NTNU
Norwegian University of
Science and Technology

Achieving loose coupling in the component-based Miles software development Platform

A Proof of Concept

Erling Wegger Linde

Master of Science in Computer Science
Submission date: June 2008
Supervisor: Maria Letizia Jaccheri, IDI
Co-supervisor: Andre Heie Vik, Miles AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This Master Thesis is intended to implement and evaluate the Proof of Concept that was planned in the Specialization Project conducted by Erling Wegger Linde, the fall 2007.

The Proof of Concept involves implementing a layered interface/intermediate/wrapper between an issue tracker and its surrounding dependent components. The layered interface shall use the following technologies and architectural styles:

*REST
*Atom feeds and the Atom Publishing Protocol
*Semantic Web technology

Dependent components of the issue tracker are Eclipse, Hudson and customised Web Interfaces which will be used by developers, project managers and customers.

This Master Thesis are intended to answer the following research questions:

1. To which extent is it possible to implement the different layers of the Proof of Concept?
2. How does the addition of each layer affect the degree of coupling?
3. How does these layers affect the amount of work spent on replacing a component?

Assignment given: 15. January 2008
Supervisor: Maria Letizia Jaccheri, IDI

Abstract

The overall aim of this Master Thesis was to achieve a long life-time for the Miles Platform by enabling loose coupling between its various components and tools. The Miles Platform is a software development platform consisting of several interconnected tools and components. For this platform to survive future changes in technology it must be possible to replace the various components without requiring large changes to the surrounding dependent components.

Based on a preceding prestudy, a layered Proof of Concept was implemented and evaluated with respect to the success of the implementation, coupling (modifiability) and amount of work. Simple prototypes involving the two first layers, namely the RESTful and the Atom Publishing Protocol based layers were implemented. The final prototype included all three layers, which involved using Semantic Web technology in addition the RESTful Atom Publishing Protocol. Only a few non-blocking issues are unresolved for this final prototype.

Both the RESTful, Atom Publishing Protocol and Semantic layer contributed to maintaining a stable interface on top of the issue tracker component. By enforcing stable interfaces and serving as wrappers or intermediares between the issue trackers and their dependent components, each layer contributed to achieving looser coupling, if not in all dimensions.

If the issue trackers have many dependent components that needs customization in order to communicate with the issue trackers, and the issue trackers are expected to be replaced one or preferably several times, a positive return on investment are expected from this Proof of Concept.

Preface

This work is a contribution to the Miles software development Platform that is being developed by Miles. This Master Thesis is a continuation from my work in a preceding specialisation project [9]. Based on a literature study, the specialization project illustrated a Proof of Concept that could help to achieve loose coupling in the Miles Platform. This previous work was intended to act as a foundation for this Master Thesis.

Acknowledgements

I want to thank my supervisor at Miles, André Heie Vik, for his supporting enthusiasm. You and everyone else at Miles have made this project a great joy.

Furthermore, I want to thank Henry Story for initiating and welcoming me on the Baetle Project. You have helped me a lot with your ideas and guidance. I also want to thank all other communities and people on mailing lists such as REST-Discuss, Jena-dev, ROME, Trac, Grails, Hudson and probably many more.

Finally, a big thanks to my dear Anne-Linn for supporting me throughout this exciting process.

Contents

I Introduction and Design 3

1 Introduction 5
1.1 Problem statement 5
1.2 Aim 5
1.3 Scope 5
1.4 Overview 6

2 Background 7
2.1 Prestudy 7
2.1.1 Miles Platform 7
2.1.2 REST 8
2.1.3 Atom Publishing Protocol 9
2.1.4 The Semantic Web 10
2.2 Related Work 11
2.3 Tactics 11
2.4 Conclusion 12

3 Research Design 13
3.1 Research Questions 13
3.2 Development Method 14
3.3 Comparing Coupling 14
3.3.1 Framework 15
3.4 Estimating work 17
3.5 Conclusion 17

II Results 19

4 Initial scenario 21
4.1 Design 21
4.2 Coupling 22

4.3 Work 25
4.4 Conclusion 26

5 RESTful layer 27
5.1 Design 27
5.2 Issues 28
5.2.1 Oustanding issues 29
5.2.2 Resolved issues 29
5.3 Coupling 29
5.4 Work 33
5.5 Conclusion 33

6 APP layer 35
6.1 Design 35
6.2 Issues 38
6.2.1 Outstanding issues 38
6.2.2 Resolved issues 38
6.3 Coupling 39
6.4 Work 42
6.5 Conclusion 42

7 Semantic layer 43
7.1 Design 43
7.1.1 Sparql Endpoint 44
7.2 Issues 46
7.2.1 Oustanding issues 46
7.2.2 Resolved issues 46
7.3 Coupling 46
7.4 Work 49
7.5 Discussion 50
7.6 Conclusion 50

III Synthesis 51

8 Discussion and conclusions 53
8.1 Summary 53
8.2 Issues 54
8.3 Coupling 55
8.4 Work 56
8.5 Research approach 56

8.6 Further work 58

IV Appendix 65

A Template for coupling discussion 67

B Test Environment 71
B.1 Jira 71
B.1.1 Jira Plugin 71
B.2 Trac 72
B.3 Project and Customer Web 72
B.4 Hudson 72
B.4.1 Plugin 72
B.5 Eclipse 73
B.5.1 Mylyn Plugin 73
B.6 Subversion 73
B.6.1 Expand log script 73
B.7 Wrapper layer 74
B.8 Conclusion 74

C Source Code 75

Part I

Introduction and Design

Chapter 1

Introduction

This chapter will present the problem statement as well as present the aim and scope for this Master Thesis. Section 1.1 presents the problem statement. Section 1.2 gives the aim of this Master Thesis. Section 1.3 limits the scope, while Section 1.4 gives an overview of this document.

1.1 Problem statement

The Miles Platform is a software development platform consisting of several interconnected tools and components. For this platform to survive future changes in technology it must be possible to replace the different components without requiring large changes to the surrounding dependent components. By achieving loose coupling between these components one can hopefully extend the life-time of the Miles Platform, as well as reduce the amount of time and money needed to implement and deploy a change.

1.2 Aim

The overall aim of this Master Thesis is to ensure a long life-time of the Miles software development Platform by enabling loose coupling between its different components and tools.

1.3 Scope

Previous to this Master Thesis I wrote a Specialization Project [9] where a Proof of Concept was planned. This Proof of Concept suggested to use a layered approach based on various specified technologies between an issue tracker and its surrounding components (described in Chapter 2). It is within the scope of this Proof of Concept I will try to reach the aim of this Master Thesis.

Furthermore, this Master Thesis is not intended to be a case study in order to find out if elements from this Proof of Concept can enable loose coupling on a more general level, although this might be the case. The goal is primarily to deal with loose coupling, not other quality attributes such as performance, concurrency and security.

1.4 Overview

- This chapter presented the aim deduced from the problem statement, as well as the scope for this Master Thesis.
- Chapter 2 presents relevant background material from the preceding Specialization Project [9], as well as discusses related work and methods.
- Chapter 3 presents research questions and methods.
- Chapter 4 presents the initial situation before any of the layers are implemented.
- Chapter 5 presents Layer 1, the RESTful Layer, and discusses its effect on coupling etc.
- Chapter 6 presents Layer 2, the Atom Publishing Protocol (APP) Layer, and discusses its effect on coupling etc.
- Chapter 7 presents Layer 3, the Semantic Layer, and discusses its effect on coupling etc.
- Chapter 8 discusses and concludes this Master Thesis, as well as suggests further work.

Chapter 2

Background

The previous Chapter 1 presented the aim and scope of this Master Thesis. The goal of this chapter is to provide relevant background material. Section 2.1 summarizes the most important topics from the preceding prestudy. Section 2.2 presents related work, while Section 2.3 discusses tactics. Finally, Section 2.4 concludes the chapter.

2.1 Prestudy

In advance of this Master Thesis I conducted a prestudy: [9]. This prestudy presented the Miles Platform as well as different technologies that could be used to enable loose coupling in this Software Development Platform. Furthermore, it illustrated a layered approach that was intended to be implemented as a Proof of Concept in this Master Thesis. A set of research questions were also deduced. To give the current reader some more insight, a summary of the most important parts is presented here. The research questions are however not presented until Section 3.1.

2.1.1 Miles Platform

The Miles Platform is to be an efficient and long-lived software development platform. It will consist of both Open Source Components as well as In-House developed applications. A key concept is to provide efficient tools for work and communication to both the project manager and developers as well as customers. Code repositiories are intended to boost up start-up time for a project, while automatic building tools and support for efficient development processes could increase effectivity. Requirements management and customer interfaces are intended to boost satisfaction of customers, and at the same time avoid discussions due to lack of information.

Many of the platform's components will be interconnected in various ways. Over time it is also very likely that new components will replace others. If a lot of components depend on a component that is replaced or upgraded, this could mean that a lot of work has to be carried out in order to successfully integrate the new component. In order to reduce the amount of work one can hopefully find a way to reduce the coupling between the different components. This goal coincides with the aim of this thesis.

Issue Tracker

The part of the Miles Platform that was considered to be most relevant to first enable loose coupling was between the issue tracker and its surrounding components. An issue tracker is a tool that helps a project team to keep track of bugs and other issues that needs to be dealt with.

In the Miles Platform the issue tracker is referred to by several components. The developers uses an IDE or Integrated Development Environment. Currently Eclipse [7] is the preferred IDE. Eclipse can be integrated with several issue trackers, allowing the programmer to monitor and report issues while programming. Furthermore, an automatic building tool such as Hudson [17] scans the commit logs from code repositories such as Subversion [33] in order to find issue keys mentioned. If an issue key is mentioned in the commit log, Hudson can provide links to relevant issues, as well as post comments to the related issues regarding the status of the builds etc.

Components called Project Web and Customer Web are additional intended components for the Miles Platform. These components are yet not specified, but may be developed in-house by Miles. They should provide useful interfaces to both developers and customers, and allow them to monitor and change requirements etc. These web interfaces are also intended to be connected to the issue tracker, for instance by showing which issues relates to which requirements etc.

Besides from the components that depends on the issue tracker, the issue tracker itself monitors Subversion commit logs as well in order to find code and revisions that relates to different issues. There are currently two issue trackers of interest for the Miles Platform. Jira [19] is a mature issue tracker being used today, while Trac [36] is a potential future candidate to replace Jira.

2.1.2 REST

REST or REpresentational State Transfer can be explained best as an Architectural Style. The term REST was introduced by Roy Thomas Fielding in his Phd Thesis [27]. REST is actually based on the successful architectural principles that are used by the internet today. The concept is based on *Resources* that can be referred to by *URIs*. One can for example refer to the resource “Issue Tracker” in the Miles Platform as: *http://miles.no/issuetracker*. However, although the URI refer to the “Issue Tracker”-resource the data you receive when doing a HTTP GET to that URI might be a HTML document. In REST such a document is not considered to be the resource itself, but a *Representation* of it.

A constraint of REST is that the server side should be entirely stateless. This breaks with the modern web’s use of cookies etc, but it provides a huge potential for scalability. For instance a resource can be accessed with HTTP GET, updated with HTTP UPDATE, created with HTTP POST and deleted with HTTP DELETE. Out of these four operations, three of them are *idempotent*, which means that the response from the server will not change whether the operations are executed one or hundreds of times. A resource can only be deleted once, and since the new representation of a resource is always included in the HTTP UPDATE the server will always know if the update has already been made. This implies that every operation except from HTTP POST can be cached. This is also one of the most argued benefits of REST ws. SOAP-based web service, as every SOAP operation uses a HTTP POST.

Finally, the term “State Transfer” indicates that a client navigates through *states* by traversing representations using URIs. For instance a client can be said to move from State1 to State2 if it jumps from http://www.example.org/state1 to http://www.example.org/state2. Such URLs should be provided as hyperlinks contained in the representation of the resources in order to indicate possible state transitions to a client.

RESTful Layer

The prestudy [9] suggested that a RESTful layer should be added on top of the Issue Trackers. Although a Wrapper Layer was indicated as a convenient Layer 0 (see Section B.7), this RESTful layer should be the first externally visible layer that enables a looser coupling between an Issue Tracker and its dependent components. By providing a layer between the issue tracker and its surrounding components, one could potentially reduce the direct coupling a great deal. Furthermore, by providing the constraints of REST one should have a limited set of operations available to use on the representations belonging to the issue tracker, and hence limit the effects of a change to the clients.

2.1.3 Atom Publishing Protocol

The Atom Publishing Protocol is a RESTful protocol for interacting with the Atom syndication feed format [2]. Atom is a content syndication format that was created in response to the mess of different RSS formats. The basics of the Atom format is a Feed that contains Entries. The Atom Publishing Protocol describes how one could for instance fetch a collection using HTTP GET on *example.org/feed*, or do a HTTP PUT *example.org/feed/entry1* to update entry1 belonging to the feed.

The Atom format contains several default elements such as title, created, published etc. that are common attributes for many types of data. Some of these attributes may overlap with the attributes of data belonging to an issue tracker, hence one can take advantage of these built-in semantics of Atom and at the same time provide a RESTful interface to the issue tracker.

APP Layer

Using the Atom Publishing Protocol as a basis for the RESTful interface to the issue tracker can help structure the data in a standardized way, making it even easier for clients to understand the data.

2.1.4 The Semantic Web

The Semantic Web was invisioned by Tim Berners-Lee et al. [35] as a web where the computers can understand the meaning of the data. Ontologies are key elements to make this possible. They allow us to describe concepts of different domains and how they are related. Ontologies can be expressed in languages such as RDF(S) or OWL. By adding information from ontologies as some form of metadata to different representations of a resource, a client can actually understand the meaning of this resource - creating huge possibilities for interoperability.

In addition to the ontology languages RDF(S) and OWL, SPARQL is a query language that can be used to query "semantic data". Such queries could be a very powerful tool that can find results across different, but somehow related ontologies. Another powerfol feature of the Semantic Web is inference; for instance if an individual is a member of class A, and class A is a subclass of class B, then it can be inferred that the individual is also a member of class B. Other much more powerful inferred statements can be made from many other RDF(S) and OWL statements, for instance OWL restrictions. [5]

Semantic Layer

Even if the Atom Publishing Protocol has standard elements for describing some common attributes to data, the data belonging to an issue tracker might have a lot of other attributes. If one could express these attributes in an ontology that could be understood by other applications, one could increase the possibility for new or external applications to understand this data.

2.2 Related Work

The **Google Data APIs** or **GDATA** is an example of how one can extend the Atom Publishing Protocol to expose all sorts of data. GDATA has also dealt with different issues such as concurrency. However, the way Google has extended the Atom Publishing Protocol indicates that GDATA is almost a new protocol on its own. It is possible that GDATA actually becomes as widespread as the Atom Publishing Protocol but one might suspect that sticking with the original Atom Publishing Protocol makes it easier for clients to understand your data. [13]

Queso is a RDF server where you can post data in the form of Atom entries. The data is then stored in RDF triples, and it is also possible to query it through a SPARQL endpoint [8]. There is however only minimal information to find on this server through various blog entries. [38]

Henry Story has created the **AtomOwl Vocabulary Specification** which is an ontology that describes the Atom format.[15]

Baetle is short for Bug And Enhancement Tracking LanguagE and is an on-going open source project that aims to develop an ontology to describe issues and their properties, which is highly relevant for the Semantic Layer of this Proof of Concept.[16]

Semap is a server where you can store and retrieve data through a RESTful interface. Semap also provides the possibility to query data related to a resource using SPARQL. Semap is however dependent on that you inject data into it, hence it can't easily be layered on top of an issue tracker. [30]

2.3 Tactics

Len Bass et al. [20] describes different tactics to prevent ripple effects. Two tactics that can be related to this Proof of Concept are "Maintain existing interfaces" and "Use an intermediary". The goal of the first tactic is to keep the interface of a component or service stable, so that dependent components do not need to change. The second tactic involves inserting a layer between a component and its dependants, in order to deal with the dependencies. It can be argued that "use an intermediary" is actually a method to "maintain existing interfaces", hence the two tactics may in fact overlap. Both tactics are well known and widely used, however little work has been done involving the combination of technologies chosen for this Proof of Concept, at least not within the same scope as this Master Thesis.

2.4 Conclusion

There is a need for enabling loose coupling in the Miles software development Platform in order to ensure it a long life time. Furthermore, no similar combination of REST, Atom Publishing Protocol and the Semantic Web is available for putting directly on top of the issue trackers. Hence these layers will have to be implemented in order to demonstrate or evaluate this Proof of Concept. The Proof of Concept will consist of a layered intermediary or wrapper between issue trackers and their dependent components, such as an IDE or a building server.

Chapter 3

Research Design

The previous Chapter 2 introduced the context and background for this Proof of concept. In order to find out whether such a Proof of concept can help to reach the aim stated in 1.2 I need a set of reserach questions to answer, as well as methods and metrics that could be used to answer these. Section 3.1 presents the research questions found in [9]. Section 3.2 presents the development method that will be used when developing this Proof of Concept. Section 3.3 presents a framework for comparing coupling between the different layers. Section 3.4 presents a method for estimating the work effort required to adapt to a change to or of the issue tracker. Finally Section 3.5 concludes the chapter.

3.1 Research Questions

The following research questions were suggested in my previous prestudy [9]. They are still considered to be the best candidates, so that answering them will help me conclude to what degree I have reached the aim of this Master Thesis or not.

1. **RQ1:** To which extent is it possible to implement the different layers of the Proof of Concept? By answering this question one can prove whether it is possible to implement either parts of or the entire Proof of Concept.
2. **RQ2:** How does the addition of each layer affect the degree of coupling? Answering this question will state how each layer affects the coupling, although in combination with lower layers.
3. **RQ3:** How does these layers affect the amount of work spent on replacing a component? Answering this question would give an indication of whether applying the layers to the platform would be a good investment in the short or long run.

3.2 Development Method

The development method that will be used when implementing this proof of concept is called Evolutionary Prototyping [10]. The basic principle of Evolutionary Prototyping is to start with a simple prototype that fulfills some of the few known requirements. When this prototype is created and new requirements are revealed, perhaps due to an increased understanding of the system, the prototype can be extended and refined. Eventually the prototype turns out to be the final system.

When developing this Proof of Concept the known requirements are to create three layers that provides access to the underlying issue trackers. I am familiar with the overall architecture of these layers as well as a few details about the API and datamodels of the issue trackers, but many details most be uncovered. I will start by creating a vertical prototype for each layer, that means that just a few attributes, such as the summary of an issue can be accessed in the beginning. Eventually new attributes will be supported, as well as options to add, edit or delete issues etc. Note that due to time constraints, only the last layer (which is assumed to incorporate the two previous layers) will be extended significantly both horisontally and vertically.

Furthermore as stated in [10]; “For a system to be useful, it must evolve through use in its intended operational environment”, there is a need to develop the Proof of Concept within a realistic environment. However, since not all the surrounding components of the Miles Platform yet exist, and that only Jira has been used etc., I decided to set up a Test Environment on my local machine. In this Test Environment both the issue trackers, as well as instances of the surrounding components needs to be installed and adapted to work with this Proof of Concept. This could also involve development of several Plugins to these components. Some prototype of the Project or Customer web must also be created. The test environment are described in Appendix B.

Finally, in order to answer **RQ1**, a list of outstanding and solved issues shall be given for each layer. The outstanding issues should be described detailed enough to indicate whether it is architectural constraints, limitations of libraries or lack of time that prevents them from being solved.

3.3 Comparing Coupling

In order to answer **RQ2**, a set of metrics and methods to measure the achieved coupling is needed. I have found several papers that presents different metrics and methods for measuring and calculating coupling. However, many metrics apply only to programs written in functional programming languages where one has to count different types of in and out parameters, for instance as described in

Franck Xia [12] and Gregory A. Hall et al. [14]. The majority of the other metrics found are directed towards the Object-Oriented paradigm such as Martin Hitz and Behzad Montazeri [21]. That existing coupling metrics are mostly classified into metrics for Procedural and Object-Oriented programs was also recently confirmed by Jarallah S. Alghamdi [18].

Charles Zhang and Hans-Arno Jacobsen [3] approaches the Web Service paradigm by suggesting that metrics can be captured based on data mining from CORBA IDL or WSDL files. However, the planned RESTful Semantic Web Service does not have a formal service description (i.e. WADL) to capture these characteristics from. Furthermore, Anthony M. Orme et al. [1] presents three different metrics to predict coupling for ontology-based systems. These metrics are calculated from the number of references, imports etc. an ontology has to other ontologies. However, this measure cannot be used to compare the different layers for this Proof of Concept due to the fact that only the final layer is directly based on ontologies.

Mikhail Perepletchikov et al. [23] provides a model for describing service oriented systems. In Mikhail Perepletchikov et al. [22] the same authours [1] present a set of metrics for measuring coupling in Service-Oriented designs. They argue that existing metrics are not directly applicable to service-oriented designs and hence they decided to develop such metrics themselves. They present nine different metrics to indicate coupling for the Service-Oriented paradigm. However, an initial evaluation I performed on these metrics, indicated that these metrics are not sufficient to separate the three layers of this Proof of Concept from each other, although they do support the idea of putting a wrapper on top of the issue tracker.

Len Bass et al. [20] presents a set of tactics to improve modifiability. They state that the goal of the modifiability tactics are to ensure that future changes to a system occurs within an affordable amount of time and cost. Furthermore, they specify a set of eight types of dependencies between modules of a system. I believe that this set of dependencies can serve as a basis for comparing the effect on coupling from each one of the layers.

3.3.1 Framework

Based on [20] I have created a framework for comparing coupling between the different layers. The input to the framework is specified in the following scenarios:

SCENARIO1: An issue tracker is replaced with another issue tracker

SCENARIO2: The issue tracker's interface changes due to an update or upgrade

SCENARIO3: The implementation of the Proof of Concept changes within the constraints of its architecture.

[1]Except for Heinz Schmidt / Zahir Tari

SCENARIO1 is a scenario that are likely to occur in the future. Perhaps Jira are replaced with a the less expensive alternative Trac. If many components that previously communicated with Jira must be changed and adapted to communicate with Trac, this could cost a lot of time or money. If it is to expensive, perhaps one is stuck with Jira, or worse the platform cannot be used without major changes.

SCENARIO2 could involve new attributes added for an issue, or changes to the remote api etc. If dependent components have to be changed for each minor update this could be very expensive in the long run.

SCENARIO3 goes beyond the changes to the issue trackers themselves. It also deals with the changes that programmers can do to the Proof of Concept in the future. The architecture or technologies used in the Proof of Concept might limit the options a future developer has in order to change the Proof of Concept. This could help clients anticipate the scope of future changes.

The effect of these scenarios should then be discussed agains the following eight dependency types from [20]:

D1: **Syntax** of data and service. The type or format of data as well as signature of services must be consistent with a client's assumptions.

D2: **Semantics** of data and service. The semantics of data and services must be consistent with a client's assumptions.

D3: **Sequence** of data and control. Either the sequence of data are important, or there exists timing constraints.

D4: **Indentity** of an interface. The name or handle of a service's interface must be consistent with a client's assumptions.

D5: **Runtime location** of a module/service. The runtime location of a module/service must be consistent with a client's assumptions.

D6: **Quality of service/data**. The quality of data or service must be consistent with a client's demands.

D7: **Existence** of a module/service. The module/service must exist.

D8: **Resource behavior** of a module/service. The resource consumption or ownership of a module/service must be consistent with a client's assumptions.

Such a discussion should point out benefits and drawbacks from implementing the different layers of this Proof of Concept. A template for such a discussion can be found in Appendix A and should be used when evaluating each of the layers.

3.4 Estimating work

The implementation of the three layers of this Proof of Concept could be considered as tactics to prevent ripple effects [20]. Hence the work that needs to be carried out when a new issue tracker should be supported would hopefully be limited to only a small part of the system. This amount of work should then be less than or equal to the amount of work that is needed without this Proof of Concept, at least in the long run, for it to be economically feasible. A comment on the amount of work needed when an issue tracker is changed should then be given for each layer.

3.5 Conclusion

Outstanding issues should be presented for each layer in order to answer to what degree it is possible to implement the different layers. A framework for measuring or comparing coupling between the different layers of the Proof of Concept were not easy to find, but were eventually created. Each layer should be evaluated against this framework. Finally, a comment on the amount of work required when any of the scenarios should occur should also be given for each layer. The latter should indicate the possible return on investment of this proof of concept.

Part II

Results

Chapter 4

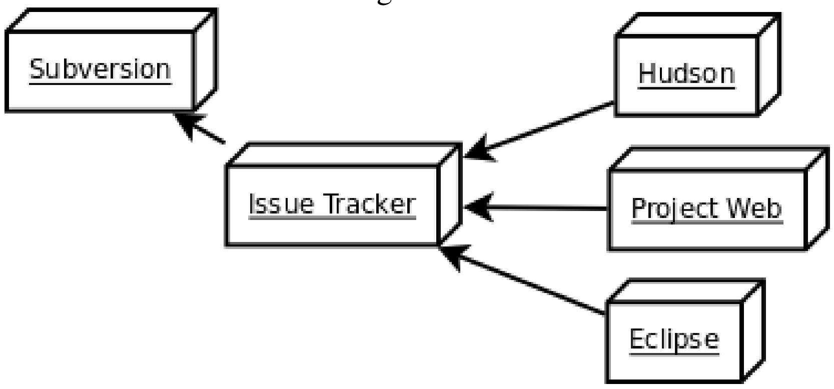
Initial scenario

This chapter will discuss the initial coupling and effect of a change before the Proof of Concept is implemented. This chapter will hence serve as a baseline to which the above layers could be compared to. This discussion is made on the background of my gained familiarity with Jira and Trac's interfaces.

4.1 Design

Figure 4.1 illustrates the initial scenario where Hudson, the Project Web and Eclipse all are using the Issue Tracker directly. The issue tracker itself can be connected to Subversion in order to link issues to relevant code and revisions.

Figure 4.1: Initial Scenario



4.2 Coupling

This section will discuss this layer's effect on coupling within in the framework presented in 3.3. Recall the scenarios from Section 3.3:

SCENARIO1: An issue tracker is replaced with another issue tracker

SCENARIO2: The issue tracker's interface changes due to an update or upgrade

SCENARIO3: The implementation of the Proof of Concept changes within the constraints of its architecture.

D1: Syntax of data and service. The type or format of data as well as signature of services must be consistent with a client's assumptions.

SCENARIO1: The type and format of data can change completely if an issue tracker is replaced with another. The same goes for the the services signature.

SCENARIO2: If the issue tracker is upgraded or updated, it is unlikely that its interface will change completely, at least some backward compatibility are expected. However in the long run Trac could for instance get a RESTful interface instead of the provided XML-RPC interface available today. So the effect on the syntax are likely to increase over time. Taking Jira's SOAP interface as an example, it would probably continue to provide a SOAP API, although new APIs might be added or gain more focus. The methods and parameters would certainly change, while it is more likely that XML is kept as the preferred format of exchanged data. Hence method names can change, as well as input parameters etc., but the format will most likely remain as XML.

SCENARIO3: The only architectural constraint that exists on this initial scenario is that other components or tools needs access to the data used by an issue tracker. This could be through a remote API, e.g. XML-RPC, SOAP or REST. In theory it could also be more directly for instance by accessing the database. Hence it is very hard for a client to predict future changes.

D2: Semantics of data and services. The semantics of data and services must be consistent with a client's assumptions.

SCENARIO1: The semantics of the data and service can of course change completely if an issue tracker is replaced. However it is likely that most issue trackers will have many concepts in common, although specific methods that are related to workflow etc. can change dramatically from issue tracker to issue tracker. The problem is of course if methods or attributes with similar names actually have completely different semantics.

SCENARIO2: It is unlikely that the semantics of data and methods will change significantly due to an update. There's of course a potential that some methods will create new side-effects etc. Hence the client may find itself executing a method that gives a different result than what it expected.

SCENARIO3: Potentially, a method called changeStatus(newStatus) can perform a completely different operation than before, it is however unlikely that significant changes will occur, but there is no constraints in e.g. SOAP or XML-RPC etc. which can prevent a programmer from making such drastic changes. Perhaps even more important is the fact that various issue trackers might provide a very different set of methods, hence it could be very difficult to figure out which method to use in order to get the same results as with another issue tracker.

D3: Sequence of data and control. Either the sequence of data are important, or there exists timing constraints.

SCENARIO1: Different workflow models etc. can require that different sequences of methods are called in order to achieve the intended status.

SCENARIO2: Again, changes to the workflow model can affect the sequence of methods one needs to execute in order to get the intended result.

SCENARIO3: As there are few constraints limiting the scope of change for sequence of data and control on this initial scenario, almost anything could change.

D4: Identity of an interface. The name or handle of a service's interface must be consistent with a client's assumptions.

SCENARIO1: The endpoint adress are very likely to change for a new issue tracker, as these endpoint adresses are often URLs using some implementation details. E.g. like trac has ../xmlrpc/. One could of course use a standard URL and redirect this to the specific endpoint.

SCENARIO2: Different versions of interfaces can lead to new endpoint adresses e.g. Jira's WSDL are fetched from "jirasoapservice-v2?wsdl" which clearly indicates that this is related to a specific version of the remote interface. However, this also implies that one should be able to continue to use the old interface even if a new one is added.

SCENARIO3: The endpoint adress to the remote interfaces can potentially change to almost anything.

D5: Runtime location of a module/service. The runtime location of a module/service must be consistent with a client's assumptions.

As long as the issue tracker is available on the inter- or intranet it should be irrelevant to clients which CPU or whatever hardware the issue tracker uses. It is important though that the URL to the issue tracker's API doesn't change even if the issue tracker is moved to another server.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D6: Quality of service/data. The quality of data or service must be consistent with a client's demands.

SCENARIO1: Bad response times may lead to a too early time out at the client application.

SCENARIO2: Hopefully updates to an issue tracker shouldn't affect quality of service negatively, although this could be the case if for instance a new version is released early.

SCENARIO3: Again, within the wide scope of change for this scenario, the quality of service or data can change both to the better or for the worse.

D7: Existence of a module/service. The module/service must exist.

SCENARIO1: If a new issue tracker is installed, the old one is likely to be removed, and hence will not exist anymore. If no issue tracker is installed or used, the surrounding components must be able to continue to work. The data belonging to the old issue tracker might be lost if no action is taken.

SCENARIO2: An update doesn't remove the issue tracker, although it could be unavailable for a certain amount of time.

SCENARIO3: -

D8: Resource behavior of a module/service. The resource consumption or ownership of a module/service must be consistent with a client's assumptions.

As the amount of dependent components increase, the network traffic can increase too, otherwise a client should not be affected of an issue tracker's resource consumption.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: .

4.3 Work

With respect to the three scenarios, a lot of work would be needed to be carried out for each of them. However, that depends also on the issue tracker, if for instance Jira are replaced with Trac, then one has two options; develop a plugin for Eclipse and Hudson yourself or use the already existing ones, which certainly would be the obvious choice here. However, there is not guaranteed that there will exist such plugins for all future issue trackers.

For the projectweb and other potential in-house developed applications, one would have to rewrite or replace the part of the code which communicates with the issue tracker. Furthermore, it is possible that a tight integration with the issue tracker's architecture could require that other parts of these applications would need to be rewritten as well. Table 4.1 illustrates the units of work that has to be carried out with regards to the number of dependent components and the number

Table 4.1: Work

#Dependants	Initial issue tracker	First replacement	Second replacement
1	1	2	3
2	2	4	6
3	3	6	9
4	4	8	12

of times an issue tracker is replaced. This model presumes that one has to modify the dependent components, i.e. not use existing plugins. I chose to define the amount of work as “units of work”. What a unit of work really is, is impossible to determine. Writing code that communicates with an issue tracker would depend a lot on the remote interface of the actual issue tracker. Furthermore, it would also depend on how much information and/or functionality a client would use. Hence, a generic “unit of work” seemed the most intuitive and correct way to use for measuring or estimating work.

4.4 Conclusion

The changes that may have to be made to the clients (dependent components) can be very drastic as many things can potentially change with respect to the three scenarios. Hence it should be possible to reduce the likelihood of having to make changes to the dependent components if some of the three scenarios should occur. The amount of work required were estimated as generic units of work. This chapter could serve as a baseline for comparing the three layers of this Proof of Concept against.

Chapter 5

RESTful layer

The previous Chapter 4 presented the initial situation before any part of this Proof of Concept was implemented. This chapter will present the first layer, namely the RESTful layer. Section 5.1 will present the design of this layer. Section 5.2 will present and discuss both solved and outstanding issues. Section 5.3 will discuss this layer's effect on coupling while Section 5.4will discuss how this layer could affect the amount of work that must be carried out on this layer and the depending components. Finally, Section 5.5 will conclude the chapter.

5.1 Design

The overall goal of this layer is to provide a generic RESTful interface to any issue tracker. The first thing that was done was to create a URI scheme for the resources. First of all a URI was needed in order to refer to an issue. Furthermore, issues do belong to a project. The following URIs were possible to refer to:

- /projects/{project}/config/ - Referring to the configuration of the project.
- /projects/{project}/issues/ - Referring to all issues belonging to the project.
- /projects/{project}/issues/{issue}/ - Referring to a specific issue belonging to the project.

Restlet [25] is a Java framework that was used as a basis for implementing this layer. By fetching {project} and/or {issue} from the incoming request URL one can then use the BugTrackerWrapperInterface from the Wrapper Layer described in Section B.7 to fetch one or all issues belonging to a project.

The URI: /projects/{project}/config/ was used to configure which bugtracker should be used. This information could then be used to instantiate the correct instance of the BugTrackerWrapperInterface provided by the Wrapper Layer. The

Figure 5.1: RESTful Layer

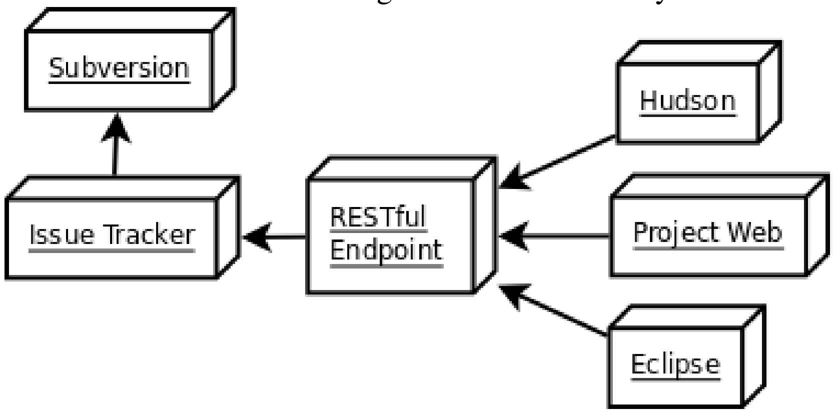


Figure 5.2: Example Representation

```
<issues>
  <issue>
    <url>http://localhost:8182/projects/LC/issues/634</url>
    <summary>summary..</summary>
  </issue>
  <issue>
    ...
  </issue>
</issues>
```

project is named no.miles.mpl.restful.endpoint, see Appendix C for details on where to find the source code. A simple test client was also made, see the project no.miles.mpl.restful.client.

Figure 5.1 illustrates the basic architecture where this layer serves as a wrapper or mediator between the issue tracker and the dependent components. Figure 5.2 illustrates how a collection of issues could be represented in an XML representation. Note that the representation of an issue was not very detailed due to time constraints.

5.2 Issues

This section presents issues that are either unresolved or resolved.

5.2.1 Oustanding issues

This section presents outstanding issues that have not yet been resolved for this layer.

1. **Extend this layer to support all attributes of issues etc.** This layer was not fully implemented due to time constraints. I had to start working on the above layers before this layer was horizontally and vertically completed. Therefore, only a few attributes of an issue, such as the summary were supported in this early version.

2. **Concurrency.** If a PUT request A is executed before PUT request B, but arrives after B. Then the resource is most likely put into the state specified by request A. Some form of concurrency control may need to be implemented, so that clients can be notified if they are trying to update an resource and they are not aware of that this resource has changed.

5.2.2 Resolved issues

No significant issues were resolved while implementing this layer.

5.3 Coupling

This section discusses this layer's effect on coupling within in the framework presented in 3.3. Recall the scenarios from Section 3.3:

SCENARIO1: An issue tracker is replaced with another issue tracker

SCENARIO2: The issue tracker's interface changes due to an update or upgrade

SCENARIO3: The implementation of the Proof of Concept changes within the constraints of its architecture.

D1: Syntax of data and service. The type or format of data as well as signature of services must be consistent with a client's assumptions.

SCENARIO1: If an issue tracker is replaced with another, this does not have an effect on what methods such as PUT, POST etc. will do. However the dataformat of the representations may change. For instance new attributes must be implemented. As for Trac, only one component per issue is supported while Jira issues can have many. This could mean that an XML representation may need to change from

```
<component>comp1</component>
```

to

```
<components>
  <component>comp1</component>
  <component>comp2</component>
</components>
```

Which could potentially break a clients ability to fetch any component names if a client for instance assumes that only one <component> element exists on the root node.

SCENARIO2: An update may lead to an attribute being added. This means that the XML document must be extended etc.

SCENARIO3: As for the format of the representation, REST gives almost no constraints as long as it is within a valid mime type. Hence it is possible that a future developer may want to change the default representation to JSON or RDF.

D2: Semantics of data and service. The semantics of data and services must be consistent with a client's assumptions.

Common for all three scenarios are that the operations e.g. PUT, POST and GET has well-known meanings.

SCENARIO1: If a new issue tracker is installed, the mappings to the XML elements etc. may be wrong or hard to do. Maybe the previous format doesn't capture the correct semantics at all.

SCENARIO2: An update to an issue tracker may change the semantics of an attribute. Such a change could be difficult for a client to uncover and/or deal with.

SCENARIO3: If one were to implement the PUT, POST, GET and DELETE operations in another fashion than the intended way, such as adding an issue with GET, this could have huge consequences. However, these operations are so clearly agreed upon (at least the major differences, such as that a PUT updates a resource and DELETE deletes it) that a client should not have to deal with ambiguity in the same way as if it should use an operation named for instance updateIssue(Array[] data, boolean delete).

D3: Sequence of data and control. Either the sequence of data are important, or there exists timing constraints.

Common for all three scenarios are that the REST architecture requires that the server are stateless. However, concurrency issues as mentioned in 5.2 must be dealt with.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D4: Identity of an interface. The name or handle of a service's interface must be consistent with a client's assumptions.

By following the guidelines of Tim Berners-Lee [34] there are no reasons for the URIs of the resources e.g. /projects/{project}/issues/ to change. The only topic that should be discussed is whether to apply a date prefix, e.g. /2008/projects/ to the URIs. This could allow applications to access old versions of interfaces as long as the old applications are kept running.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D5: Runtime location of a module/service. The runtime location of a module/service must be consistent with a client's assumptions.

One must assume that the RESTful interface are reachable from the internet.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D6: Quality of service/data. The quality of data or service must be consistent with a client's demands.

Common for all three scenarios are that REST supports extensive use of caching, which should make it possible to achieve a reasonable response time.

SCENARIO1: A new issue tracker may not support large attachments. Its external API may impact response times etc.

SCENARIO2: -

SCENARIO3: -

D7: Existence of a module/service. The module/service must exist.

If for some reason an issue tracker should not be used in a project, it is important that the surrounding components keep on working, e.g. handles a 404 error. However, if an issue tracker are unavailable for a certain amount of time, information will most likely be lost or unavailable.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D8: Resource behavior of a module/service. The resource consumption or ownership of a module/service must be consistent with a client's assumptions.

As REST supports extensive use of caching, it may be possible to reduce the load on the network.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

Table 5.1: Work

#Dependants	Initial issue tracker	First replacement	Second replacement
1	2	3	4
2	3	*4*	**5**
3	4	**5**	**6**
4	5	**6**	**7**

5.4 Work

Assuming that all the surrounding components already communicates with this layer, the impact of the three scenarios would be limited as to supporting new attributes, or perhaps a new format etc. The work invested in writing this layer, plus the client code, should then be less or equal to the amount of work discussed in Section 4.3. This depends on both the number of clients that one has to write code for and also the number of times the issue tracker are replaced. Table 5.1 illustrates the units of work required with respect to the number of dependent components and the number of times an issue tracker is replaced. Assuming that the interface remains stable, one only has to do changes at the dependent components (client side) once. Furthermore, when a new issue tracker is installed, one only has to change the server-side implementation (wrapper). The table shows break-even points in *italic* and less amount of work in **bold**. This implies, that if one has only one dependent component, the amount of work invested will always be greater with this Proof of Concept. However if one has two dependent components, the break-even point is reached when the issue tracker is replaced for the first time. Hence, as the number of dependent components increase and the issue trackers are replaced several times, the return on investment are expected to increase.

One could argue that the amount of work required to implement this Proof of Concept is proportionally larger than just writing the client code needed to communicate with the issue tracker, something which could mean that the break-even point should be shifted one place to the right and/or down. However, the investment could still be beneficial in the long run.

5.5 Conclusion

This layer was implemented trying to follow the architectural constraints of the architectural style REST [27]. Due to time constraints, only a simple prototype that returned a partial XML representation of an issue was implemented. However, apart from a few issues that should be possible to overcome, it should be possible to extend this layer to include all relevant properties and functionality.

Both the addition of an intermediary in general, as well as the REST specific constraints had positive impact on reducing coupling between an issue tracker and its surrounding components. By using the HTTP methods such as GET, PUT, POST and DELETE the interface to the the issue tracker are expected to remain more stable (than the initial scenario).

Regarding the amount of work invested in this Proof of Concept, it is difficult to determine the exact point where the return of investment breaks even. However, if more than one dependent component has to be adapted to be able to communicate with the issue tracker, and if the issue tracker is replaced several times, one can assume that the investment will pay off, at least in the long run.

Chapter 6

APP layer

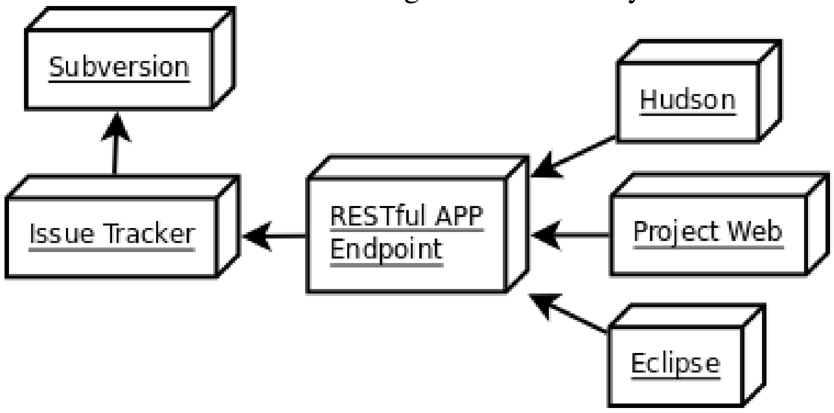
The previous Chapter 5 presented the first layer of this Proof of Concept. This chapter will present the next layer by extending the RESTful layer to implement the Atom Publishing Protocol. Section 6.1 will present the design of this layer. Section 6.2 will present and discuss both resolved and outstanding issues. Section 6.3 will present the effect on coupling while Section 6.4 will discuss the amount of work. Finally, Section 6.5 will conclude this chapter.

6.1 Design

Rome [26] and its subproject Propono were used as a framework to implement this layer. Basically, a class implementing the com.sun.syndication.propono.atom.server.-AtomHandler had to be written. This class has methods that are executed when GET, PUT, POST or DELETE requests are received. The appropriate actions must then be taken. For a GET request, the Wrapper Layer presented in Section B.7 is used to fetch no.miles.mpl.wrapper.datastructures.Issue(s). These issues are then converted to com.sun.syndication.feed.atom.Entry(ies) and a com.sun.syndication.feed-.atom.Feed is returned containing these entries. The URI scheme below illustrates which feeds and entries are available through this interface. See Appendix C for details on how to view the source code.

Figure 6.1 illustrates that the basic architecture has not changed much from the previous layer, although the endpoint now returns and accepts Atom feeds and entries. Figure 6.2 shows an example Atom feed. Additional information such as a detailed XML representation of an issue should be put in the content element of an entry.

Figure 6.1: APP Layer



URI scheme:

The URI scheme from the previous layer 5 was modified and extended to support the following request URIs:

../projects/{projectname} - the project - with configuration

../projects/{projectname}/issues/ - all the issues belonging to the project

../projects/{projectname}/issues/{issue}/ - an issue

../projects/{projectname}/versions/ - all versions belonging to the project

../projects/{projectname}/components/ - all components belonging to the project

../projects/{projectname}/types/ - all issuetypes possible for issues belonging to the project

../projects/{projectname}/priorities/ - all priorities possible for issues belonging to the project

../projects/{projectname}/issues/{issue}/comments/ - all the comments on the issue

../projects/{projectname}/issues/{issue}/states/ - all possible states (transitions) for the issue

One can GET, PUT, POST and DELETE projects and issues. Comments support GET and POST. The other feeds only support GET.

Figure 6.2: Example Atom feed

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:app="http://www.w3.org/2007/app"
  xmlns:sy="http://purl.org/rss/1.0/modules/syndication/"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:taxo="http://purl.org/rss/1.0/modules/taxonomy/">
  <title>LC issues</title>
  <id>http://localhost:8070/no.miles.mpl.semantic.app.endpoint/
      projects/LC/issues/</id>
  <updated>2008-05-14T08:23:45Z</updated>
  <entry>
    <title>summary..</title>
    <link rel="alternate" href="http://localhost:8080/jira/
            browse/LC-634" />
    <link rel="alternate" type="application/xml"
            href="http://localhost:8070/no.miles.mpl.semantic.app.endpoint/
            projects/LC/issues/634?contentformat=application/xml" />
    <link rel="about" href="http://xmlns.com/baetle/#Issue" />
    <link rel="edit" href="http://localhost:8070/
            no.miles.mpl.semantic.app.endpoint/projects/LC/issues/634" />
    <author>
      <name>test</name>
    </author>
    <id>http://localhost:8070/no.miles.mpl.semantic.app.endpoint/
       projects/LC/issues/634</id>
    <updated>2008-04-09T10:05:15Z</updated>
    <content type="application/xml">
      < XML REPRESENTATION HERE! >
    </content>
    <summary type="text" />
  </entry>
  <entry>
    ....
  </entry>
</feed>
```

6.2 Issues

6.2.1 Outstanding issues

This section presents the outstanding issues for this layer.

1. **Extend this layer to support all attributes of issues etc.** This layer was not either fully implemented due to time constraints. I had to start working on the above layer before this layer was horizontally and vertically completed. Therefore, only a few attributes of an issue, such as the summary were supported for this layer too.

2. **Content negotiation on the <content> element is outside of HTTP scope.** The media type for an atom feed are always application/atom+xml. Hence if the type attribute of the content element could be both application/xml or text/plain, HTTP negotiation in the form of using the Accept header, cannot be used. Instead an alternate link with the type attribute set to the different media types can be used to indicate which format the content should be in. This works, but requires additional interaction between a client and a server, not to mention that the client must know the details of the Atom format.

6.2.2 Resolved issues

This section presents issues that been resolved for this layer.

1. **Concurrency.** I implemented the GDATA approach [13] by providing a versioned edit-uri to each entry. If an entry is updated the entry's edit uri are incremented. Hence, the server can easily tell if a client is trying to update an entry that has changed since the client fetched it. A 409 -Conflict HTTP code is returned if a client tries to update or delete an entry with an old edit-uri.

2. **Security.** This issue might only be partially resolved, but what the APP endpoint does, is really only to pass the user credentials forward as arguments to the various issue tracker's specific APIs. Hence if a user isn't allowed to view issues from Jira, he won't be allowed to view them with this APP endpoint either. However, this layer may introduce increased vulnerability to man-in-the-middle attacks etc. Using SSL or a simliar solution should probably have a positive impact on security. As this is a bit out of scope of this Master Thesis I have not dealt with this issue any further .

6.3 Coupling

This section will discuss this layer's effect on coupling within in the framework presented in 3.3. Recall the scenarios from Section 3.3:

SCENARIO1: An issue tracker is replaced with another issue tracker

SCENARIO2: The issue tracker's interface changes due to an update or upgrade

SCENARIO3: The implementation of the Proof of Concept changes within the constraints of its architecture.

D1: Syntax of data and service. The type or format of data as well as signature of services must be consistent with a client's assumptions.

In addition to the uniform interface applied by the previous layer, this layer specifies parts of the syntax in the form of the Atom syntax. Additional attributes are put inside of the <content> element of an entry. Furthermore, the Atom format describes how to represent collections of data, as entries in a feed.

SCENARIO1: The attributes or properties that fit into the Atom format such as title, summary etc. will not change. However, the representation put inside the <content> elements - probably an XML representation similar to the one in the previous layer may have to change if new attributes must be supported.

SCENARIO2: Just like the previous layer, an attribute may need to be added, hence if it is not supported by the Atom format it would lead to an extension of the representation included in the <content> element.

SCENARIO3: As mentioned above, the Atom format specifies the basic syntax. However, the representation included in the <content> element could change from text to XML or XHTML etc. Atom adds some constraints to how binary files or media content should be included. One should either include it as a base64 encoded document, or one should just specify the URI to the representation.

D2: Semantics of data and service. The semantics of data and services must be consistent with a client's assumptions.

In addition to the meaning of the HTTP operations, the Atom format elements are well defined and one could assume that they are commonly understood. For instance an author of an entry is the one who created or wrote the entry. However, the "one-liner" summary attribute in Jira, actually maps better to the <title> element of the Atom format. Furthermore, the description attribute of an issue in Jira and Trac, actually maps better to the <summary> element of Atom. This means that although the Atom format's semantics are intuitive, different implementations may actually map different attributes to the Atom elements.

SCENARIO1: If a new issue tracker has a different set of attributes with slightly different semantics, then it could be hard to map them to the same elements of the Atom format. A client may experience a different use of the same Atom elements.

SCENARIO2: -

SCENARIO3: -

D3: Sequence of data and control. Either the sequence of data are important, or there exists timing constraints.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D4: Identity of an interface. The name or handle of a service's interface must be consistent with a client's assumptions.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D5: Runtime location of a module/service. The runtime location of a module/service must be consistent with a client's assumptions.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D6: Quality of service/data. The quality of data or service must be consistent with a client's demands.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D7: Existence of a module/service. The module/service must exist.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D8: Resource behavior of a module/service. The resource consumption or ownership of a module/service must be consistent with a client's assumptions.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

6.4 Work

On a general level, the amount of work should be similar to what was presented in Section 5.4. However, as discussed in the previous Section 6.3 the interfaces are expected to remain a bit more stable than the RESTful layer. Hence it is more unlikely that the dependent components would be impacted of a change.

6.5 Conclusion

This layer added more functionality to this Proof of Concept. A new issue regarding content negotiation were uncovered, but it is not necessarily hard to work around in practice. Furthermore, it had some impact on coupling beyound what the previous layer added. The likelihood of maintaining a stable interface has increased due to the introduction of the Atom format. This should then reduce the likelyhood of changes needed to be implemented on dependent components.

Chapter 7

Semantic layer

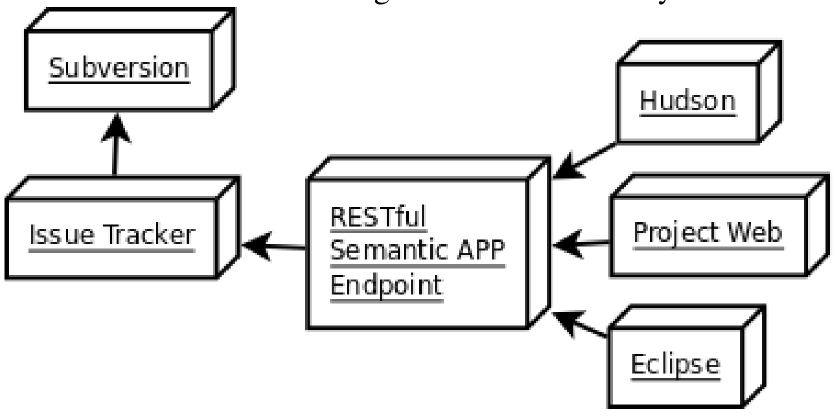
The previous Chapters 5 and 6 presented the two first layers of this Proof of concept. This chapter presents the final layer, namely the Semantic layer, extending the previous layers with Semantic Web technology. Section 7.1 presents the design of this layer while Section 7.2 presents outstanding issues. Section 7.3 discusses this layer's effect on coupling, while Section 7.4 discusses work. Section 7.5 discusses a few topics regarding ontologies and Semantic Web technology. Finally, Section 7.6 concludes this chapter.

7.1 Design

The RDF data about an issue etc. is contained in an text/rdf+n3 representation inside the <content> element of an Atom entry. The issue is described using the envisioned Baetle ontology, which I have contributed to during my work on this Master Thesis [16]. The Baetle ontology links an issue and its properties to several other public ontologies such as FOAF, SIOC and DOAP [6] [11] [31]. These ontologies were also reused in the project feeds as well as in the comment feeds.

A design issue that I got into was whether to use OWL-DL or OWL-FULL. OWL-FULL does not come with a computational guarantee, hence one can end up with infite loops etc. However, when importing public ontologies that are OWL-FULL, my ontologies became OWL-FULL too. Hence I was left with a question of loosing value in form of abandoning these public ontologies, or risk that reasoning on these ontologies might stall or crash the application. I received good help from different Semantic Web communities and learned that the best practice was to include everything and test it - if it doesn't crash, it's okay. This approach is also a match with the open world mindset of the web today, one had to accept the 404 Error in order to allow the web to scale. In [5] this is described

Figure 7.1: Semantic Layer



with the AAA slogan, which is short for; “Anyone can says Anything about Any topic”.

When choosing a URI to represent the actual issue I consulted the W3C article “Cool URIs for the Semantic Web” [37]. It is difficult to decide whether an issue is an abstract thing or in fact the actual representation given by either Trac or Jira. I landed on using URIs such as http://localhost:8080/jira/browse/LC-105 for issues. These URIs make it easy for a human to understand that this is an issue. It is also probably a better approach than to use the same URI as the Atom entry, because that is more of a temporary wrapper/container for the issue. However, one should perhaps include both URIs to ensure future applications can recover this issue, even if it has been moved into another issue tracker. This could be implemented using the rdfs:seeAlso property.

Figure 7.1 illustrates almost the same basic architecture as the two previous layers. However, the endpoint now returns and accepts Atom feeds and entries bundled with semantic data. Figure 7.2 illustrates how a semantic representation (N3) of an issue is included in the <content> element of an Atom entry.

7.1.1 Sparql Endpoint

The same URI scheme as mentioned in Section 6.1 was used, but with a small extension; namely the support for SPARQL queries on issues.

- ../projects/{projectname}/issues/?sparql=SELECT * WHERE ..

This was implemented by fetching all the issues to the specific project, merging them together in an in-memory model together with the Baetle (meta model) ontology itself. Then a sparql query is executed and an Atom feed with one entry for each SPARQL Resultset is returned.

Figure 7.2: Example Semantic Atom feed

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" ... >
  <title>LC issues</title>
  <id>http://localhost:8070/...</id>
  <updated>2008-05-14T08:23:45Z</updated>
  <entry>
    <title>summary..</title>
    <link rel="alternate" ... />
    ...
   <content type="text/rdf+n3">
    @prefix :        &lt;http://xmlns.com/baetle/#&gt; .
    @prefix sioc:    &lt;http://rdfs.org/sioc/ns#&gt; .
    @prefix rdfs:    &lt;http://www.w3.org/2000/01/rdf-schema#&gt; .
    @prefix foaf:    &lt;http://xmlns.com/foaf/0.1/&gt; .
    @prefix owl:     &lt;http://www.w3.org/2002/07/owl#&gt; .
    @prefix doap:    &lt;http://usefulinc.com/ns/doap#&gt; .
    @prefix wf:      &lt;http://www.w3.org/2005/01/wf/flow#&gt; .

    &lt;http://localhost:8080/jira/browse/LC-634&gt;       a       :Task ;
         wf:state                  [ a       :Open           ] ;
        :assigned_to               [ a       sioc:User ;
                                     foaf:accountName "test" ] ;
        :created "2008-04-9T12:05:15.0" ;
        :priority                  [ a       :Major          ] ;
        :project                   [ a       doap:Project ;
                                             doap:name "LC"  ] ;
        :reporter                  [ a       sioc:User ;
                                             foaf:accountName "test" ] ;
        :title  "summary.." ;
        :updated "2008-04-9T12:05:15.0" .

        ...

   </content>
   <summary type="text" />
  </entry>
</feed>
```

7.2 Issues

This section presents both outstanding and resolved issues.

7.2.1 Oustanding issues

This section presents the outstanding issues for this layer.

1. **Enable an external application to automatically deduce what this feed is about and that RDF data can be found in the content.** This issue is a result of that this Proof of Concept does not use a standardized combination of Atom Feeds and Semantic Web technology. Although Atom feeds are a nice way to express collections of data, and that RDF and OWL are a uniform way to express this data, few clients would be able to know that this is actually a “Semantic Feed” with semantic data in the <content> element. It would certainly find out by studying the feed, but it is unclear whether an entry is an issue, contains data about an issue or just mentions an issue. I used a link with rel=”about” and a URI to the baetle ontology. However, this is not yet a standardized way to specify what a feed is about. If the rel=”about” practice was to be standardized, one could use the Awol ontology [15] to describe each feed and state that the content contains the data etc. One would then need to specify the URI of the about link to point to for instance the concept IssueFeed.

7.2.2 Resolved issues

This section presents issues that been resolved for this layer.

1. **Performance.** Long response times were experienced when dealing with hundreds of issues. However, by caching all the entries, and only fetch the ones that are updated (and leave out the ones that have been deleted), the response times have improved. To be able to do this I had to write a plugin to Jira (see Section B.1.1). However, as a feed containing thousand issues carries a lot of data, it might be beneficial to partition the feed. For instance one could get only issues assigned to a user, or only recently updated issues etc. in order to decrease the response times even further.

7.3 Coupling

This section will discuss this layer’s effect on coupling within in the framework presented in 3.3. Recall the scenarios from Section 3.3:

SCENARIO1: An issue tracker is replaced with another issue tracker

SCENARIO2: The issue tracker's interface changes due to an update or upgrade

SCENARIO3: The implementation of the Proof of Concept changes within the constraints of its architecture.

D1: Syntax of data and service. The type or format of data as well as signature of services must be consistent with a client's assumptions.

SCENARIO1: If an issue tracker is replaced, a property or attribute could be added or removed from the RDF representation of the issue, however a client should be able to parse the model anyway. The risk is of course that a client may fail provide an attribute that is mandatory by the new issue tracker.

SCENARIO2: See SCENARIO1.

SCENARIO3: The syntax of the data contained within the <content> element of the Atom entries should now be text/rdf+n3, application/rdf+xml or any other widely used textual representation of RDF. This means that most RDF parsers could understand this part. Hence the scope of change in syntax is very limited. By using SPARQL as a query language to query issues, one can be sure that the query language does not need to change. Even if totally different ontologies are used, one can always use SPARQL as a query language in the future. However, the "SPARQL to Atom mapping" described in Section 7.1 are not necessarily intuitive and at least not standardized. Hence an external application might expect another format in return after executing the SPARQL query. This should however be possible to overcome using HTTP Content Negotiation and providing different formats for displaying SPARQL results (e.g. the standard SPARQL XML serialization). [32]

D2: Semantics of data and service. The semantics of data and services must be consistent with a client's assumptions.

SCENARIO1: If an issue tracker is replaced, a property may need to be added to the representation (Atom + N3). A client may be able to understand this property, either partially or fully. For instance, if a new state is added as a subclass of baetle:New, the client may not understand the details, but it would probably be able to make sense of it.

SCENARIO2: See SCENARIO1.

SCENARIO3: The model describing an issue could for instance be linked to many other new ontologies, without enforcing any trouble on the client. As mentioned in SCENARIO1, a client that for instance uses inference and SPARQL could potentially understand new properties without any further implementation, given that they are related to already known concepts.

D3: Sequence of data and control. Either the sequence of data are important, or there exists timing constraints.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D4: Identity of an interface. The name or handle of a service's interface must be consistent with a client's assumptions.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D5: Runtime location of a module/service. The runtime location of a module/service must be consistent with a client's assumptions.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D6: Quality of service/data. The quality of data or service must be consistent with a client's demands.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D7: Existence of a module/service. The module/service must exist.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

D8: Resource behavior of a module/service. The resource consumption or ownership of a module/service must be consistent with a client's assumptions.

No additional impact from this layer.

SCENARIO1: -

SCENARIO2: -

SCENARIO3: -

7.4 Work

On a general level, the amount of work should be similar to what was presented in Section 5.4. However, this layer could potentially keep the interfaces even more stable than the previous layers. Hence even less changes should propagate to the client side. Furthermore, this layer makes it possible to create somewhat intelligent agents, that could be preprogrammed to understand a large set of widespread ontologies, hence increasing the possibility of understanding changes or addons to this Proof of Concept in the future. It is however doubtful that the latter possibility would pay off within a reasonable amount of time.

7.5 Discussion

As presented in Section 7.1 there's being developed an ontology for describing issues (Baetle [16]). A few other public ontologies are used in this Proof of Concept too. As different ontologies are linked, new links between different concepts are uncovered. There should be possible to link many other parts of the Miles Platform together in a uniform way using different ontologies. For instance should all users be described using both FOAF [11] and SIOC [31]. Each project described with DOAP [6] could be linked to many other things besides issues, for instance SVN Repositories, wikis or websites. One could then end up with a Platform consisting of semantically interlinked concepts or resources. New relationships or properties could potentially be deduced and exploited.

Furthermore, external applications which are familiar to any of these public ontologies could automatically understand and use parts of the Miles Platform. One could also see this from the opposite viewpoint, namely allowing the Miles Platform to understand data from external applications or projects. One potential feature would be to let some part of the Miles Platform search through issue trackers for different projects that has been included in a Miles Platform project in the form of for instance a library. Hence, the Miles Platform could potentially suggest that an issue reported on the Miles Platform issue tracker is very similar to an issue reported on a library being used by the current project. A developer could find a potential solution on this library's issue tracker, or at least contribute with more information on the issue.

However, as these ontologies help providing a stable interface to a client, what is indicated in [1] is that coupling are expected to increase as ontologies imports or refers to each other. Hence, there is likely to exist a trade-off when using ontologies between interconnectability vs. loose coupling.

7.6 Conclusion

This layer extended the previous layer by describing issues etc. with public ontologies. This layer was also extended both vertically and horizontally to support full CRUD on an issue, with almots every relevant attribute.

Once clients are told where to find this data describing the issues, they could interpret it in a uniform way. Furthermore, this layer further narrowed the possible scope of change, as well as increased a clients possibility to understand or cope with future changes to this Proof of concept's interface. Hence this layer should potentially reduce the amount of work required even further, at least in the medium or long run.

Part III

Synthesis

Chapter 8

Discussion and conclusions

The previous chapters presented background, design and results for creating this Proof of Concept. This chapter will discuss and conclude this Master Thesis with respect to the aim stated in Section 1.2. Section 8.1 will summarize this Master Thesis. Section 8.2 will discuss issues, Section 8.3 will discuss coupling and Section 8.4 will discuss work. Section 8.5 will discuss the research approach. Finally, Section 8.6 will suggest further work.

8.1 Summary

This Master Thesis involved implementing and evaluating the Proof of Concept I planned in [9]. The Proof of Concept consists of three layers. The first layer I implemented was the RESTful layer. This layer was placed on top of the issue trackers; Jira and Trac. This initial layer served as an intemediary or wrapper between the issue trackers and their surrounding tools and components. While still following the constraints of REST I extended the Proof of Concept by implementing the second layer using the Atom Publishing Protocol. Finally, I added the third layer which involved Semantic Web technology by using RDF(S) and OWL to represent issues and their properties.

Before I had implemented this Proof of Concept I first started out with designing the research approach. I decided that the Research Questions produced in [9] were still the best candidates to answer in order to evaluate if I would reach the aim of this Master Thesis. The aim was to "*ensure a long life-time of the Miles software development Platform by enabling loose coupling between its different components and tools*" 1.2.

To be able to test and use the Proof of Concept, a test environment had to be installed, configured and set up. I had to implement several plugins and prototypes myself (see Appendix B for details). The source code for both this Proof of

Concept and some configuration details for the test environment were submitted together with this thesis (see Appendix C).

8.2 Issues

To answer **RQ1:** “To which extent is it possible to implement the different layers of this Proof of Concept?“, I decided to present all the important issues that were either resolved or not, when this Proof of Concept was finished or I ran out of time. The following issues have been resolved:

- **Concurrency.** The issue of concurrency has been dealt with, using a similar approach as GDATA [13]. The edit-uri of an Atom entry are incremented each time it is updated. Hence the server can tell if a client is not aware of the latest update.
- **Performance.** By caching entries one can fetch only the updated issues from the issue trackers. Hence the response times have improved significantly.
- **Security.** When a client uses the Semantic APP endpoint, the credentials are passed on to the issue tracker. If the user does not have the correct privileges on the issue tracker, the server returns a HTTP error code such as 403 Forbidden.

The following issues have not yet been resolved:

- **Enable an external application to automatically deduce what a feed is about and that RDF data can be found in the content.** Based on my understanding of the current state of the art, it is unclear whether a so called intelligent client would interpret an entry containing a semantic description of an issue as an entry with some data about an issue, as the issue itself or something else. This is not at all a blocking issue. It is more a symptom of that I have used somewhat immature technologies, in a special combination, where few best practices yet exist. Future versions of Atom might prove to be more integrated with Semantic Web technologies, or I might explore another solution, but for now the approach I used seemed like the best alternative.
- **Content negotiation on the <content> element is outside of HTTP scope.** A client cannot use the HTTP Accept Header to specify the content type of the <content> elements of an Atom Feed as this header is already been used for application/atom+xml. This should not be treated as a blocking

issue either. It does require a client to perform an extra step if links are provided to other representation formats, but following hyperlinks to other representations is also clearly a part of the REST architectural style.

Although the prototypes of the RESTful and the Atom Publishing Protocol layers were not fully implemented, they are both a part of the final prototype containing all three layers. Hence I have proved that it is possible to successfully combine these architectural styles and technologies within the scope of this Master Thesis. All unresolved issues are considered non-blocking and almost trivial, although they have resulted in interesting discussions. I have dealt with important issues such as concurrency, performance and partially security. However, these solutions could possibly be improved even further. Although the quality attribute in focus for this Master Thesis has been modifiability, I have shown that it is possible to deal with other quality attributes too, by overcoming the initial negative effects that this Proof of Concept had on them.

8.3 Coupling

Answering **RQ2:** "How does the addition of each layer affect the degree of coupling?" was done by discussing each layer's effect on coupling and modifiability within a chosen framework (see Appendix A). This framework was considered the best framework available (see Section 3.3). Each layer contributed especially to keeping interfaces stable, either because the methods or the formats were not expected to change significantly.

As I have worked with this Proof of Concept I have experienced that a very important aspect of achieving loose coupling is about maintaining stable interfaces. First of all, stable interfaces reduces the possibility for ripple effects. First the RESTful layer applied a standard set of methods. Second that the Atom Publishing Protocol layer introduced the Atom formats, making the syntactical room for change a bit narrower. Finally, the Semantic layer narrowed this scope of change even further by standardizing on the attributes that didn't fit well into the Atom elements. This Proof of Concept was implemented as a wrapper or intermediary between the issue trackers and their dependent components. This approach served as a foundation for keeping the interfaces stable. [20]

The dependency type **"D7: Existence of a module/service**" was not significantly reduced by any of the layers. It is unclear how a client can cope with a 404 error. In the best case, it should just keep on doing whatever it could do without data from an issue tracker. A topic that could shed some light on this issue is perhaps the dimension of time. What if an issue tracker is currently unavailable? Then information could be lost due to a client giving up. This is a shortcoming of this Proof of Concept.

This Proof of Concept involved implementing three layers, one on top of the other. Hence, there might exist other combinations of the layers that might be more beneficial. One could then ask the question “do we actually need the APP layer?”. What I learned from creating the RESTful layer, was that it was difficult to know how to express collections of elements. Atom solved this in a very nice way. However, by combining only the RESTful and the Semantic layer, a number of alternatives exists that might solve this problem. One could for instance just include all the issues belonging to a project in an N3 document. This might even remove some overhead from Atom, and potentially increase performance. However, by excluding Atom you would remove the possibility for people to easily make sense of these feeds using a feed reader.

Each layer contributed to keep the interfaces stable, although other dimensions, such as time, have not been dealt with from a modifiability perspective. I expect the three layers or different combinations of these to serve as a good foundation for ensuring the Miles Platform a long life-time.

8.4 Work

The last research question, **RQ3:** “How does these layers affect the amount of work spent on replacing a component?” were discussed on a general level for each layer. The discussion focused on whether one could expect a return on investment by implementing this Proof of Concept in the short or long run. As the sections describing **Work** discussed, using an intermediary/mediator/wrapper only pays off if there are several dependent components, and the central component is actually replaced one or preferably several times during its life-time. Using these techniques can therefore not be defended, at least not from a modifiability perspective alone, if none of the Scenarios presented in Section 3.3 are expected to occur.

8.5 Research approach

By answering **RQ1:** “To which extent is it possible to implement the different layers of this Proof of Concept?“, I could state how successful I have been with implementing this Proof of Concept. Listing solved and unresolved issues was an easy and effective way to indicate to which extend I was able to implement this Proof of Concept. However, before the Proof of Concept has been thoroughly tested by other people than me, it is reasonable to assume that a lot of bugs and issues have not yet been uncovered.

Using Evolutionary Prototyping [10] some of the code used for the prototypes of the two first layers was actually reused in the prototype containing all three layers. The development method was easy to use when working within a previously unfamiliar field. As I tested out new libraries and designs I could just continue to build on top of what already worked. However, as I gained new knowledge and became familiar with the technologies and libraries, new better solutions were uncovered, and a lot of changes had to be made to the prototype. If one could gain this knowledge by experimenting with each technology or library one at a time before starting on the final system, one might end up with a better design or architecture on the final system.

RQ2: "How does the addition of each layer affect the degree of coupling?", was designed so that I could evaluate each layer's effect on coupling. As argued in Section 3.3, it was difficult to find any suitable coupling measures for this kind of RESTful (Semantic) Web Services. However, by considering coupling as a part of the more general quality attribute **modifiability**, I was at least able to perform a qualitative discussion on this topic for each layer. This disussion or evaluation of each layer's effect on coupling and modifiability, could however be influenced by my subjective values or beliefs. Hence a reader of this Master Thesis should perhaps be considering my conclusions more as an advice from me, rather than quantitatively confirmed scientifical results.

The last research question, **RQ3:** "How does these layers affect the amount of work spent on replacing a component?", was only answered on a general level. It was impossible to create exact estimates as it is hard to predict the future. However, this question was perhaps more directed towards the general approach of using an intermediare and keeping interfaces stable, than towards the specific architectural styles and technology used in this Proof of Concept.

As the above discussion indicates, **RQ2** was probably the most central research question for this Master Thesis. It clearly correlates with the aim of this Master Thesis and it uncovered several aspects that have been thoroughly discussed. The coupling was mostly reduced due the fact that the interface(s) a client should use to communicate with the issue tracker(s) is expected to remain stable. Both format and methods, i.e. syntax of data and service, are not expected to change significantly. Although all the layers contributed to maintaining stable interfaces, the use of public ontologies might have introduced new dependencies through referring to other ontologies, hence there exists a trade-off here one should be aware of.

"Ensure a long life-time of the Miles software development Platform by enabling loose coupling between its different components and tools" as stated in Section 1.2 was the aim of this Proof of Concept. Based on the results and previous discussions it is reasonable to believe that this Proof of Concept will contribute to extending the life-time of the Miles Platform by enabling loose coupling between its components. Either the Proof of Concept are adopted (fully or partially) into the Miles Platform or the the experience and knowledge gained from this thesis could be used as a basis for new solutions or decisions.

8.6 Further work

This Section presents further work that could be conducted in order to improve this Proof of Concept.

- To deal with the dimension of time, and implement the asynchrounous publish-subscribe pattern, an Atom Publishing Protocol (APP) extension to the XMPP protocol could be implemented on top of this Proof of Concept as suggested in [28]. Atom entries could be passed as the payload of messages. Furthermore, by listening to incoming messages instead of polling an APP endpoint directly, one could potentially reduce the load on the APP endpoint.
- The Atom Format attributes such as <title>, <summary> etc. can be parsed by most feed readers and displayed to a human in a readable way. However, the representations used inside the <content> element are in this Proof of Concept text/rdf+n3. One can argue that this format is more readable than application/rdf+xml. However, many web sites uses for instance HTML or XHTML together with CSS to provide a far better user experience than if a user would have read pure N3. RDFa makes it possible to include semantic meta-data in XHTML. This has the impact that an XHTML document can be both human and machine readable (understandable). One could potentially change this Proof of Concept to include RDFa (XHTML) in the <summary> element of an Atom Feed, hence making the representation presented to the user equal to the representation used by the applications. [24]
- Using SWRL to map from one ontology to another, it should be possible to create one ontology for Trac and one ontology for Jira etc. and then specify how these ontologies map to Baetle and other ontologies. This could make it easier to make changes to the Wrapper Layer (see Section B.7) because some of the changes could probably be made at run-time. [29]
- As the data describing issues etc. are available on a uniform format across different issue trackers, there should be fairly easy to backup this data and use it in the future regardless of whether the issue tracker are replaced or not. The latter could be considered as a contribution to extending the life-time of the Miles Platform, by actually extending the life-time of its data.

Perhaps this is equally or more important that achieving loose coupling with stable interfaces.

Bibliography

[1] Anthony M. Orme, Haining Yao, and Letha H. Etzkorn. Coupling Metrics for Ontology-Based Systems. *IEEE Software*, 2006.

[2] Atomenabled. URL `http://www.atomenabled.org/`. Last accessed: 22 January 2008.

[3] Charles Zhang and Hans-Arno Jacobsen. Quantifying Aspects in Middleware Platforms. *Proceedings of the 2nd international conference on Aspect-oriented software development*, 2003.

[4] Codehaus. Grails. URL `http://grails.codehaus.org/`. Last accessed: 14 April 2008.

[5] Dean Allemang and Jim Hendler. *Semantic Web for the Working Ontologist*. Morgan Kaufmann, 2008.

[6] Edd Dumbill. Doap. URL `http://trac.usefulinc.com/doap`. Last accessed: 6 May 2008.

[7] Eclipse. URL `http://http://www.eclipse.org/`. Last accessed: 22 January 2008.

[8] Elias Torres. Queso. URL `http://torrez.us/archives/2006/07/17/471/`. Last accessed: 14 April 2008.

[9] Erling Wegger Linde. Achieving Loose Coupling in the Component-Based Miles Software Development Platform. *Specialization Project*, 14 December 2007.

[10] Evolutionary Prototyping. URL `http://en.wikipedia.org/wiki/Evolutionary_prototyping#Evolutionary_prototyping`. Last accessed: 16 April 2008.

[11] FOAF. URL `http://www.foaf-project.org/`. Last accessed: 6 May 2008.

[12] Franck Xia. On the concept of coupling, its modeling and measurement. *The Journal of Systems and Software 50*, 2000.

[13] Google. Google Data APIs. URL `http://code.google.com/apis/gdata/overview.html`. Last accessed: 14 April 2008.

[14] Gregory A. Hall, Wenyou Tao, and John C. Munson. Measurement and Validation of Module Coupling Attributes. *Software Quality Journal 13*, 2005.

[15] Henry Story. Atomowl vocabulary specification, . URL `http://bblfish.net/work/atom-owl/2006-06-06/AtomOwl.html`. Last accessed: 14 April 2008.

[16] Henry Story. Baetle, . URL `http://code.google.com/p/baetle/`. Last accessed: 14 April 2008.

[17] Hudson. URL `http://hudson.gotdns.com/wiki/display/HUDSON/Meet+Hudson`. Last accessed: 22 January 2008.

[18] Jarallah S. Alghamdi. Measuring Software Coupling. *Proceedings of the 6th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems*, 16 February 2007.

[19] Jira. URL `http://www.atlassian.com/software/jira/`. Last accessed: 22 March 2008.

[20] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice Second Edition*. Addison-Wesley, November 2006.

[21] Martin Hitz and Behzad Montazeri. Measuring Coupling and Cohesion In Object-Oriented Systems. *Proc. Int. Symposium on Applied Corporate Computing*, 1995.

[22] Mikhail Perepletchikov, Caspar Ryan, Keith Frampton, and Zahir Tari. Coupling Metrics for Predicting Maintainability in Service-Oriented Designs. *Proceedings of ASWEC'07*, 2007.

[23] Mikhail Perepletchikov, Caspar Ryan, Keith Frampton, and Heinz Schmidt. Formalising Service-Oriented Design. *Journal of Software, Vol. 3, No. 2*, February 2008.

[24] RDFa Primer. URL `http://www.w3.org/TR/xhtml-rdfa-primer/`. Last accessed: 09 May 2008.

[25] Restlet. URL `http://www.restlet.org/`. Last accessed: 16 April 2008.

[26] ROME. URL `https://rome.dev.java.net/`. Last accessed: 16 April 2008.

[27] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[28] P. Saint-Andre. Atomsub: Transporting atom notifications over the publish-subscribe extension to the extensible messaging and presence protocol (xmpp). URL `http://www.xmpp.org/internet-drafts/draft-saintandre-atompub-notify-07.html`. Last accessed: 09 May 2008.

[29] SWRL: A Semantic Web Rule Language Combining OWL and RuleML. URL `http://www.w3.org/Submission/SWRL/`. Last accessed: 27 May 2008.

[30] Semap. URL `http://code.google.com/p/semap/`. Last accessed: 8 April 2008.

[31] SIOC. URL `http://sioc-project.org/`. Last accessed: 6 May 2008.

[32] Sparql Query Language for RDF. URL `http://www.w3.org/TR/rdf-sparql-query/`. Last accessed: 09 May 2008.

[33] Subversion. URL `http://subversion.tigris.org/`. Last accessed: 22 January 2008.

[34] Tim Berners-Lee. Cool URIs don't change. 1998. URL `http://www.w3.org/Provider/Style/URI`. Last accessed: 17 April 2008.

[35] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 17 May 2001.

[36] Trac. URL `http://trac.edgewall.org/`. Last accessed: 25 January 2008.

[37] W3C Interest Group. Cool URIs for the Semantic Web. URL `http://www.w3.org/TR/cooluris`. Last accessed: 31 March 2008.

[38] wingerz. A queso example. URL `http://wingerz.com/blog/?p=38`. Last accessed: 14 April 2008.

Part IV

Appendix

Appendix A

Template for coupling discussion

The following is a template for discussing the coupling of a Layer for this Proof of Concept. Comments are given in parantheses:

D1: Syntax of data and service. The type or format of data as well as signature of services must be consistent with a client's assumptions.

(Place for general discussion across scenarios)

SCENARIO1: (Place for scenario specific discussion)

SCENARIO2: (a "-" can be given if no scenario specific discussion is relevant)

SCENARIO3:

D2: Semantics of data and service. The semantics of data and services must be consistent with a client's assumptions.

SCENARIO1:

SCENARIO2:

SCENARIO3:

D3: Sequence of data and control. Either the sequence of data are important, or there exists timing constraints.

SCENARIO1:

SCENARIO2:

SCENARIO3:

D4: Identity of an interface. The name or handle of a service's interface must be consistent with a client's assumptions.

SCENARIO1:

SCENARIO2:

SCENARIO3:

D5: Runtime location of a module/service. The runtime location of a module/service must be consistent with a client's assumptions.

SCENARIO1:

SCENARIO2:

SCENARIO3:

D6: Quality of service/data. The quality of data or service must be consistent with a client's demands.

SCENARIO1:

SCENARIO2:

SCENARIO3:

D7: Existence of a module/service. The module/service must exist.

SCENARIO1:

SCENARIO2:

SCENARIO3:

D8: Resource behavior of a module/service. The resource consumption or ownership of a module/service must be consistent with a client's assumptions.

SCENARIO1:

SCENARIO2:

SCENARIO3:

Appendix B

Test Environment

As indicated in 3.2, in order to develop the Proof of Concept in a realistic environment, I had to install and also develop some scripts, plugins and prototypes. This chapter gives an overview of the different components of the Test Environment that has been used. Section B.1 and B.2 presents the two candidate issue trackers. Section B.3 presents future important components of the Miles Platform, namely the Project and Customer Web. Section B.4 presents Hudson and gives an overview of the plugin that was written for it. Section B.5 presents Eclipse and describes the extension to the Mylyn Plugin that was implemented. Section B.6 presents Subversion and explains the motivation for creating a script that expands the commit logs. Section B.7 presents the Wrapper Layer that was used as a basis for the three layers, by providing a transparent interface to Trac and Jira. Finally, Section B.8 concludes the chapter.

B.1 Jira

Jira is a partial Open Source, widely used issue tracker. It is currently used by Miles. Jira was installed on a Tomcat web container. Version 3.12.1 of Jira was used for this Proof of Concept.

B.1.1 Jira Plugin

In order to be able to fetch the ids of only the recently updated issues I had to write a plugin to Jira. This plugin adds another SOAP interface to Jira. See Appendix C for additional details on the source code.

B.2 Trac

Trac is an Open Source issue tracker that could be a future competitor to Jira. Trac was configured to be run with Apache 2. Several plugins had to be installed in order to access Trac through a remote interface. Version 0.10.4 of Trac was used for this Proof of Concept. See Appendix C for additional details.

B.3 Project and Customer Web

The Project Web is planned to be an interface for developers working with the Miles Platform. Customer Web is respectively planned to be an interface for customers. These web interfaces should provide services and features that are beneficial to customers and developers as well as project managers.

As these interfaces are currently on the planning stage, it is hard to test them against this Proof of Concept. However, as they will most likely be developed in-house (at least partially, IBMs Jazz is a candidate for Project Web), no plugins for communicating with Trac or Jira or other Bug Trackers exists. It could be reasonable to believe that the Project and Customer Web could be the applications that would benefit the most from this Proof of Concept, as they would be very tightly coupled to the issue trackers without this Proof of Concept.

To demonstrate how such applications could benefit from the Proof of Concept, I created a prototype for the Project Web using Grails [4]. This simple client was used as an administrator interface for configuring which project uses which issue tracker. See Appendix C for details on the source code.

B.4 Hudson

Hudson is an automatic building system, which implies that when you check in code in for instance Subversion, Hudson executes a build and creates a report and notification from the result of the build. Furthermore, there exists plugins to integrate Hudson with both Trac and Jira. Hudson recognizes issue keys in the commit logs at the Subversion repository and then includes links to the issues in the generated report.

B.4.1 Plugin

In order to reduce the coupling between Hudson and the Issue trackers, I developed a Miles Platform Plugin for Hudson that recognizes Miles Platform issue keys formatted as {projectkey}/{issuekey} and verifies that these issues exists.

However, it might be more beneficial for the Hudson plugin to actually fetch the issuekeys from some part of the Proof of Concept. See Appendix C for details on the source code.

B.5 Eclipse

Eclipse is a widely used Integrated Development Environment (IDE). There exists plugins to integrate Eclipse with both Subversion and Issue Trackers such as Jira and Trac.

B.5.1 Mylyn Plugin

To avoid the direct coupling between Eclipse and the Issue Trackers, the candidate developed a Miles Platform extension to Mylyn. Mylyn is an Eclipse Plugin that enables integration with Jira, Trac and many other issue trackers. Mylyn provides a framework that makes it easy to add support for other issue trackers, or more specifically this Proof of Concept. Due to time constraints this Mylyn Extension was not finished. But it is able to fetch issues and a few of their attributes from the endpoint. Creating a new issue is also partially supported. See Appendix C for details on the source code.

B.6 Subversion

Subversion is a repository that is used to store source code etc. Eclipse, Jira, Trac and Hudson are all possible clients to Subversion repositories. They also interact indirectly by committing to and reading from Subversion. Jira and Trac has plugins that links issues to source code and revisions. Hudson could also link to Trac and Jira based on issue keys found in Subversion commit logs. See the Appendix C for configuration details.

B.6.1 Expand log script

Both Trac and Jira has support for Subversion (through widely used plugins). These plugins recognizes issue keys in the commit logs in Subversion repositories. If a known issue key is found, then an issue can be linked to files and revisions in Subversion. However, this would imply that a user of the Miles Platform needs to be aware of which Bug Tracker is being used in order to specify the correct issue key. Jira keys are formatted as {projectname}-{issue} while Trac keys are formatted as #{issue}. To avoid this tight coupling a script was created that would expand the commit log with keys for the relevant issue trackers. The script takes a Miles Platform issue key, formatted as {projectname}/{issue} and appends the Jira and Trac equivalent keys to the commit log. The script are executed using two Subversion hooks:

post-commit hook is a hook that is executed by Subversion when a commit is finished. Such a hook was written in order execute the Perl script that expands the log with Jira and Trac issuekeys if a Miles Platform issue key is found in the commit log.

pre-revprop-change hook is a hook that is executed before a revprop(erty) is changed. The svn:log is the revprop used by the script. A simple pre-revprop-change hook was needed in order to allow changes to the commit log.

However, as discussed in Section B.4.1 it might be more beneficial for the Hudson plugin etc. to just fetch the issue keys from the Proof of Concept directly in order to avoid duplication etc.

B.7 Wrapper layer

In order to communicate with the issue trackers, a wrapper layer was written in Java. The most important part of this wrapper is the BugTrackerWrapperInterface and its implementing classes TracWrapper and JiraWrapper. The JiraWrapper and TracWrapper connects respectively to the Jira SOAP interface and the Trac XML-RPC interface. All another class that uses this interface needs to know is which Wrapper to instantiate. Besides that, transparent access to the two issue trackers are given. Other issue trackers could of course be supported to by providing a suitable wrapper. The project is named no.miles.mpl.wrapper.api.

Furthermore, to provide a reusable datamodel, the classes Issue, Project, Comment etc. was created and maintained in the project no.miles.mpl.wrapper.datastructures. These classes could then be reused on the client side. This project is of course referenced by no.miles.mpl.wrapper.api. See Appendix C for details regarding the source code.

B.8 Conclusion

In order to test and use the Proof of Concept, many applications needed to be installed and configured. Furthermore, I developed a few applications and plugins myself.

Appendix C

Source Code

This is an index of the contents of the .zip file provided together with this Master Thesis. The .zip file was uploaded in DAIM (see http://daim.idi.ntnu.no/) together with this Master Thesis.

\README.txt - Includes this index as well as a few installation instructions.

\projects - This folder includes the source code for every project I have implemented.

\environment - This folder contains configuration files for the Test Environment.

\libraries - This folder includes external libraries that are hard to find in the online Maven repositories are included here. Use the ROME and Propono libraries from here, as they have been recompiled by me.