Model Driven Development in a Bundle Based Environment

Authors:
Sebastiaan Daniël Verburg
(0769727)
Rik van Ballegooijen
(0801094)

November 22, 2010
# Contents

Preface 3

1 The Magic Word 4
   Information vs. Software architects .......................... 5
   OSGi ......................................................... 6

2 A Business Case 8
   Design ......................................................... 8
   Functional .................................................. 9
   Data ............................................................ 10
   Non-Functional ............................................. 12
   Technical .................................................... 12
   Implementation .............................................. 13
   Data Domain ................................................ 15
   Controller Domain ......................................... 21
   Presentation Domain ........................................ 22
   Testing ....................................................... 26

3 The Real World 29

4 Conclusion 31

Glossary 33

Acronyms 35

List of Figures 35

Bibliography 37
Preface

In the past various holy grails of software development and design have been found, and buried again. New ideas would always pop up and render older ideas useless. Sometimes the older ideas were revamped, or sometimes simply discarded. As authors of this article we do not claim the described methodology is the ultimate solution to every software related problem, as there is always room for innovation, new ideas and improvement.

Being passionate software developers, always eager to learn better and more progressive techniques, we took a ride through the wonderful world of design and building methods. On this journey new things were constantly discovered raising the excitement, only to drop again after being unable to live up to the expectations. Continuing the discovery of even better ways of achieving the desired utopic software design methodology.

At some point in time we stumbled upon a group of people on a similar journey. This group called themselves the OSGi alliance, on a mission to "design a service platform that ensures interoperability of applications and services delivered and managed via networks". While browsing what they had come up with until now, a sort of eureka feeling overtook us. Rather than reinventing the wheel, we decided to embrace their philosophy.

Doing so, the list seemed endless: scalability, modularity, testability, reusability. But perhaps the greatest was the tolerance for learning on the job. New and revamped ideas will keep surfacing. However, incorporating them without throwing away the experience you gained so far is what we see as the one true advantage.

We hope that the reader finds this article inspiring. We will guide him along a composition of the projects we have done in the past few years and give him a peek on our pitfalls, demanding customers and software projects.
Chapter 1

The Magic Word

Modularity, there, we’ve said it. This word is used in almost every project and on almost every management level, from the code grunt to the not-hindered-by-any-technical-knowledge manager. What is it that makes this word so special? It’s actually quite simple. As a kid most of us have played with blocks. Simple wooden (or any material for that matter) blocks. Every kid, and probably we are no exception to that rule, will try to make a tower out of these blocks. Of course we will start out by just stacking the blocks as high as we can and will fail utterly because the combination of gravity and lack of a solid basis step in. During this process we will come to know that, although we keep using the same blocks, we can create different forms and shapes each having their own advantages and disadvantages. This is where we come to the basis of this metaphor. Compare the wooden building blocks to software architecture and imagine each block, while still being a block, able to execute logic in cooperation with its nearest neighbours. This gives us two big advantages:

- **Separation.** For any system to have a certain level of maintainability separation is crucial to keep a system understandable and maintainable.

- **Re-usability.** Logical blocks can be re-used, thus using proven software without having to go into the process of re-designing the wheel.

The block metaphor can be applied on almost every level of a software project. If we look at the blocks from a programmers point-of-view we can identify them as logical components. An example of a logical component is a logging library. It is a component which provides ways of logging application output to a human readable format. A logging component is something every application needs, it would therefore make no sense to redesign it every time
a new application is developed. Not only would redesigning mean that there is a different set of logging functions available for every application, but also that for every application we are using untested and non proven logging software. Therefore it is in the programmers best interest to work with proven software components which have a strict Application Programming Interface (API) to minimize testing and development effort.

Now, let’s take a look at this metaphor from the management perspective. The manager is usually someone who has little understanding of the technical side of the project, but he does have to keep the customer happy. To achieve this, a manager has to keep track of the development progress in his project. This is best done by splitting up the project into functional components. A functional component has strict boundaries on its functionality and is part of the bigger system. These components are not to be compared with the logical components we discussed earlier in this section. The difference between the two is its relationship to each other. A functional component is comprised out of one or more logical components. By example the functional component "Printing" can use the logical component "Printer Driver", "Printer Status" and "Logging" to perform its task of printing a page. By splitting up the project into these functional components, a manager can now assign budgets, keep track of dependencies between these components and re-assign resources whenever one of the subprojects are at risk of not being delivered on time.

Often modularity is presented as the solution for most of the problems in a project design and management. However as this is partly true, since it supports both the programmer and the manager, its effect is hugely dependant on a number of factors. A manager can decide to split up a project into functional components, but when this decision is not backed up by a logical component approach by the developer the manager will still have a hard time tracking progress and dependencies. So basically, although not to be compared, the blocks, or components, need to share a common ground to fully benefit from their modular design throughout the project. This means that we also need the architects to cover both the functional and technical part in close cooperation.

Information vs. Software architects

At the earliest stage of building a new system, a subtle difference can have huge impact on the resulting software. The designers, usually called archi-
tects, can have different types of backgrounds forming the basis for this difference. The two main types we deal with in the software development world are information- and software-architects. Judging by the two names, one might already guess what the main difference between the two will be. Information-architects will base their design on the flow of information. Business processes are the key to their design. What the end user wants to achieve is the main point of their focus. During development, this approach makes process implementation easier and faster. On the other hand, software-architects often base their design on all processes together. A model holding all information necessary for the entire application is key to their work. Generally speaking, a software-architect will come up with a simpler data model. However, the information-architect based designs will have simpler process flows. Finding the balance between the two is what a good software design is all about.

**OSGi**

The OSGi Alliance (formerly known as the Open Services Gateway initiative, now an obsolete name) is an open standards organization founded in March 1999 that originally specified and continues to maintain the OSGi standard.

OSGi technology is Universal middleware. OSGi technology provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle. These capabilities greatly increase the value of a wide range of computers and devices that use the Java platform. The OSGi specification [3] is developed by the members in an open process and made available to the public free of charge under the OSGi Specification License.

The OSGi framework specification [3] is a module system and service platform for Java that implements a dynamic component model, something that is not provided by the standalone Java/Virtual Machine (VM) environments. Applications or components (coming in the form of bundles for deployment and management) can be installed, started, stopped, updated and uninstalled without requiring a reboot. Life cycle management is done via API’s which allow for remote downloading of management policies. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly.
The OSGi specification [3] allows for a great amount of freedom in the lifecycle of a software product. Because of predetermined API’s and clear specifications on how to design and structure bundles designers can focus mainly on the functional parts of their application and therefore not having to consider any future design implications of the framework.
Chapter 2

A Business Case

In this chapter we will consider a ficticious company that uses a simple two-tier application to handle its user management tasks. This application was designed specifically for the company when it was in the startup phase. Now, because of the growth the company has been through over the years there is a need for a more flexible approach to the user management tasks. The company now has two options; to either upgrade the existing software to cope with the new requirements, or rewrite the software. In this particular case the software was developed by a single developer who left the company, there were no proper requirements and there is a complete lack of technical documentation. Considering these issues the decision was made to rewrite the application entirely. To minimize risk, management also decided to roll out the new application in two phases; phase 1 will be the roll out of the new application and phase 2 will be the conversion and rollout of the new database. With this roll-out scenario the old application can still be used as a fall-back in case any problems arise with the new application. Only when the new application has been tested and accepted by the users, the old data will be migrated into the new database, therefore completely rendering the old application obsolete.

In the initial release, the new application will be functionally the same as the old application. We will not be covering any new feature designs, so we can focus on redesigning the existing features.

Design

The design of a system is always preceded by a set of business requirements. These requirements will have to be translated into a detailed design which will
be used for the technical implementation. We can split up these requirements in two categories:

1. **Functional** requirements define the primary functions and tasks the system should be able to perform once implemented.

2. **Non-Functional** a non-functional requirement is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors.

**Functional**

For the functional design we will be looking at the tasks which the system should be able to perform. This will be done on a high-level overview. We will not be bothering ourselves with minor details about the how, when, where, etc. A functional design should be focused on the end-users of the system. We will not be biased by technical or political limitations. We will only answer the question: "What should the system be able to do?". Now, what should it do? Looking closely at the existing application we are able to derive these use cases:

- **Add User** will allow the system user to add a non-existing user to the system. The process will consist of the user filling in the required data and subsequently submitting that data to the persistent storage. After submitting the new user data, the information should be available to the other use cases.

- **Retrieve User Information** will allow the system user to retrieve existing user information from the system which was filled in during the **Add User** case. This information is a snapshot of the information persisted in the system at the time of retrieval. It will not reflect changes done by another process in the persistent storage.

- **Authenticate User** will compare the input from a user to data stored in the persistent storage. It will retrieve the user information matching to the given input and verifies if the authentication token (password), given in the input, matches the token in the persisted information.

In this business case we will concentrate on these use cases as shown in diagram figure 2.1.
As we can derive from the diagram in figure 2.1 we not only have three separate usecases, but two of them are also related to each other. A use case is not limited to being used by a actual user, in fact there is shared ground between them. In this case, to authenticate a user, the system needs to retrieve information about the user it is trying to authenticate to compare it to the information entered by the user himself.

**Data**

Data modeling often takes place in the early stages of software development. Based on functional requirements, decisions have to be taken on where to store which data, and relationships between certain data. In this phase the model is often referred to as "conceptual schema". Objects, or "things", of any relevance are drawn as a schema, and the relations between these objects are then added to the schema. Later on, when development of the software begins, the model is converted to a "logical schema". For a database model it means converting the conceptual schema into tables with certain columns, and relations between tables. The type of model we will be discussing in this paragraph is called a data model. It is an abstract representation of the data used by the application. Various types of data models exist, but the idea behind them all is the same: provide a definition for storing data in a structural way.

Modern applications often use a database as means of data storage. The organization of the data in the database is called a database model. Often there is a large synergy between the database model and the applications data model. We will discuss this further in the 2 section, in the implementation section. Obviously, this large synergy does not mean full equality. A business
process could easily affect various parts of the data model individually. One solution would be to design the data model based on the business logic, so every single business process is in the same part of the data model. This is in general, however, not a good idea since both models serve a different purpose. To address this problem, software patterns are often used, which will also be discussed in the implementation section.

In this particular case, an old data model existed. In the early stage of the project we found out that it is not a very good one. Since compatibility with the old database (model) is one of the requirements, we have no choice but to write software that works with the old database model.

![Figure 2.2: Old Database Design.](image)

However, another requirement is maintainability. New software is a big investment, and the customer learned from this experience that a complete rewrite is expensive. For the next decade the new application should be easily maintainable and extendable. This is the main reason we have chosen to redesign the database model, and start using it in phase 2 of the project.

![Figure 2.3: Database Re-Design](image)

As indicated in figure 2.3, some more tables were introduced in comparison to the old situation. This is to have single-purpose tables. In the old model, the Users table was also used to store groups and roles. While this is possible, it introduces a lot of room for error. For example, a user could be a descendant of another user. Also, a field indicating if the record was a group or not had type "integer". While a yes/no field was probably meant,
the integer type allows for a numeric value like "5". Obviously, the behavior in such a case is undetermined.

**Non-Functional**

Users have implicit expectations about how well the software will work. These characteristics include how easy the software is to use, how quickly it executes, how reliable it is, and how well it behaves when unexpected conditions arise. The nonfunctional requirements define these aspects about the system and should be defined as precisely as possible. Often, this is done by quantifying them. Where possible, the nonfunctional requirements should provide specific measurements that the software must meet. The maximum number of seconds it must take to perform a task, the maximum size of a database on disk, the number of hours per day a system must be available, and the number of concurrent users supported are examples of requirements that the software must implement but do not change its behavior.

**Technical**

A part of the design that is often overlooked at first, is the technical design. Thinking about the environment the software will run in is nevertheless important. Some questions that should be addressed:

- Will the application and it’s database(s) be able to run on one machine?
- If so, is this desired? Are there objections?
- Which operating system should be used to run the application? Are there restrictions on which ones we can use?
- Can the network manage the amount of data we wish to transfer?
- Is a firewall used in the network? and what impact does it have on our application?

Obviously, many more questions can arise. Trying to address them before the implementation starts has great benefits. To give an example: During implementation a developer might want to use operating system specific features. If nothing is known about the operating system of choice, the developer will have to keep all options open and refrain from using these operating system specific features. This can increase the complexity of the implementation,
even though it might prove unnecessary in the end. In this business case, the application will run with Microsoft Windows as the operating system, as the old application ran on it, and the company does not want to replace their client systems. The choice for the server machine is affected in the same way. The customer prefers not to buy a new server, so their existing unix system will serve the application and databases: both the new and old.

Another topic of interest regarding the technical design is maintenance. A good technical design is key to low maintenance costs. Keeping the amount of systems as low as possible reduce the time necessary to keep each system running. Also, limiting the types of systems used lowers the level of knowledge required by system administrators. In the long run, both have high impact on the maintenance costs. Projecting this back on the business case we’re discussing, our choice of keeping existing hardware and infrastructure in place fits within our strategy. Only one server system is used and an existing system administrator is present. He has knowledge of the system as he has been maintaining it for several years now.

Furthermore, future software upgrades are also something to take into account when creating the technical design. They make up a large portion of maintenance, and thinking about how upgrades can be rolled out can have big influence on the amount of time needed for it later on. Asking questions about the release frequency and extendability of the application on beforehand should result in the desired upgrade strategy best fit for the application. In this case, a high release frequency is expected as the first version of the application will only contain a minimal feature set: a copy of the existing applications functionality. A short release cycle is expected, introducing new features in a step by step fashion. As a result of this, we chose to use an update server in combination with an auto-update feature in the application. At any given time the system administrator can publish a new version of the software or even a single bundle to this server. The next time the application is started, the update will automatically be installed. This approach extends the implementation a bit, as this feature will have to be built. However, the estimation is that this one-time extension will be cheaper than the increase in maintenance costs over time.

**Implementation**

For the implementation, we will divide the application in 3 different layers:

1. **Data**, in this domain all the data will be persisted and retrieved. This
layer has no specific knowledge of the data contents, it only concerns are with the structure of the data being valid.

2. **Controller**, in this domain the data is retrieved from the data domain and formatted, merged or split to business specification.

3. **Presentation**, in this domain the data is finally presented to the end-user. This can be through any kind of media.

The biggest advantage of splitting up an application into layers (figure 2.4) is modularity. Since the presentation layer has no functionality for data storage we can alter, remove or replace the layer without having to worry for the data breaking down. It also works the other way around; we can alter, remove or replace the data layer as well, as long as we honor the agreements made in the API.

![Figure 2.4: Implementation Domains](image)

Looking at figure 2.4 we can see another big advantage of the layered approach. The layers only talk to the layer directly beneath them, they have no clue to which layer is on top of them nor which layer comes after the one directly below. Now we know that whenever a problem arises in one of the components we only have to look at the component itself, or the one directly
beneath to find the cause of the issue. Now we must define the components within these layers. We will not go into extreme detail, we will just specify high-over components and their interface definitions.

![Component Diagram](image)

**Figure 2.5: Component Diagram**

The component diagram in figure 2.5 shows us in which domain we will be placing the system’s components and how they communicate with each other. We are now ready to look at each layer separately.

**Data Domain**

The data layer is the most basic layer of the entire system. It’s sole purpose is to retrieve and store data. This means that the data layer will only look at the structure of the data presented to him. If the structural definitions are met, the data layer will pass the data through. It will not be bothered about the contents of the data.

Back to the business case. We designed the data layer handling as follows...
The data layer as designed and discussed in section 2 has a twist. Normally we would be dealing with one data provider and one data structure to provide the system of data. However in this case we will be dealing with two providers and two different structures.

This poses a serious problem. The datalayer has an obligation to the rest of the system. It must provide it’s data in a single structure. Imagine yourself peeling bananas and each time you are done peeling one banana there is a guy handing you a new one to peel. Now, you are happily peeling the bananas till, at some point, the guy hands you a pineapple…yes, that will turn your world upside down. The same will happen to the system, it will not be able to cope with the change in datastructure and will probably refuse to work. We are, of course, not the only one who have faced this problem and there are numerous things we can do to cope with this problem. However the solution must not break our separation of domains. Therefore
we can not fix it in such a way that the Controller domain becomes aware of
the two different data structures.

The solution lies within perhaps one of the most widely used software
patterns. The adapter pattern. It’s task: to turn an incompatible interface
into a compatible one. The translation between these two interfaces is done
in a special object, the adapter.

As we can see from 2.8 we have a client object which needs to call
\texttt{adaptee.MethodB() }but alas, this method is out of reach for 	exttt{Client}, because
it has no knowledge of the \texttt{Adaptee} object. Of course we could make a refer-
ence to the \texttt{Adaptee} object, but that would add a lot of complexity to the
\texttt{Client} object. It does however have a reference to the \texttt{Adaptor} object. This
is where the adapter pattern works it’s magic, it forwards the call to \texttt{adap-
tor.MethodA()} to \texttt{adaptee.MethodB()} on the \texttt{Adaptee} object. There we have
it, the \texttt{Client} object still only has to deal with the \texttt{Adaptor} object and does
not require any extra knowledge about the \texttt{Adaptee} object!

If we apply the Adapter pattern in our data domain, we can solve our
problem with the two data structures. We are now ready to begin designing
how our classes in the data domain will look like.
Figure 2.9 gives us a detailed view of the data layer structure. We have created the interfaces User, Group, and UserFactory as our definitions to the other components. In the old system, the user and group data was combined in one data structure, but now we have created a separation between the user and group data and both now reside in their own data structure. This means that we will have to create two adapters to satisfy the need of the controller domain for a separate user and group structure. We will start with the UserAdapter:
Implementation

Chapter 2. A Business Case

Figure 2.10: Operation sequence, with the use of adapters

```java
package nl.codecast.example;
public class UserAdapter implements User {
    protected OldUserBean oldUserBean;
    ... implementation ...
    @Override
    public String getFullName() {
        return oldUserBean.getFullName();
    }
    @Override
    public Group getGroup() {
        GroupAdapter adapter = new GroupAdapter(oldUserBean);
        return adapter;
    }
}
```

Listing 2.1: UserAdapter

This adapter holds a reference to the OldUserBean and uses that as it’s source of data. The adapter is the only one with knowledge about the old structure. The rest of the system will simply address the UserAdapter as if it were a entity on it’s own.

Whenever the system calls for the group object belonging to this user the adapter will create a GroupAdapter object with the argument being his own
datasource. The same principle is applied to the GroupAdapter:

```java
package nl.codecast.example;

public class GroupAdapter implements Group {

    protected OldUserBean oldUserBean;

    ... implementation ...

    @Override
    public String getName() {
        return oldUserBean.getGroupName();
    }
}
```

Listing 2.2: GroupAdapter

We have solved the problem of the two different datastructures. The datastructure which was not compatible with our interfaces defined in the component diagram is adapted to pose as the structure defined by the interfaces.

```java
package nl.codecast.example;

public interface UserFactory {
    public User getUser(String username);
}
```

Listing 2.3: nl.codecast.example.UserFactory.java

The UserFactory allows the controller to utilize different ways to retrieve Users from the system. Lets have a look at how the system would retrieve an olduserbean, which has been adepted by the system.

```java
package nl.codecast.example;

public class OldUserFactory implements UserFactory {

    @Override
    public User getUser(String username) {
        OldUserBean oldUserBean = ...Get OldUserbean from ORM Mapper...
        UserAdapter userAdapter = new UserAdapter(oldUserBean);
        return userAdapter;
    }
}
```

Listing 2.4: OldUserFactory

```java
package nl.codecast.example;

public class NewUserFactory implements UserFactory {
```
Implementation

Chapter 2. A Business Case

3
4    @Override
5    public User getUser(String username) {
6        UserBean userBean = ...Get userbean from ORM Mapper...
7        return userBean; //userBean implements User so we can
8            return it directly
9    }

Listing 2.5: NewUserFactory

Depending on the factory which is used by the system, we can return any kind of object to the system, as long as it implements the User interface.

Controller Domain

Like the name suggests, the controller domain is responsible for controlling the application. It collects input and adapts the model or changes the view where necessary. In modern programming languages a controller often uses events to achieve its goals.

Figure 2.11: Event Listeners

The event source could be the user pressing a key on his keyboard, or moving his mouse. But also something like a webservice being called, or even a piece of code being executed. An event object is then created and sent to whoever is interested. The common terminology here is to say an event is fired, and several listeners are interested, as described in figure X. These listeners are still members of the controller domain, and are a common place for business logic. When certain events happen, the implementation behind the listener can apply the desired business logic and commit any changes made to the data model.

The business case presented in this chapter is no exception. User interface input will trigger events: mouse clicks and key presses will be captured. Depending on the state of the view, for example a text field might be focused and pressing a key should add the key typed to its current value. What will
happen is the key pressed will trigger an event. A listener picks up the event, and changes the text fields model. The view is then notified to update its model, so the user gets visual confirmation.

![Diagram showing the relationship between User, View, Controller, Model, and Application domains.](image)

**Figure 2.12: Controller in relation to other domains**

While updating the model we see the benefit of putting the adapter in the data domain. The differences between the old and the new database model are completely irrelevant to the controller domain, which simply talks to "the data model". Even the complete absence of a database is no problem for the controller. This decreases the impact of the dual database model design. The same thing goes for any future database changes. They will remain hidden for the view and controller domains until either one is explicitly changed to do something with the model change. For example: the view must now show a field that was added to the data model.

Reflecting back on our bundle based design: a bundle can fire events through event routing structures provided by the framework. This will allow other bundles to attach a listener to the same structure, and get notified of the events fired. A simple mechanism for bundle cooperation without creating dependencies between them.

**Presentation Domain**

The presentation domain is the domain the customer will actually get to see. A lot of developers either try to hard to create a fancy presentation domain that they forget the most important thing: it’s *purpose*. The purpose of
a presentation domain is that information gets conveyed to the client and actions get processed by the controller. Nothing more, nothing less. Before everybody starts screaming, that doesn’t mean a design is not needed either, we wouldn’t want the client to have to look at Extensible Markup Language (XML) or something even less readable. The information must be shown to the client in a orderly natural manner. It is therefore absolutely necessary to create impressions of how the Graphical User Interface (GUI) eventually will look like, so we know for sure a client has the same idea about orderly and natural as we do. We will do this by creating a mockup of the screens we will be building.

Figure 2.13: Mockup for the user overview screen

Figure 2.13 shows the administrative overview of the users already in the system. We have a few basic tasks of which some have been described in the functional design 2. Some of you might jump up in your chair and yell: "Hey! But its all white and sloppy…This is not how it will look like!". Indeed, this is not how it will look like. It is an impression!
Figure 2.14: Mockup for the edit/add user screen

Figure 2.14 shows the add process for a user. It looks simple and that is exactly the idea.

Figure 2.15: Mockup for the group overview screen
Mockup 2.15 shows the administrative overview of the users already in
the system. Note that we have less columns than with the user table. This
immediately changes the look and feel of the screen.

Figure 2.16: Mockup for the edit/add group screen

Figure 2.16 shows the add process for a group. Although not specified,
this is something which should be shown. Because of the small amount of
data contained within a group, the management screen looks empty. For a
client this could be a reason to request this screen to be merged with another
screen to prevent cluttering of functionality.
The authentication screen shown in figure 2.17 is designed in such a way that the client has no way of not knowing what to do. The entry for credentials is centered and there is only one button to press. Simple, but effective and that is what a GUI should be.

Testing

There are two main types of testing when it comes to software development: technical and functional tests. The first is often referred to as unit testing. In the following paragraph we will explain the both, focusing on the benefits they offer in this particular case.

Technical Tests

Unit testing is a often neglected part of software implementation. As project deadlines draw closer, these tests seem an easy target for saving time (often second to technical documentation). Even though these technical tests take quite some time to implement, the time spent is usually won back during the maintenance stage. Especially core functions of the application are good candidates for unit testing, as they are used frequently and any error in them will have high impact on the application.
A typical unit test consists of two parts: input and expected output. Given the input, and our idea behind a certain function, we expect it to respond in a certain way.

```java
@Test(expected=IllegalArgumentException.class)
public void testServiceReferenceImplNullBundle() {
    new ServiceReferenceImpl(serviceRegistrationImpl,
    serviceRegistry, null);
}
```

Listing 2.6: A typical unit test

As code example 2.6 shows, a test can be small and simple. We expect that our input will result in the function returning an exception of type "IllegalArgumentException". If something else happens, the unit test will fail. This means either our expectations were incorrect or the software is not working in the way we want it to.

With our bundle based working style, unit testing offers an additional benefit based on the principle of separation of concerns. Every bundle offers a limited amount of features to the rest of the framework. When we unit test these features and all tests succeed we know for sure that the bundle is working correctly. This helps a great deal in pinpointing the location of issues.

**Functional tests**

The main goal of functional tests is user acceptance. Does the software work as expected by the end user? A common problem is the difference between what the customer wants, and what is written down in the functional requirements document. While these two match in theory, they often differ in the real world. Vague description of certain elements force the developer to interpret, and eventually create what he think was meant. This room for interpretation is the cause of the concern. However, while high quality functional documents reduce the problem, it is impossible to write a fool-proof specification. Following the common software development cycle, functional testing would occur at the end. This poses a real threat, as we find out the wrong things might have been built when the build is completed.

Luckily, we use a modular approach wherein a module is tested on completion. Depending on the type of module, a combination of technical and function tests can be executed. If the module only delivers technical features, unit tests will suffice. When end user functionality is added by the
module, the functionality can be tested. This does not only counter the threat mentioned, but it also allows for testing and development in parallel.

In our case, defining the functional tests should not be too complicated, as the new applications functionality should be the same as the old one. With both applications running on the same database (in phase 1), changing anything in either application, should reflect in the other. A team of experienced users will execute the test scenarios defined. They will work with the application on a daily basis, and the acceptance of the new application depends on this group.
Chapter 3

The Real World

In the current software development industry, various implementations of the OSGi specification exist [13] [11] [10]. As they all confirm to these specifications, most bundles are interchangeable. This offers us with a huge amount of bundles already available for use. While it seems the advantages are endless, there are also various drawbacks. Perhaps one of the main advantages is also the major drawback: strictness. Guaranteed compatibility even with other projects using the same specification allows for a huge library of bundles ready to use. However, this compatibility exists only for bundles. Actually, any other type of software is almost certainly incompatible. With the majority of software out in the world not following OSGi specifications, it might still be necessary to rewrite existing software to be used in the framework.

Another advantage and disadvantage in one is the completeness of the specification. It is very detailed and goes down to the core of the Java programming language. This can cause compatibility issues with native tools delivered by Java’s maintainer Sun. A good example of this problem is a tool called WebStart. It is used to deploy and start a Java application through a web environment. The user first downloads the latest version of the application, which is then started on the local machine. This mechanism of loading and starting is incompatible with OSGi’s own loading and starting mechanism. While a workaround might be possible, most companies will opt for an own deployment tool. Obviously, this gives a feeling of reinventing the wheel.

Both disadvantages mentioned are about a decrease in production hence, extra work that needs to be done. This effect is more or less countered by the fact that the extra work has to be done only once. If a certain organization develops a OSGi-compatible deployment tool, and releases it to the public,
others will not have to build their own.

Key to success here is the widespread use of this methodology. A large number of people, companies and organizations use the OSGi framework. Its open source characteristics reduce the chance you will be the first to need a certain feature and increases the chance that it’s already available. Of course this is a philosophy that should fit the person or organization using it.
Chapter 4

Conclusion

The approach taken in this document is an example of how the authors think a software project should be executed. Splitting up the title into two parts gives us the two core steps in our strategy.

**Model driven development** is our first target: deriving a model from the requirements. At first the business model is created, with the data model following suit. This ensures us a proper way of dealing with data inside the application. This is a key step in the design, as the entire application will be using data through the models created here.

Next, **the bundle based way of working**. As we have seen in the case described in chapter 3, using a modular approach in software design can have great advantages. Probably the most important advantage is the so called "separation of concerns". Every bundle has certain tasks and these can be tested and accepted individually. When an issue arises, pinpointing its source will be easier due to the strict responsibilities known for every bundle. Bundles can depend on other bundles, or even specific versions of these other bundles. This is a great help when dealing with software upgrades while maintaining the desired backward compatibility.

This case has taught us how modular software helps us in replacing existing code with ease and with little impact on the rest of the application. However, to fully profit from the advantages of our modular design, we need to extend its use to outside the software design. That’s where the bundle based environment comes in. Having modules that can easily be replaced by newer versions is no use when the software ends up in one big package for distribution. The architecture and environment will also need to support this modular approach. Let’s consider something like a bundle repository, periodically updated with newer versions of bundles. Or a management interface for
installing new bundles and updating existing ones. A system administrators life will become considerably easier with such simple tools, maintaining the application with a relatively low chance on human error.

In the end, both customer and the software company benefit from this strategy. The customer will get an application that has been tested and accepted in a step-by-step fashion, ensuring all requirements have been met. Also the maintenance costs will be reduced and application stability and maturity increased compared to a more static approach. On the other hand, the software company has made an application which is likely to have reusable assets, that can then be used in other projects. Bugs in modules can be resolved faster and will only have to be done once, rather than on a per customer basis. Compared to the static approach, this results in more profitable service level agreements.
Glossary

**bundle** The Framework defines a unit of modularization, called a bundle. A bundle is comprised of Java classes and other resources, which together can provide functions to end users. Bundles can share Java packages among an exporter bundle and an importer bundle in a well-defined way. In the OSGi Service Platform, bundles are the only entities for deploying Java-based applications. 13, 27, 29

**framework** A software framework is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code, thus providing specific functionality. 6, 7, 27, 29, 30

**Java** Java is a programming language originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995 as a core component of Sun Microsystems’ Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. 6, 29

**middleware** Middleware is computer software that connects software components or some people and their applications. 6

**mockup** A user interface prototype. A mockup is an simple impression of the final product. They are used for early impressions of a system and as a guideline for the User Interface developer. 23

**OSGi** The OSGi framework is a module system and service platform for the Java programming language that implements a complete and dy-
namic component model, something that does not exist in standalone Java/VM environments. 6, 7, 29, 30
Acronyms

API Application Programming Interface. 5–7, 14

GUI Graphical User Interface. 23, 26

VM Virtual Machine. 6

XML Extensible Markup Language. 23
## List of Figures

1.1 OSGi framework layer overview ........................................... 7
2.1 Use Case Diagram ............................................................ 10
2.2 Old Database Design ......................................................... 11
2.3 Database Re-Design .......................................................... 11
2.4 Implementation Domains ..................................................... 14
2.5 Component Diagram .......................................................... 15
2.6 Operation sequence, without the use of adapters ....................... 16
2.7 Data structures ............................................................... 16
2.8 The Adapter Pattern .......................................................... 17
2.9 Class Diagram With Adapters ............................................. 18
2.10 Operation sequence, with the use of adapters ........................... 19
2.11 Event Listeners ............................................................... 21
2.12 Controller in relation to other domains .................................. 22
2.13 Mockup for the user overview screen .................................... 23
2.14 Mockup for the edit/add user screen ..................................... 24
2.15 Mockup for the group overview screen ................................... 24
2.16 Mockup for the edit/add group screen ................................... 25
2.17 Mockup for the authentication screen ................................... 26
Bibliography


