Integration of Agile Planner into IBM/Rational Jazz Development Environment

Kai Nehring

Acknowledgements

I d like to thank several people who supported me during my work. First, Prof. Dr. Schmucker-Schend from University of Applied Sciences in Mannheim and Prof. Dr. Maurer from the University of Calgary for supervising my work and allow me to carry out my Master-Thesis in Calgary as well as Prof. Dr. Knauber from University of Applied Sciences in Mannheim for being my co-supervisor.

Last, but not least, the members of the ASE workgroup and Brady Lill for their support and patience.

Declaration

Ich versichere, dass ich diese Master-Thesis selbststandig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Diese Arbeit hat in dieser oder ahnlicher Form keiner anderen Prufungsbehorde vorgelegen.

Calgary, 28. August 2008 _____

Contents

List of Tables

List of Figures

Listings

Abstract

More and more development teams are working in a distributed manner. Even though such teams are distributed, they still must work on the same project. Distribution makes project planning very difficult because it is neither economic nor realistic to bring all of the teams together for short, but necessary meetings.

Every project is based on decisions that influence the progression of the project directly. These decisions are based on the information that is available to a developer or project manager. Reporting tools are therefore very important to keep track of the project s status.

Virtually no tool supports distributed synchronous planning and reporting capabilities that are necessary for effective project planning in distributed teams. This thesis discusses a solution to this problem. That is, how to create a toolset which allows distributed synchronous planning and the ability to create reports.

The focus of this thesis is set on the integration of Agile Planner with IBM/Rational Jazz development environment.

Chapter 1

Introduction

1.1 About this Thesis

Project management is important in software development projects in order to prevent them from losing their focus. Project management is comprised of two important tasks: *project planning* and *progress tracking and reporting*. This thesis sets its focus on progress tracking and reporting.

1.2 Motivation and Goals

Progress tracking and reporting is a very important task in project management. At the same time, more and more projects are carried out in a distributed manner. The demand for software tools which cover both, distributed planning and progress tracking and reporting is high. The goals of this thesis are

1. To create an environment which offers distributed synchronous planning and the ability to create reports
2. To discuss an approach which allows Agile Planner to work with Jazz
3. Realization of the approach to allow Agile Planner to work with Jazz
4. Future aspects how can Agile Planner/Jazz be improved

1.2.1 Planning environment

Although a variety of planning tools exist, non of them supports distributed synchronous planning and the ability to track progress and create reports. The goal of this thesis is to create such an environment.

1.2.2 Approach

Chapter 3 discusses how it can be achieved that Agile Planner is able to work with Jazz. It explains how synchronization works as well as requirements. It finally discusses several ways how both tools could work together.

1.2.3 Realization and future aspects

Chapter 4 and chapter 5 explain how the adapter is actually built and what the problems were.

1.2.4 Future aspects

Chapter 6 describes useful information for an evaluation in the event of a possible migration to Jazz/Agile Planner. Chapter 7 introduces improvements to Agile Planner s/Jazz functionality.

Chapter 2

Related Work

2.1 Background

2.1.1 Story Card Metaphor

One task per **story card**

Agile approaches often use the so called *story card metaphor* to define functionality that should be implemented[1]. The actual functionality is broken down into small *user stories* that explain what the system is supposed to do. These user stories are then written on index cards. Since index cards are limited in space, the user story must be concise to fit on one side of the card as shown in Figure 2.1. If a user story is too long and therefore needs to much explanation, the user story will be divided[2] into further stories. Furthermore, each story card is comprised of a meaningful title and often time estimations, that is how much time that particular user story approximately takes to carry out.

2.1.2 Distributed Planning

Participants can **join** meetings from **different locations**

More and more projects are carried out in a distributed manner. Big development groups are divided into smaller groups which are often in charge of only a single duty. Many of these groups are specialized to a single sector, e.g. databases, user interfaces, and so forth. Larger companies often build competence centres for these teams and these centres are rarely housed in a single building.

Even though such teams are distributed, they still must work on the same project. Distribution makes project planning very difficult because it is neither economic nor realistic to bring all of the teams together for short, but necessary meetings. One solution for this problem is video conferencing which connects all of the teams together during a planning meeting. However, one problem still remains: who keeps track of all the decisions?

[1] In fact, everything, even problem reports are written down on story cards
[2] Sometimes called *triaged*

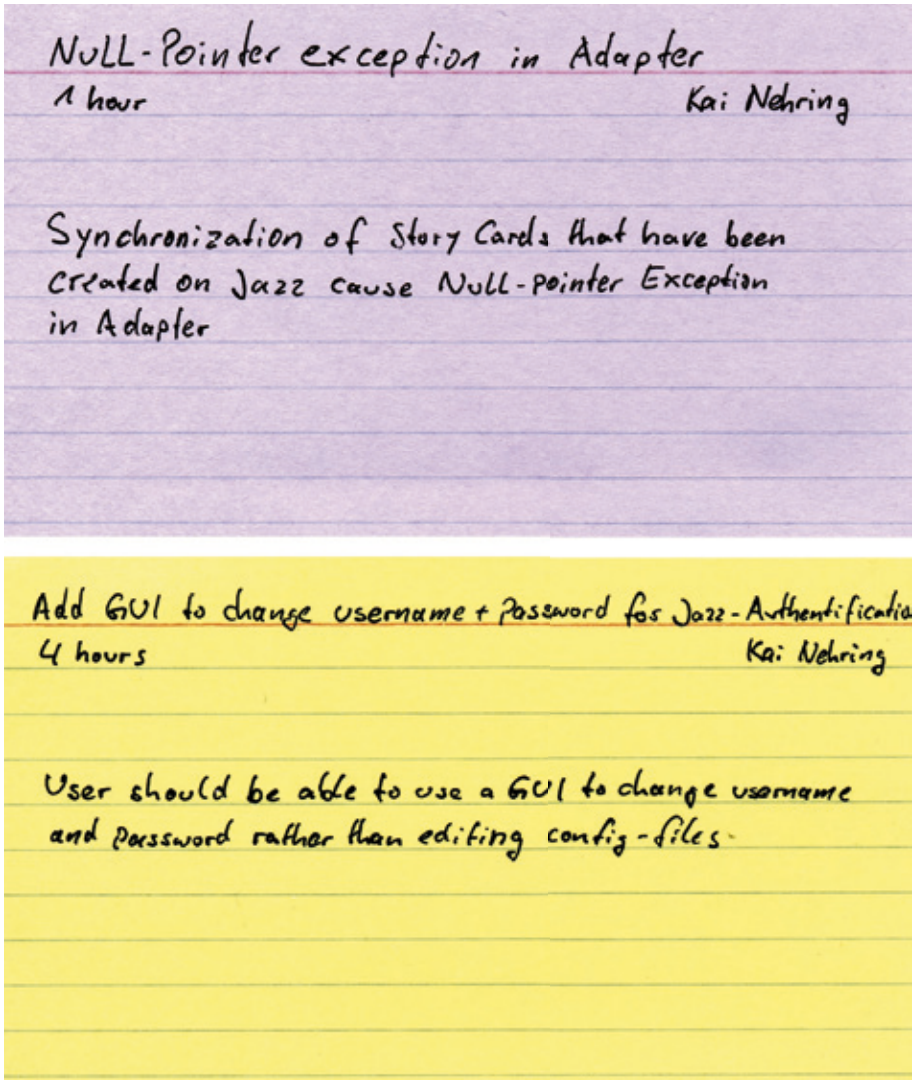

Figure 2.1: Story Cards in an Agile Environment. The violet story card shows a failure. The yellow story card shows a new feature. Both cards include an estimated time to work on these tasks and the responsible developer.

2.1.3 Synchronous Planning

Synchronous planning meetings are suitable for distributed teams. Each team is able to keep track of any given event, e.g. a new feature will be generated. Modern software solutions allow realtime updates of a workspace which holds planning artifacts such as story cards; refer to 2.1.1. Synchronous means that all teams will experience each update, such as adding text to an artifact or move a story card on the desktop within a reasonable amount of time.

Participants experience **real time updates**

2.2 Progress Tracking and Reports

Reporting and therefore progress tracking is important to determine the project s health. This information is also required to plan future steps which keep the project alive.

> During project execution, task progress and task deadlines need to be monitored while any issues and problems that arise need to be addressed. When actions are taken to address specific problems, those corrective actions need to be accounted for as well. Project progress tracking ensures that misunderstanding and confusion over project process is reduced or eliminated and that there are smooth transitions when tasks are handed off among team members. Project progress tracking further ensures that project changes are monitored and the impact of change is well understood and anticipated.[CBC+06]

2.2.1 Progress Tracking

2.2.1.1 Why progress tracking?

> The overall objective of project progress tracking is to increase awareness and visibility of a project s process, which in turn increases the likelihood of project success. Insufficient project progress tracking can result in the entire process being treated as a black box. The black box phenomenon can cause a number of potential problems to arise. For example, project members may engage in unproductive efforts or lose track of a critical change in the customer s requirements, project development approach, or individual task assignments. Without closely monitoring such changes, it is virtually impossible for project members to estimate change related risks and to identify alternatives that can be employed to mitigate such risks in a timely fashion. As a result, projects are frequently subject to cost over-runs and cancellation.[CBC+06]

Make the project s **progress visible**

2.2.1.2 How to accomplish progress tracking?

> One technique for task progress tracking is to use technology to maintain a task list for team members.[CBC+06]

Tools offer **task-lists** including their current **status**

Specialized tools offer lists of tasks. Each developer can see at least his or her tasks often combined with priority and due dates. The developer who is responsible for a specific task marks it *in-progress* if it is executed or *completed* if it is finished. Such to-do lists are better suited for self-monitoring than plain text descriptions. Furthermore, project managers can easily see if a task has been finished and has the ability to change the priority if required.

2.2.2 Reports

Once progress information have been acquired it must be processed into a meaningful form. To do this a report generator must perform multiple steps[3], such as

- Sorting data
- Manipulate[4] data
- Create charts
- Deliver report, e.g. as PDF

Custom representations of project information

Reports are created for a particular target group. Each group requires its own format depending on the intended use, e.g. for a presentation for a customer. That also means that one set of data can be represented in different ways. A report generator regards this by using templates. A template defines how a report should be created. It contains information about

- Header, footer
- What data should be used, i.e. which column in a table
- Which type of chart should be created, if any
- Where should the data be placed

2.3 Agile Planner

Agile Planner is a project planning tool that uses story card metaphor to perform planning tasks, illustrated in Figure 2.2. It also supports distributed synchronous planning.

Although Agile Planner is a adequate for planning sessions, it lacks in few important features, such as *Reporting*.

[3] Not necessarily all steps are required to create a report
[4] For example, convert hours in days

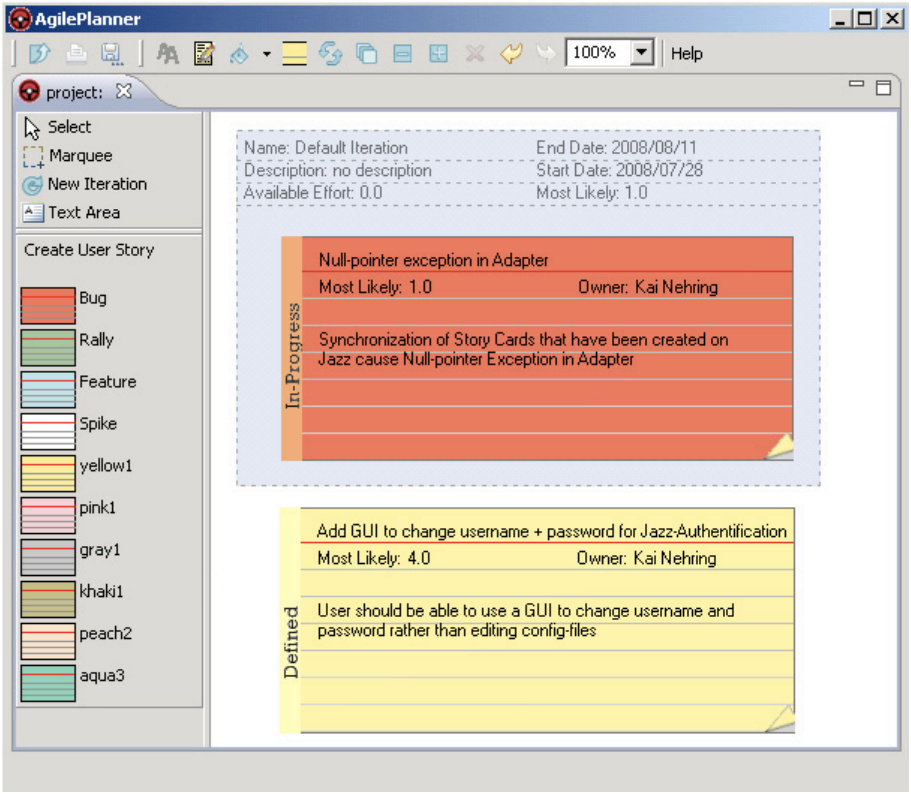

Figure 2.2: Story Cards and Iteration in Agile Planner. The red story card is part of the current iteration. The yellow story card is in the backlog task is defined but not executed in the current iteration.

2.3.1 Planning history

Agile Planner s capabilities to preserve the planning history are very limited. The workspace becomes unreadable if all previous iterations and story cards are kept. Even though it would be possible to keep previous versions of a project plan by storing them in a repository, it is difficult to handle and error prone.

2.3.2 Reporting

Agile Planner offers no reporting capabilities.

2.3.3 Support for different input devices

Agile Planner supports multiple input devices such as

- Desktop and notebook computers
- Digital Tables
- PDAs

It is possible to use different input devices simultaneously during a planning meeting. For example, a development team uses a digital table to perform a planning meeting. A developer can join this meeting on his or her laptop while he or she is out of the office.

2.4 Jazz

IBM/Rational Jazz is a complete development environment that offers project planning, reporting capabilities, and many other features.

2.4.1 Work Items

Work items represent tasks similar to story cards described in section 2.1.1. However, unlike Agile Planner, Jazz offers no graphical representation of work items.

2.4.2 Reporting

Every project is based on decisions which influence the progression of the project directly. These decisions are based on the information that is available to a developer or project manager. Jazz offers a data warehouse that stores historical data about the project. The developer can then use BIRT to access this information and create a report based on a report template. This template describes not only what information should be displayed, it also describes the way in which the information will be shown, e.g. as a table, a chart, ...

Reports are generated based on *information* from the **data warehouse**

Although Jazz offers the possibility to create customized reports, it also comes with few of the most often used report templates, such as:

- How many work items are open?
- How many bugs contain a specific component?
- How much time has been spent on a component?
- ...

2.5 Adding support for reporting tools to Agile Planner

The ability to create reports is very important for effective project planning. Although there is a need to add reporting capabilities to Agile Planner it is not wise to reinvent everything. Instead, existing reporting tools could be used to fulfill user s needs.

There are many tools available which offer more or less advanced reporting capabilities, such as BIRT and Jazz. The conclusion is therefore to add the ability to use these tools with Agile Planner.

Create an **interface** between Agile Planner and Jazz

A reporting tool needs access to all current planning information and to previous planning information to create meaningful reports. Agile Planner s ability to store information about previous iterations is very limited. Therefore, tools such as BIRT, which simple generates a report out of a data set, can not be used. Jazz on the other hand uses its own repository to store all planning information and is therefore the best choice.

Table 2.1 compares important features of Agile Planner and Jazz. The table also shows that Jazz offers exactly the features that Agile Planner needs.

2.5.0.1 Synchronous Planning

Every user who participates on a planning meeting will see updates in real time.

2.5.0.2 Distributed Planning

The planning information will be stored on a dedicated server. All team member need access to this server in order to retrieve or update planning information.

2.5.0.3 Task based planning

Planning is performed in tasks as explained in story card metaphor.

2.5.0.4 Reporting capabilities

The ability to create reports out of (planning)information. It is desirable that the tool is able to create custom reports but it is not a requirement.

Table 2.1: Feature Overview: Agile Planner and Jazz

Feature	Agile Planner	Jazz
Synchronous Planning	X	-
Distributed Planning	X	X
Task Based Planning	X	X
Reporting Capabilities	-	X
Store Planning History	-	X
Extendable	X	X

2.5.0.5 Store planing history

Since some reports require a complete planning history, the tool must preserve it and be able to hand it over to the reporting tool.

2.5.0.6 Extendable

The tool must be extendable in order to add or use its functionality. This can be through public interfaces or though the source code of the tool itself.

2.5.1 Advantages of Agile Planner - Jazz integration

Jazz supports work items which is basically just another version of story cards. Jazz lacks in the planning process itself that means it doesn t support synchronous planning meetings. Distributed teams especially need this functionality to make planning meetings easier and less error prone. Agile Planner offers not only distributed synchronous planning but also a variety of input devices as explained in Section 2.3.3. Therefore, Jazz is not capable of replacing Agile Planner despite of its enormous set of functionality.

Use the best **features** from **both** systems

Chapter 3

Approach

To take advantage of Agile Planners sophisticated planning features and Jazz advanced tracking and reporting capabilities, both servers must be able to synchronize with each other. The planning information must be transferred from Agile Planner Server to Jazz Server and vice versa. This must happen before and after a planning meeting to keep both systems synchronized[1].

3.1 Synchronous and asynchronous synchronization

The Agile Planner Server is responsible for persisting a projects planning data. The server runs on a local system or on a remote system and is necessary for distributed planning. All clients connect to the server which is often a dedicated server. Agile Planner uses two different methods to synchronize with either connected clients or remote services, such as Rally or Jazz in the future:

- Synchronous
- Asynchronous

3.1.1 Synchronous synchronization

During a planning meeting, Agile Planner uses synchronous synchronization to update all clients which are connected to the server if one or more client perform a change on the current plan. Clients send their changes to the server which propagates to all other clients, illustrated in Figure 3.1. Since updates will be performed in real time they are *synchronous*.

Send **real time** updates to all the **clients**

[1]The first synchronization is necessary to use the latest planning information during the planning meeting, the second to update Jazz' database after the planning meeting

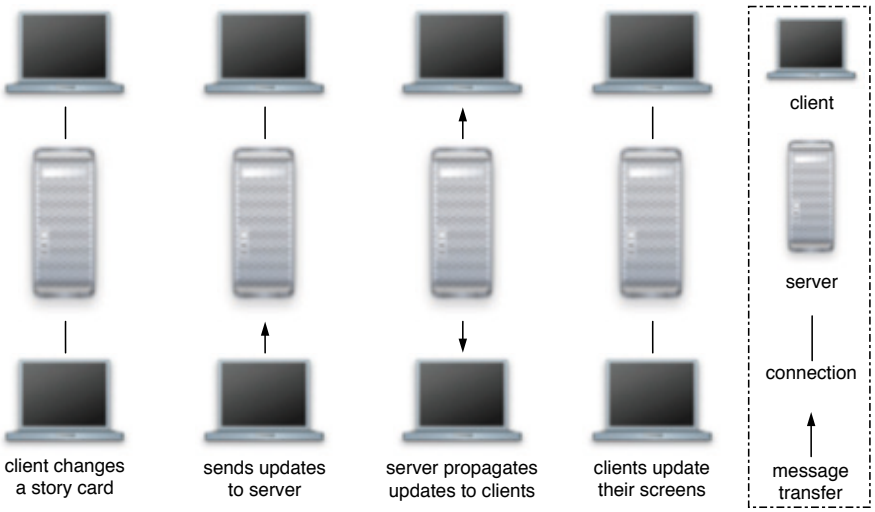

Figure 3.1: Synchronous Planning. The client updates planning information and sends changes to the server. The server propagates the updates to all clients, which update their screens.

3.1.2 Asynchronous synchronization

It is not necessary to update remote services in real time during a planning meeting. In fact, real time updates rely on fast network connections. Remote services which are connected through the internet often use a slow connection, potentially a slow dial up modem connection. Hence, the emerging delay would slow down the planning meeting because all users have to wait until all updates have been performed.

Agile Planner Server uses asynchronous synchronization to synchronize planning data with other services, such as Rally or Jazz in the future. Unlike with synchronous synchronization mode, the server connects the target system on demand. For example, no network connection must be maintained during the planning meeting.

Establish **connections** to external services **on demand**

The user triggers the synchronization process on the client which sends a message to the server. The server connects the remote service and synchronizes its data with the remote service. If the update has been performed successfully, Agile Planner server disconnects itself from the remote service and sends the updated project plan to all connected clients which update their workspaces, e.g. display the updated planning information, shown in Figure 3.2

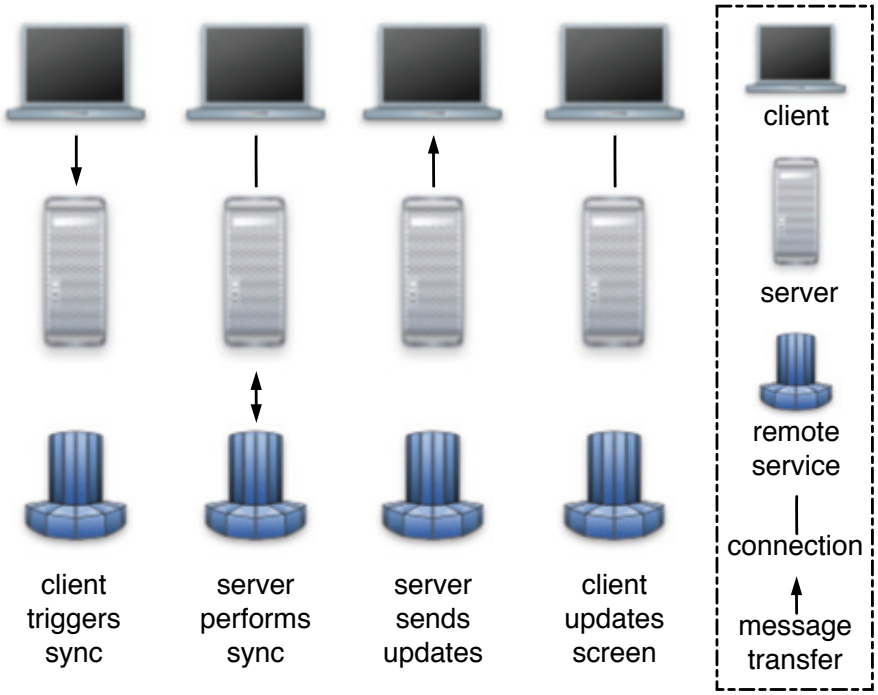

Figure 3.2: Asynchronous Synchronization. The client triggers the synchronization operation. The server synchronizes the project with the remote service and sends an update to the clients after the synchronization process has been finished successfully. The clients update their screens to display the current project.

3.2 Functional Requirements

The following situations can occur during an Agile Planner-Jazz-synchronization (a more detailed use case description can be obtained from Appendix A):

- Story Card added on Agile Planner
- Work Item added on Jazz
- Story Card deleted on Agile Planner
- Story Card altered on Agile Planner
- Work Item altered on Jazz
- Work Item and the associated Story Card are simultaneously altered
- Iteration added on Agile Planner
- Story Card added to Iteration on Agile Planner
- Story Card removed from Iteration on Agile Planner
- Work Item added to Iteration on Jazz
- Work Item removed from Iteration on Jazz

The following descriptions use the term **task** to refer to either a story card or a work item, depending of the system which initiates the described action.

3.2.1 Story Card added on Agile Planner / Work Item added on Jazz

Agile Planner and Jazz shall be able to create new tasks. After the synchronization, the new task shall be available on both systems.

3.2.2 Story Card deleted on Agile Planner

It shall be possible to delete a task on Agile Planner. The changes shall be reflected on Jazz after synchronization.

3.2.3 Story Card altered on Agile Planner / Work Item altered on Jazz

Agile Planner and Jazz shall be able to alter existing tasks.

3.2.4 Work Item and the associated Story Card are simultaneously altered

It shall be possible to alter the same task simultaneously on both systems.

3.2.5 Iteration added on Agile Planner

Agile Planner shall be able to create new iterations. Those iterations shall be available on Jazz after the next synchronization.

3.2.6 Story Card added to or removed from an Iteration on Agile Planner

It shall be possible to associate a task with an iteration on Agile Planner. Furthermore, it shall be possible to remove this association later if required. Both cases shall be reflected on Jazz after the next synchronization.

3.2.7 Work Item added to or removed from an Iteration on Jazz

Jazz shall be able to associate a task with an iteration and remove this association if required.

3.3 Non-Functional Requirements

Unlike functional requirements, non-functional requirements are not reflected through the systems functionality but might be reflected in its *response time* during operation or its *testability* during development or maintenance.

3.3.1 Testability

Since Agile Planner and Jazz are developed by distinct companies, changes in their interface or behaviour will occur. It shall be possible to test Jazz functionality without Agile Planner so that behavioural changes in Jazz can be detected early.

3.4 Accessing Work Items

There are two ways to access work items which are stored on the Jazz server;

- through the Server plug-in
- through Jazz Plain Java Client Library

3.4.1 Server plug-in

The Jazz server could be extended by a custom plug-in that offers an easy to use access-point to the repository and Jazz planning tools. Agile Planner would need a communication facility to send and receive data to and from the Jazz server. The actual server plug-in would be divided into two distinct plug ins a

Service plug-in which allows communication with Agile Planner and a *Common plug-in* to access work items on the server as shown in Figure 3.3.

However, this approach has several disadvantages. The service plug-in must use Jazz authentication and user management facility to keep access to the server restricted. Otherwise the plug-in would be a potential security risk. Furthermore, the high coupling between the common plug-in and the internal Jazz services and its plug-in registration procedures could make the plug-in very sensitive to any changes within Jazz.

Less code in Agile Planner but **higher coupling** with Jazz

3.4.2 Client

Jazz Eclipse plug-ins can be used like any other Eclipse plug-ins[2] but Jazz also offers *Plain Java Client Library*. This can be used in any Java application and is not restricted to Eclipse plug-ins. Although the Plain Java Client Library is more flexible, it needs some more attention. The Client Library offers a set of interfaces that can be used to gain access to Jazz server extentions, which are responsible for the actual functionality. The Jazz server sends most of the necessary instances of the classes which implement the interfaces to the client at runtime. The client library uses a service interface to communicate with the Jazz service on the actual server, which offers the counterpart to the client service interface as shown in Figure 3.4.

Lower coupling with Jazz but **more code** in Agile Planner

The Client Library offers interfaces to virtually all of Jazz features. The amount of dependent Jazz services could be reduced by using the client library instead.

3.4.3 Conclusion

Server plug-ins need to interact with lots of Jazz components. The high coupling to Jazz increases the probability that changes in Jazz could have an impact on the plug-in. The use of the Plain Java Client Library would cause an increased amount of Jazz related code on Agile Planner Server but offers fewer dependencies to Jazz since access to fewer Jazz components is required. Therefore, using the Plain Java Client Library to access Jazz is the better choice in this particular case.

3.5 Adapter

In order to enable both systems to exchange information an adapter must be placed between both systems. To prevent high coupling between the systems the adapter must transform all data types. This way Agile Planner only has to deal with story cards and Jazz with work items.

The adapter[3] (illustrated in Figure 3.5) would be comprised of multiple subcomponents, such as

[2] via their extention points

[3] This is the *adapter-component*

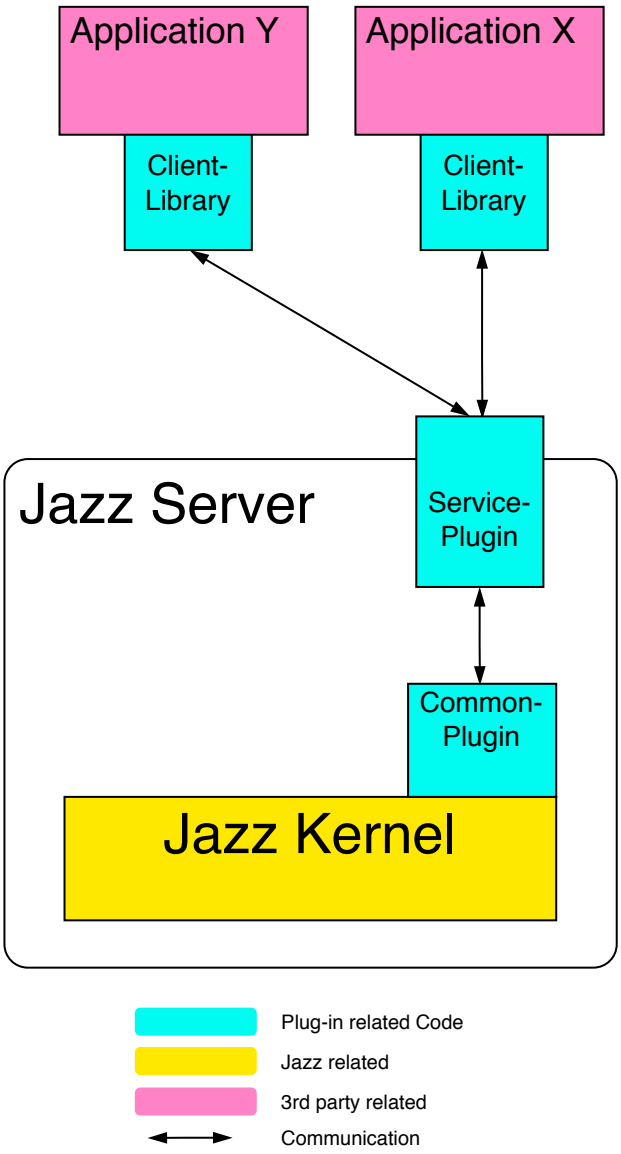

Figure 3.3: Jazz Plug-in overview. The Jazz Server Plug-in is comprised of a common- and a server-plug-in. 3rd-party application use the client library to connect with the service plug-in, which delegates method calls to the common plug-in.

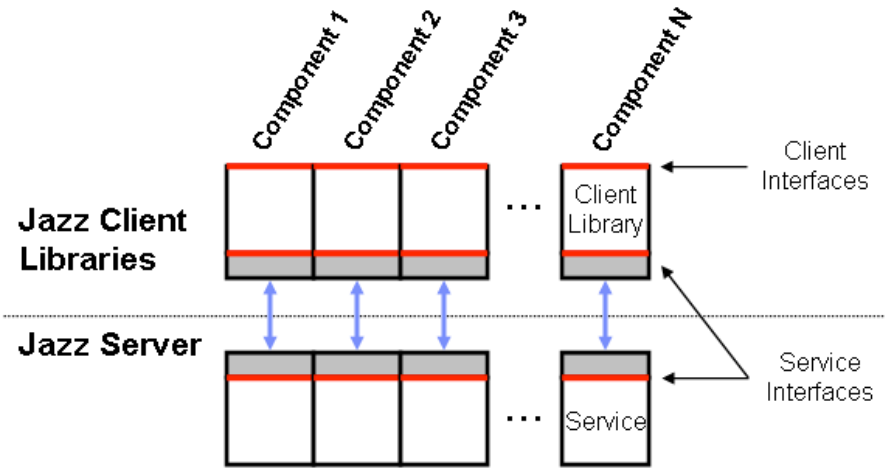

Figure 3.4: Jazz Services. The client interface communicates through a communication layer with a server interface which offers a specific functionality. [taken from http://jazz.net]

- Adapter[4]
- Converter
- Error Facade
- Session Manager
- Component Manager

3.5.1 Adapter

The adapter offers functionality which will be used by Agile Planner to retrieve necessary information from Jazz or to update information on Jazz. Methods such as

- Retrieve all work items from Jazz
- Update work item on Jazz
- Create new iteration on Jazz
- ...

[4]This is the actual *adapter-class*

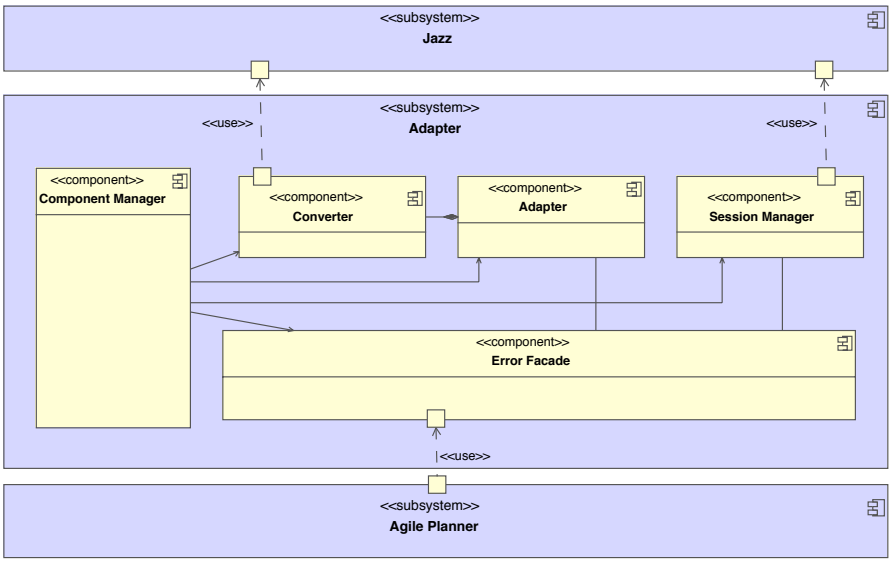

Figure 3.5: Static View of the adapter. [Key: UML Component Diagram]

3.5.2 Converter

The converter transforms all data objects which have to be exchanged between the two systems. Besides work items, iterations need to be transformed as well. The converter is the only instance which knows everything about particular Agile Planner and Jazz data objects.

3.5.3 Error Facade

Jazz throws exceptions in case of a problem. This exception is Jazz-specific and therefore for internal use only. Agile Planner is not part of Jazz and should not know internal details of Jazz. Therefore Agile Planner should not know Jazz *TeamRepositoryException*.

De-couple both systems

The Error Facade is a layer between the adapter and Agile Planner. It forwards all method calls to the appropriate (sub-)component inside the adapter and returns results to the caller if required. The most important task is to catch exceptions. Very simple error facades just catch exception and maybe write a note into the log-file or the console.

More sophisticated error facades often try to *repair* the problem. The facade could call a *Diagnostic&Repair-Agent*, such as SQL-Expert, which tries to create a detailed diagnosis and then tries to repair the problem if possible. If the agent fails too or if no agent is available, the error facade creates an appropriate exception which will be caught by the original caller, illustrated in Figure 3.6. More about error facades can be obtained from [Sie].

Possibility to add **diagnosis and repair** in the case of an **error**

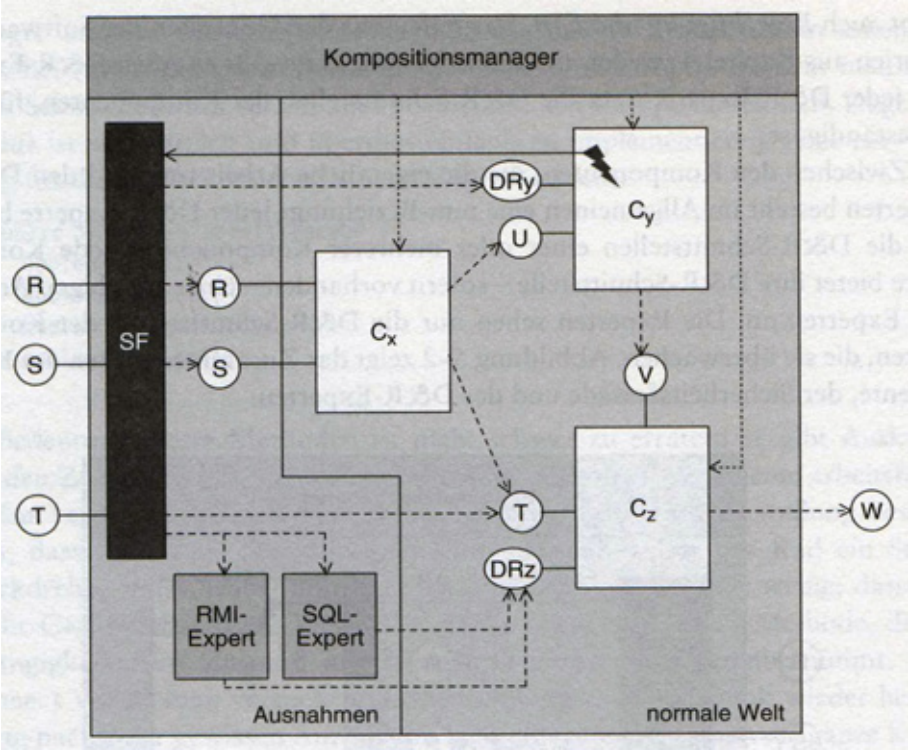

Figure 3.6: Example of an error facade [Sie].

3.5.4 Session Manager

Session management will be part of the adapter as well. This reduces the amount of components which Agile Planner has to work with. The session manager creates and maintains a connection to Jazz.

3.5.5 Component Manager

Configures the adapter

The component managers responsibility is to instanciate all adapter-subcomponents, link them together, and to deliver a fully functional adapter to the caller. This approach offers multiple advantages.

The complexity of the actual adapter is hidden from the user who uses the adapter. The user only knows the adapter-interface and the component manager, which delivers an instance of the adapter.

Each subcomponent of the adapter can be replaced by another without notifying the actual user. The component manager is the only authority who knows each of the subcomponents and how to instanciate and configure them.

3.5.6 Entity–Mapping

It is most unlikely that both systems use the same data model. It is therefore necessary to analyse both data models and categorize all attributes into one of the following categories:

identical Fields which store the same information in the same representation[5].

adaptable Fields which store the same information in different representations. It is possible to transform the representation between the different formats.

emulated A field in one model has no correspondent field in the other model but the information must be preserved anyhow.

insignificant A field in one model has no correspondent field in the other model and there is no need to transfer the information between the systems.

Table 3.1 and 3.2 give a quick overview of the data models. Table 3.3 applies the categories to the the models.

3.6 Prototypes

New systems always create a high learning curve. It is wise to build at least one prototype to increase the developers personal experience. Then, when most of the problems are known and hopefully solved, the actual component(s) can be designed and implemented. The prototypes should comprise at least *session management*, *work item access* and *data conversion*.

[5]The representation refers not only to the data types but also to the content itself, e.g. fields of type String could store a description as plain- or XML-Text.

Table 3.1: Agile Planner Attributes

Field	**Datatype**	**Description**
id	long	Unique identifier to distinguish story cards
name	String	Short description of the task
description	String	Full description of the task
acceptanceTestText	String	Description of the FIT test
acceptanceTestUrl	String	Uniform resource location address of the FIT test
fitID	String	Identifier of the FIT test
color	String	Colour of the story card in the workspace
height	int	Height of the story card in the workspace
width	int	Width of the story card in the workspace
locationX	int	Location of the start point of the story card in X position
locationY	int	Location of the start point of the story card in Y position
rotationAngle	float	Rotation Angle of the story card in table top view
parent	long	Identifier of story card s parent entity, i.e. an iteration
bestCaseEstimate	float	Minimum time to finish the task
mostLikelyEstimate	float	Estimated time to finish the task
worstCaseEffort	float	Maximum time to finish the task
actualEffort	float	Actual time spent
owner	String	Developer who s in charge of this task
rallyID	boolean	Has this card a Rally ID
handwritingImage	byte	Stores a picture of the card s surface if it contains content that has been written by hand

Table 3.2: Jazz Attributes

Field	**Datatype**	**Description**
summary	String (HTML)	Quick summary of the task, used as headline
description	String (HTML)	Full description of the task
id	int	Unique identifier to distinguish work items
internalState	String	Points out the state, such as *new* or *in-progress*
internalResolution	String	Reason for resolution, such as *fixed* or *invalid*
resolutionDate	Timestamp	Date when the work item has been resolved
internalSeverity	String	Describes the severity of the task
creationDate	Timestamp	Date when the work item has been saved
creator	Contributor	Developer or Manager who created this task
internalPriority	String	Work item s priority
dueDate	Timestamp	Task must be finished by that date
owner	Contributor	Developer who s in charge of this task
category	Category	Points to a sub-project
internalComments	Comment	Useful information or discussion about the current task
internalSubscriptions	Contributor	Contributor who should be notified if a change occurs
workflowSurrogate	String	
tags	String	Keywords that describe the task
workItemType	String	Type-information, such as *task* or *bug*
duration	long	Estimated time to finish the task
timeSpent	long	Actual time spent on that task
projectArea	ProjectArea	Project area which *contains* the work item
resolver	Contributor	Developer who has set the work item to resolved
internalApprovals	Approval	Points to an approval such as *approved by Quality Assurance*
internalApproval-Descriptors	Approval-Descriptor	
target	Iteration	Iteration that this work item belongs to
internalSequence-Value	String	
foundIn	Deliverable	

Table 3.3: Attribute Mapping

Agile Planner	**Jazz**	**Category**
id	id	adaptable
name	summary	adaptable
description	description	adaptable
acceptanceTestText	-	emulated
acceptanceTestUrl	-	emulated
fitID	-	emulated
color	-	emulated
height	-	emulated
width	-	emulated
locationX	-	emulated
locationY	-	emulated
rotationAngle	-	emulated
parent	-	emulated
bestCastEstimate	-	insignificant
mostLikelyEstimate	duration	adaptable
worstCaseEstimate	-	insignificant
acutalEffort	timeSpent	adaptable
owner	owner	adaptable
rallyID	-	insignificant
handwritenImage	-	insignificant
-	internalState	insignificant
-	internalSeverity	insignificant
-	resolutionDate	insignificant
-	internalResolution	insignificant
-	creationDate	insignificant
-	creator	insignificant
-	internalPriority	insignificant
-	dueDate	insignificant
-	category	insignificant
-	internalComments	insignificant
-	internalSubscriptions	insignificant
-	workflowSurrogate	insignificant
-	tags	insignificant
-	workItemType	insignificant
-	projectArea	insignificant
-	resolver	insignificant
-	internalApprovals	insignificant
-	internalApprovalDescriptors	insignificant
-	target	insignificant
-	internalSequenceValue	insignificant
-	foundIn	insignificant

3.6.1 Session Management

The session management establishes a connection to Jazz and performs user authentication.

3.6.2 Accessing Work Items

Once a connection has been established, access to the actual work items is required. The most important actions are to read, write, and alter work items. Iteration and user management are also associated with work item management and have to be performed as well.

3.6.3 Converting Data

Jazz work items have to be converted to Agile Planners story cards and vice versa. The required conversion leads to further tasks such as creation of attributes to store customized data as shown in Table 3.3. It is difficult to predict the impact of these tasks. Therefore a prototype is necessary to clarify this first.

Chapter 4

Realization

4.1 Prototypes

As mentioned in section 3.6, multiple prototypes have first been created to learn more about Jazz and its API. The prototypes highlighted the complexity of Jazz.

4.1.1 Create and save a Work Item

A *work item* is the most important object that is used among Jazz agile planning tool. Therefore, the first step is to create a new work item. The code in Listing 4.1 illustrates how to create a work item, while Listing 4.2 shows the necessary code to save the work item to a Jazz server.

Jazz-**API** is too **complex**

Listing 4.1: Code to create a work item

```
1  IWorkItem workItem = null;
2  IWorkItemClient client = repository.getClientLibrary(
       IWorkItemClient.class);
3
4  IWorkItemType workItemType = client.findWorkItemType(
       projectArea, WorkItemTypes.DEFECT, monitor);
5
6  ICategoryHandle category;
7  List<ICategory> findCategories = client.findCategories(
8          projectArea.getProjectArea(),
9          ICategory.FULL_PROFILE, monitor
10         );
11
12 category = findCategories.get(0);
13
14 IWorkItemHandle handle = client.
       getWorkItemWorkingCopyManager().connectNew(
15         workItemType, monitor
```

```
16          );
17
18  WorkItemWorkingCopy wc = client.
        getWorkItemWorkingCopyManager().getWorkingCopy(
19          handle
20          );
21
22  workItem = wc.getWorkItem();
23  workItem.setCategory(category);
```

Listing 4.2: Code to save a work item

```
1   IWorkItemWorkingCopyManager manager =
2   repository.getClientLibrary(IWorkItemClient.class).
        getWorkItemWorkingCopyManager();
3   manager.connect(
4           workItem,
5           IWorkItem.FULL_PROFILE,
6           monitor
7           );
8
9   WorkItemWorkingCopy copy = manager.getWorkingCopy(
10          workItem
11          );
12  workItem = copy.getWorkItem();
13
14  copy.save(monitor);
15  manager.disconnect(workItem);
```

The code in listing 4.1 and listing 4.2 demonstrate that even *simple operations*, such as creating an empty work item or saving a work item, can be quite complex. In fact, the code has been simplified for readability by omitting exception handling.

4.1.2 Simplify the API by adding an Abstraction Layer

It is obvious that the Jazz API is too complex to *simply use it*. If used in the adapter, the complexity would confuse developers and would set their focus on *how to implement* the functionality instead of the method s purpose. An additional software layer must be introduced to simplify the Jazz API, the so called *Abstraction Layer*.

4.2 Building an Abstraction Layer

The abstraction layer will be placed between Jazz and the actual adapter and consists of several methods. Each of these methods calls one or more methods

An additional layer **simplifies** the Jazz-**API**

on Jazz and returns the result if necessary. Since abstraction layers are common in software architecture, many design patterns exist which can be applied. Depending on the realization of the adapter, the *Remote Facade* [ACM] or the *Business Delegate* [ACM] design pattern can be used to encapsulate Jazz API. The most important difference between both patterns is that the Remote Facade is implemented on the server while the Business Delegate is used on the client side.

As discussed in section 3.4, access takes place via the Jazz client library. Therefore, the Business Delegate design pattern will be used.

4.2.1 The Business Delegate Design Pattern

Delegator **forwards** method calls to the server

This design pattern is often used to hide the client from the complexity of the server API and to reduce the network load by caching important objects. Furthermore, it allows the client to test the server functionality through a set of tests without worrying about client related code. If the client application suddenly behaves different or some functionality fails the test set can be used to verify that the server still works as expected. In the latter case, the failure has been caused by the client application.

The sequence diagram in figure 4.1 illustrates the principles of how the delegator operates. First, the client calls a method on the delegator. Next, the delegator invokes one or more methods on the server and delivers the results to the caller. The client uses a simplified method that hides the complex server API.

Unlike the pattern description in [ACM], the abstraction layer doesn t translate exceptions nor data types. Data objects transformation is not necessary since the adapter will perform those. Exceptions remain untouched because an error facade (3.5.3) will transform them if necessary.

4.2.2 Abstraction layer code

Heavily relied upon methods separated into **distinct interfaces**

The abstraction layer is comprised of all methods that are necessary for the adapter to fulfill any given task. Most of the code will be in one class called *JazzUtilities*. This includes the creation of work items, iterations, users as well as methods to alter work items.

Although many methods will be in a single class they are separated into two distinct interfaces to make them easier to use.

Management Includes all methods to manage the development environment, such as adding users, iterations, ...

Utility Includes all methods that are very often used by the adapter, such as creating and altering work items, ...

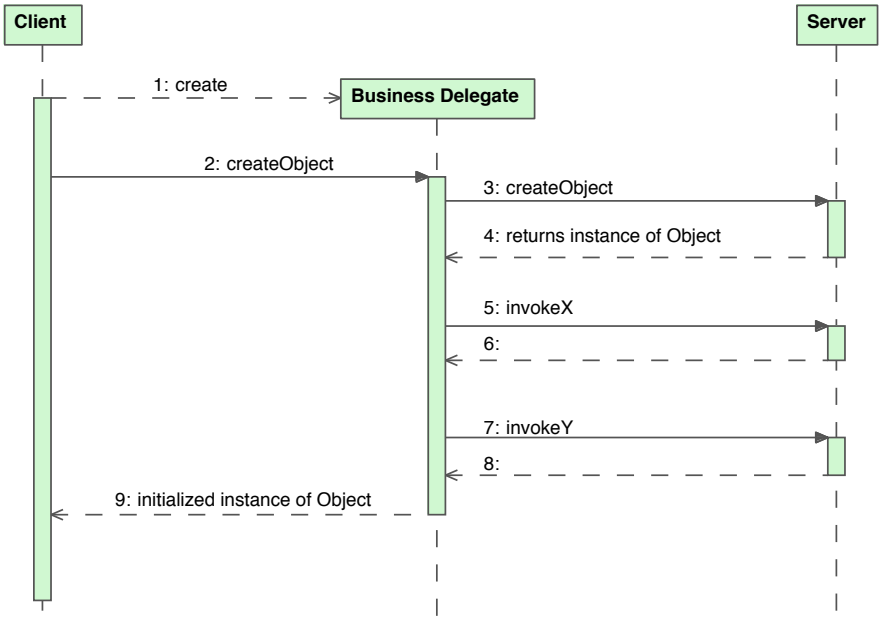

Figure 4.1: Example of the Business Delegate Design Pattern.

4.2.3 Service Lookup

The actual Jazz client library only offers a few concrete classes. In fact, the developer uses interfaces most the time. At runtime, the Jazz server sends an instance of the proper class, that implements a specific interface, to the client. This allows for easy upgrades on the server side since no client code will be affected as long as neither the interfaces nor the concrete classes change.

Caches instances of classes from Jazz

Although the client could ask for an instance of a class if needed, storing the instances on the client reduces the network load. Therefore, a *Service Lookup* class will be responsible for fetching the instances from the server, store them locally and return them upon a clients request.

4.2.4 Query Builder

The adapter often needs access to a specific work item. Thus, it is not practical to retrieve all work items if just a single one is needed. To alleviate this problem, the adapter uses different queries to retrieve specific items. A query consists of two parts:

Separate the **creation** and **execution** of queries

Query A logical expression that is comprised of the attribute that should be compared, a value to compare with and a boolean expression, such as **equals**, **includes**, ...

Query Execution Sends the expression to Jazz query engine, which executes the query and returns the result.

Since the query execution code remains identically, the creation of the expression will be separated from the execution code. A Query Builder [Sie] offers an easy to use interface that consists of all possible queries[1], shown in Listing 4.3. The Query Builder class builds the actual query and returns an expression object, demonstrated in Listing 4.4.

Listing 4.3: Query Builder Interface

```
1  public Expression workItemByID(AttributeOperation op, int
       id)
2  throws TeamRepositoryException;
3
4          public Expression workItemBySummary(
5                          AttributeOperation op,
6                          String text
7                  ) throws TeamRepositoryException;
8
9          public Expression workItemByIteration(
10                         AttributeOperation op,
11                         IIteration iteration
12                 ) throws TeamRepositoryException;
13
14         public Expression workItemByState(
15                         AttributeOperation op,
16                         State state
17                 ) throws TeamRepositoryException;
18
19         public Expression workItemByAPID(
20                         AttributeOperation op,
21                         long id
22                 ) throws TeamRepositoryException;
```

Listing 4.4: Query Builder:

```
1  public Expression workItemByState(AttributeOperation op,
       State state)
2  throws TeamRepositoryException {
3
4          Expression exp = null;
5          IQueryableAttribute attribute;
6
7          attribute = QueryableAttributes.getFactory(
               IWorkItem.ITEM_TYPE).
```

[1] This approach can be used since no user defined queries are necessary

```
8
9           findAttribute(
10                  projectArea,
11                  IWorkItem.STATE_PROPERTY,
12                  lookup.getAuditableClient(),
13                  monitor
14                  );
15
16          exp = new AttributeExpression(attribute, op,
                state);
17
18          return exp;
19  }
```

New queries can be easily added to the Query Builder without affecting other classes. The execution code will be in JazzUtility.

4.2.5 Dealing with copies of work items

Simplify the **management** of work item copies

Often work items need to be altered but Jazz always sends copies of the actual work items to the client. For this reason, the clients need to request a mutable copy of the work item that they want to alter. The client can then alter and save the work item, but they also have to keep track of each copy since they are marked as *delivered mutable* on Jazz. The client has to close the connection in order to reset this mark on the Jazz server. Therefore, a *Copy Manager* is needed to deal with this procedure.

The *Work Item Copy Manager* checks out copies of a specific work item. It stores the copy locally in order to deliver multiple copies to clients. It then closes the connection to the Jazz Work Item Copy Manager when a work item copy is no longer needed.

4.2.6 Component Manager

Configures the abstraction layer

The *Component Manager* creates instances of all the classes and glues them together. It then delivers the fully configured abstraction layer to the client.

The client doesn t need to know what classes are actually necessary in order to perform all tasks nor need to know the names of the classes. This makes it easier to replace a sub-component since only the component manager needs to know about the change.

4.2.7 Resulting architecture of the abstraction layer

Separate the **public** and **private** components

The abstraction layer component is the result of all previously mentioned classes. They are divided into multiple packages depending on their visibility. All public classes and interfaces are stored in a package (ca.ucalgary.jazzconnect) while all implementations, except for the component manager, are stored in a different

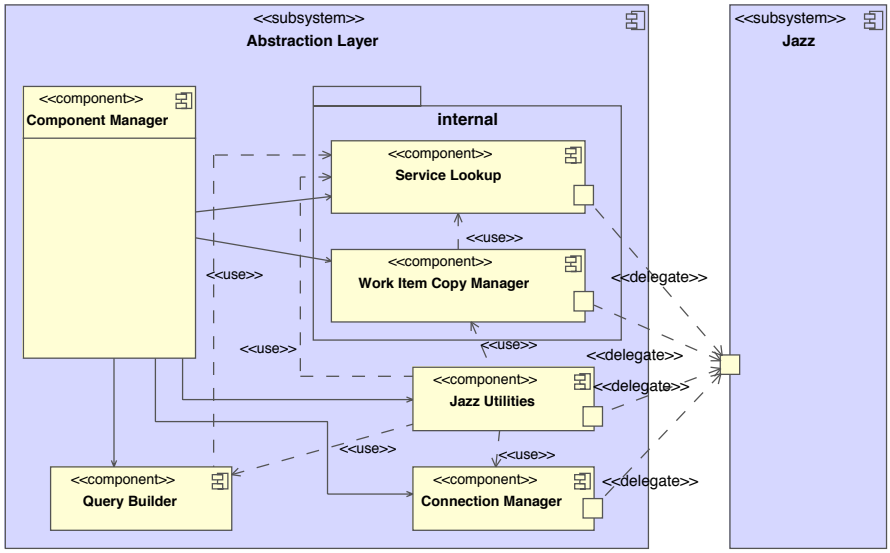

Figure 4.2: Overview of the resulting abstraction layer.

one (ca.ucalgary.jazzconnect.internal). By convention, no component or class that uses the abstraction layer should ever use one or more classes from the *.internal package directly. If it does, it s considered highly coupled to this specific implementation and an update of the referenced class can cause incompatibilities. Figure 4.2 illustrates the resulting structure of the abstraction layer.

4.3 Abstraction Level

The prototypes have shown that it is very difficult to use the Jazz API directly. A single operation can require multiple lines of Jazz related code. Using the Jazz API in the adapter would make it difficult to keep the focus on the actual adapters functionality. Instead, an abstraction layer will be introduced that encapsulates Jazz specific code, similar to the *Business Delegate* design pattern [ACM].

4.3.1 Differences from the Business Delegate design pattern

Doesn't transform objects and exceptions

Unlike the Business Delegate design pattern presented in [ACM], the abstraction layer will not hide Jazz completely. Hiding Jazz completely would cause to introduce new data objects to hide Jazz work items, iterations and exceptions. Instead, only the functionality will be encapsulated while the data objects and

exceptions remain the same. The resulting abstraction layer is pretty similar to the *Remote Facade* design pattern [ACM] except that the facade would be implemented on the server side.

The adapter is comprised of a converter that transforms data objects into suitable representation, that includes

- Work Items to Story Cards
- Story Cards to Work Items
- Jazz Iterations to Agile Planner Iterations
- Agile Planner Iterations to Jazz Iterations

Exceptions will not be transformed until they reach the ErrorFacade, refer to 3.5.3.

4.4 Composition of the different components

So far, there are two major components, the abstraction layer and the actual adapter. It would be possible to separate both in to their own projects and resulting jar-files but it is not feasible because of *problems caused by indirect references* and a lack of *reusability*.

4.4.1 Problems caused by indirect references

The abstraction layer encapsulates most parts of Jazz so that it is easier to use. Therefore, this component must contain all necessary Jazz libraries. On the other hand, the adapter needs knowledge about Jazz internal workings such as exceptions (refer to 4.3.1).

Combine abstraction layer and adapter into **one project**

Since the libraries are already available in the abstraction layer, there is no need to copy them into the adapter-project too. Furthermore, if one or more library must be replaced, it must be done in multiple places.

Problems can arise at runtime when the adapter needs access to Jazz libraries which are in the abstraction layer package since they must be *indirectly referenced*. If the libraries are not exported correctly, then a runtime exception will occur.

4.4.2 Reusability

The abstraction layer is highly customized to meet the adapters requirements. It is doubtful that the abstraction layer can be reused in another project without major changes. A generic version of the abstraction layer could cause the loss of the simplified and easy to use interface, and in the worst case, could interfere with the adapters requirements.

Abstraction layer can **not** be **reused** in other projects

4.4.3 Resulting composition

According to 4.4.1 and 4.4.2, separation of both components is not useful. Therefore, both components will be merged into a single project as shown in Figure 4.3.

4.5 The Adapter Component

Section 3.5 has given a quick overview of the adapter component. The following sections explain the important sub-components in more detail.

4.5.1 Converter

The converter offers methods to convert *story cards* into *work items* and vice versa.

4.5.1.1 Convert Story Cards to Work Items

Two situations can occur which have a huge influence on the whole conversion process:

1. The story card counterpart already exists on Jazz
2. The story card is new, e.g. no counterparts exists on Jazz

If a story card s counterpart already exists on Jazz, the associated Jazz work item might have be altered if the related information has been changed. Therefore the converter must query the work item. The query searches for the Agile Planner ID in the Jazz repository. The retrieved work item can then be altered and saved to Jazz.

The query returns an empty list of work items if the counterpart to the current story card does not exist on Jazz yet. The converter must then create a new work item, add Agile Planner specific attributes to the newly created work item and copy all story card information to the new work item, illustrated in Figure 4.4.

Creates work item **on demand**

4.5.1.2 Convert Work Items to Story Cards

Unlike the conversion of story cards into work items, the converter always creates a new story cards even if it already exist in Agile Planner. The convert-method simply copies all attributes from the work item to a new story card.

It is possible that work items do not have Agile Planner specific attributes. This occurs if they have been created on Jazz and not yet copied to Agile Planner. In that case the story card ID will be set to 0.

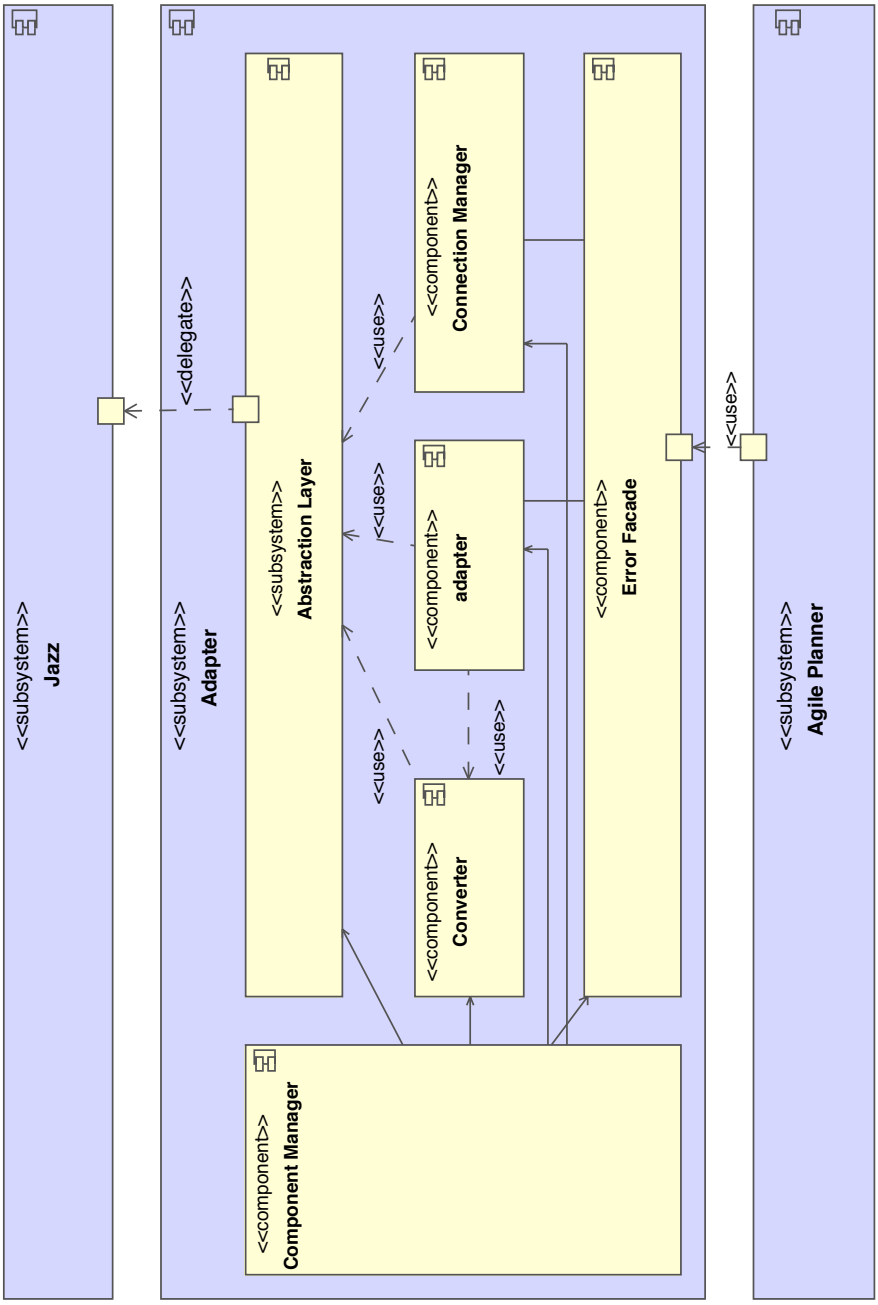

Figure 4.3: Overview of the resulting adapter. [Key: UML Component diagram]

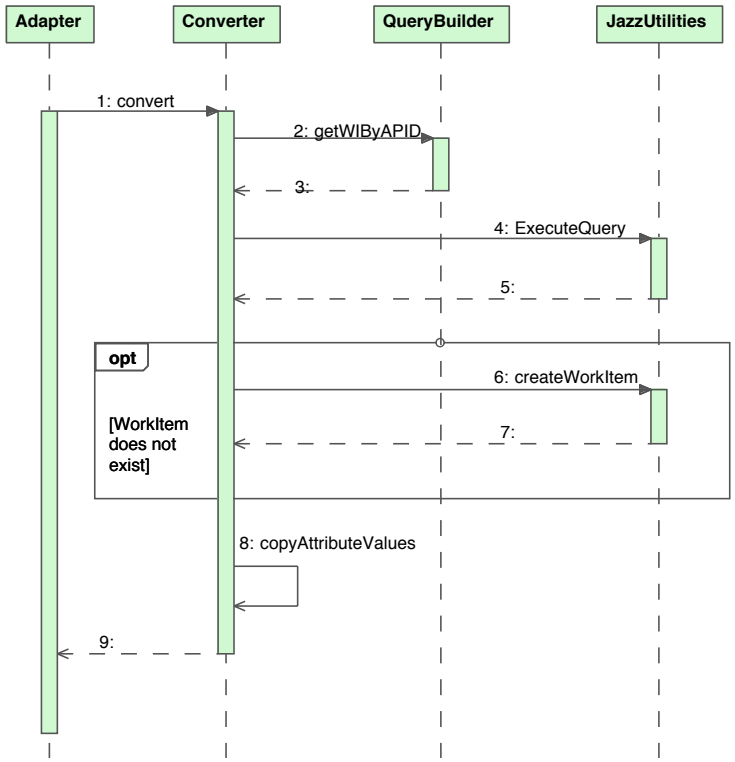

Figure 4.4: Conversion: Story Card to Work Item. [Key: UML Sequence diagram]

4.5.2 The adapter class

The actual adapter class simply offers methods to retrieve specific information, such as story cards, iterations, and methods to update this information on Jazz. To achieve this, the adapter uses queries which are defined in *QueryBuilder*. It then converts the delivered work items to story cards by using the converter.

Provides **high level** functionality to work with Jazz

Once converted, all story cards which have newly been created on Jazz must be filtered out because they require special treatment.

4.5.2.1 Filtering out story cards

As discussed in Section 4.5.1.2, the ID 0 has been assigned to story cards which have been created on Jazz and not been copied to Agile Planner yet. The filter simply removes cards with ID 0 from the list of story cards and adds them to a new list for later processing.

Items created on Jazz need **special treatment**

4.5.2.2 Story Cards with ID 0

All story cards with the ID of 0 have been added to a *special treatment list*. Since they have been created on Jazz, the associated work items on Jazz do not include the custom attributes which store Agile Planner specific information.

Add Agile Planner specific **attributes** to work items

The first step is therefore to add Agile Planner specific attributes to the work items. Before the attributes can be copied from the story card to the work item, the story card s ID must be set. Agile Planner offers a method *createID* which will be used to create a unique identifier. The work item can now be saved to Jazz.

Since the work item now includes all custom attributes and the Agile Planner ID, the adapter can update the work item just like the others, illustrated in Figure 4.5.

4.6 Agile Planner's Synchronization Facility

Agile Planner Server must perform several tasks to synchronize its planning data with Jazz. The whole synchronization process is encapsulated in one single class on Agile Planner Server. The process, shown in Figure 4.6, can be divided into a few logical steps:

1. Synchronizing Iterations
2. Update story cards on Jazz
3. Add story cards which have been newly created on Jazz to Agile Planner

4.6.1 Synchronizing Iterations

A new iteration might have been created since the last synchronization. The synchronization facility requests the latest iteration from Jazz, compares this with Agile Planner s current iteration and creates a new one on Jazz if required.

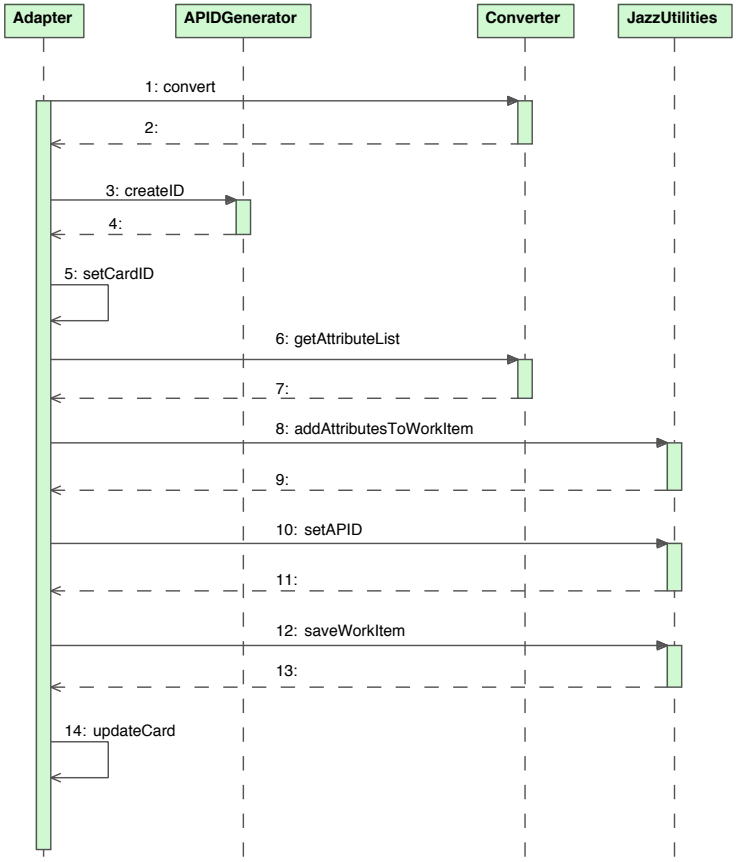

Figure 4.5: Story Cards which have been created on Jazz. [Key: UML Sequence diagram]

4.6.2 Update Story Cards on Jazz

New story cards might have been created or already existing story cards might have been altered during the planning meeting. Those changes must be copied to Jazz.

The synchronization facility first receives all story cards from Jazz which are either associated with the current iteration or not associated with an iteration and not resolved yet. The latter indicates that the story cards belong to the backlog. It then updates each story card on Jazz, that includes all story card attributes and the status. It also resolves all story cards on Jazz which have been deleted on Agile Planner since the last synchronization using the difference quantity of Agile Planners current story cards and the story cards received from Jazz at the beginning of the process, as shown in 4.1.

Update only those story cards which **already exist** on Jazz

$$JazzCards \backslash AgilePlannerCards \equiv \{card | (card \in JazzCards) \wedge (card \notin AgilePlannerCards)\} \tag{4.1}$$

4.6.3 Add story cards which have been newly created on Jazz to Agile Planner

Story cards may have already been created on Jazz since the last synchronization operation. The facility receives all story cards which have been newly created on Jazz and adds them to the current project.

4.7 Architectural overview

Different people are often involved in integration projects, such as developers and integrators. Each of these *roles* are interested in different aspects of the overall system. The architecture will therefore be illuminated from different perspectives using Kurchten s 4+1 View [Kru].

4.7.1 Logical View

The logical view represents all logical parts of the overall system, illustrated in Figure 4.7. Agile Planner uses the adapter to receive and update information from Jazz. Since Agile Planner can not handle Jazz data objects, a converter translates data objects back and forth. The abstraction hides Jazz complexity by offering convenient methods to access Jazz functionality. The abstraction layer itself communicates directly with Jazz and uses its functionality.

4.7.2 Development View

The development view represents software layers and services and how they are built on each other, illustrated in Figure 4.8

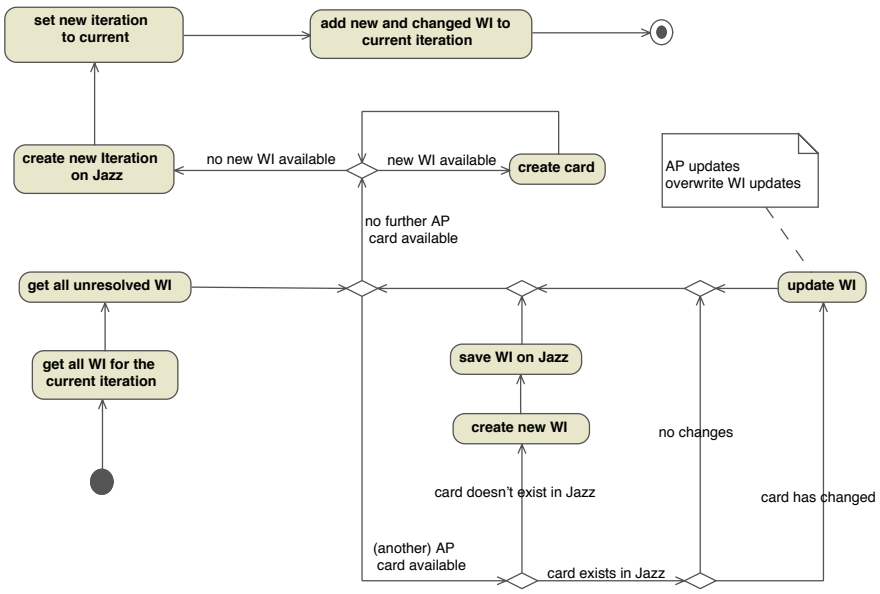

Figure 4.6: Overview of the synchronization process. [Key: UML Activity diagram]

4.7.2.1 Jazz Service Supporter

Jazz Service Support offer Jazz specific services to the Jazz Utilities, such as Query Builder or the Work Item Copy Manager. Supporter classes offer services to simplify Jazz Utilities or to extract redundant code.

4.7.2.2 Jazz Utilities

Jazz Utilities offer low level Jazz related services to layers above. They offer a simple interface to Jazz functionality, and delegate method calls to Jazz.

4.7.2.3 Adapter Service Supporter

Adapter Service Supporter offer services which are low level from the adapter s point of view. They are also specific to Agile Planner as well as to Jazz.

4.7.2.4 Adapter Service

Adapter Service is comprised of high level functionality which is needed by the synchronization process. It offers methods to retrieve or update Jazz work items.

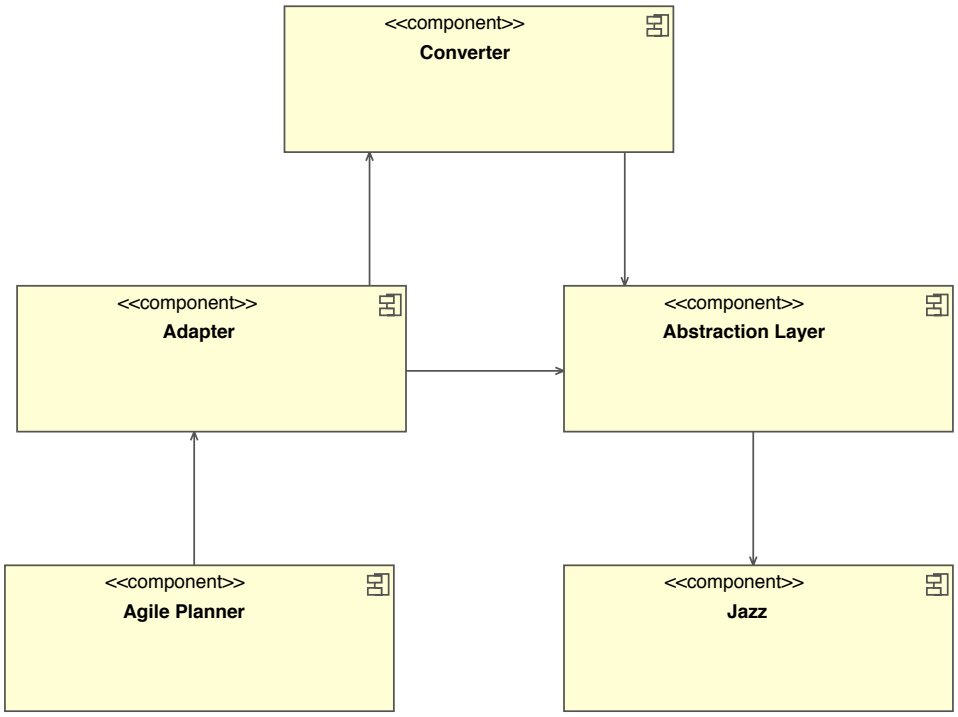

Figure 4.7: Logical Overview. [Key: UML Structure Diagram]

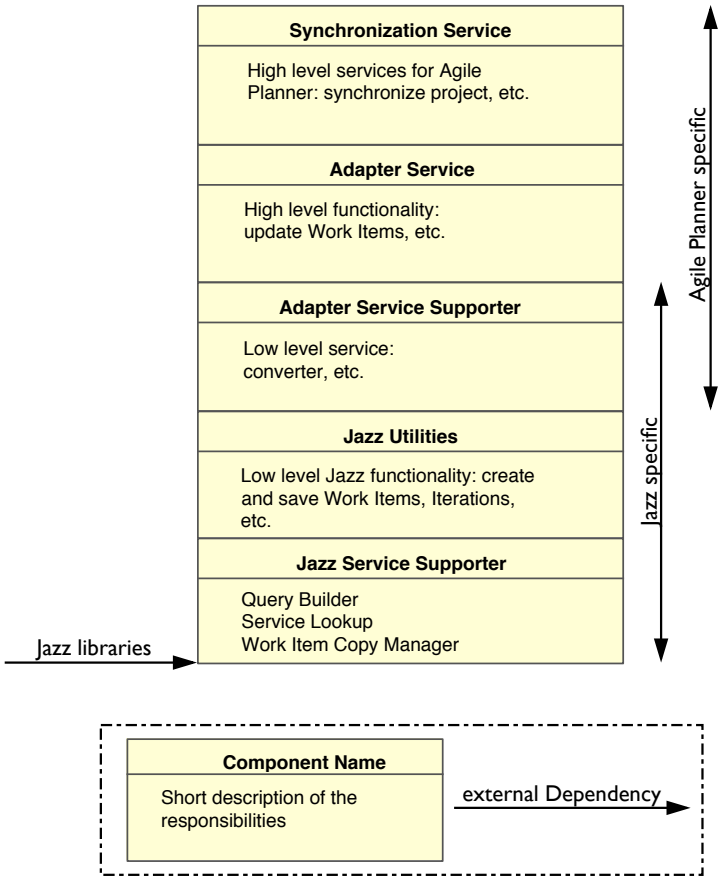

Figure 4.8: Development View

4.7.2.5 Synchronization Service

The synchronization Service is responsible to synchronize an Agile Planner project with an external service, such as Jazz.

4.7.3 Process View

The process view shows the different processes and how they communicate with each other, illustrated in Figure 4.9.

4.7.3.1 Agile Planner

Agile Planner client is used by developers during planning meetings. There are a number of different versions of Agile Planner, such as a stand-alone version

and a version for table top[2] systems.

4.7.3.2 Agile Planner Server

The server is responsible for persisting planning data and for distributing planning information to all clients.

4.7.3.3 Synchronization Facility

The synchronization facility gets the current project from the Agile Planner server and synchronizes its planning information with Jazz.

4.7.3.4 Adapter

The adapter offers methods to retrieve and send planning information to and from Jazz. The converter is part of the adapter and is responsible for converting data objects between Agile Planner and Jazz.

4.7.3.5 Abstraction Layer

The abstraction layer simplifies the access to Jazz. It simply delegates method calls to Jazz and returns the result to the caller.

4.7.3.6 Jazz

Although Jazz can be distributed among multiple servers[3], it often runs on a single Server.

4.7.4 Physical View

The physical view shows how each component can be distributed across multiple computers, illustrated in Figure 4.10. Although Jazz can be further divided so that the application and the DBMS run on separate servers, they often run on the same server.

4.7.5 Use Case View

The use case view represents the functionality of the system, e.g. what the user *can do* with the system. A more detailed description can be obtained from Appendix A.

[2]Digital Tables

[3]The Jazz Application (Server) and the Database Management System can run on dedicated Server.

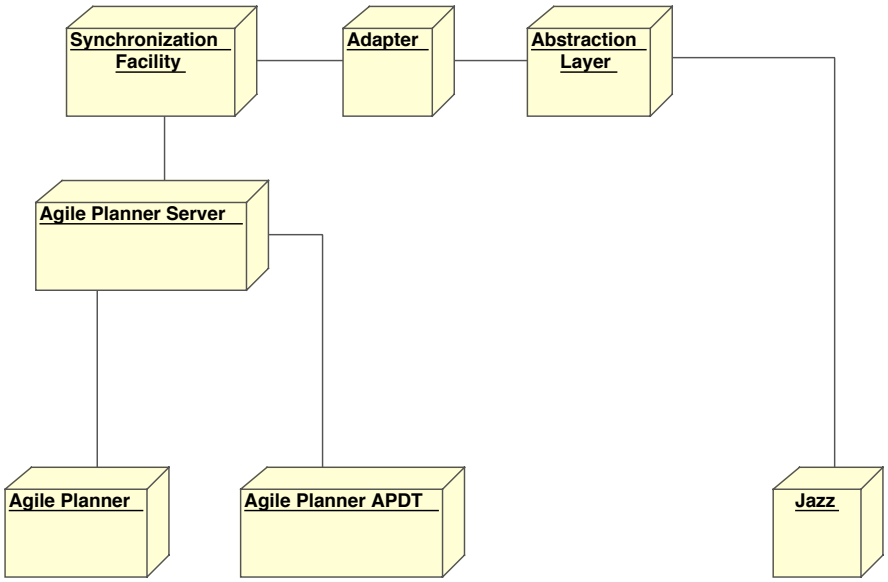

Figure 4.9: Process Overview. [Key: UML Structure Diagram]

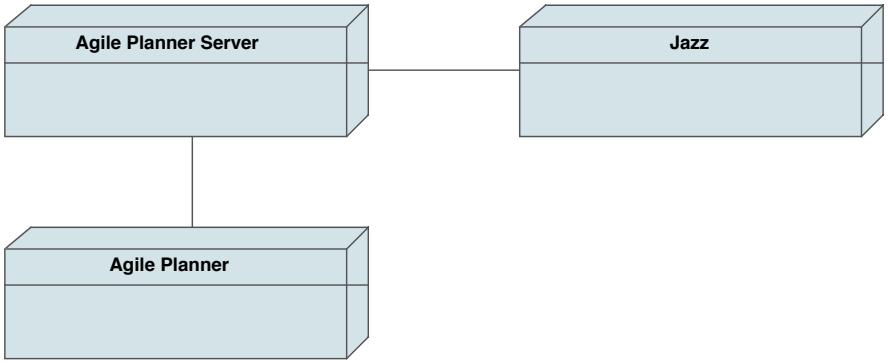

Figure 4.10: Physical View. [Key: UML Structure Diagram]

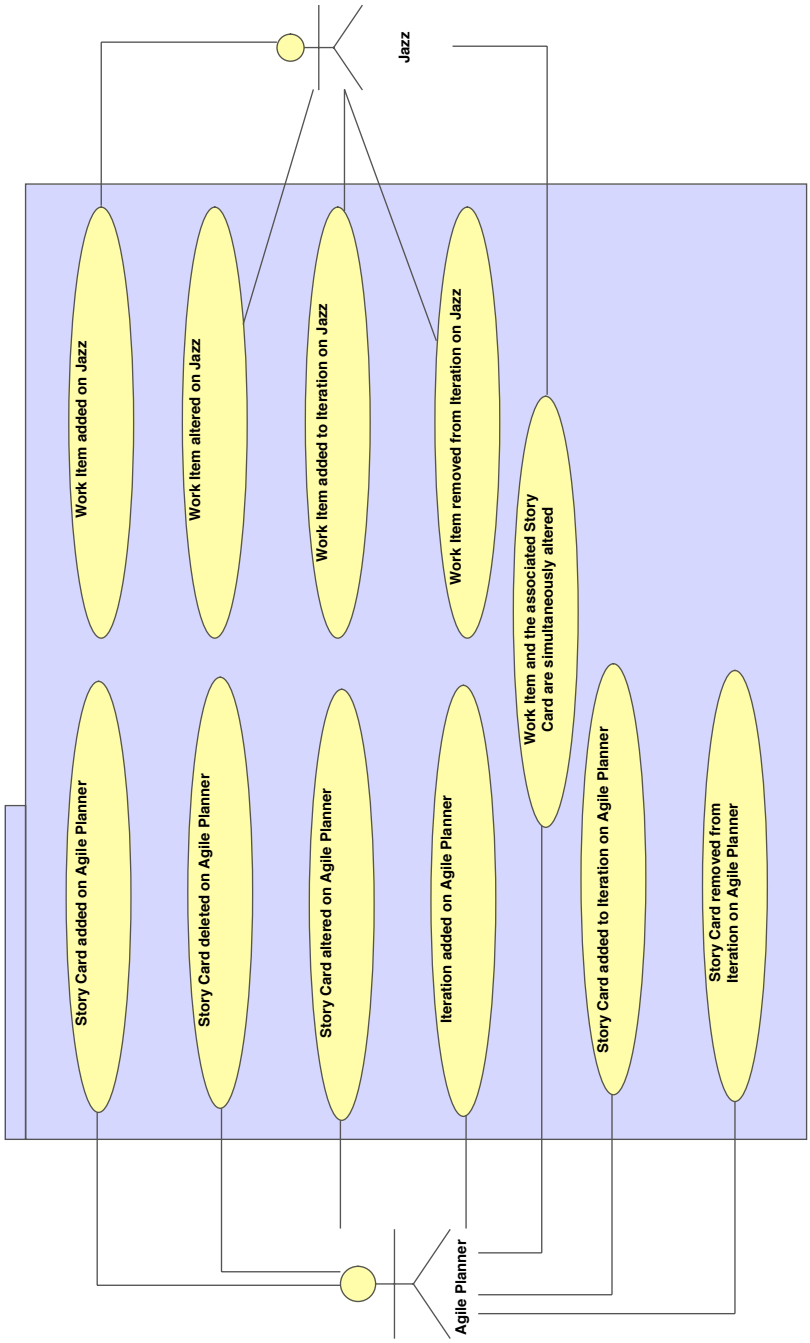

Figure 4.11: Use Cases of the Adapter. [Key: UML Use Case Diagram]

4.8 Integration into Agile Planner

Agile Planner server already has the ability to synchronize with a remote service the Rally web service. Therefore, the sync-message, which will be send from the client to the server, already exists an be reused. Agile Planner server then has to invoke the sync-method of the synchronization facility to trigger the synchronization process.

Chapter 5

Problems and solutions

Systems which are as big as Agile Planner and Jazz are usually quite complex. Therefore, integration often causes lots of problems.

5.1 Use cases

Some problems arose during development which were covered during planning of the adapter/abstraction layer. Solutions for some of the following problems have been omitted because chapter 7 offers general enhancements which covers these problems.

5.1.1 Work Item and the associated Story Card are simultaneously altered

Planning meetings will be performed on Agile Planner. Therefore, Agile Planner s planning information will be up-to-date and Jazz planning information can be overwritten.

A more sophisticated version of the adapter could offer abilities to merge information from both sources if required, discussed in chapter 7.1.

5.1.2 Jazz can not delete items

Once a work item has been created on Jazz it is not possible to delete it afterwards. The only possibility to *remove* a work item is to set it to *RESOLVED*. The latest version of Jazz offers the possibility to add further information about *why* a work item has been resolved.

Resolved is uses as a **synonym** for deleted

Unfortunately, the Jazz version that has been used to create the prototypes and the abstraction layer didn t support this feature in the same way the final version of Jazz does. Therefore, support for this additional feature has been omitted.

5.1.3 Support to merge changes

It is most likely that changes will be performed on both systems. Although Agile Planner offers up-to-date planning data after a planning meeting some information, such as *time-spent*, should simply not be overwritten. Future versions of the adapter could offer the possibility to customize the synchronization process and even the ability to merge manually if required.

5.2 Jazz

Jazz has been built from scratch. Despite IBM s effort to grant 3rd party developer early access to Jazz the beta versions lacked in functionality as well as in usability.

5.2.1 Documentation

The documentation of Jazz interfaces itself and how to use them was incomplete and not completely available. JUnit-Tests which were meant to test Jazz interfaces as well as to document how the client library can be used often contained deprecated code. Fortunately, IBM offered a developer forum where 3rd party developers can ask questions which were answered by other 3rd party developers and IBM employees.

5.2.2 Functionality

Some of Jazz functionality, such as user authorization, was not implement during development of the abstraction layer. Therefore, tests which were meant to test the abstraction layer s user management failed and had to be omitted. An updated version, which included user authorization, caused lots of problems because a few operations were limited to specific roles such as *administrator* or *developer*. It took some effort to configure a user before all required functionality could be used.

5.2.3 Usability

The usability suffered from unclear interfaces which contain several dozens of methods and vague descriptions. It was not always obvious how a method works and what is required to use it. For example, an iteration depends on a development line and vice versa. Therefore, it was not possible to add a new iteration without modifying the development line. Both, the new iteration and the altered development line, have to be save simultaneously since Jazz checks dependencies. Besides the JUnit-Test examples from IBM, Trial-and-Error lead to the goal during the prototype phase.

5.2.4 Jazz updates

Each major Jazz update brought in new client libraries. Previous versions of Jazz libraries were not compatible with newer Jazz versions. Therefore, each Jazz update caused an update of all libraries.

5.3 Agile Planner

Agile Planner has been built by students with different designing and programming skills. Furthermore, most of its code is either undocumented or not sufficient documented which makes it difficult to understand. JUnit-Tests covered less than a quarter of the whole code. Therefore, no regression tests exist for many classes and changes on one or more classes may cause problems everywhere.

Fortunately only a few classes had to be altered to integrate the Jazz adapter to Agile Planner so that its impact was limited. The performed changes were covered by JUnit-tests so that it was possible to show that the system s behaviour didn t change.

Chapter 6

Evaluation

This section discusses ways to evaluate Jazz. Although a full evaluation is very time consuming, the potential benefit can be tremendous.

6.1 Improved accuracy

The Jazz client allows the user to keep track of the time spent on a work item. The client s GUI includes a start-stop button to indicate that work on a specific task has been started or stopped. Jazz counts the minutes spent on a work item and saves this information in the work item. Unlike counting the time manually, Jazz time-count is less error prone.

6.1.1 Time tracking

Developers should be asked to both write down their time manually as usual and use Jazz time count. They should start this procedure at the beginning of an iteration and evaluate the data after the iteration to determine the degree of accuracy. They should repeat this procedure for two more iterations to improve overall statistics.

6.1.2 Interview

Developers should be interviewed after a short learning period [1].

- Did Jazz time tracking interfere with your workflow?
- How often did you forget to start or stop time tracking?
- How can we improve the workflow?

[1] the learning period is important since developers have to get used to the new workflow

6.2 Consistent user interface and behaviour

Jazz is a complete development environment which offers

- Planning tools
- Reporting tools
- Source control
- Build Environment
- ...

Since each tool has been developed by the same company it is likely that they are following the same usability guidelines. Therefore, the configuration and administration of all parts of Jazz are consistent in its usage. Developers as well as administrators do not have to learn how to use multiple tools.

6.2.1 Evaluate the time spent on daily tasks

Developers and administrators should track the time they need to perform daily tasks. After a short learning period, they should repeat the same tasks with Jazz. An evaluation will show whether Jazz reduces the amount of time to fulfill the work or not.

6.3 Migration to Jazz

A migration raises multiple questions which have to be solved first, such as

- Does Jazz offers the functionality we need?
- Can Jazz be (easily) extended?
- Can we customize Jazz to fulfill our needs?
- How can we transfer our current development process to Jazz?
- How much time will a migration take?
- How much time is necessary to train developers and administrators?
- Is it possible to migrate to other systems once we migrated to Jazz?
- How much will the migration costs be (Hardware, Software, Training, Support, Licenses)?
- How much time/money will we save if we use Jazz?
- Does scale Jazz with our needs (bigger projects)?

6.3.1 Using Jazz with 3rd party tools

Jazz offers the possibility to use Jazz own components or (partly) 3rd party components, such as subversion. Each organization should evaluate if Jazz functionality is capable to replace their existing tools if required. Once a Jazz component has been replaced by a 3rd party component, how does it impact on the daily workflow?

6.3.2 Role based workflow

Jazz forces a development team to take roles for each task. A role defines what a person is allowed to do. For example, a developer is allowed to submit code to the repository but not to create new iterations. An administrator can create new users but can not submit files to the repository.

How does these restrictions impact on the current workflow?

Chapter 7

Prospects

Although the Jazz adapter is fully functional, it only offers basic functionality. In order to become more practical for everyday use, some enhancements to Agile Planner and Jazz are required.

7.1 Enhancements

7.1.1 Preserve status changes from Jazz

Do **not overwrite** Jazz information

During synchronization, a Jazz work item will be overwritten by its corresponding Agile Planner story card. Although this is appropriate most of the time, specific information, such as *status* and *time-spent-information* should be preserved. An even more advanced approach will be described in section 7.2.1.

7.1.2 Work Item Types

Support **different types** of tasks

New work items, created from the adapter, always use the *type* **bug**, even if it s a new feature or an enhancement. Although the user can change this type information later in Jazz, the adapter should be able to set the appropriate type if possible. Since Agile Planner defines type information by the colour of the card and the colour-type-association can be changed by the user, minor changes on Agile Planner itself might be necessary.

7.1.3 Reason for item resolution

Distinguish completed from deleted

A work items status can be set to *Resolved* which indicates that the task has been completed. Even though this might be adequate in most cases, additional information about *why* a work item has been resolved can be useful to improve project statistics. Jazz offers multiple resolution choices, such as *Fixed*, *Duplicate*, *Invalid*, ... The adapter should make use of this to distinguish completed cards from deleted cards.

7.1.4 Enhanced Iteration Support

The abstraction layer only offers basic support for iterations. Therefore, the adapter can not implement all required functionality to offer a broad spectrum of features. Even though some functionality is applicable, minor changes to the abstraction layer might be necessary.

7.1.4.1 Multiple Iterations

Support **more than one iteration**

Currently, the Jazz-adapter supports just one iteration. If more than one iteration exist in Agile Planner, all but the first will be ignored. Since it is most likely that multiple iterations exist in a plan (e.g. a plan for the iteration after the current one) the adapter should be able to deal with multiple iterations.

7.1.4.2 Change Iteration Information

Information, regarding to an iteration, can change over the time, such as start- or end-date. Thus, the adapter should be aware of these changes.

7.2 New Features

Although the current adapter and its integration in Agile Planner has gone beyond a proof-of-concept, it still lacks particular functionality that would make it more useable.

7.2.1 Configurable synchronization

Support **customizable** synchronization

Currently, Agile Planner story cards overwrite Jazz work items on each synchronization; changes on Jazz are not considered yet. However, section 7.1.1 showed that some information should be preserved. The user should be able to customize the synchronization process. He or she could choose the following behaviour for each attribute

Always Agile Planner Indicates that Agile Planners information always takes priority.

Ask Agile Planner provides a dialog that shows the conflicting values and offers the user the possibility to choose the *right one* that will be used on both systems.

Always Jazz Indicates that Jazz information always takes priority.

7.2.2 Work Item (Change) History

The information that a work item carries can change over the time. A history of changes could be useful, especially if fields such as *description* or *summary* are affected. Jazz offers the item type *auditable* to keep track of these changes.

Once implemented, the change history could be used to restore to a previous state if they have been changed by accident.

7.2.3 Categories

Support sub-projects

A project can be divided into sub-projects, e.g. the Agile Planner project is comprised of Agile Planner itself, Fitclipse, Green Pepper, Jazz connectivity, ...

Jazz supports this sort of sub-project association. This also makes it also possible to generate more detailed reports, such as

- How much time has been spent on a sub-project?
- Which sub-project needs the most attention?
- ...

7.2.4 Milestones

A milestone is a collection of tasks or goals which are combined into a single event. A milestone has been reached if all tasks, designated to this specific milestone, have been finished. Jazz offers support for milestones by associating iterations to milestones. This information can be used later to display how many iterations and therefore how much time has been spent to achieve the goal.

7.2.5 Advanced GUI-support

The GUI-support is rudimentary and not very attractive at the moment. Usability suffers and therefore the motivation to use it.

7.2.5.1 Configuration dialog

A configuration dialog would make the configuration easier and less error-prone. Besides a dialog to enter username and password, the configuration-requirements discussed in 7.2.1 are best represented by GUI elements.

7.2.5.2 Change History

If work item change history (refer to 7.2.2) is implemented, a dialog could offer all available versions and a quick preview of the entire story card. An additional option **restore** could offer a user-friendly way to restore previous states.

Chapter 8

Conclusion

The need for more advanced tools exists, such as tools which support synchronous distributed planning and reporting. Although no single tool encompasses all of these demands, the combination of these tools do. At the same time it is not necessary to reinvent the functionality that the chosen tool doesn t support. Agile Planner now successfully works with Jazz and uses Jazz features to make Agile Planner more valuable.

As a result, Agile Planner can be used to perform distributed synchronous planning meetings. Jazz can be used to track the progress of a project and to create reports, illustrated in Figure 8.1, 8.2, and 8.3.

Although Agile Planner Jazz integration caused a few problems, it took less effort to combine both system than to reinvent reporting capabilities from scratch in Agile Planner.

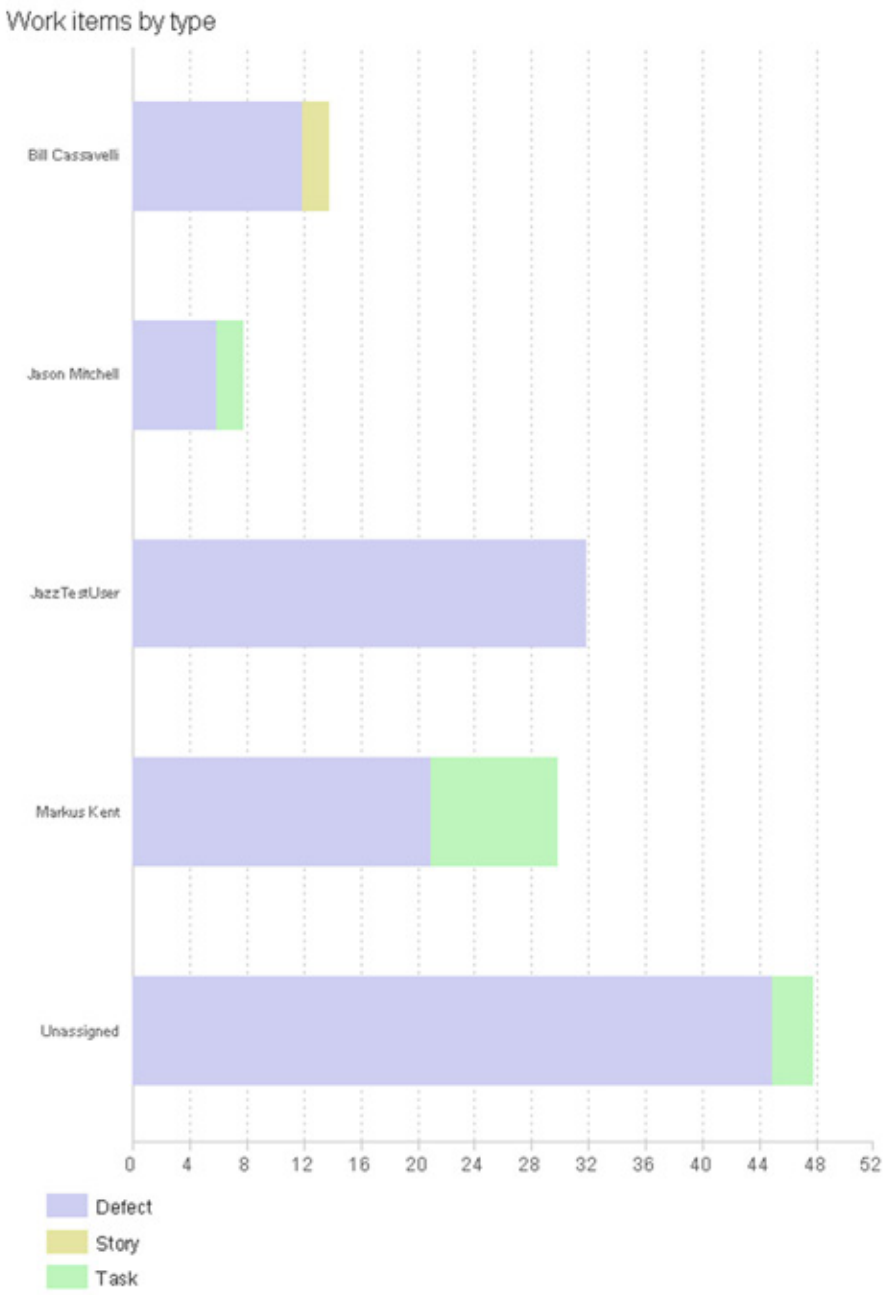

Figure 8.1: Report: number of tasks assigned to developers.

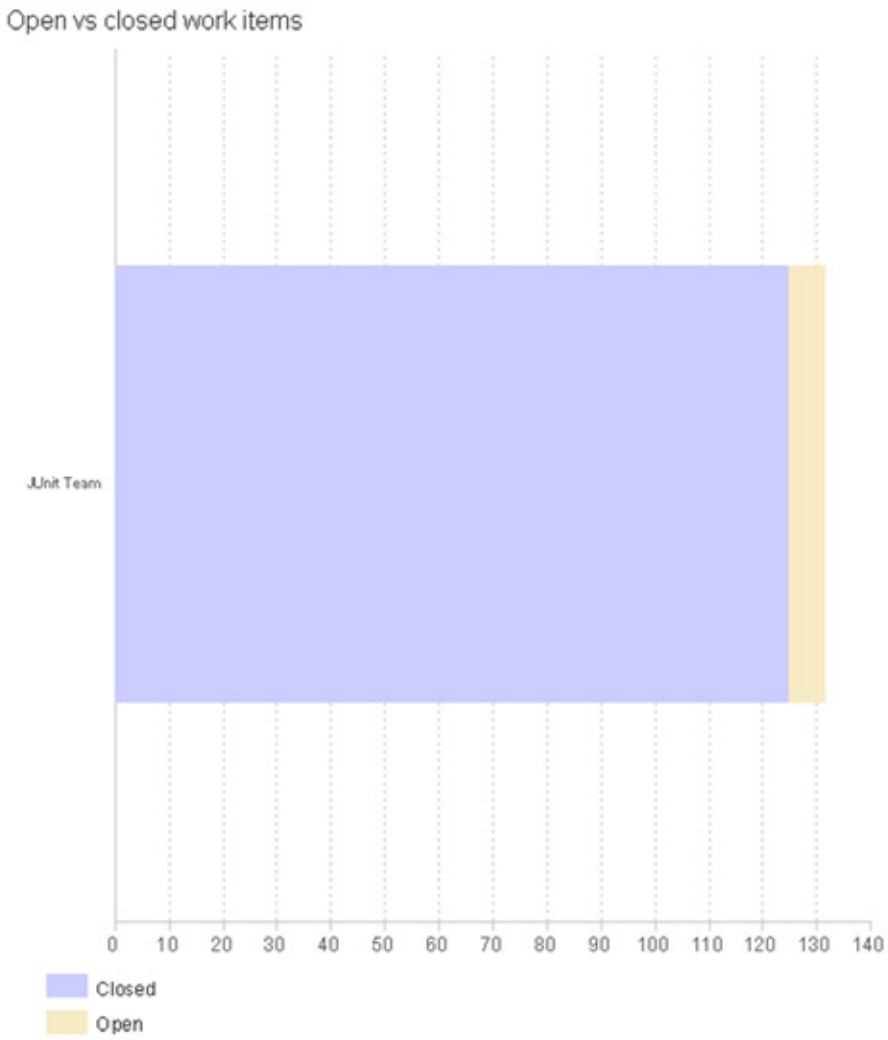

Figure 8.2: Report: number of open tasks vs. closed tasks.

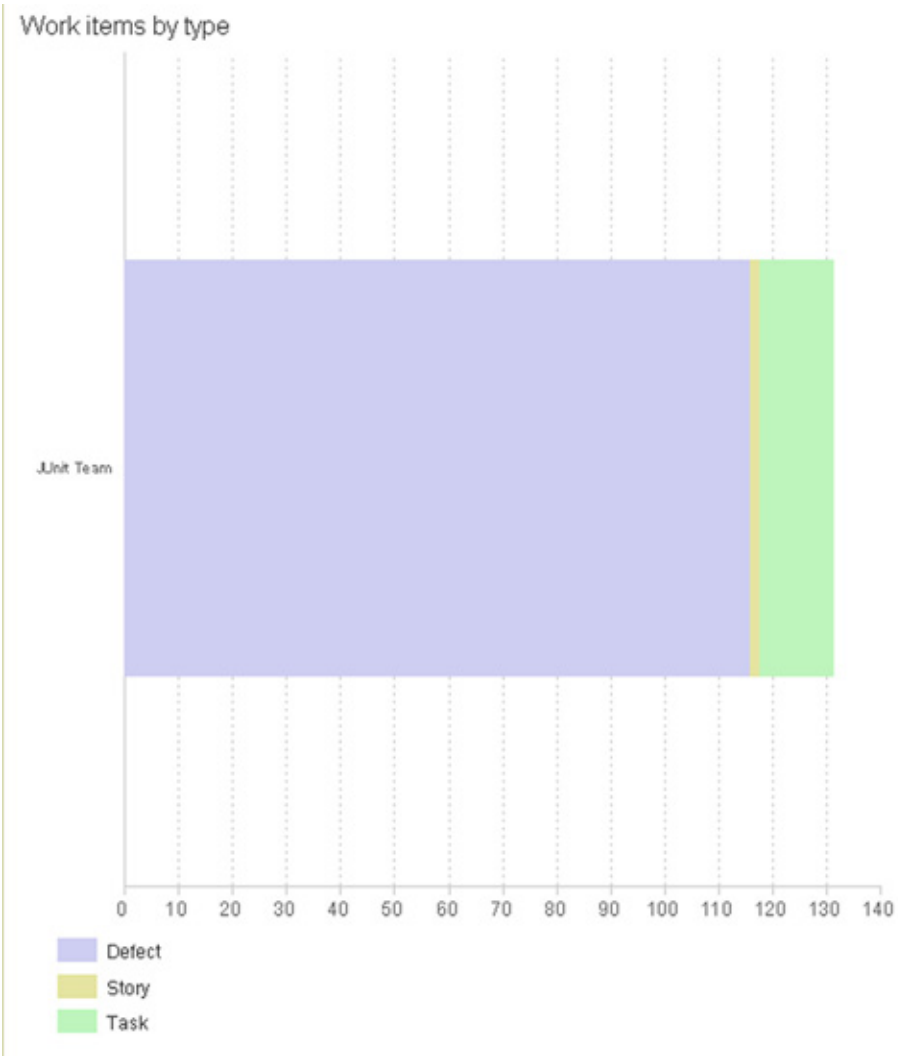

Figure 8.3: Report: number of distinct tasks differentiated by their type.

Appendix A

Use Case Description

This chapter documents use cases in a formal way. Each use case is described by several sections.

Summary A brief description of the use case.

Actors Several actors might be involved. Each use case is comprised of a primary actor that initiates the use case or provides essential information to start with. There might be one or more secondary actors who are involved in the use case.

Precondition All preconditions must be true to start a use case.

Description A brief description of the action. The sequence of actions will be executed in the same order as described. It describes the *behaviour* of the system and not the *implementation* itself.

Exceptions An exception can occur during execution. Each exception belongs to a line in the description field.

Postcondition Describes the condition if the execution stops, usually by execution of the last line of the description field.

The following use case example explains a login procedure.

Summary	Login to the system using username and password
Actors	User (primary), System (secondary)
Precondition	User s account must exist and be active
Description	[1] Users enters username and password into the fields on the login screen [2] Users presses **login**-button [3] System checks username and password
Exceptions	[3] Password is invalid
Postcondition	User has been logged in successfully or must provide login-information again when login-procedure was not successfull

A.1 Story Card added on Agile Planner

Summary	Add work item to Jazz when a story card has been added on Agile Planner
Actors	Agile Planner (primary), Jazz (secondary)
Precondition	Story card has been added successfully on Agile Planner, corresponding work item must not exist on Jazz
Description	[1] Check that corresponding work item does not exist [2] Create work item [3] Copy attributes from story card to the work item [4] Save work item on Jazz
Exceptions	[1] Corresponding work item already exists [4] Can not save work item
Postcondition	Work item has been added successfully

A.2 Work Item added on Jazz

Summary	Add story card to Agile Planner when a work item has been added on Jazz
Actors	Jazz (primary), Agile Planner (secondary)
Precondition	Work item has been added successfully on Jazz, corresponding story card must not exist on Agile Planner
Description	[1] Check that corresponding story card does not exist [2] Create story card [3] Copy attributes from work item to story card [4] Save story card on Agile Planner
Exceptions	[1] Corresponding story card exists [4] Can not save story card
Postcondition	Story card has been added successfully

A.3 Story Card deleted on Agile Planner

Summary	Story card has been deleted on Agile Planner, corresponding work item has to be set to resolved
Actors	Agile Planner (primary), Jazz (secondary)
Precondition	Story card counterpart exists on Jazz
Description	[1] Check that corresponding work item exists [2] Set corresponding work item to resolved [3] Save altered work item
Exceptions	[1] Work item does not exist [3] Can not save work item
Postcondition	Work item has been set to resolved

A.4 Story Card altered on Agile Planner

Summary	Story card has been altered on Agile Planner, Jazz counterpart has not been changed
Actors	Agile Planner (primary), Jazz (secondary)
Precondition	Story card exists on Agile Planner and it s counterpart on Jazz
Description	[1] Retrieve corresponding work item [2] Story card changes are copied to work item [3] Save altered work item
Exceptions	[1] Work item does not exist [3] Work item can not be saved
Postcondition	Jazz Work item has been updated

A.5 Work Item altered on Jazz

Summary	Work item has been altered on Jazz, Agile Planner s counterpart has not been changed
Actors	Jazz (primary), Agile Planner (secondary)
Precondition	Work item exists on Jazz and it s counterpart on Agile Planner
Description	[1] Retrieve corresponding story card [2] Copy changes to story card [3] Save altered story card
Exceptions	[1] Story card does not exist [3] Story card can not be saved
Postcondition	Agile Planner Story card has been updated

A.6 Work Item and the associated Story Card are simultaneously altered

Summary	Resolve conflicts and update Work item
Actors	Jazz (primary), Agile Planner (secondary)
Precondition	Work item and Agile Planner s counterpart has been altered simultaneously
Description	[1] Retrieve corresponding work item [2] Story card changes are copied to Jazz Work item changes will be overwritten [3] Save altered work item
Exceptions	[1] Work item does not exist [3] Work item con not be saved
Postcondition	Conflict has been resolved and work item has been updated

A.7 Iteration added on Agile Planner

Summary	Add previously created iteration on Jazz
Actors	Agile Planner (primary), Jazz (secondary)
Precondition	Iteration has been created on AgilePlanner
Description	[1] Check that iteration doesn t exist on Jazz [2] Create iteration on Jazz [3] Save iteration on Jazz
Exceptions	[1] Iteration already exists on Jazz [3] Can not save iteration on Jazz
Postcondition	Iteration has been added on Jazz

A.8 Story Card added to Iteration on Agile Planner

Summary	Add work item to an iteration on Jazz
Actors	Agile Planner (primary), Jazz (secondary)
Precondition	Story card and iteration exist on Jazz
Description	[1] Check that corresponding work item exists on Jazz [2] Check that iteration exists on Jazz [3] Set work item s iteration on Jazz [4] Save work item on Jazz
Exceptions	[1] Work item does not exist [2] Iteration does not exist [4] Work item can not be saved
Postcondition	Work item s iteration has been set to current iteration

A.9 Work Item added to Iteration on Jazz

Summary	Add story card to current iteration on Agile Planner
Actors	Jazz (primary), Agile Planner (secondary)
Precondition	Story card and iteration exist on Agile Planner
Description	[1] Check that corresponding story card exists on Agile Planner [2] Check that iteration exists on Agile Planner [3] Set story card s iteration on Agile Planner [4] Save story card on Agile Planner
Exceptions	[1] Story card does not exist [2] Iteration does not exist [4] Story card can not be saved
Postcondition	Story card has been added to iteration

A.10 Work Item removed from Iteration on Jazz

Summary	Remove the association of an iteration to a story card on Agile Planner
Actors	Jazz (primary), Agile Planner (secondary)
Precondition	Story card counterpart must exist
Description	[1] Check that story card exits [2] Remove iteration from story card [3] Save story card
Exceptions	[1] Story card doesn t exist [2] Story card can not be saved
Postcondition	Association with iteration has been removed

A.11 Story Card removed from Iteration on Agile Planner

Summary	Remove the association of an iteration to a work item on Jazz
Actors	Agile Planner (primary), Jazz (secondary)
Precondition	Work item counterpart must exist
Description	[1] Check that work item counterpart exists [2] Remove iteration from work item [3] Save work item
Exceptions	[1] Work item doesn t exist [2] Work item can not be saved
Postcondition	Association with iteration has been removed

Appendix B

Acronyms

API Application Programming Interface

BIRT Eclipse Business Intelligence and Reporting Tools

GUI Graphical User Interface

Bibliography

[ACM] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns*. Prentice Hall, second edition.

[CBC+06] Fang Chen, Robert O. Briggs, Gail Corbitt, Jay F. Nunamaker Jr., James Sager, and Stanley C. Gardiner, editors. *Project Progress Tracking Template - Using a Repeatable GSS Process to Facilitate Project Process Management*, 2006.

[Kru] Philippe Kruchten, editor. *Architectural Blueprints - The 4+1 View Model of Software Architecture*.

[Sie] Johannes Siedersleben. *Moderne Softwarearchitektur*. dpunkt.verlag, first edition.

Index