<tr>
<td>User Manual</td>
<td>3.5 days</td>
<td>Tuesday 10/15</td>
<td>Friday 10/25</td>
<td>2 days</td>
<td>1 man-month</td>
<td>$10,000</td>
<td></td>
</tr>
<tr>
<td>Quality Assurance</td>
<td>5 days</td>
<td>Tuesday 10/15</td>
<td>Friday 10/25</td>
<td>2 days</td>
<td>2 man-months</td>
<td>$25,000</td>
<td></td>
</tr>
<tr>
<td>Acceptance Test Plan</td>
<td>3.5 days</td>
<td>Tuesday 10/15</td>
<td>Friday 10/25</td>
<td>2 days</td>
<td>1 man-month</td>
<td>$10,000</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.1. A low-end project management tool.**

MS-Project presents a hierarchical grid-based view of tasks and helps managers to imbibe a lot of tracking information at a glance. However, MS-Project is not capable of matching predicted work product quantity based on the available resources, and this is one of the shortcomings of MS-Project. However, third-party add-ons for MS-Project are available, which serve to enhance its capabilities. Mid-range tools typically cost more than a low-end tool, but significantly less than high-end tools. Popular tools that fall into this category are CA-SuperProject from Computer Associates and Project Scheduler from Scitor. High-end tools are those that can handle multiple projects simultaneously. They boast several complex functionalities and are significantly more expensive than low-end and mid-range project management tools.
management software. Some of the popular high-end project management tools are the Primavera Project Planner and JIRA. See Figure 2.2.

Figure 2.2. The Primavera Project Planner.

JIRA is a project management tool that may be classified as a high-end tool, since it is capable of handling many projects in an organization. JIRA is also becoming popular with Small and Medium Enterprises (SMEs) and research organizations that need little or no task/issue tracking capabilities. In the past, several research studies have concentrated on identifying factors that would give a project management tool the definitive edge over other tools in its genre. See Figures 2.3 and 2.4.

Yet another suggestion made by Fabac et al. [2] is to classify project management tools according to their interfaces/mode of operation. This classifies project management tools as classic or online; the online tools are the ones that generally expose a web or browser-based interface, and the classic tools are the ones that expose a non-web interface.
Figure 2.3. JIRA local installation home page.

Figure 2.4. Issue home page in JIRA.
Ahmad and Laplante [3] have used a process called the Analytical Hierarchy Process to solve the problem of determining what features are most important in a project management tool. Their study isolates a set of twelve different features on which the suitability of project management tools to a project may be evaluated and designates them as the most important evaluation points in the project management tool selection process. These features are mentioned here in the order of mention in the paper: the ability of the tool to do task scheduling, resource management, collaboration among stakeholders, time tracking, task estimation, risk assessment, change management, reporting and charts, file attachments (allowing crash dumps or design documents to be attached to issues), email notifications, workflows (representing process/methodologies) and portfolio management. See Table 2.1.

Task scheduling and time tracking are very important from a project management as well as an individual perspective. Smith [4] makes a strong case for time-tracking, saying that tracking time spent on tasks leads to better utilization of time. The author recommends that graduate students maintain a log of their activities each day, so that they are able to identify tasks on which they spent more time than was warranted, and tasks that were missed out due to erroneous distribution of time. The author explains the significance from the perspective of a graduate student, but it is evident that this scenario easily holds true in any multiple-tasking environment where time is of the essence. From the perspective of the software development industry, this means that developers, project leaders and managers could make better use of their time if there was a way to log their activities and the time that they spend on each of their tasks. Indirectly, it also has the effect helping users to prioritize their tasks effectively. JIRA allows project managers and leads to break down large projects into progressively smaller chunks, which may then be allocated to developers. Scheduling in JIRA is accomplished by means of setting an estimated allocation of development time and a date on which the deliverable is due. Tracking time is accomplished by using worklogs [5], thereby affording management and developers the benefits of tracking time. The necessity and benefits of individual process and time tracking has been cognized by the Workflow Management Group and the Object Management Group, who are pushing for this feature to be included in the diversity of project management and tracking tools that are flooding the market today. As such, the ability to track individual time is an important criterion that must be evaluated prior to zeroing in on a project management tool for acquisition. A very
<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Task Scheduling</td>
<td>Task scheduling refers to the assignment of start and end times to a set of tasks. This feature lets software project managers track important project milestones and note who is responsible for each task.</td>
</tr>
<tr>
<td>2 Resource Management</td>
<td>This feature lets the software project manager organize and trace requirement details to ensure that proper resources are committed to the project. The software project manager can establish information relationships between multiple documents, assign attributes to the information, such as task assignment, priority and status, and change these over time to reflect changes in the project.</td>
</tr>
<tr>
<td>3 Collaboration</td>
<td>Collaboration enables both structured and free-flow sharing of knowledge and best practice. It includes project status reports that are accessible via a Web page, integrated e-mail or discussion boards.</td>
</tr>
<tr>
<td>4 Time Tracking</td>
<td>Time tracking allows recording, analyzing and reporting associated with project working routine. Software project managers can use the time tracking feature to manage employee timesheets and expenses, calculate salaries, prepare project estimates, and get invoices based on personal or client work rates.</td>
</tr>
<tr>
<td>5 Estimating</td>
<td>The estimate feature allows the project manager to generate, manage, and validate estimates of effort for a wide variety of projects. It evaluates the project plan, project requirements, information about working environment, and even different aspects of company’s culture.</td>
</tr>
<tr>
<td>6 Risk Assessment</td>
<td>Risk assessment helps the software project manager in identifying and planning for potential project risks. It could also help the software project manager in describing the various risk factors and how to score them.</td>
</tr>
<tr>
<td>7 Change Management</td>
<td>This feature lets the software project manager control schedules, resources, and deliverables of project. It can manage the impact that changes have on project objectives, and it lets the software project manager trace changes to see how each requirement's changes affect multiple other requirements.</td>
</tr>
</tbody>
</table>
Table 2.1. (continued)

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Reporting/ Charts</td>
</tr>
<tr>
<td>9</td>
<td>File Attachment</td>
</tr>
<tr>
<td>10</td>
<td>E-mail notification</td>
</tr>
<tr>
<td>11</td>
<td>Process/ Methodology</td>
</tr>
<tr>
<td>12</td>
<td>Portfolio Management</td>
</tr>
</tbody>
</table>

A desirable side effect of using these tools is that data collected and archived by these tools may be mined and analyzed by these or other tools. The analysis so performed enables developers and other users to identify their shortcomings and helps in instituting processes that contribute to personal development. Another feature that is desired in project management tools is the ability to track tasks.

Zimmerman et al. [6] discuss the advantages of being able to track bugs, which, though beyond the scope of this discussion, raises the need for support for bug-tracking. The authors highlight the importance of the completeness of initial bug reports. By having all the relevant information in the initial bug reports, developers may quickly get down to the task of identifying the root causes of the bug and fixing them. Without complete information, developers and reporters alike spend several cycles in the information gathering phase before meaningful work can start getting done. One way of capturing all the pertinent information is by asking all the right questions as the bug is being reported. For example, some of the most important questions that may be asked of the reporter are:
• How severe is the bug (not severe, critical, showstopper, etc.)?
• What component of the system does it affect (and the affected release)?
• What platforms are affected by the bug?
• What operating systems are the bug noticed on?

JIRA facilitates answers to these and other pertinent queries by the large number of attributes that an issue can have. By giving values to these attributes, in essence, the reporter is supplying all pertinent information that is required to triage and fix bugs. JIRA supports tasks, bugs, new feature requests, questions, etc. by defining a generic entity called an issue.

Mockus [5], in the paper titled Analogy Based Prediction of Work Item Flow in a Software Project: A Case Study, makes a very valid point by stating that in spite of all the research and studies that have gone into the software project management area, it still remains a formidable task to predict what will happen next in a software project. The significant questions that are asked several times in a software development cycle are:

• Will the product release goals be achieved by the set deadline, without compromising on the quality requirements?
• Is the progress of the current project on track with respect to the progress of past successful projects at the corresponding milestones?

All issues in JIRA progress through well-defined workflows [7] and this answers the need to be able to track issues and bugs. Collaboration among stakeholders takes place for a variety of reasons. The most common reasons are the elicitation of system requirements and acquisition of information for problem solving. A good project management tool must provide means for stakeholders to collaborate from within the context of the issue that they are dealing with. This means that any information regarding the issue, any documents that pertain to the issue and the like must be accessible to stakeholders as they deliberate on an issue. This will reduce the turnaround time involved in issue assignments and information gathering. According to Ahmad and Laplante [3], it is highly desirable to incorporate these features into a project management tool. JIRA makes accommodations for this need by allowing users to comment and vote on issues. Requirements elicitation in JIRA takes an unobtrusive form by defining many attributes that an issue may have, such as the product versions that the issue pertains to, the person who reported the issue, the person who the work item is assigned to and so on [7]. JIRA collects operational data and presents statistics in the form of reports and charts. This, while helping in task estimation through use of
historical data [7], also keeps project management updated on the current state of projects in terms of development tasks, bug fixes and patches. From our reading of the literature as outlined above, we see that JIRA comes pretty close in fulfilling the expectations that business users across the operating size spectrum harbor on a project management tool. This surmise is further strengthened by the fact that JIRA is used by a growing number of software development and research organizations across the world. The developers of JIRA are constantly busy incorporating newer and more desirable features into JIRA while phasing out the old and obsolete ones. This allows JIRA to remain highly competitive in the project management tool space.

2.2 THE NEED FOR A CLIENT TOOL

The current usage paradigm is based around how JIRA exposes itself to users. The JIRA installation has a home page where users can log on. Once logged on, users can pull up a list of tasks by project, by favorite or non-favorite filters or by running a JQL query. The list of tasks may span several web pages. Each task is listed as a URL which points to the task’s web page where the user can view attributes of that task and from where he/she can navigate to different web pages where work can be logged, comments added and the issue edited. The time spent in waiting for the different web-pages to load may be tolerable for users who have just one or two tasks to update, but in the case of administrative users who update a huge number of tasks at one go, this delay cumulatively builds up.

In a study conducted on Australian software development organizations, Verner and Cerpa [8] expound on the role of project management personnel and how their activities greatly influence the success of a software development project. The authors devised a questionnaire consisting of seven questions that were circulated among the stakeholders of a software development project. According to their evaluation of the responses to the questionnaire (the part of the study pertinent to our discussion), the success of projects directly correlated to the time and attention that project managers were able to give to their projects. This observation is best backed up by another reported fact – the success rate of software development projects directly correlated to management’s acknowledgement and appreciation of long working hours put in by software developers. Albeit in an indirect way, this fact underscores the argument that management involvement (and consequently, the time
invested in the project by management personnel, directly related to the outcome of a 
project).

Davison et al. [9] discuss the importance of tracking progress and effort put in by 
individual contributors into a project. The authors are of the view that it is difficult to 
determine the exact right degree of tracking in a project that can help manager make 
informed choices. From a detailed study of planning and how planning impacted project 
management decisions, the authors find that it makes sense to break down the tracking 
process into three separate components as below:

- Iteration planning
- Progress tracking
- Release tracking

Iteration planning deals with breaking down the set of goals or the entire development project 
into iterations, with a pre-agreed set of goals being achieved in each iteration. Iteration plans 
and refinements to iteration plans are made by actually looking at the goals outlined for an 
iteration and the progress actually achieved against those goals. In progress tracking, 
managers ask questions and try to determine whether the progress made in the project is 
satisfactory, relative to progress made in other projects in the past that have been successful, 
and at the corresponding milestones achieved. Progress tracking needs to capture a lot of 
metrics like the number of bugs fixed, number of new features added, and so on. This is very 
important from a project management perspective, since it gives managers an idea of how 
individual progress parameters actually contribute to the success of the project. Progress 
tracking includes the following methods:

- Team communication – where team members interact and peers discuss among 
  themselves how each other’s tasks and the project goals are being approached.
- Getting feedback from the customer – project management meets with customers on a 
  regular basis to present the progress made, and to elicit feedback about the 
  effectiveness of the developed work products.
- Documentation - as progress is made on individual tasks (developers) and on project 
  goals (team leads/technical managers), progress documentation is created, for higher-
  level managers and for archival purposes.
- Running test cases – this point is especially valid in the extreme programming 
  environment, where test cases are written prior to development. As code is developed 
  and test cases begin to succeed, progress becomes evident.
In release tracking, managers try to determine whether an initial allocation of time for a set of goals seem accurate in the current situation. The progress tracking activity becomes so important because it is not always carried out by technical personnel. There might be non-technical managers who are trying to understand the achieved progress, and therefore, tools must be developed in such a way that it is easy to capture, analyze and present tracking information. It is also important that progress tracking be easy and simple, since managers will need the changes in progress information very frequently in order to make slight and continuous changes to the way a project is operated. These phases of tracking make it mandatory to designate some of the stakeholders as iteration planning manager, progress tracking manager and release tracking manager, in order to ensure ownership for each of these activities. The argument in [10] that shows a direct correlation between the success of a project and the time invested by project management into the projects that they manage, is further bolstered by the preceding argument. Project managers need to invest a lot of time into tracking and managing the efforts that go into software development. They also discuss how collecting time metrics is but another small activity that demand a project manager’s time, so now we can fully appreciate the fact that there has to be a faster way to update tasks in JIRA, rather than wasting a lot of time in waiting for two or three pages to load just to complete updating a single task.

Every software development organization actively brings in several tools that are variously engaged in improving processes, quality, developer output and so on. However, it is also true that the initiation of every tool is met with some resistance by the stakeholders of the project, particularly those who have to use the new tool. It is clear that there is a certain amount of psychology at play here. Riemenschneider et al. [11] explore the role that user psychology (in this context, user is implied to mean a software developer) plays in the adoption of a new tool or methodology at the workplace. As a means of structuring the research into this area, they model their research on a few models that are listed and explained below:

- The technology acceptance model – 1
- The technology acceptance model – 2
- Perceived characteristics of innovating
- The theory of planned behavior
• Model of personal computer utilization

The technology acceptance model – 1 is a methodology that is used to gauge the acceptance of information technology applications like email and work trackers at the workplace. This model claims that an individual’s intention to use a system or application is dependent upon the perceived usefulness and ease of use of the new system or tool. The technology acceptance model – 2 is an extension of the technology acceptance model – 1, and it shows that usage of a tool is dependent upon the feeling of stakeholders that other stakeholders too must be motivated to use the new process or tool. The theory of planned behavior explores the relationship between an individual’s intention to use the new process or tool and the perceived internal and external constraints from the individual’s point of view. The model of personal computer utilization concerns itself with the relationship between an individual’s intention to use the new process or model, and the degree to which that individual has imbibed and internalized the organization’s culture with respect to adoption of new processes and tools. It also takes into account the historical trends in which new processes or tools have been adopted, and the circumstances surrounding those adoptions. The authors find that a tool or methodology is more readily accepted if it makes existing methodologies simpler to follow. Especially in regards to a tool, ease of use and reliability rank high on the factors that decide whether a tool is to be adopted or avoided [12] and [13].

The ideal process of selecting tools in an organization has been explored and described very thoroughly by Tran et al. in their paper titled Tool Selection Process and its Management for Small and Medium Enterprises in Defense Projects [14]. The authors identify the following steps as very important to the tool selection process. From our reading of the literature, it is evident that these steps are not specific to the defense industry, but are relevant to our investigation of tool selection processes in the software development industry as well. The steps are listed below, along with a brief explanation.

• Gather stakeholders – In this step, all the stakeholders of the processes that may be affected by the introduction of the new tool, are identified and gathered, in an effort to elicit their views, requirements and concerns on introducing the new tool into the operating environment.

• Identify rationales – list out the problems currently encountered in the operating environment, and identify which of those problems may be addressed by introducing the new tool.
1. Define operating scenarios – this is a complex process in which several activities are carried out. Firstly, the activities that mandate the introduction of the new tool are identified and listed. Next, a listing is made of the fiduciary and other criteria that are central to acceptance of the tool. An estimation is made of the potential fiduciary gains that the new tool could ultimately responsible for. Scenarios for demonstrating the tool to the stakeholders are determined and set up. A Request for Information (RFI) is sent out to potential vendors who may be able to supply such a tool. Lastly, a list of candidate tools and vendors is drawn up.

2. Financial preparations are made for the acquisition of the tool.

3. A team is set up and made responsible for managing the entire tool selection and acquisition process.

4. Vendors are invited to participate in the tool selection process, by way of providing more information regarding the tools that they provide.

5. Vendor responses are evaluated with respect to stakeholder expectations and other constraints.

6. Product usage evaluations are conducted.

7. Results are tabulated, documented and presented to all stakeholders on a need-to-know basis.

8. In the final step, the tool is selected and the purchase process begins.

This discussion provides a comprehensive insight into an ideal process for tool selection and acquisition.

A study conducted by Zari et al. [15] concludes that slow performance of web-based applications is said to cost the software industry over $5 billion in lost revenues each year. Internet users typically notice differences in the response time of various different applications. The difference in response times are due to latencies that arise from different sources. Latency introduced by applications running on the requestor system due to various other processes running on that system, is termed as system latency. The latency introduced by the network and the delay in propagating requests and responses to and from the clients and servers is termed as network latency. These are but very few examples of the different kinds of latency that may cause slow performances of web applications. Therefore, it is easy to imagine the amount of savings an organization could make if there could be a way to marry the fast access technologies to the rich feature-set of JIRA. Developing such a tool would most definitely improve operating efficiency at organizations that currently use JIRA and also provide an impetus to organizations that are holding off on using JIRA due to the high time costs associated with using it.
From a usability perspective, there are many arguments why JIRA’s web-based usage model may not be suitable for a large organization that is dynamic in terms of its projects and human resources. In their paper titled *Rapid scalability of complex and dynamic web-based systems: Challenges and recent approaches to mitigation*, Malarvannan and Ramaswamy [16] talk about how web-based systems must be able to scale up rapidly in response to dynamic workloads. They are referring to web companies that provide services, but this argument is also true in the context of a JIRA installation at a large and dynamic organization. A web application typically receives several requests concurrently. The authors are of the view that data storage must be distributed across several geographically diverse locations in order to satisfy user requests in an acceptable way. It is worth investigating whether it is possible to reduce the flux in scalability to some extent by using this suggestion of the authors. The scenario that is the cornerstone of our discussion involves a fair amount of relational database management systems. The authors suggest several methods that may help in the geographical distribution of relational data. They are mentioned here along with a brief explanation.

- Master-Slave replication – the database management system is replicated in such a way that the data to be read is distributed across several slave databases. The reads may happen from any slave database (based on geographical factors) while the writes must all happen on the master database.

- Multi–master replication – this model is similar to the preceding one; the difference being that we now have more than one master database, so that the reads and writes may happen in a geography-specific fashion.

- Synchronous multi-master replication – the data distribution across the master databases happen in a synchronized fashion.

- Asynchronous multi-master replication – the data distribution across the master databases happen in an unsynchronized fashion.

The issues of session workloads and scalability requirement analysis have been studied closely and Goseva-Popstojanova et al. [13] present an empirical study in their paper. The authors’ study of the problem has been structured as follows:

- Characterization of web workload in terms of sessions.
- Characterization of session-based and request-based reliability.
- Identification of invariants of the web workload and reliability.
The authors have presented an empirical analysis of session-based workload and reliability based on actual usage patterns. From their analysis, it is clear that there must be a constant and tight watch on the session workload and availability analysis at every organization that uses JIRA, in order that users may continue to use the web interface provided by JIRA and enjoy high availability and quick access. This again translates into more spending for the organization. This will help overcome some of the adoption challenges associated with a web interface as outlined in [15].

Lam and Maheshwari [17] argue that requirements elicitation time can be reduced by giving a large number of attributes to an issue. The truth in this argument can be easily seen by looking at the information captured when the user provides values for the different attributes when an issue is reported. If the attributes are given values carefully, it is easy to see that most of the information required to start working on the issues is already present along with the issue. This also forms the foundation towards establishing a coherent collaborative environment between the various stakeholders of an issue that has been identified and populated into the project management tool. An issue in JIRA has a large number of attributes, but a downside is that users update many of these attributes constantly during the requirements elicitation process. The gains made by giving a large number of attributes to an issue threaten to be undone by the delays experienced in editing these attributes via the web interface. These arguments further underline the need for a client side application that can provide fast access to JIRA, while supporting all of its many features. The admission of this need is probably best demonstrated by the fact that the developers of JIRA have exposed a SOAP API that is capable of exchanging data with the JIRA installation.

2.3 CANDIDATE TECHNOLOGIES

There are many technologies that can be used to develop a client interface to JIRA. There is a great variety of programming languages, communication technologies and reflection options available to us today. The quest to find the right mix of technologies leads us to further review literature that reflect the research in this area. Adoption of a tool hinges heavily on whether or not the usage of the tool is intuitive. At the very least, the design should be done in such a way that the learning curve is kept as flat as possible [14].
be easily done by keeping the layout of the screens and fields similar to that of JIRA web pages. In some web-applications, the concept of pre-fetching is used to avoid the delay in accessing the web page [16]. This concept is invaluable to us in designing the client interface to JIRA, but cannot be used as proposed by the authors. This is because JIRA is a large and complex system and contains huge amounts of data. Pre-fetching all of the data in JIRA might actually prove counter-productive to our efforts. In order to make use of this idea, we need to exploit the concept of intelligent pre-fetching as described in [18]. We can exploit this concept to our advantage by pre-fetching to the cache all of the user-specific data. Through client-side caching as explained in [16], we can, in effect, replicate locally the user-relevant part of the JIRA installation.

The delay incurred in waiting for web pages to load can be reduced to a great extent, but cannot be altogether eliminated by using the SOAP API to access it. The goal for us, however, is to altogether eliminate the communication overhead incurred in communicating with JIRA, thereby only incurring the cost in displaying the data. In order to do this, we make use of the principle of user-perceived service availability [6]. Caching and pre-fetching user-relevant data from JIRA helps us to take advantage of this principle. It is possible to know in advance a chunk of the data that the user will access in a session. We fetch this data and hold it locally in caches before the user requests the data. In actuality, pre-fetching could happen any time after the tool logs on to the JIRA installation. On request, we do not waste time in communicating with JIRA; instead we just use the data in the local caches. The user perceives that the data is local, and is not aware of how much time was spent in fetching the data from JIRA.
CHAPTER 3

SYSTEM DESIGN OVERVIEW

The review of the literature presented in the previous chapter highlights the inherent problems in the current usage paradigm of JIRA and how those problems may slow down the adoption of the tool in large software development organizations. We also reviewed a few studies that presented possible solutions to overcome these problems, and those solutions form the foundations on which we build the design of this client interface tool.

3.1 DESIGN PHILOSOPHY

As we have seen in previous discussions, users may not be very enthused to use the tool if it is not intuitive to use and the learning curve for users is steep. For this reason, we have to keep the design of the tool as close to JIRA as possible. This we can achieve by keeping the layout of screens and fields just as they are on the JIRA web page.

None of the original functionality of JIRA must be obscured by the tool. In simpler words, the tool must provide a way to perform every function that the user is able to perform on JIRA. The tool has all the functionalities that are available on the JIRA web page and actions may be performed faster than they may be performed on the web page.

The tools and technologies that are used must be platform independent and must ensure that users may be able to use the tool on Windows, Unix and Mac machines with very little or no configuration changes necessary.

The tool is meant to be used in a software development environment, where main memory is always at a premium. More often than not, there would be situations where users run analytical tools on their work products. Therefore, care must be taken to ensure that the tool uses a minimal amount of main memory and the memory and network resource consumption is low and even, rather than spiking on and off sporadically.

3.2 FEATURES REQUIRED IN THE TOOL

Reviewing the literature in this area has given us a good idea of the features that users would like to see in a client interface tool that interfaces with JIRA. It is our aspiration to
include as many of these features as reasonably possible, from a resource consumption and usage standpoint of view. We list those features here:

- Must enable simultaneous access to a group of issues at the same time.
- Must be able to search JIRA and display the current user’s assigned and reported issues by project.
- Must be able to search JIRA and display the current user’s assigned and reported issues that match favorite filters, as designated by the user.
- Must be able to accept a JQL query and display issues that match that query.
- Must support editing the tasks from right within the display.
- Must support creating issues.
- Must support deleting issues.
- Must allow users to track time on issues and publish auto-generated worklogs.
- Must allow users to publish worklogs without tracking time.
- Must allow users to comment on issues.
- Must synchronize the local data cache with the JIRA server at regular intervals.

### 3.3 HIGH LEVEL DESIGN

We now present the architecture of the proposed client interface to JIRA and explain the various components in the architecture bottom-up.

At the lowest layer is the JIRA installation of the organization. JIRA uses its own database and schemas to manage its data; JIRA’s internal data management is beyond the scope of our discussion.

The next layer is the local cache layer. This layer is responsible for scheduling and running data fetch and store operations. The next layer, the data interaction layer, hosts functions that are scheduled to run at user-configurable intervals and perform mass fetches and writes from/to the JIRA system.

The highest layer in our hierarchy is the user interface layer. In this layer are the graphical user interface elements that allow the user to visually request for and view data from JIRA.
3.4 CLASS – LEVEL DESIGN

In this section, we provide an overview of the important classes that encapsulate the functionality of the tool. We will introduce these classes organized by the layer as depicted in Figure 3.1.

![Architecture diagram of the client.](image)

3.4.1 The Presentation Layer

The presentation layer consists of the visual elements that act as an interface between the user and the tool. The user makes use of the screens in this layer to make choices and requests and subsequently view the results of those choices and requests. Below we introduce the classes that are part of the presentation layer. The screens will be developed using the java swing APIs on the Netbeans IDE. See Figure 3.2.

3.4.1.1 MAINWINDOW

This class encapsulates the logic necessary to trigger various operations like login, create an issue, delete an issue, choose columns to view, go offline, view unpublished worklogs and so on. Besides this, the main window also supports the display of issues that are retrieved according to the user’s search criteria.
Figure 3.2. Class diagram of the presentation layer.
3.4.1.2 SEARCHSCREEN

This class encapsulates the logic that helps the user to specify search criteria. The user may enter a JQL query to search for issues.

3.4.1.3 CREATEISSUESCREEN

This class contains visual elements that help the user to specify attributes for a new issue that is being created. The user interface screen closely mimics the create issue page in JIRA. The screen accepts all the inputs from the user, creates the issue on user confirmation and returns the issue key after the issue has been created successfully.

3.4.1.4 EDITISSUESCREEN

This class contains visual elements that help the user to modify the existing attributes for an issue. The current values for each attributes are clearly displayed in this screen. All of JIRA’s permissions are respected by the tool, meaning that attributes that cannot be edited by the user on the JIRA web page cannot be edited via the tool either. Before writing the final changes to JIRA, the tool checks to see whether a concurrent edit has taken place. If so, the user is alerted to that fact, with the final option whether or not to override the most recent change being vested in the user.

3.4.1.5 WORKLOGSCREEN

This class contains visual elements that help the user to create a worklog, adjust time spent on that work item, write referential comments and publish the worklog.

3.4.1.6 COMMENTSCREEN

This class contains visual elements that help the user to create a comment on an issue, restrict the visibility to any of the user groups defined inside JIRA and to then publish the comment.

3.4.2 The Data Interaction Layer

The data interaction layer is responsible for operations such as setting up a SOAP connection session with the JIRA server, maintaining the session, ending the session, fetching data from the JIRA server and writing back data to the JIRA server. The classes in this session also propagate connection and network exceptions to the presentation layer to
ensure that they are handled gracefully, with the involvement of the user if need be. The automatically generated code, explained in Section 3.4.3, may also considered to be part of the data interaction layer.

The main classes in the data interaction layer that we implement, are introduced below.

3.4.2.1 SESSIONMANAGER

The session manager class is responsible for setting up the SOAP session with the JIRA server. The operations of this class are never invoked directly by the presentation layer, but rather by the other class in the data interaction layer. The session manager is responsible for logging in to the JIRA server. When a session is set up, the JIRA server provides an authentication token, which is needed for all future transactions. This token is passed on by the SessionManager to the other class in this layer. This class is also responsible for tearing down the SOAP session when the user has finished working. See Figure 3.3.

![Figure 3.3. Class diagram of the data interaction layer.](image)

3.4.2.2 CLIENTMANAGER

The client manager class activates the SessionManager class to set up a SOAP session with the JIRA server, provides it with a username and password to log on, and retrieves the
authentication token. The authentication token is used by the ClientManager class to perform operations such as searching for and fetching issues by project, filter or JQL query and writing back data to the JIRA server.

### 3.4.3 The Local Cache Layer

The local cache layer contains the classes that implement the scheduling of data fetch and write jobs and also contain the cache objects. The classes that make up this layer are introduced below. See Figure 3.4.

![Class diagram of the local cache layer.](image)

**Figure 3.4. Class diagram of the local cache layer.**

#### 3.4.3.1 TaskScheduler

This class contains the functions that invoke data interaction layer functions at regular intervals to fetch data from the JIRA server and load them into the local cache. Functions also run at regular intervals to write accumulated changes back to the JIRA server at one go.
3.4.3.2 **DataCache**

This class contains the actual cache objects that hold the issue data locally. These caches are filled with all the issue data that pertains to the current user, and hence, is a superset of the results of any request that the user is bound to make. The class also contains the functions that enable writing into and reading from the cache objects.

3.4.4 **JIRA Installation**

This layer refers to the organization’s local JIRA installation. The JIRA installation publish a Web Services Definition Language (WSDL) document, which is a specification of how requests must be made to the JIRA server, and the results that may be expected. By using tools such as Axis-Maven, it is possible to generate Java code that communicates with the JIRA server in a format that it understands. The current user must be registered on the local JIRA installation as a user to be able to use the tool.
CHAPTER 4

SYSTEM IMPLEMENTATION OVERVIEW

JIRA exposes SOAP and XML-RPC APIs for communication, but is said to be more expressive over SOAP. JIRA exposes a Java SOAP API for this purpose, therefore Java turns out to be the path of least resistance when deciding upon an implementation language. The Java swing APIs make it very easy to develop some very sturdy user interface screens. There are also very good code generation and automated test frameworks available in this area.

4.1 DEVELOPMENT TOOLS USED

We have made use of a number of tools to hasten and simplify our software development. Those tools are listed below:

- We use the Eclipse IDE to develop the Java classes.
- The classes for the user interface are developed on the Netbeans IDE.
- To generate Java classes matching the Web Services Definition Language (WSDL) document of JIRA automatically, we use the Axis-Maven plugin system. The SOAP endpoint classes are generated once for the duration of development; they do not change unless the WSDL document changes.
- We use the OpenSymphony caching API to implement caching in our tool. OSCache is an open-source API.
- The test framework is developed using the UISpec4J API. We write automated tests to cover every line and branch of the source code.

4.2 THE PRESENTATION LAYER

The presentation layer consists of all the classes that help the user to make requests such as those to search issues by project, search issues by filter, search issues by a JQL query, create an issue, delete an issue, edit an issue, start, pause and stop time tracking, publish a worklog and comment on an issue. The implementation of those user interface screens are explained in details in the text below.

The main window is the only window that is designed to stay alive and visible from the time the tool is started, until the time the user exits from the tool (see Figure 4.1). The
main window enables the user to launch many operations like searching for issues, creating and editing issues, publishing worklogs and comments, and tracking time on issues.

The main window has one table each to display the names of the project and the favorite filters of the current user. There is also a table to display the tasks and a table to display the sub-tasks. The behaviors built into these tables will be discussed subsequently.

The menubar has four main menus as seen in the Figure 4.2. The four menus are System, Tools, Actions and Help. The menus, the items present in each menu and the operations that they enable are described below.

As seen in Figure 4.3, the System menu has two items within itself.

The Login menu item launches the login dialog when clicked (see Figure 4.4). The user may enter login information like username and password, choose whether to save credentials on the current machine, and then attempt to log on.
The Exit menu item shuts down the tool gracefully, after making sure that unpublished worklogs are persisted on the disk and unsaved edits are written to JIRA.

As seen in Figure 4.5, the Tools menu has three menu items, View Unpublished Worklogs, Select Viewed Attributes and Set Refresh Rate under it.

The Tools menu provides options for the user to View Unpublished Worklogs, Select Viewed Attributes and to Set Refresh Rate. These options are explained below.

- The View Unpublished Worklogs option displays a table that lists all the unpublished worklogs that are present in the system (see Figure 4.6). From this table the user can choose to publish or discard a particular worklog.

- The Select Viewed Attributes option displays a menu where each attribute of the issue are represented as a check box (see Figure 4.7). By toggling the state of the check box between selected and unselected, the user can decide whether or not to see the values of that attribute in the display grid.
The *Set Refresh Rate* option displays a menu where the standard refresh rates are represented as checkboxes (see Figure 4.8). This interval is measured in minutes.

The *Actions* menu provides the user with options to perform a *JQL Search*, *Create Issue* or *Refresh Now* (see Figure 4.9).
Clicking on JQL Search displays a dialog where the user can enter a JQL query, as shown in Figure 4.10.

Selecting the Create Issue option will display a window where the user will be able to enter values for attributes (based on the JIRA permissions for that user). The screen where the user can specify attributes for the new issue, is shown in Figure 4.11.

Selecting the Refresh Now option forces cache data to be refreshed, by invalidating the cached objects and fetching them anew from the JIRA server.
The Help menu leads the user to the documentation of the tool, which consists of an FAQ list, guidelines to using the tool and solutions to frequently encountered problems.

### 4.3 The Data Interaction Layer

The first step in creating the data interaction layer is to implement the classes that are able to communicate with the JIRA server over the SOAP protocol. The list of SOAP requests that are accepted, the arguments that must qualify those requests and the formats of the responses are described in the Web Services Definition Language (WSDL) document published by the JIRA installation. To communicate with the JIRA server over SOAP, we must implement classes according to the WSDL document. This is a complex task since the WSDL is not designed to be read and understood by humans.

However, to our rescue is the Axis plugin provided by the Apache software foundation. This plugin is capable of reading WSDL documents and generating classes that match the SOAP endpoints defined in the document. The axis plugin works on a project object model, which means that it has a single file that specifies the different resources that are required for each code generation/packaging run of the plugin. We generate the required classes using the axis plugin. These generated classes constitute the bulk of the data interaction layer. The other two classes that we implement to complete the data interaction layer are explained below.

The `SessionManager` class is responsible for setting up a SOAP session with the JIRA server and tearing it down when user shuts down the tool or decides to go offline. The outline of the `SessionManager` class is shown in Figure 4.12.

As seen in Figure 4.13, most of the functions that we implement in the data interaction layer are packaged into the `ClientManager` class. This sums up our discussion about the data interaction layer.

### 4.4 The Local Cache Layer

As discussed in preceding sections and chapters, the local cache layer is where data fetched from the JIRA server are persisted, and where the edited changes are written, before they are saved. The local cache layer has been incorporated into the design of the tool to reduce the delays incurred in waiting for issues to be fetched from JIRA. Since we fetch a large number of issues at a time, the delays might be large enough to discourage users from
using the tool. Therefore, we implement the pre-fetching strategy, where we fetch all of the current user’s assigned and reported issues before the user requests for them. This data is held in the local cache layer until the user asks for it. Also, the edits made by the user are not written to JIRA immediately. Data to be written is sorted out based on time-sensitivity. Only time-sensitive data is written to the JIRA server immediately; the other data is held in the local cache layer and then written in one go. This helps us to reduce the overhead of writing to JIRA each time the user makes an edit.
Figure 4.13. Outline of the *ClientManager* class.
public class SOAPClient implements Serializable
{
    /*This function returns the names of the fields of the
     *specified issue that may be edited by the current user
     */
    public HashMap<String, String> getFieldsForEdit(String issueKey)
    {
        RemoteField[] fieldsForEdit =
            jiraSoapService.getFieldsForEdit(token, issueKey);

        return fieldsForEdit;
    }
    /*This function updates the issue specified by the issue
     *parameter with the new values in the updated parameter.
     */
    public void performUpdate(Vector<RemoteFieldValue> updated, String issue)
    {
        RemoteFieldValue[] rfv = new
            RemoteFieldValue[updated.size()];
        jiraSoapService.updateIssue(token, issue,
            updated.toArray(rfv));
    }
    /*This function creates a new issue as represented by the
     *newIssue parameter
     */
    public RemoteIssue createIssue(RemoteIssue newIssue)
    {
        return jiraSoapService.createIssue(token, newIssue);
    }
    /*This function deletes the issue whose key matches the issue
     *parameter
     */
    public void deleteIssue(String issue) throws Exception
    {
        jiraSoapService.deleteIssue(token, issue);
    }
}
public HashMap<String, String> getUserFilterList()
{
    RemoteFilter[] filterList =
        jiraSoapService.getFavouriteFilters(token);
    return filterList;
}

public RemoteIssue[] getTasks(String projectKey)
{
    JqlQueryBuilder queryBuilder =
        JqlQueryBuilder.newBuilder().where()
            .sub().assigneeIsCurrentUser().or()
            .reporterIsCurrentUser().endSub()
            .and().not().issueTypeIsSubtask().and()
            .project(projectKey).endWhere();
    String jqlQuery = queryBuilder.buildQuery().toString();
    return jiraSoapService.getIssuesFromJqlSearch(token, jqlQuery, maxIssues);
}

public synchronized void getSubTasksList(String parentKey)
{
    JqlQueryBuilder queryBuilder =
        JqlQueryBuilder.newBuilder().where()
            .sub().assigneeIsCurrentUser().or()
            .reporterIsCurrentUser().endSub()
            .and().issueTypeIsSubtask().and()
            .issueParent(parentKey).endWhere();
    String jqlQuery = queryBuilder.buildQuery().toString();
    RemoteIssue[] remoteIssuesForSubTask =
        jiraSoapService.getIssuesFromJqlSearch(token, jqlQuery, maxIssues);
    return remoteIssuesForSubTask;
}

public String[][][] getTasksListFromJQLQuery(String jqlQuery)
{
    RemoteIssue[] jqlRemoteIssues =
        jiraSoapService.getIssuesFromJqlSearch(token, jqlQuery, maxIssues);
    return jqlRemoteIssues;
The local cache layer needs functionalities for the fetched issues to be inserted into the cache, cached objects to be retrieved, and the cache to be cleared. All these functionalities can be packaged into a single object. Another functionality that we need is for the cache to be cleared out and repopulated at regular intervals, so that users may not see stale data. For this purpose, we can use timers to schedule jobs that run at regular intervals and fetch issues and
populate them into the cache. The scheduler mechanism and associated methods may be put into a separate class. The classes in the local cache layer are introduced below.

The *CacheManager* class allows the implementation to create a cache object, to put items into the cache object, retrieve it and delete it (see Figure 4.14). Finally, when the cache is no longer needed, (for instance, when the tool is shut down) the implementation can destroy the cache.

```java
public class CacheManager {

    /* This method creates a new cache object with the name as specified by the 'cacheName' parameter */
    public GeneralCacheAdministration createCache(File propertyFile) {
        return new GeneralCacheAdministration(propertyFile);
    }

    /* This method inserts the item into the cache with the given key */
    public void putItemInCache(GeneralCacheAdministration cache, String key, Object item) {
        cache.putInCache(key, item);
    }

    /* This method returns the cached object stored in cache with the given 'key' */
    public Object getItem(GeneralCacheAdministration cache, String key) {
        return cache.getFromCache(key);
    }
}
```

Figure 4.14. Outline of the *CacheManager* class.
The \textit{JobScheduler} class defines a timer for each task that needs to be run at pre-specified intervals. Each timer is spawned as a separate thread to facilitate concurrent and non-intrusive execution. The outline of the \textit{JobScheduler} class is shown subsequently in Figure 4.15.

\section*{4.5 The JIRA Layer}

JIRA is available in various configurations from its makers, Atlassian Software Systems [19]. JIRA may be run locally on a server at the user organization, or hosted at Atlassian. There are different pricing catalogs for commercial organizations, academic organization and not-for-profit organizations.

For developing our SOAP client and demonstration purposes, we use a local JIRA installation (running on the local machine) supporting up to 10 users.
Figure 4.15. Outline of the JobScheduler class.
public class JobScheduler {
    java.util.Timer projectTimer;
    java.util.Timer filterTimer;

    public RefreshCache(ClientManager client, long interval) {
        clientManager = client;
        refreshRate = interval;
    }

    public void startRefresh() throws Exception {
        if (null != filterTimer) {
            filterTimer.cancel();
            filterTimer.purge();
        }
        filterTimer = new Timer();

        if (null != projectTimer) {
            projectTimer.cancel();
            projectTimer.purge();
        }
        projectTimer = new Timer();
        filterTimer.scheduleAtFixedRate(new PrefetchFilterIssues(
            clientManager), 0, refreshRate);
        projectTimer.scheduleAtFixedRate(new PrefetchProjectIssues(
            clientManager), 0, 120000);
    }
}
class PrefetchFilterIssues extends TimerTask implements Runnable {
    public PrefetchFilterIssues(ClientManager client) {
        clientManager = client;
    }
    public void run() {
        try {
            client.prefetchFilterIssues();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    ClientManager clientManager;
}

class PrefetchProjectIssues extends TimerTask implements Runnable {
    public PrefetchProjectIssues(ClientManager client) {
        clientManager = client;
    }
    public void run() {
        try {
            client.prefetchProjectIssues();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    ClientManager clientManager;
}
CHAPTER 5

THE TEST FRAMEWORK

We have designed a comprehensive test framework to make sure that the tool satisfies all the aspirations of the user that we discovered during the literature survey phase. In the sections below, we discuss the rationale for developing a test framework, the design methodologies that we follow and a discussion about the implementation of the test framework.

5.1 RATIONALE FOR DEVELOPING A TEST FRAMEWORK

Software testing has taken on more and more importance as software systems have grown in size and complexity. Gone are the days when software was shipped out to users after little more than a few cursory unit tests being run on them by none other than their developers. Software development organizations have realized the importance of testing, identifying bugs or design flaws and fixing them before the system is shipped out into the market. The cost of identifying and fixing a bug in the field is many times more than the cost of identifying and fixing them while still in the lab. Fuelled by this philosophy, software test technology has been keeping pace with the developments in software design and development technology.

The SOAP client for JIRA developed by us has a huge variety of features. Many of these features are inter-dependent, meaning that any bug fixes or changes in one feature may easily propagate into others. The reverse is also true, in that a bug introduced in one feature may easily find its way into other features. Thus, it is not only enough if we test the tool once before shipping to a customer. Comprehensive tests should be run each time there is a small bug fix or a design patch. The presence of a graphical user interface means that we have well over a hundred test cases in order to ensure complete code coverage. Running all the test cases each time we make a bug fix or implement a design modification becomes too much of a hassle. In order to avoid the hassle of manually running the test cases by launching
operations through the user interface, it would be nice if we could have an automated test suite, which ensures that we could run through all the test cases at once.

5.2 Design Methodologies

From our discussion of test framework design philosophies in the preceding section, our needs and expectations from a test framework may be distilled down as listed below.

- We need a software testing framework to ensure that all the functionalities outlined in the design document are certified to be working.
- We must be able to test elements of the graphical user interface as well as core feature functions.
- Tests must be designed and developed in a way that they may be run repeatedly, without resorting to the graphical user interface of the client tool under development.
- It must be possible to activate all the tests by launching a single program.
- It must be possible to determine the percentage of code covered by the tests, and this determination must be automated and fast as well.

In developing the SOAP client for JIRA, we have used the extreme programming development paradigm. In this paradigm, immediately after requirements are collected (as in Chapter 2) and the design phase is concluded, we embark on developing test cases rather than developing the code. We write unit tests for each designed functionality. By default, the tests all fail at this stage, since no code exists to satisfy that functionality. Once unit tests are developed to cover every designed functionality, we start writing code to pass the developed unit tests. As new features are added, we follow the same procedure of writing unit tests first and subsequently developing the code to implement the functionality. This helps us in our endeavor to develop a test framework that has 100% function coverage.

5.3 Test Framework Technologies

There are several testing tools available to the software developer today. Keeping in mind the size and complexity of our tool, and the development framework that we have, we determine that an open source tool that is capable of testing object-oriented software in general, and Java code in particular will fit our needs very nicely. Following the extreme programming technique of developing unit tests before implementing the features helps us to test our system/feature design. Most significant design flaws will be identified by these unit tests.
JUnit [20] is an open-source test framework that has been very popular in the Java software development community. We use JUnit to develop unit tests for our feature implementations in the data interaction and local cache layers as described in Chapter 3.

Aside from the feature implementations, we also have a significant amount of logic built in to our display framework, which resides in the presentation layer. The display framework has a large number of java swing and awt components that may not be tested by the JUnit framework. In order to test these components and their working, we need a framework that specializes in testing java graphical user interface components. The answer to our needs is UISpec4J [21], which is built on top of JUnit and specializes in test java graphical user interfaces. The advantage of using UISpec4J in our project is that it integrates seamlessly with the test suite that we develop for testing the feature implementation, using JUnit.

Following the extreme programming practice ensures that our test framework covers 100% of the feature implementation. However, unless we make sure that every branch of code in our software has been exercised during the testing process, we may not be sure that we have identified all potential bugs and design flaws. In principle, we strive for 100% code coverage, although in reality, the expected code coverage is below 100%. For example, the Federal Aviation Administration has laid down guidelines for the expected code coverage expected for avionics software to make them eligible for FAA certification. In order to measure the code coverage provided by our test framework, we use EclEmma [22].

5.4 TEST FRAMEWORK IMPLEMENTATION

Developing unit tests for the feature implementation and display logic are very similar to each other. To get started with writing test cases, we first have to download the test libraries from the JUnit and UISpec4J sites as discussed in [20] and [21]. Once we have linked them to our project development environment in the Eclipse IDE, we may use these libraries to develop unit tests. We write unit tests to cover each feature implementation using JUnit. With UISpec4J, it is possible to obtain handles on windows, buttons, textfields and the like. We may use these handles to trigger and populate values into the different components of the display framework. We develop tests to cover every graphical user interface using the UISpec4J library.
Figure 5.1 shows the outline of an example unit test developed for the login functionality. This example is only a partial unit test.

```java
public class ClientManagerTest extends TestCase
{
    public void loginTest()
    {
        ClientManager clientManagerObject = new
            ClientManager("user", "password", null);
        String token = clientManagerObject.login();
        assertFalse(null == token && token.isEmpty());
    }
}
```

**Figure 5.1. Outline of unit test for login functionality.**

Figure 5.2 shows the outline of an example unit test developed to test the graphical user interface elements of the login dialog. The example unit test is only a partial one.

To measure the percentage of code covered by our test framework, we download and install the EclEmma tool into the Eclipse IDE as explained in [22]. After the IDE is restarted, we have a new program execution mode available. This mode is called *Coverage mode*. JUnit (and hence, UISpec4J) classes may be launched in the coverage mode. At the end of execution, the tool provides visual and textual data about the percentage of data covered by the test cases that were run in coverage mode. See Figure 5.3.

This concludes our discussion of the design and implementation of our test framework.
```java
public class LoginTest extends UISpecTestCase
{
    Window mainWindow;
    Window loginWindow;
    Window searchWindow;
    Window exceptionPane;

    public void testLoginSuccessUnsaveCreds()
    {
        setAdapter(new MainClassAdapter(LaunchClient.class, "testLoginMenu"));

        mainWindow = getMainWindow();

        loginWindow = WindowInterceptor.run(mainWindow.getMenuBar().getMenu("Start"),
                getSubMenu("Login").triggerClick());

        assertTrue(loginWindow.titleContains("Login"));
        Component[] components = loginWindow.getSwingComponents(JTextField.class);

        for(int i = 0; i < components.length; i++)
        {
            if(components[i].getClass().getSimpleName().
                    equalsIgnoreCase("jtextfield")
                ((JTextField)components[i]).setText("user");
            else ((JTextField)components[i]).setText("passwd");
        }

        searchWindow = WindowInterceptor.run(loginWindow,
                getButton("Login").triggerClick());
        assertTrue(searchWindow.titleContains("Search"));
        File file = new File("./config.jsoc");
        assertFalse(file.exists());
    }
```

Figure 5.2. Outline of unit test for login screen.
Figure 5.3. Output of the code coverage tool.
CHAPTER 6

SUMMARY, CONCLUSIONS AND FUTURE WORK

In Chapter 2 on a study of the relevant literature in this area, we have distilled a list of expectations that users hold from a SOAP client interface tool, as identified by several reputable studies. In Chapter 3 on system design overview, we have set the list of expectations from Chapter 2 as implementation goals for the SOAP client for JIRA that we developed. In Chapter 4 on system implementation overview, we have discussed the technologies and methodologies involved in developing a SOAP client for JIRA. In Chapter 5, we have described the philosophies, methodologies and technologies that go into implementing the SOAP client interface for JIRA.

6.1 SUMMARY OF ACCOMPLISHMENTS

We are pleased to report that most of the goals that we set out for development have been achieved. We have been able to develop a client interface tool that communicates with JIRA over the SOAP protocol. We have been able to implement all of the features that ranked high on users’ expectations from a tool of this genre, according to several reputable studies in this area. The design of the tool was geared towards the extreme programming paradigm, where unit tests for each feature were developed as soon as the design was completed. This helped us to identify many design flaws at the outset, and tweak the design as needed. The tool was developed in the agile model, with each feature being developed and the tool being tested in its entirety before embarking on development for the next feature. We have also been successful in developing a comprehensive test-bed for the development framework. This not only saved us valuable time in terms of testing during the iterative project development phases, but also guaranteed functioning of all existing features prior to a feature release. We also measure the percentage of code covered by the tests, each time a new test is added; this ensures the efficacy of the developed tests. In addition, the test-bed
that we have created is open to enhancement as new features are added in the future. We have used open source tools to keep things simple.

### 6.2 Conclusions

In conclusion, we have developed a SOAP client interface for JIRA that meets the most significant expectations that users harbor from such a tool. By using the client interface tool developed by us, users may completely bypass the JIRA web version since we have been successful in replicating all of JIRA’s features in our tool. As users start to get familiar with the tool and use it for more and more day-to-day activities on JIRA, they will see that they are saving a lot of time that would have otherwise been wasted in waiting for the various web pages to load and refresh. The savings will be especially noticeable in the case of managers and leads who need to update several issues at once. Managers and leads may now use a fraction of their time towards JIRA-related activities and more towards other activities that require their time and attention. When used across a large organization, the savings will be enormous. The architecture, design and the development/test frameworks of the tool have been developed in such a way that the further feature-additions and other development may be accomplished with the minimum effort.

### 6.3 Future Work

From our discussions in the preceding chapters and sections, we have seen that the SOAP client interface for JIRA, developed by us, satisfies a majority of the expectations that user harbor against a project management tool of this genre. However, we have identified a set of tasks that may further the usability and efficiency of this tool. Those tasks are listed below and are open to research.

- It will be interesting to re-write the entire application in a language that does not need its instructions to be interpreted to machine code each time it is run. This may increase the speed of the tool and make it more suitable for mission critical scenarios.
- The *Offline* mode of the tool was not a feature that a majority of users listed high on their expectations, but it would be a good feature to try and test for feasibility.
- The graphical user interface that we have developed is built in Java. This was developed more as a proof-of-concept; it will be good if the user interface could be re-written in a sprightly technology such as Adobe Flex [23] that communicate well in SOAP.
It is our expectation that the tool will be widely across the length and breadth of organizations that rely on JIRA for issue and bug tracking, and project management. JIRA is a tool widely respected in the project management circles, and the client interface tool developed by us helps to unleash its full power.
BIBLIOGRAPHY


APPENDIX

USER MANUAL FOR JIRA DESKTOP CLIENT
USER MANUAL FOR JIRA DESKTOP CLIENT

This appendix is meant as a primer for users who wish to use the desktop client as an interface to JIRA. All of the features present in the desktop client are exercised in this primer. The primer presents a general usage pattern; the user is in no way constrained to follow this pattern.

A.1 CONFIGURING THE CLIENT

The client has been designed in such a way that configurations made once will be persisted across sessions, until the user wishes to modify them. In the subsequent text, we describe how the different configuration choices may be made.

A.1.1 Setting the Refresh Rate

The refresh rate is the rate at which issue data fetched from the JIRA server is invalidated and fetched again. The refresh rate may be set by selecting Tools->Set Refresh Rate from the menubar of the main window and choosing one of the displayed options as shown in Figure A.1.

![Figure A.1. Setting the refresh rate.](image)

A.1.2 Choosing Issue Attributes for Display

The user may choose the attributes for display by choosing Tools->Select viewed attributes and then selecting the required attributes from the options shown, as shown in Figure A.2.

Now that the client has been configured, we will demonstrate how a user may log in to the tool and view issues (tasks and sub-tasks) by project, by filter or by a simple JQL query.
A.2 Logging in to JIRA

The user may log in by selecting Start->Log in from the menubar of the main window. The login dialog pops up, where the user must enter the username and password assigned by the JIRA administrator. The user may decide to store his credentials on the system. When the credentials have been entered, the Login button must be clicked to complete the login process. See Figure A.3.

Figure A.3. Login to JIRA.

When the user has been logged in successfully, the login dialog will disappear and the Projects and Filters tables will be populated. The Projects table will be populated with all
projects in which the current user has reported or assigned issues, and the Filters table will be populated with all the current user’s favorite filters. See Figure A.4.

Figure A.4. On successful login.

A.3 Viewing Issues

Sections A.3.1 through A.3.3 introduce the various task viewing capabilities that we have built into the tool.

A.3.1 Viewing tasks

The user may choose to view tasks by project or by filter. Clicking on any row in the Projects table populates the Tasks table with issues reported by or assigned to the current user, in that project. See Figure A.5.

Figure A.5. Viewing tasks by project.

To view tasks by filter, simply click on a row in the Filters table. All tasks matching that filter will be displayed in the Tasks table in Figure A.6.

To view issues matching a JQL query, select Actions-> JQL Search from the menubar of the main window. On entering the JQL query in the resulting dialog, issues matching that query will be displayed in the Tasks and Sub-Tasks tables. See Figure A.7.
A.3.2 Viewing Sub-Tasks

To view sub-tasks, simply click on a row in the *Tasks* table. If there are any sub-tasks under that issue, reported by or assigned to the current issue, they will be displayed in the *Sub-Tasks* table shown in Figure A.8.

A.3.3 Creating a Task

Users may create a task by selecting *Actions->Create issue* from the menubar of the main window. A new window pops up. Initially all fields except the *Project* are empty, and the user will not be able to make choices. Once the project is selected, the other fields will be populated with project-specific values that the user may choose from. Finally, on clicking *Create Issue*, the issue will be created. A confirmation dialog reports the success of the operation, along with the key of the newly-created issue. See Figure A.9.

A.4 MODIFYING ISSUES

Sections A.4.1 through A.4.4 describe the various actions that result in the modification of issue(s) via the tool.
Figure A.8. Viewing sub-tasks.

Figure A.9. Creating an issue.
A.4.1 Editing an Issue

Users may edit any issue in the Tasks or Sub-Tasks table by performing a right-click on the row for that issue and selecting Edit Issue from the context menu that pops up. A new screen will be displayed with the fields for all attributes showing their current values. The user may make the required changes, and on clicking Update Issue, the changes are written back to JIRA. See Figure A.10.

![Editing an Issue](image)

Figure A.10. Editing an issue.

An alternate way of editing issues is from the display tables; to launch this operation, simply double-click on the cell for which the value has to be modified. If the user has permissions to modify this attribute, a popup element will present the user with ways to do this. See Figure A.11.

A.4.2 Deleting an Issue

To delete an issue, simply right click on the row representing that issue in the Tasks or Sub-Tasks and choose Delete Issue from the context menu that pops up. Once the user confirms the intention to delete the issue, if the user has permissions to delete the issue, it will be deleted. See Figure A.12.
Figure A.11. Editing an issue in-place.

Figure A.12. Deleting an issue.
A.4.3 Logging Work on an Issue

To log work on an issue, right-click on the row representing that issue in the *Tasks* or *Sub-Tasks* table, and select *Log your work* from the context menu that pops up. A new screen will be displayed, where the user will be allowed to set values for the worklog, and on clicking *Log*, the worklog will be published. See Figure A.13.

![Logging work on issues](image)

Figure A.13. Logging work on issues.

A.4.4 Commenting on an Issue

To comment on an issue, right-click on the row representing that issue in the *Tasks* or *Sub-Tasks* table, and select *Add Comment* from the context menu that pops up. A new screen will be displayed, where the user will be allowed to set values for the comment, and on clicking *Add*, the comment will be published. See Figure A.14.

![Commenting on issues](image)

Figure A.14. Commenting on issues.

A.5 AUTOMATED TIME-TRACKING

A.5.1 introduces the behavior of the automated time tracking feature of the tool. A.5.2 explains how the user may force a cache refresh.
A.5.1 Tracking Time on an Issue

To start tracking time on an issue, right-click on the row representing that issue in the Tasks or Sub-Tasks table and select Start time tracking. A dialog pops up, where the user has to enter description of the activity. One the description is entered, the tool starts tracking the time.

Time tracking is designed to pause automatically when four hours of work have elapsed. The user may also pause time tracking by choosing Pause time tracking from the context menu of that issue row.

Time tracking is designed to stop automatically when eight hours of work have elapsed. The user may also stop time tracking by choosing Stop time tracking from the context menu of the issue row.

When time tracking is stopped, the user is presented with options to publish the generated worklog immediately, or retain it in the system for publishing later. See Figures A.15 and A.16.
Unpublished worklogs may be viewed by selecting Tools->View unpublished worklogs from the menubar of the main window. The user may choose to publish or discard these worklogs by right-clicking on the rows representing the worklogs, and choosing Publish worklog or Discard worklog, as appropriate. See Figure A.17.

A.5.2 Refreshing the Local Cache

The local cache is designed to refresh itself as per the refresh rate configured by the user, as shown above. In addition to this, the user may also force a cache refresh by selecting Actions->Refresh Now from the menubar of the main window.
Figure A.17. Viewing unpublished worklogs.