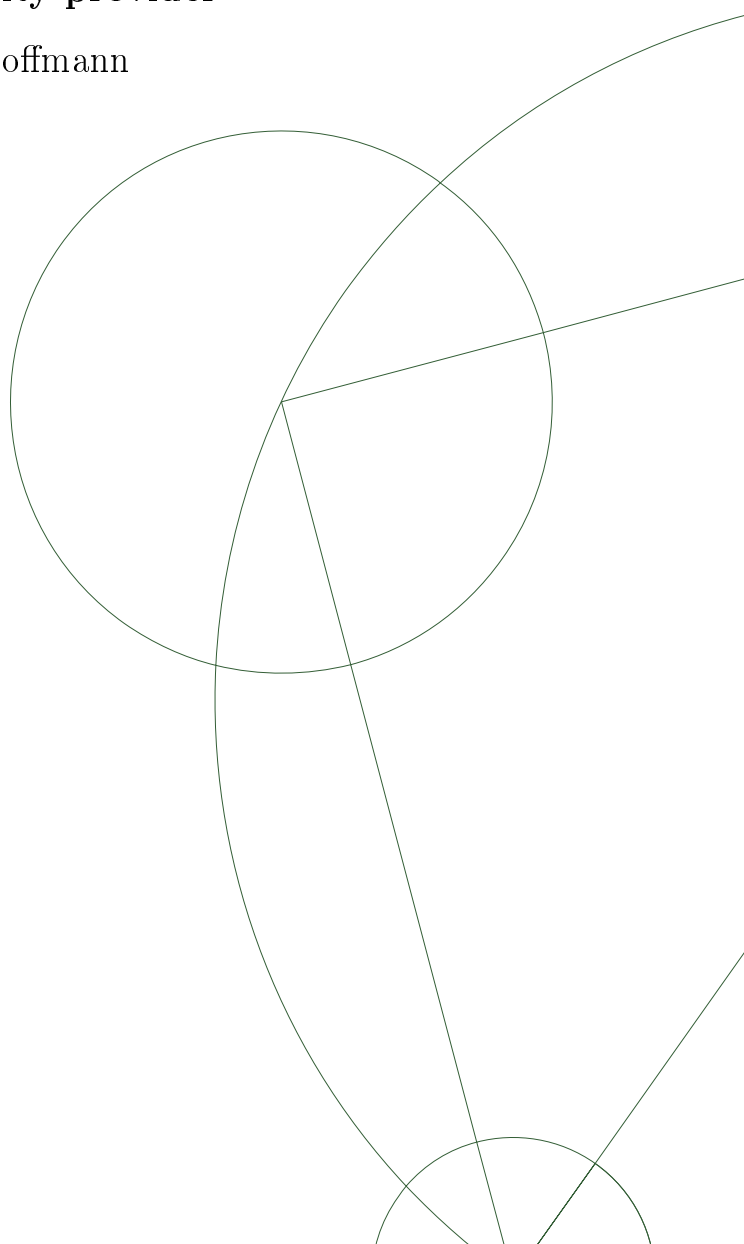
Master's thesis, Department of Computer Science
University of Copenhagen, spring 2009
Supervisor: Jyrki Katajainen

Domain-driven design in action

Designing an identity provider

Klaus Byskov Hoffmann



Abstract. In many scientific disciplines "complexity" is one of the most exciting current topics, as researchers attempt to tackle the messiness of the real world. a software developer has that same prospect when facing a complicated domain that has never been formalized.

In this thesis the principles of domain-driven design are used to model a real-world business problem, namely a framework for an extensible identity provider. A specification for the software is presented and based on this specification a complete model is created, using the principles of domain-driven design which are also presented. The thesis also includes an implementation of a generic domain-driven design framework that leverages object-relational mappers. It is then showed how this framework can be used to implement the designed model. Finally, the quality and completeness of the model is validated through a series of reviews and interviews.

The work shows that applying the principles of domain-driven design is a good approach to modelling a complex business domain.

Resumé. I mange videnskabelige discipliner er kompleksitet et af mest spændende aktuelle emner, idet forskere forsøger at takle den virkelige verdens rod. En software udvikler har den samme udsigt når han står overfor et kompliceret domæne der aldrig er blevet formaliseret.

I dette speciale bruges principperne fra domænedrevet design til at modellere et forretningsproblem fra den virkelige verden, nemlig et rammeværk for en udvidbar identity provider. En specifikation for softwaren præsenteres, og baseret på denne specifikation udvikles en komplet model ved brug af principperne fra domænedrevet design, der ligeledes præsenteres. Specialet indeholder også en implementering af et generisk domænedrevet design rammeværk der gør brug af objeckt-relationelle oversættere. Herefter bliver det vist hvordan dette rammeværk kan bruges til at implementere den designede model. Slutteligt valideres modellens kvalitet og komplethed igennem en række reviews og interviews.

Arbejdet viser at det at benytte principperne fra domænedrevet design er en god tilgang til at modellere komplekse forretningsdomæner.

Contents

Preface

This master's thesis concludes my cand. scient degree in Computer Science at the University of Copenhagen. The thesis was written in the period from September 1st 2008 to May 29th 2009.

Introduction

The thing that determines how complex a problem that can be solved by a piece of software, is not how many programmers you hire to code it but how well you design it. If the design is not thought through from the beginning, and everyone involved in the programming solves their tasks from a varying understanding of the problem domain, then chances are that you will end up with a cluttered code base which is both hard to understand, maintain, and extend. The premises for domain driven design are that each software project should be based on a model and that focus should be on the domain and the domain logic instead of on the technology that is being used. When focusing on the model it is easier to achieve a shared terminology between the domain experts and the programmers, thus breaking down the language barriers that often exist between them. Domain driven design is not a technology or a method, but a mind set and a set of priorities. Domain-driven designdeals with choosing a set of design principles, design patterns and best practices in order to achieve faster development of software dealing with a complex business domain, while at the same time achieving a code base that is maintainable and extensible.

In order to discuss how domain-driven design can be used to design complex software, it is important to me that the discussion is based on a real life problem, and not just on fictitious examples. Therefore, this thesis is based on a project that I am working on with my current employer, a company called Safewhere. On this project, I alone am responsible for the development of the software, which makes the project well suited for this one-man thesis. The project does of course have other stakeholders, such as the technical director and the CEO of Safewhere.

The project at hand concerns the development of a framework needed to implement a multi-protocol identity provider, as described in Chapter 2. The thesis does not include a fully functional version of the product, but does include a complete design and implementation of the core functionality.

To make sure that I have achieved making an easily maintainable and extendible model I have asked a few of my peers to review the model. I have interviewed each person and had them assess the quality of the design and the benefits of using domain-driven design in general.

Chapter overview

The first three chapters contain a general introduction to the problem domain and to domain-driven design. Thus, these chapters mainly contain material from books and papers and my contribution to the contents found herein is merely that of passing on knowledge in a suitable and more compact format. The last six chapters contain my main contribution, where I take the theory from the first chapters and put it into practical use.

Identity provider: This chapter explains, in general terms, what an ***Identity Provider*** (IdP) is.

Specification: This chapter contains the feature specification of the software that is going to be developed.

Domain-driven design: This chapter contains an introduction to domain-driven design and the concepts it uses.

Model: This chapter contains the design model of the software using domain-driven design.

Object-relational mappers: In this chapter I discuss how object-relational mappers can be leveraged in domain-driven design.

A domain neutral component: In this chapter I will present the implementation of a domain neutral component that implements a lot of basis functionality that is useful for implementing a system based on domain-driven design.

Implementation: In this chapter I will showcase the source code for some of the most interesting parts of the implementation of the system.

Design validation: This chapter is based on interviews with some of my peers, and its purpose is to let me assess the quality of the model and the software presented herein.

Conclusion: This chapter contains a conclusion on what has been learned from working with domain driven design.

Tips for the reader

This thesis is written in British English. The thesis is intended to be read from beginning to end, since most of the chapters build on theory explained in previous chapters.

The reader should be familiar with object oriented programming and design. All code samples in this report are in C#, a language of the Microsoft .NET™ family. A code sample may look like this:

A code sample

```
public void DoNothing(int i, int j)
{
    //Do nothing with the input data and return
    return;
}
```

If you are not familiar with C# you may refer to the Appendix A, which contains a short introduction to the language.

The developed code can be found on the enclosed disk. Having Microsoft Visual Studio 2008 installed is a prerequisite for opening the solution file, however each code file can be inspected in any text editor.

Acknowledgements

First of all I wish to thank my supervisor, professor Jyrki Katajainen for his support and thorough dedication to the project. I also want to thank the technical director at Safewhere, Peter Haastrup, for his sincere interest in the project, for shielding me from things I did not need to bothered with, and for keeping encouraging me throughout the entire process. I also wish to thank the CEO at Safewhere, Niels Flensted-Jensen, for letting me write my thesis in collaboration with the company, and Mark Seemann, Mikkel Christensen and Peter Haastrup for taking time to review the model and participate in interviews. Finally I wish to thank my beautiful wife Gitte Byskov Hoffmann for bearing with me on the late nights and weekends where a substantial part of my time has been spent studying for and writing on this thesis.

1
Identity provider

An identity provider (IdP) is a centralized identity service whose primary responsibility is identifying a user and stating something about that user, such as the users name, email or CPR number. The IdP identifies the user based on some credentials and the statements that it makes about the user are known as claims. Figure 1 shows the primary work of an identity

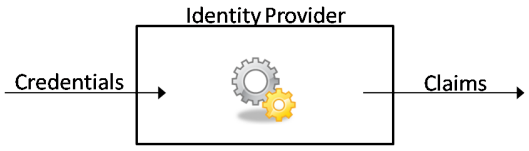


Figure 1. An identity provider.

provider. In general terms it can be said that an IdP performs the function $f(\epsilon) = \zeta : \zeta \in \phi$ where ϵ is a given set of credentials and ζ is a set of claims from the complete set ϕ of claims that can be issued by the IdP. Throughout this thesis I will refer to $f(\epsilon)$ as ***token issuance***. ζ will be referred to as ***token***, ϵ will be referred to as ***user credentials*** and finally I will refer to ϕ as ***identity provider claims***

1.1. IdP use case example

The advent of a range of new technologies such as SOAP web services, Ajax, new browser capabilities and higher internet bandwidth has made it feasible for ISV's to provide software that is hosted in the cloud. This type of software is referred to as software as a service. Buying software as a service is becoming a more and more popular for a number of reasons. Primarily because the total cost of ownership for software as a service vs. self-hosted software tends to be lower. In fact, according to [Daarbak 2008], the analyst company Gartner predicts that 30 percent of all CRM systems will be running on the software as a service model in the year 2012. Software as a service solutions are hosted in large data centers and provide the ability

to quickly scale the application for a given customer should the need arise. This extra flexibility is another selling point for "software as a service" solutions. ISV's that sell software as a service solutions may choose to use claims based access control in their applications. Figure 2 below shows a fictitious company that uses two SaaS applications in the cloud.

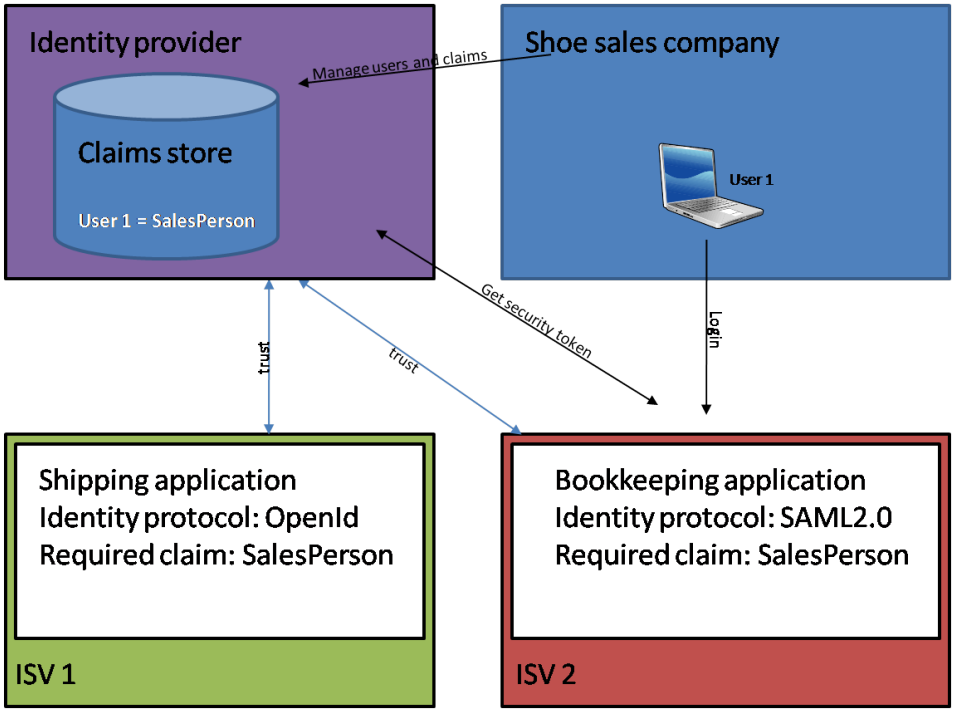


Figure 2. IdP use case.

The shoe sales company uses a shipping application and a bookkeeping application in the cloud. The bookkeeping application uses the SAML2.0 identity protocol and the shipping application uses the OpenId protocol. When user 1 wants to log in to the bookkeeping application, he is asked to go and get a security token from an identity provider that is trusted by the bookkeeping application. The identity provider now issues a security token containing a claim that the holder is a "SalesPerson". Since this is the claim required by the bookkeeping application, User 1 is now allowed to use its features. When user 1 later wants to go and use the shipping application, the same thing happens. The only difference is that the shipping application uses another identity protocol, namely OpenId. This setup has several benefits. First of all, user 1 only has to log in once within a session, since the identity provider will remember him. Secondly, the shoe sales company has one single

place where it can manage it's users and the claims that are issued about them. This means that it would be possible to revoke user 1's access to both the shipping and bookkeeping application by revoking his SalesPerson claim in the IdP. Last, the IdP allows the shoe sales company to easily integrate with software as a service products that use different identity protocols. Please note that the IdP shown here is actually a "software as a service" product itself, although conceptually it could just as well be a stand-alone application in the shoe sales company.

It is the design and implementation of an identity provider such as the one in Figure 2 that is the goal of this thesis.

2 Specification

This chapter includes the feature specification of the identity provider software that is going to be designed and implemented. The chapter will contain requirements to both the software and the platform on which the software runs. The main focus of this thesis is, of course, the software, however, considerations about the platform are important for a complete understanding of the system, and therefore they are included here for completeness.

Before we go into the specific details, I want to present the following text, taken from a marketing brochure made by Safewhere, to give you an idea of what the project is about, from a business perspective. In the following text the product is referred to as Safewhere*Identify, a term that I will ***not*** use throughout the remainder of the thesis.

> *Safewhere*identify is a new kind of user identification solution providing for seamless and heterogeneous authentication across the supply chain of web applications and web services. With Safewhere*identify an organization may handle user identification centrally and outside of all web applications and web services. Safewhere*identify supports any kind of authentication including traditional methods such as username/password and X.509 as well as various identity federation mechanisms. Selected benefits and advantages include:*
>
> **Externalization of authentication.** *Applications and services move all authentication to the Safewhere*identify "identity broker", which in turn integrates with any and all authentication mechanisms*
>
> **Federated identities.** *User identities are seamlessly and securely transferred from their origin — the place where the user first logged in — to your infrastructure, thereby removing the need for operation and administration of a local extranet user database*
>
> **Traditional user authentication.** *For external users not ready for federated authentication Safewhere*identify provides integration to local user databases, thus allowing for seamless leverage of your existing investments in user directories.*

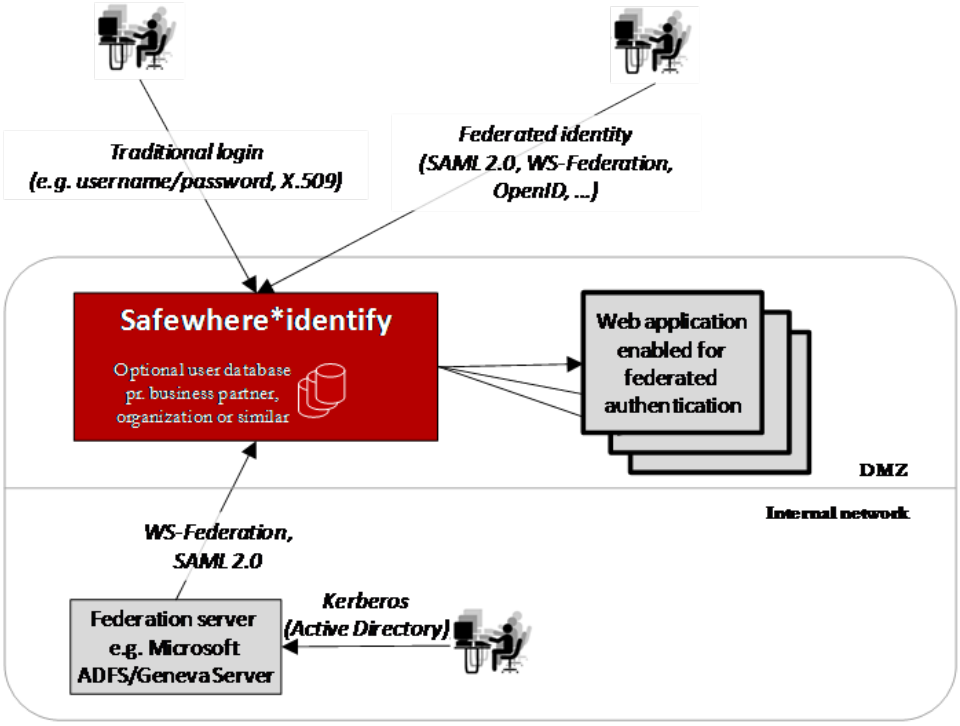


Figure 3. Logical identity flow from users to web applications

Delegated user administration. *Safewhere*identify automatically provisions a user database per organization for authentication purposes and maintenance of other user attributes. Each database belongs to, and is administered by, one organization.*

Service provider integration. *As more and more of your applications leverage the new identity solution, Safewhere*identify provides identity conversion/mapping to successfully transfer identities between applications*

Provided both "as a service" and as traditional on premise software. *Due to the unique and standards compliant architecture and communication patterns, Safewhere*identify may equally well be leveraged as an external service (Software as a Service) or as software installed in your own infrastructure. Whichever is chosen has no effect on feature set or security.*

*Safewhere*identify provides a rich set of features with the aim to remove entirely all need for local administration and authentication*

of users. The key capabilities include:

Browser based federation. *Safewhere*identify implements a number of federation protocols including SAML 2.0 and WS-Federation for browser based authentication*

Federated authentication for Web services *— aka. "active" federation — through WS-Trust and possibly WS-Federation*

One Identity Provider *instance per organization (e.g. each of your extranet partners) providing for full separation of data. Separate IIS Application Pools under different service accounts for each instance ensure very tight security around each organizations data all the way to the file system and database level*

Claims mapping. *Applications leveraging Safewhere*identify potentially all need different kinds of user attributes, or claims, and often with slightly different names and formatting. (E.g. a role to one application is a group to another). Safewhere*identify provides delegated administration of attribute mappings to transparently and correctly transfer identity between applications.*

Self registration *of organizations and users. Workflows support the signing of new organizations — e.g. new business partners — as well new uses of each organization. The first requires review and approval of you whereas the latter leverages the distributed nature of user administration and leaves it up to the user's home organization.*

2.1. IdP

The main difference between this IdP and "standard" IdP implementations is that it must support more than one "instance". An instance is an isolated IdP that runs side by side on the same server as other instances, but without being coupled to the other instances in any way other than sharing storage. Systems that host multiple instances of the same software for different customers are often called multitenant systems. Designing a multitenant system requires extra considerations regarding data storage layout and data security. Another key feature of the IdP is that it must be extensible in a way that it can support more than one identity exchange protocol and more than one authentication mechanism.

2.2. Technical requirements

The IdP must be developed in C# and Asp.NET. Configuration data will be stored in SqlServer 2008. The IdP will be running on Windows Server 2003/2008 on IIS 7.

2.3. Terms

This chapter makes use of the following terms:

Customer: An organization that has bought an IdP instance.

Administrator: A person who administrates a single IdP instance.

User: An end user who uses an IdP instance to identify himself to a service provider.

2.4. Overview

Figure 4 shows an overview of the SaaS IdP.

The figure shows the following concepts:

IdP instances: The SaaS IdP contains a number of IdP instances. Each IdP instance belongs to a customer, except for one instance, the system instance. There is no implementation difference between the system IdP instance and the customer's IdP instances. The system IdP instance is used to authenticate administrators when they log in to administrate their own IdP instance through the Admin web.

Admin website: The admin website acts as a service provider against the system IdP, and relies on a set of claims issued by the system IdP to determine which IdP instance the administrator belongs to. On the admin website an administrator must be able to configure his IdP instance in a number of ways which will be discussed in further detail below.

Registration website: The registration website is where new customers can sign up for the services provided by the SaaS IdP.

Personal data store: All changes made to IdP configurations through the admin website are written to the personal data store. Each IdP instance

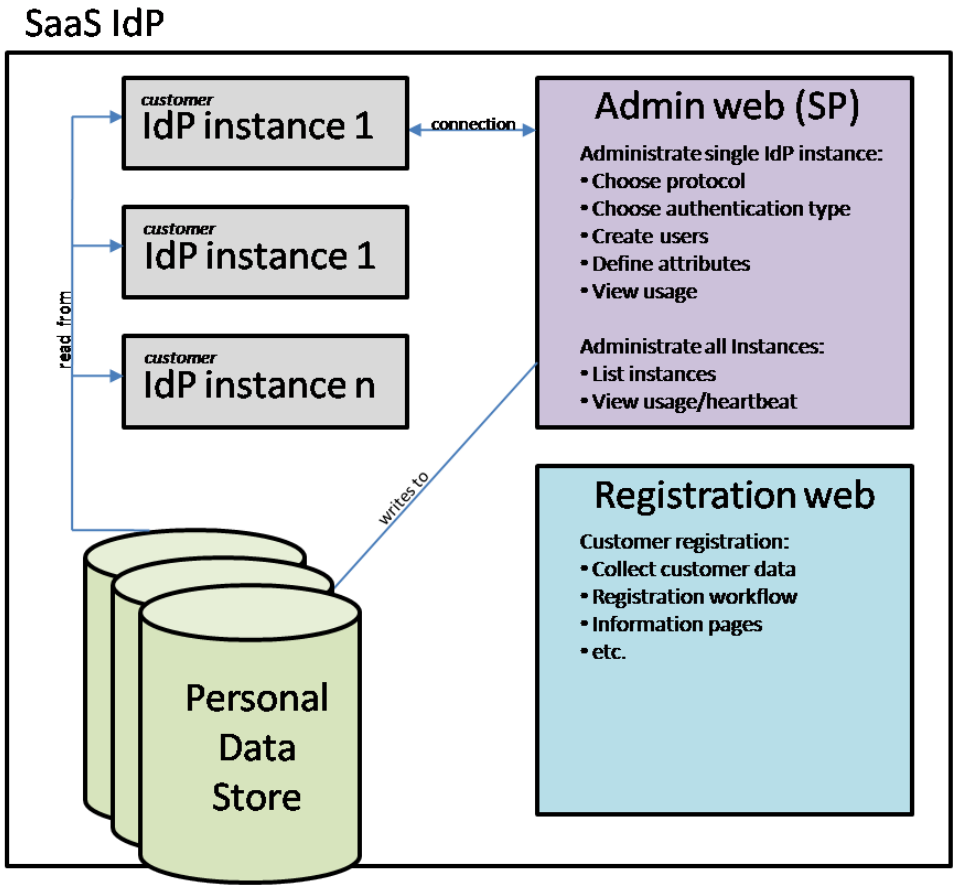


Figure 4. IdP overview.

uses the personal data store to read it's configuration, retrieve claims, authenticate users etc. The personal data store is either a separate database or a separate database schema.

2.5. IdP instance

This section contains the specification of which features a single IdP instance contains. An IdP instance is driven by its configuration. The configuration mainly specifies the following five things:

Credentials: Which types of credentials are accepted.
Protocols: Which protocols does the IdP support.

Claims (attributes): Which claims does the IdP issue.

Certificates: Which certificates does the IdP use for signing and decryption, and which SP certificates are trusted.

Service Providers: Which service providers is the IdP connected to.

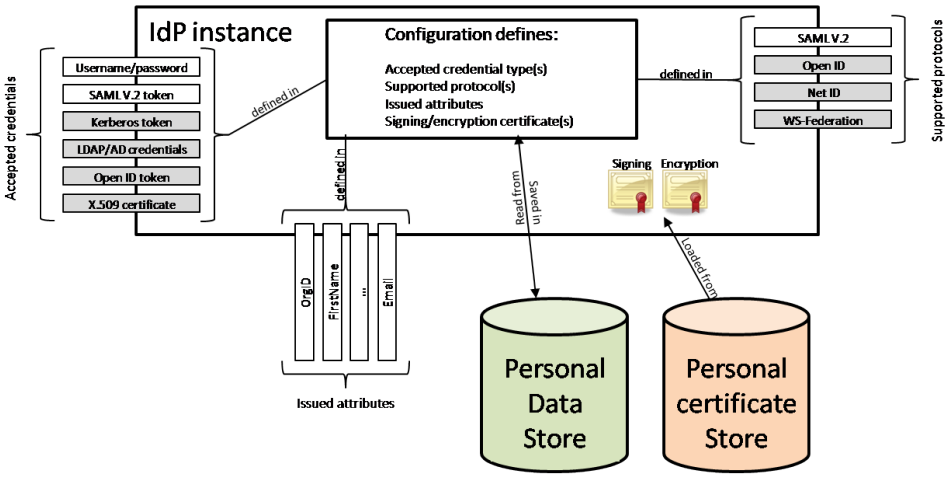


Figure 5. IdP instance.

Figure 5 shows a single IdP instance. The IdP instance may be configured to accept a large range of different credentials. Initially, the only requirement is to support "username/password" and SAML2.0 credentials, but it is important that checking credentials is done in way such that further types of credentials can be added later. The same requirement for being pluggable applies to the protocols that the IdP supports. The only initial requirement is to support the SAML2.0 protocol, however this must be done in a way that does not impede future support for other protocols such as OpenId or WSFederation. As Figure 2 shows, the IdP may also be configured to issue a set of different claims. All the different configuration options are saved in the personal data store. Most identity protocols use certificates for encryption and signing of messages. Therefore the configuration must contain information on which certificates are to be used for this purpose. The certificates themselves must be stored in a personal certificate store on the Windows server. In windows, a certificate store is tied to a system account, so in order to achieve having a personal certificate store for each IdP instance, every instance must run under a separate user account.

2.6. Hosting

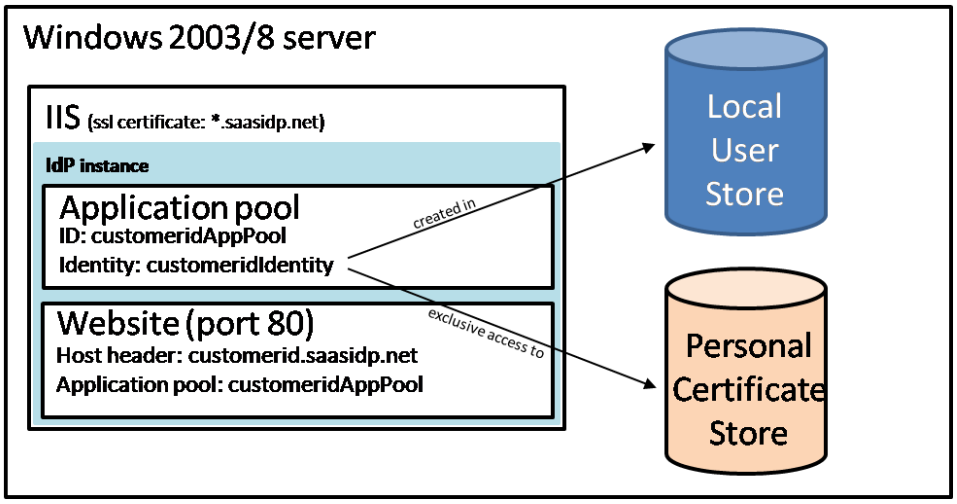


Figure 6. IdP instance hosting.

Figure 6 shows how an IdP instance should be deployed in IIS (Internet Information Services). The IdP instance runs in its own IIS website. Running in its own website means that it can have its own unique domain name, e.g. somecustomeridp.saasidp.com. The alternative would be to have each IdP instance run in an IIS virtual directory, and accessing it through www.saasidp.com/somecustomeridp. However, in order to be able to configure different SSL certificates for each IdP instance it is important that the IdP instance has its own unique domain name. An SSL certificate is always tied directly to a domain name. So if the instances were running in a virtual directory, all instances would be using the same SSL certificate, namely that pertaining to www.saasidp.com. To make SOAP requests over SSL to an IdP instance running in a virtual directory would require trusting the SSL certificate belonging to www.saasidp.com. However, by doing so, a trust relationship would be implicitly made to all IdP instances on the server. To avoid this, each IdP instance must run in its own IIS website and have a unique domain name and therefore also a unique SSL certificate. This would imply explicitly trusting every SSL certificate for each individual IdP instance. Another benefit of separate IIS websites is the ability to tie a website to an application pool. An application pool lets us define the system account under which the website runs. If each IdP instance runs under a different system account, we can achieve having personal certificate stores

for each IdP instance. By having personal certificate stores the operating system ensures that certificates in the personal store can only be accessed by the user to which the store belongs. Furthermore, personal certificate stores provide a way of configuring different trusted certificates, something that is important when importing certificates from federation partners. Having personal certificate stores requires creating a new system account for each IdP instance. This account should be created as part of the installation of each IdP instance. The password for the specific system account should be randomly generated and forgotten. When configuring the IIS application pool, the password should be entered and the remember password feature should be used. By doing this, no one will ever be able to use that account for anything other than running the IdP instance, and the account will be hard to compromise. By using the deployment structure shown in Figure 3, a clear separation of the IdP instances is achieved when it comes to certificates and certificate trust. This is deemed very important since most identity protocols rely on mutual certificate trust.

2.7. IdP instance administration

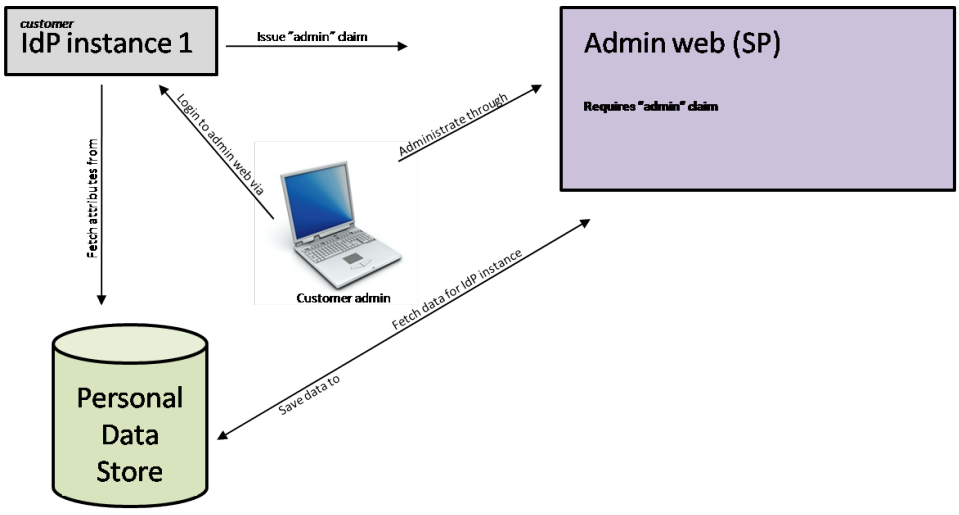


Figure 7. IdP instance administration.

Figure 7 shows how an administrator can administrate his IdP instance through the administration website. The administration website acts as a service provider to the system IdP instance, in which all administrators of

all customer IdP instances have an account. The system IdP issues a set of claims about the specific administrator, the most important claim being an OrgId. Based on the OrgId, the administration website knows which data to get from the data store, and lets the administrator edit them. When changes are made, these are saved back to the shared data store. The customer's running IdP instance uses the data from the personal data store to fetch it's configuration.

2.8. Claim definition

An administrator must be able to define a set of claims for his IdP. The definition of a claim contains the following properties:

Property	**Description**
ClaimType	The type of the claim, for example "Email". There exists a set of predefined claim types, and claim types are typically on URI form. For example: dk:gov:saml:attribute:CvrNumberIdentifier
ClaimValue	This property contains a specific value for the claim. It is used when users are not allowed to override the value, and when only one specific value for the claim makes sense.
DefaultValue	Some claims may have a default value. This property lets the administrator specify a default value for the claim.
UserEditable	This property specifies whether or not the user will be able to edit the value of this claim.
ValidationRule	The administrator must be able to add some kind of validation rule for the value of a claim. This is especially useful when claim values are typed in by users. The IdP should specify a set of predefined rules, but it would probably also be useful if the administrator was able to type in a regular expression do meet specific validation needs.

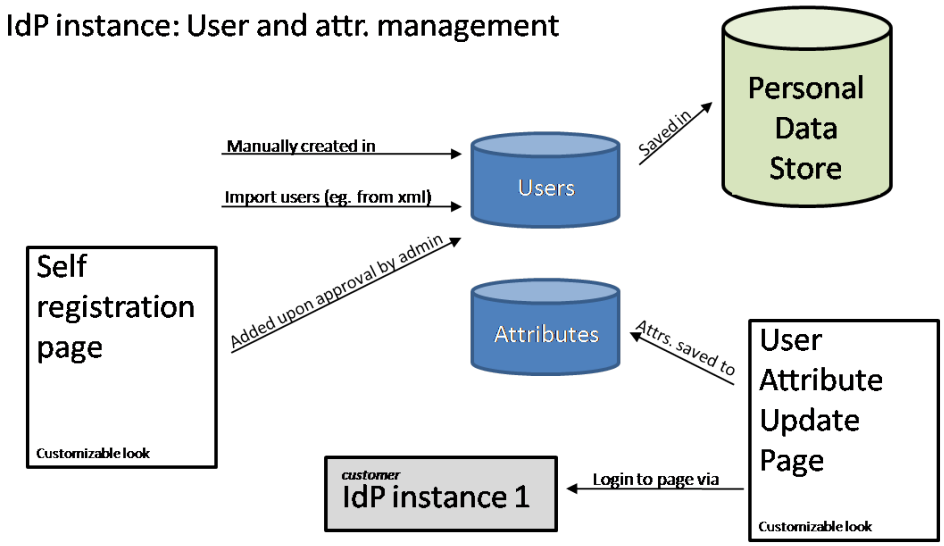


Figure 8. User and attribute management.

2.9. User and attribute administration

Figure 8 shows an overview of user and attribute management. The figure mainly focuses on the username/password case, where users are created in the data store. An organization may have several hundreds or thousands of users, so many that it would be infeasible for a single administrator to create them all. This fact calls for some import functionality. If we assume that a company has defined the claims "FirstName", "LastName" and "Email". The administrator must be able to import a list of email addresses, and the IdP should automatically send an email to each address, asking the recipient to go to a self registration page where he or she can enter a desired username, a password, and the value for the claims defined for the IdP instance, in this case "FirstName" and "LastName". The "Email" claim will be filled out on beforehand because it is already known. When the user has filled out the claim values, each user needs to be approved by the administrator. It should of course also be possible for the administrator to create users manually, but this is considered a special case. It must be possible for a user to change the value of his/her claims, which leads to the need for an update page, shown in the figure as the "user attribute update page". To access the page, the user presents his credentials, and is presented with a list of his claims. It must be possible for the administrator to define which claims can be edited

by the users and which cannot. This property must be defined when the claim is defined. As the figure shows, both the self registration page and the attribute update page must be customizable with regards to look and feel such that it is possible for the customer to make it look like other of their corporate websites.

2.10. Groups

Group claims are no different from any other type of claims. However the IdP must make it easy to add and remove users from groups. Consider a group claim with value "Managers". Behind the scenes, this claim is just another claim in the user's security token, however the administration pages for an IdP instance must provide the ability to work with groups in a familiar manner. This involves listing all groups, listing all users in a group and an easy way to add and remove users from groups. Since group claims are merely expressed as claims, and claims have no hierarchical structure, groups are not hierarchical either. This is important to note, since in some systems groups have a hierarchical structure where a group can be a member of another group. This is not the case in this system.

2.11. Claims mapping

When a connection to a service provider is created, the service provider may specify that it requires some specifically named claim. Figure 2 showed two service providers that both required a "SalesPerson" claim. Let us assume that the "SalesPerson" claim was a group claim with the value "SalesPerson". In some specific customer organization they may have a group claim called "SalesManager", which is conceptually identical to the "SalesPerson" claim. The IdP should allow a simple mapping of claims such that when a user that has the "SalesManager" claim is connecting to the service provider which requires the "SalesPerson" claim, the "SalesManager" claim is automatically mapped to the "SalesPerson" claim. Figure 9 shows the order in which this claims mapping is performed.

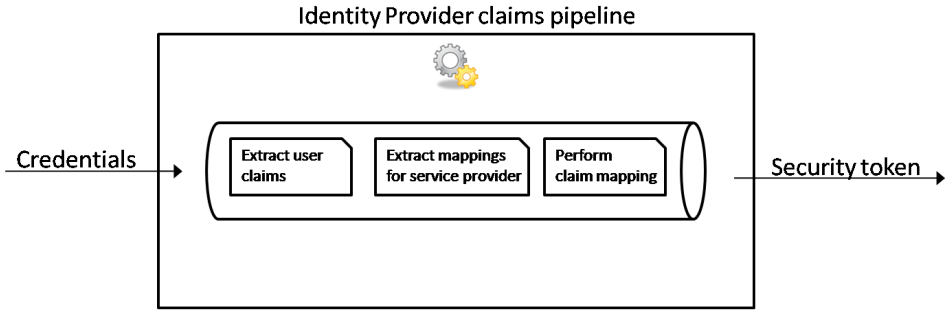


Figure 9. Claims mapping pipeline.

2.12. Customer registration

The IdP must contain a customer registration page which is the entry point for the workflow shown in figure 7. When a customer wants to buy the services provided by the IdP, the customer goes to the registration page and enters his data, such as a contact name, email, address etc. This data is saved, and a sales representative is alerted. The sales representative then contacts the user to confirm his identity and his intent to buy. A contract is then sent to the customer, and when the contact is returned the order is confirmed and the customer's IdP instance is created automatically. Thus, the IdP must support automatic creation of new IdP instances. This automatic process includes:

- Creating the administrator in system IdP instance.
- Configuring website and application pool in IIS.
- Creating SSL certificate and configuring SSL in IIS.
- Creating certificates for encryption and signing.
- Sending log in details to administrator.

2.13. Conclusion

The support for multi-tenancy is an important commercial concern for this system. However, as we have seen in this chapter, the multi-tenancy aspect can be solved by the platform alone. Tenant isolation is achieved through a per-tenant system account running each tenants instance. Having a per-tenant system account allows database isolation through either separate

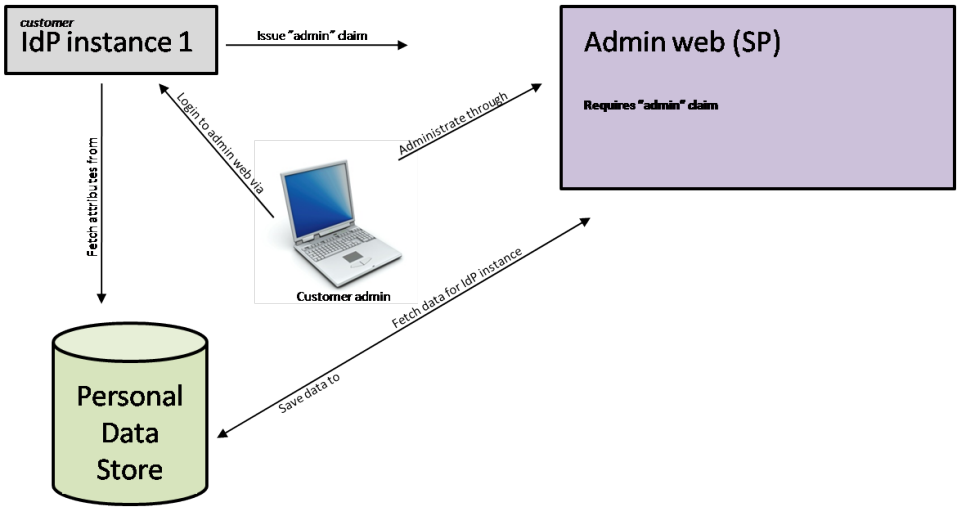


Figure 10. Claims mapping pipeline.

databases or separate database schemas, to which only that system account has access. The same goes for the certificate store, which, as mentioned, is also isolated on a per user basis. Therefore, the software design presented in this thesis will be totally multi-tenancy agnostic, without compromising the commercial demand for multi-tenancy.

3
Domain-driven design

This chapter serves as an introduction to domain-driven design. The material discussed herein is largely based on the book "Domain-driven design: tackling complexity in the heart of software" by Evans [2004]. The chapter is however not just a summary of the book, but will also contain my own perspectives and the explanatory examples provided here will be based on the problem at hand whenever possible.

Before we dive into the details of domain-driven design, let us start with an excerpt from an interview with Eric Evans, the author of the book "Domain-Driven Design" (Evans [2004]). The interview was brought in the Software Development Times (Handy [2009]) on March 12, 2009.

> ***What are the primary aspects of domain-driven design?***
> *I could boil it down into two or three basic things. The first is the ubiquitous language. On most projects, you'd have different people talking in different languages. Your technical people will discuss the system with a certain language. They will describe the actual functioning of the system in the same way. They will have words for the functional entities that are different from the words used by the business people. Some will know the language the business people use, so they act as interpreters for the technical people who don't know that language. You have a process broken into parts.*
> *The business people are talking to technical people in requirements gathering, and then it's written up and handed off in this other sort of language for implementation. That means you can never have a conversation about how the system really works. The software on the inside is actually nothing like what the business people are imagining it to be.*
> *This leads to usability problems. Translation is never a perfect thing. It also hurts estimates. The way estimates work is that the technical person says, "I have feature A and feature B, so it seems to me that feature C is a natural extension of A and B." The business person says, "No, no, no, that's a totally different thing and will take an enormous amount of time." When in fact, for the*

technical people, it's not totally different from an implementation point of view.

There's no communication there. There's no way for a non-technical person to anticipate what might be easy or what might be hard. Also, business people never propose ideas that might have been easy because it's not evident to them that they would be easy.

Does that mean that something as simple as naming your libraries and classes after the business tables they represent? *At the most basic level, it's naming things the way they would expect. There are more subtle aspects. With the application model itself, it's a system of names and relationships among things. We share this, and when we talk about it, we use that consistent language. When we build the system, we stay true to this. If the system isn't built that way, we're just talking about some fiction. I don't mean this thing where people say we take the business language and the conception, and just make software that reflects it.*

What are the other two important aspects of domain-driven design? *The second element is that you have to bring about a creative collaboration between the domain experts and the technical experts. It's very closely related and so much easier said than done. You'd have to start with an intention to do it. Some people are good at it and some are not. One of the keys to this is in recruitment. You have to hire the kind of people who are good at this, who are good at getting in a fruitful conversation with a domain expert.*

The third aspect is what I call an awareness of context. Here is one of the areas where I kind of had to invent a system because no one had really systematized this. Within any given project there are multiple models in play. I'm not talking about the kind of models I've already spoken of. In this subsystem we talk about it this way, and in [that] subsystem we talk about it that way. This we cannot eliminate. People have tried, and the results are much worse. It's one of those cases where the cure is worse than the disease.

Instead, we try to understand them and map them out. What are the boundaries that define where each one applies? If you say pot-ay-to and I say pot-ah-to, that's fine, as long as we know where pot-ay-to is used and where pot-ah-to is used. Projects that succeed have usually accomplished this.

Having a language to describe this and a terminology and a system serves to make it more reproducible.

3.1. The philosophy of domain-driven design

Most software projects, if not all, address a specific problem. Solving this problem well is important to making the software, and therefore the business, successful and profitable. The main philosophy of domain-driven design is that the primary focus of any software project should be on the domain and the domain logic, ie. the business logic. This may sound like common sense, put in practice, the book argues, many software projects are too focused on technology when designing their software. Therefore, the design models produced are cluttered with "unimportant" aspects that draw attention away from, or completely hide, the core domain logic.

Domain-driven design is not a development method per se, nor is it tied to any particular methodology. It is however oriented toward agile development as we shall see further on, and it also draws on well established software design patterns. One of the primary concepts of domain-driven design is the so-called ***domain model***. The domain model represents a large amount of knowledge contributed by everyone involved in the project and it reflects deep insight into the domain at hand. The domain model is not a particular diagram, or document, or drawing on a whiteboard, and it is definitely not just a data schema. This does not mean that the such diagrams, documents, or drawings could not exist, and they probably should, but they are not the domain model as such. The domain model is the idea that the combination of these diagrams and documents intend to convey. The domain model is an abstraction of all the knowledge about the domain, carefully organized in a way that it conveys this knowledge in the most expressive way possible, or as the book puts it, *the model is distilled knowledge*. Therefore, the model is also the backbone of the language spoken by all team members: developers, domain experts, software users, and testers. Developers and domain experts can have a different view of the problem domain. The developer's view is often not as complete as that of the domain expert and thus the model produced by the developer is not as rich and communicative as that of the domain expert. It is therefore of utmost importance that developers and domain experts collaborate intensely on the creation and maintenance of the model. The book refers to this discipline as ***knowledge crunching***.

Knowledge crunching is achieved through brainstorming and experimenting, continuous talking, diagramming and sketching, involving experiences from current systems or legacy systems and drawing on knowledge from domain experts, current users, etc. This is important because deep and expressive models seldom lie on the surface. Knowledge crunching is therefore an extensive exploration of the domain and a continuous learning experience. Distilling as much knowledge as possible in the model is important because of the natural leaking of knowledge from most software projects. Skilled and knowledgeable programmers get promoted, and teams are often rearranged, and when such things happen knowledge is essentially lost. But when a deep and expressive model exists, most of this knowledge is to a large extent preserved on the project. Using this method of modelling also has various benefits over methods such as the waterfall method. In the waterfall method, knowledge is usually passed on from domain experts to analysts, and from analysts to programmers. But the waterfall method has various shortcomings, as argued in [Evans 2004]. First of all, the method lacks feedback, and knowledge is potentially lost through the chain of information.

3.2. Ubiquitous language

A model should be based on a ***ubiquitous*** (universally present) ***language***. Usually, domain experts use the business jargon and technical team members use another, perhaps more technical jargon. But the overhead cost of translation, not to mention the risk of misunderstanding makes the use of different jargons dangerous. Thus, the principles of domain-driven design state that a conscious effort should be made to encourage a common language and that the model should be based on the ubiquitous language. This model based language should not only describe artifacts but also tasks and functionality, and as such it goes far beyond *finding the nouns*, a widespread practice when modeling. The ubiquitous language must be used in both oral and written communication within the team, and the model and language are very tightly coupled. This effectively means that any change to the language is also a change to the model and viceversa. Having a common language resolves the confusion that often occurs when domain experts and developers use different terms to describe the same thing. In [Evans 2004], Evans states that he has met many developers who initially do not like the idea of a common language because they think that domain experts will find their terms too abstract or will not understand objects, but he argues:

> *If sophisticated domain experts don't understand the model, there is probably something wrong with the model.*

I really agree on this point. Especially because the focus of the model should be on the business domain and not on technical details. So if a domain expert does not recognize his own business in what is supposed to be a model of that same business, then the model is by definition not right.

3.3. Relationship between model and code

If the ubiquitous language is one representation of the domain model, then the code is definitely another representation of that same model. It is arguable that the code is the most correct representation of the model, but one thing is for sure; the code is always the most detailed representation of the model. The vital details of the model lie in the code. Diagrams and documents can be used to clarify design intent or decisions, but they should represent skeletons of ideas and not be overly detailed. Being detailed is something that the code does particularly well. Coupling the model and the implementation from an early start through prototyping is very important. Doing this will uncover any infeasible facets of the model early on and it is also an important part of getting the developers involved in the design process. By taking part in the modelling process with domain experts, developers will improve their modelling skills and domain specific knowledge, but most importantly the will feel responsible for the model. And that is very important, because developers must realize that by changing the code they also change the model. If they do not realize this, any future refactoring of the code will weaken the model instead of strengthening it. The code should be written to reflect the domain model in a very literal way and it should use the same terminology as in the ubiquitous language. It is obvious that every application has a lot of plumbing concerned with graphical user interfaces, database and network access and other things that are not related in any way to the domain. Thus, in order to achieve the tight coupling between model and code, and to avoid cluttering the code, a layered architecture as that shown in Figure 11 should be used.

Figure 11 shows the layers that make up the layered architecture. Domain-driven design focuses exclusively on the *domain* layer, and this is the layer which should contain all domain logic. It is important to notice that the components of each layer only interact with other components in the same

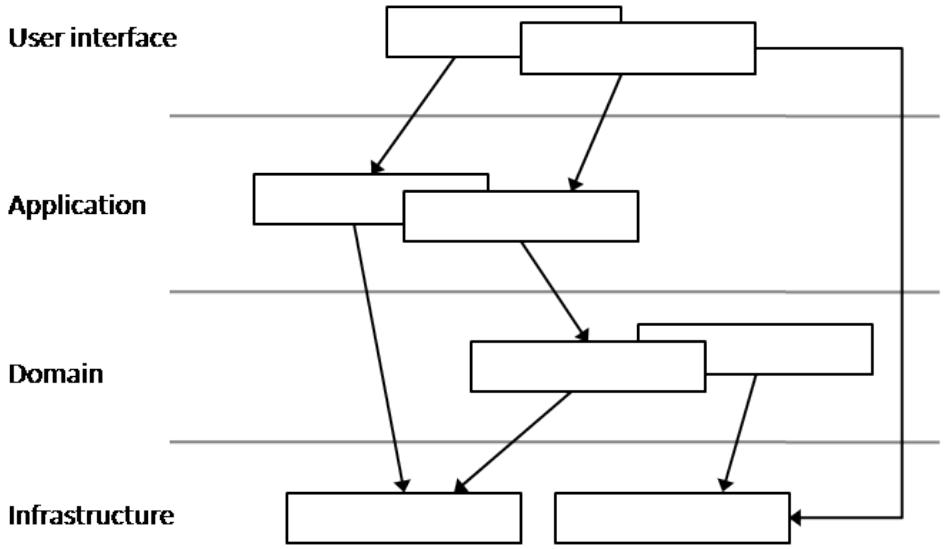


Figure 11. A layered architecture.

layer or in layers below them. The *user interface* (UI) layer contains code for interaction with the user, the *application* layer contains application specific code, such as managing long running transactions or workflows, and finally the infrastructure layer contains code related to data access, network access, logging, etc. When domain related code is scattered throughout the UI, infrastructure layer, etc, it becomes very hard to see the domain logic and reason about it. To change a business rule may require code changes in many different parts of the code and this can be very error prone. So the value of using a layered architecture is that each layer can be specialized to manage different aspects of the computer program.

Since domain-driven design deals heavily with assigning responsibilities to the right places in code, it renders itself well to object-oriented programming languages.

3.4. Building blocks of the model

Domain-driven design specifies a set of conceptual objects that should be used in code to create the domain model. Evans refers to the act of designing the code components that make up the model as ***model-driven design***.

Figure 12 shows an overview of these concepts, which will be discussed in detail next.

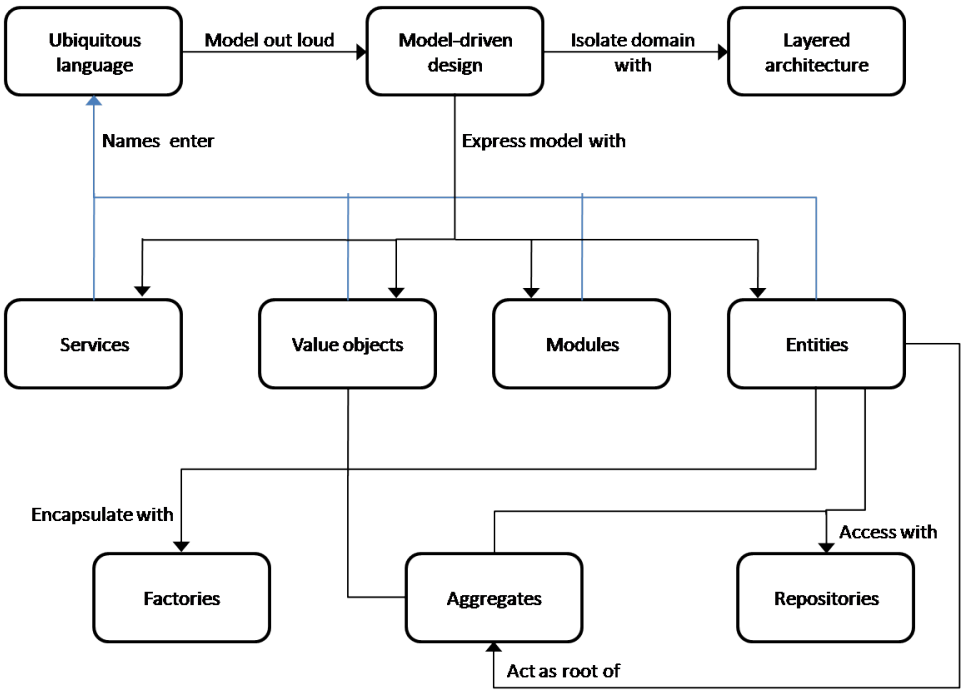


Figure 12. The building blocks of domain-driven design.

3.4.1 Entities

Entities are objects that are defined by their identity rather than by the values of their properties. The "lifespan" of entity objects is usually long, and the values of its properties can change often over time whereas the identity does not change. An example of an entity object could be a **`User`** object. Consider a **`User`** object with properties **`UserId`**, **`UserName`**, **`CreatedDate`**, and **`UserAttributes`** as shown below.

```
1  public class User{
2
3      private Guid _userId;
4      private string _lastName;
5      private DateTime _createdDate;
6      private IEnumerable<Attribute> _userAttributes;
7
8      public User(Guid userId,
```

```
                string lastName,
                DateTime createdDate){
        this._userId = userId;
        this._lastName = lastName;
        this._createdDate = createdDate;
    }

    public Guid UserId{
        get{return userId;}
    }

    public string LastName{
        get{return _lastName;}
        set{_lastName = value;}
    }

    public DateTime CreatedDate{
        get{return _createdDate;}
    }

    public IEnumerable<Attribute> UserAttributes{
        get{return _userAttributes;}
        set{_userAttributes = value;}
    }

    public overrides bool Equals(User other){
        return this.UserId == other.UserId;
    }
}
```

Even though two **User** objects have the same **LastName**, they are only identical if the **UserId** is the same. Two objects could even represent the same user regardless that **LastName** are not identical as long as the **UserId** is the same, for example, if the user changed his or her last name. Thus, when implementing entity objects, it is important to implement equality functions in such a way that the equality of two objects is based on comparing identity rather than comparing the individual properties of the object, since these can, and most likely will, change over time. The **Equals** method above is an example of such an implementation. The identity field of an entity object is often automatically generated such as it would be the case in the example of the **User** object. The identity field (**UserId**) is not always important to the user of the system, but it could be, for example if the id was a social security number, or package tracking number.

3.4.2 Value objects

Value objects, contrarily to entity objects, are objects that do not have a unique id. Value objects are often used to describe entities, and we care about them only for *what* they are, not *who* they are. A good example of a value object could be a **Money** object as the one below. After all, ten dollars are ten dollars no matter what. We usually do not care about the identity of the ten dollars (unless of course we are looking for stolen money) but only the value they represent.

```
public class Money{

    private int amount;
    private Currency currency;

    public Money(int amount, Currency currency){
        this.amount = amount;
        this.currency = currency;
    }

    public int Amount{
        get{return amount;}
    }

    public Currency Currency{
        get{return currency;}
    }

    public Money Add(Money other){
        // Make sure currency is the same
        // Add amounts
        // return new money object
        ...
    }
}
```

Note that the value objects, such as the **Money** class, are by definition immutable, meaning that after creation the properties of the object cannot be changed. When adding **Money** objects, new objects are created instead of changing any of the objects being added. This maintains the immutability of the objects.

3.4.3 Services

Services are classes that implement domain logic that does not naturally belong to an entity or a value object. A service's interface should be defined in

terms of other elements of the domain model and its operation names should be part of the ubiquitous language. Classes that implement services should be stateless. In practice, stateless classes are often comprised of static methods that do not require an instance of the class in order to be invoked. An example of a service in our problem domain could be a `CertificateService` with operations such as `CreateCertificate` and `RevokeCertificate`. Since the concept of a service is used widely in computer science in general I feel it is important to clarify that a service in terms of domain-driven design should not be confused with a service such as in web-service. This does not mean that a system built using domain-driven design should not expose web-services, however, when using domain-driven design and a layered architecture, a web-service would be part of the application layer and not part of the domain layer. In terms of the `CertificateService` mentioned above, a web-service could be created in the application and the only responsibility of that web-service would be to digest the XML input and delegate the call to the `CertificateService` in the domain layer.

3.4.4 Aggregates

Aggregates are clusters of associated objects which are worked on as one unit when it comes to data changes.

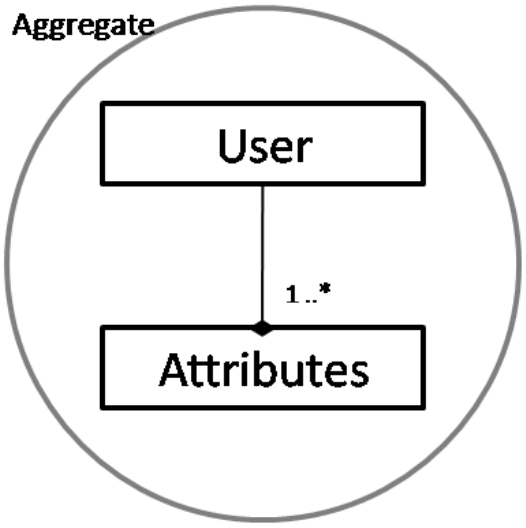


Figure 13. An aggregate.

Figure 13 shows an aggregate made up of a **User** object with various **Attribute** objects. **Attribute** objects are only interesting to modify in the context of a **User** and do not make sense outside the context of a **User**. In this example, the **User** object is said to be the *aggregate root*, meaning that it is the primary entry point when accessing the objects that make up the aggregate. Aggregates can only have one aggregate root, but the aggregate itself can be much more complex than the one shown here.

3.4.5 Repositories

The purpose of a ***repository*** is to fetch and save entities and value objects from a data store. The repository should hide all database code and give the illusion that all objects are in RAM. The following is the skeleton of a repository:

```
public class UserRepository : IUserRepository {

    public User this[Guid userId] {get{ ... }}

    public IEnumerable<User> FindByLastName(string lastName) ...

    public IEnumerable<User> FindRecentlyCreated() ...

    public IEnumerable<User> FindAllUsers() ...

    public void AddUser(User newUser) ...

        ...
}
```

The above repository lets us fetch a **User** object by a unique id, finding all users with some last name, and it also allows us to add new users. All these methods actually implement some kind of domain logic with regards to users. The **FindRecentlyCreated** method could be using a SQL statement such as **select firstname, lastname from users where createdDate > :createdDateThreshold** (where **:createdDateThreshold** would be a bound parameter). But this SQL statement expresses domain logic so if we did not have a central place to put this function, that SQL statement could potentially be duplicated many places in the code, making it harder to change the logic of finding recently created users, if for some reason the definition of "recently created" would change. Another benefit of using repositories is that they can be made interchangeable. By implementing the **IUserRepository** and referencing the concrete instance only by means of its interface, separate specific implementa-

tions of the repository can be seamlessly interchanged. This is especially useful for testing. A live system typically fetches its data from large databases, and replicating these databases to a development server can be both time consuming and sometimes even impossible. By specifying the concrete class that implements the repository in some external configuration file can make it easy to switch between the live environment and a developer or testing environment[1]. In our current example we could imagine an alternative implementation of the `IUserRepository`, called `InMemoryUserRepository`, that could be used for fetching dummy instances of the `User` class, thus facilitating a means of testing various aspects of the code without having to access the production database. However, when using this approach, it is important that developers understand the implementation of the repository being used. For instance, if someone was to write a method for counting all users in the system the function `FindAllUsers` could be used and the `Count` method of the `IEnumerable<User>` could be used to return the number of users. This would probably perform quite well with a dummy repository implementation returning only a limited number of users. However, when using a "real" repository implementation that accesses the production database the effect of performing the count this way would be that all users in the database would be loaded into memory just for the sake of counting them. That would most likely perform badly, thus stressing why it is important that developers understand the implementation of repository functions.

In Chapter 5 I will discuss how OR-mappers can be leveraged to create repositories quite easily even for complex database schemas.

3.4.6 Specifications

Specifications provide a means of encapsulating and more importantly naming those small Boolean expressions that tend to appear in most programs but whose purpose and or meaning with regards to business logic can be hard to figure out. A specification is a class with only one method called |IsSatisfiedBy|. The method has a Boolean return type and the argument of the method depends on the type of object it is a specification for. If we continue our `User` class example from above, we could imagine the need to determine if a given instance of the `User` class was created recently. The term recently is chosen on purpose because it is unspecific and definitely contains

[1] This concept is actually a well established design pattern, often referred to as ***strategy*** or ***policy***. See [Gamma et al. 1995]

business logic with regards to our current example. Let us assume that the definition for a recently created is that it is was created within the last 10 days.

```
User u = ...
if(u.CreatedDate > DateTime.Now.AddDays(-10)){
        ...
}
```

The above code snippet shows one way of determining if a **User** was created recently. However it suffers from two serious drawbacks. First of all it is not very expressive, meaning that without code comments it does not convey to the reader which business rule it is checking. Furthermore, if the same check was needed somewhere else, the code would have to be duplicated, leading to extra maintenance if the criterion ever changed.

```
//Specification class that determines if a user has
//been created recently

public class RecentlyCreatedSpecification{
        public static bool IsSatisfiedBy(User u){
                return u.CreatedDate > DateTime.Now.AddDays(-10);
        }
}
```

The above code snippet is a class that implements the recently created business logic. Now if we were to use this specification in code it could look something like the following:

```
User u = ...
if(RecentlyCreatedSpecification.IsSatisfiedBy(u)){
        ...
}
```

The above code example shows how the resulting code is much more expressive in terms of what business rule is checked, and furthermore we have created a single placeholder for the business logic, making it easy to maintain. It could be argued that what is achieved by the specification in this example could just as well have been placed in a function on the **User** class. This observation is probably right for this very simple example. However, a class with a lot of predicate functions like **IsRecentlyCreated** easily becomes very large. Furthermore, one could easily imagine cases where the object itself does not contain all the necessary information to check the condition.

3.4.7 Factories

The ***factory*** design pattern is widely used in object-oriented design. The analogy of the factory object is that sometimes the construction of an object is too complex for the object itself to perform, and it needs to be created in a factory. One important benefit of using factories as opposed to constructors is that the methods of the factories can have abstract return types or return some implementation of an interface. This way, the user of the factory method does not need to reference the concrete type of the object that is returned, which makes changing or substituting these objects completely transparent to the caller. Factories should of course not be used for everything. Simple objects that do not implement a common interface or use polymorphism should probably not use factories for their construction. I have seen examples of open source software packages (such as OpenSAML [OpenSAML 2009]) where even the simplest of objects had to be created through a factory. The overuse of factories can actually obscure simple objects and make the user think that the object is more complex than it actually is. The most important property of factory methods is that they should be atomic. This means that everything needed to construct the object should be passed to the factory such that the construction can take place in one single interaction with the factory. Furthermore, it is important that the factory ensures that all invariants for the object being created are met. This does not mean that the logic that checks these invariants should be moved outside the object being created, but it does mean that the factory should make sure to invoke this logic before returning the instance of the object in question. Factories used for construction of entity objects should behave differently when creating new instances of an object as opposed to creating a reconstituted instance of the object. Most importantly factories should not automatically assign new ids to entity objects when they are reconstituted, because if they did, the continuity of that particular object would be broken.

3.4.8 Modules

The concept of ***modules*** is used to group related concepts from the model, e.g. all classes containing `User` related logic, in conceptual packages. A module could be either a dll[2] or merely a namespace within a dll. As with everything else in the domain model, module names should make sense in

[2] Or .jar, .lib or whatever equivalent is offered by the technology used.

terms of the ubiquitous language. Determining whether to use individual dll's or just namespaces within that dll should probably be based on the number of classes involved although no details are offered in [Evans 2004].

3.5. Refactoring

When we set out to write software we never know everything about the domain. It is therefore important to make the design open to refactoring and change such that new knowledge can easily be incorporated into the design when it is discovered. A system that lacks a good design does not encourage developers to leverage existing code when refactoring, thus leading to duplicate code. Likewise, monolithic design elements also impede code reuse, just as design elements with confusing or misleading names may lead them to used inconsistently by different developers. In fact, the lack of a good design may completely stop refactoring and iterative refinement since developers will dread to even look at the existing code and they will be afraid to make changes, since a change to the existing mess may break some unforeseen dependency or just aggravate the mess, Evans argues. In domain-driven design a design that renders itself well to refactoring and iterative refinement is called a ***supple design***. There are no exact formulas to achieving a supple design, but Evans offers a set of patterns that could lead to it. These patterns are shown in Figure 14, and will be discussed in detail in this section.

3.5.1 Intention-revealing interfaces

The word interface in ***intention-revealing interfaces*** should not be confused with the keyword `interface` as known from many programming language. In this context, the word interface refers to the naming of all public elements of a software artifact. Therefore, classes, operations, and argument names should be named in a way that reveals the effect and purpose. The names should however not contain any information on how the effect is achieved. An example of an intention revealing method could be the `AddUser(User newUser)` function from the `UserRepository` discussed in Section 3.4.5 on page 31. The method name reveals that the method adds a user, and that the argument should be a new user (as opposed to an already existing one). It should be obvious that the naming of this method and parameter is more intention revealing than for example `Create(User u)`. By being explicit about

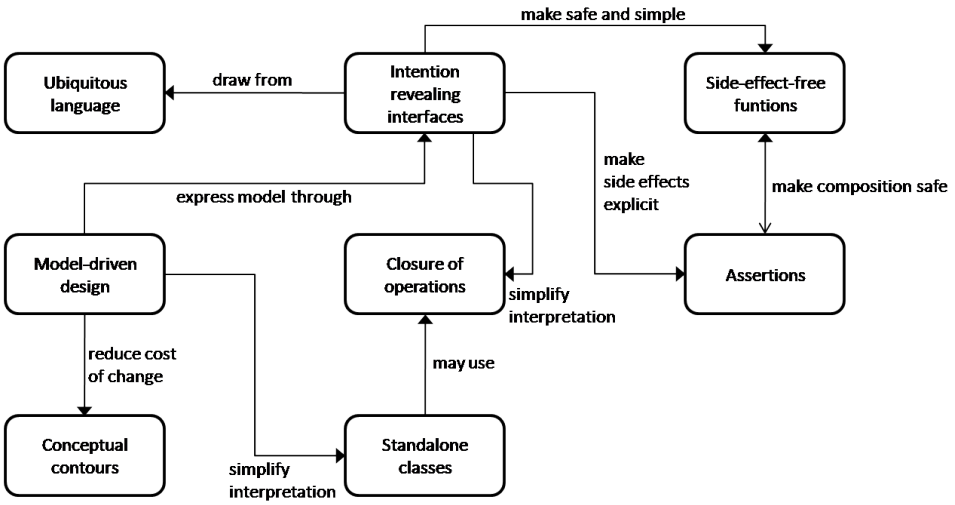


Figure 14. Patterns that contribute to a supple design.

what the method does, and what its parameters are, the risk of some developer unintentionally using the method for something else than its intent is minimized. In [Evans 2004], Evans gives the following example of why intention-revealing interfaces are important:

> *If a developer must consider the implementation of a component in order to use it, the value of encapsulation is lost. If someone other than the original developer must infer the purpose of an object or operation based on its implementation, that new developer may infer a purpose that the operation or class fulfills only by chance. If that was not the intent, the code may work for the moment, but the conceptual basis of the design will have been corrupted, and the two developers will be working at cross-purpose.*

3.5.2 Side-effect-free functions

A side effect normally means some unintended consequence, but in the context of computer science in general and domain-driven design specifically, a side effect means a change in the state of a system. Evans argues that generally there are two different types of functions in a system. Functions that query system data and functions that alter system state. Evidently,

changes to system state neither can nor should be avoided per se. However, according to the ***side-effect-free functions*** pattern, these two types of behaviors should not be mixed. This means that functions that return data should not alter the system state in any observable way. In the same way, functions that alter system state should not return data.

3.5.3 Assertions

As we have just discussed in the preceding section, side effects cannot be avoided. The ***assertions*** pattern in conjunction with intention revealing interfaces makes it even more explicit what the side effect of a function is. This is achieved through a set of pre- and post-conditions that should always be satisfied before and after invoking a function. It does not always make sense to code the pre and post-conditions as part of the program because of performance overhead or missing support in the programming language, so if that is the case, the assertions should be included in automated unit tests and in the documentation of the program.

3.5.4 Conceptual contours

The ***conceptual contours*** pattern strives to achieve a meaningful granularity of functions, classes and interfaces in the model. Evans argues that no single granularity will fit everything in our domain model, so instead of using a naive approach where every function or class is limited to a fixed number of lines of code or the like, the granularity should be based on what the function or object conceptually achieves. Thus, according to the conceptual contours pattern, each object should be a whole concept, nothing more and nothing less. In the same way, each function should perform something meaningful in its own right, but it should not span several conceptual operations. As an example of the latter, Evans states that the `add()` function should not be split up into two separate functions. In the same way the `add()` and `subtract()` functions should not be combined into one. There is of course no simple recipe for achieving conceptual contours, and therefore the decomposition of design elements into cohesive units is something that must be based on intuition and which must be expected to undergo many changes over time until a good granularity has been achieved.

3.5.5 Standalone classes

The pattern called ***standalone classes*** revolves around reducing coupling between objects. The goal of this pattern is to achieve low coupling where ever possible because it makes the model easier to understand. Evans argues that every dependency that a class has makes the class more complex and the relationship between the class and its dependencies have to be understood in order to fully understand the class. Therefore, classes with low coupling are easier to understand. It is obvious that taking the standalone classes pattern to the extreme leaves us with a (useless) model where everything is reduced to a primitive. However, it is important to reduce the number of interdependencies between classes, especially those that are not essential to the concept.

3.5.6 Closure of operations

The name ***closure of operations*** comes from mathematics. In mathematics, the addition operation is closed under the set of integers, e.g. $1+2=3$. The addition of two integers yields another integer. This property can be used when designing a good model, since it does not introduce new concepts, and it is easier to understand. In general, the closure of operations pattern is mostly used for value types, and not so often for entities. Because of the life cycle of an entity it is not natural that a function on an entity would return another instance of that entity. However, value types can often have functions that offer closure of operations. An example of a value type that has a function with closure of operations is the `Money` class presented in Section 3.4.2 on page 29. Its function `add` is closed under instances of the `Money` class.

3.6. Model integrity

Building large systems often involves several teams of developers who develop each their part of the system. It is of paramount importance that these teams have a shared understanding of the model such that the meaning of each concept in the model is the same for all teams. If this is not the case, classes that represent one concept to one team could potentially represent a different concept to another team, thus leading to misuse of that class

and in the end maybe faulty behavior in the system. Within the scope of this thesis, the system being developed is developed only by this author, nevertheless, the patterns of domain-driven design that address model integrity are still interesting to describe, not least because the future maintenance of the system being designed may have to be undertaken by more than one person or team, but also because some of these patterns are useful for modelling in general. The patterns used for maintaining model integrity, as outlined by domain-driven design are shown in Figure 15.

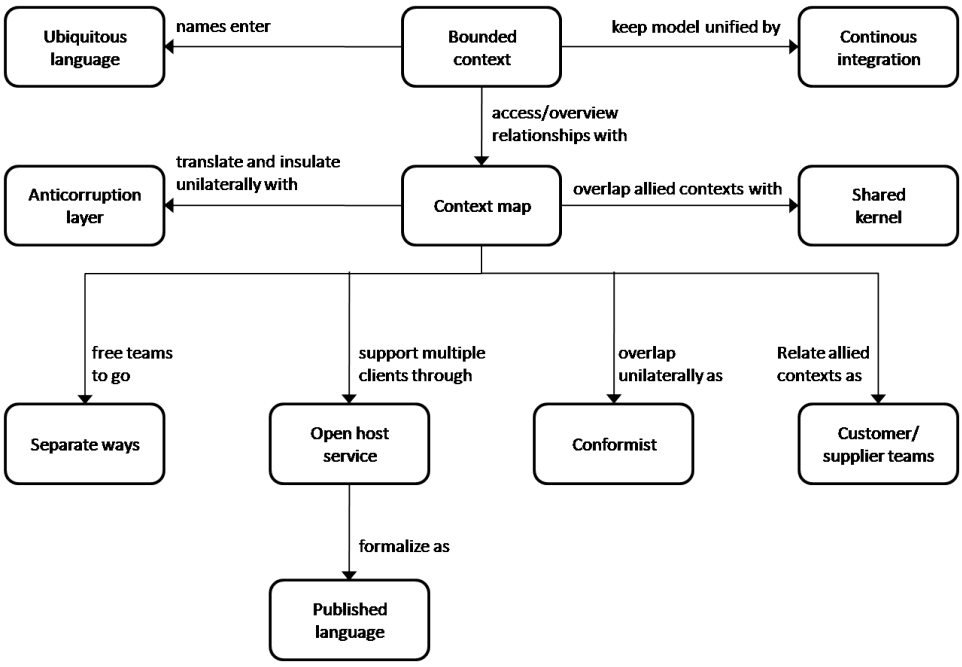


Figure 15. Model integrity patterns.

3.6.1 Bounded context

On large projects with several teams working on the same system it can become unclear whether or not the teams are working on the same model. The teams may have an intention to work on the same model and share code. But sometimes systems are too big for a single unified model to exist, especially when communication between teams is not as good as it could be. Having one large unified model requires good communication and processes to detect conflicting interpretations of the model. In order to overcome this

problem, Evans argues that it may sometimes be beneficial to split the code and model into several ***bounded contexts***. A bounded context is an explicit definition of a context in which a given model applies. Once a number of bounded contexts have been defined, the boundaries of these contexts can be explicitly defined in terms of usage within the application, code base etc., thus being able to achieve a pure model within the bounded context. Using bounded contexts makes it explicit to developers that they are working on different models, and hereby eliminates the risk that different teams will think that they are working on a unified model when in actual fact they are working on conceptually divergent models.

3.6.2 Continuous integration

Continuous integration is a well-known practice that is aimed at speeding up delivery times and decreasing integration times in software development. Domain-driven design takes the concept of continuous integration a step further, namely by not only focusing on the continuous integration of code, but also on the continuous integration of model changes. As opposed to the continuous integration of code, the continuous integration of a model is not something for which a set of automated processes exist. Therefore, continuous integration of the model is something that must be achieved by continually discussing the model and relentlessly exercising the ubiquitous language in order to strengthen the shared view of the model and to avoid that concepts evolve differently in developers' heads. Continuous integration of code is of course something that must take place in parallel with continuous integration of the model. This should happen through automated builds and automated tests. A good set of automated tests should make it more comfortable for developers to refactor existing code, because by running the automated tests, they will know instantly whether or not something is broken by the change they made.

3.6.3 Context map

When working with a number of bounded contexts, a ***context map*** serves the purpose of creating a global view of the entire system and defining the relationship between the different bounded contexts. The relationship between bounded contexts often involves some kind of mechanism to translate data between bounded contexts. The context map can be a diagram or a

text document or both, and it is important that everyone involved in the development of the system know and understand the context map, and that the names of every bound context described in the context map enter the ubiquitous language.

3.6.4 Shared kernel

A ***shared kernel*** is essentially a part of the model that is used in more than one bounded context. The obvious benefit of having a shared kernel is that code reuse between teams can be maximized. The shared kernel often represents the core domain of the system and/or a set of generic subdomains, but it can be any part of the model that is needed by all or some teams. Making changes to the shared kernel requires consultation with all its users such that the model integrity of the shared kernel is not broken.

3.6.5 Customer/supplier development teams

The ***customer/supplier development teams*** pattern suggests that when there exist a relationship between two teams where one team delivers code that the other team is dependent on, then each team should work within their own bounded context. Doing so makes it easier to define responsibilities and deliverables and the joint development of acceptance tests will validate whether these have been met.

3.6.6 Conformist

Sometimes one of the bounded contexts in a system is an off-shelf component with a large interface. When this is the case, Evans argues that a pattern called the ***conformist*** pattern should be used. The conformist patterns says that to conform to the model represented by the bounded context represented by the off-shelf component. The rationale behind this is that if there is a real need for an external component, then that component probably represents valuable knowledge and probably has a well thought through model[3]. Therefore, conforming to the model of the component is usually a good idea

[3] Evans also argues that if this is not the case the use of that component should be seriously questioned.

because less translation between concepts is necessary, and because the possibility of being "dragged into" a better design exists. The conformist pattern is most important when the interface with the component is big.

3.6.7 Anticorruption layer

Interfacing with legacy systems is often a necessity for one reason or another. The legacy system may have weak model or a model that does not fit the current project. When this is case, the ***anticorruption layer*** pattern may be used. An anticorruption layer is a technique where legacy systems are isolated through classes and functions that honor the current model and hides any conversion and translation logic from the current model to the model of the legacy system. An anticorruption layer is often made up of a number services that have responsibilities in terms of the current model and internally use the ***facade*** and ***adapter*** patterns as described in [Gamma et al. 1995]. The facade is an interface that simplifies access for the client, thus making the subsystem easier to use, and an adapter is a wrapper that is used to transform messages from one protocol to another.

3.6.8 Separate ways

Sometimes the benefit of integrating bounded contexts is small, and therefore may not be worthwhile. The ***separate ways*** patterns can be used when that is the case, thus allowing developers to find simple and specialized solutions within a small scope.

3.6.9 Open host service

Sometimes a subsystem has to be used by many bounded contexts, and making customized translators between all the subsystems and all the bounded contexts can be time consuming and hard to maintain over time. So instead of doing so, the ***open host service*** pattern can be used. The open host service pattern is about defining a protocol that gives access to the subsystem, and that can be used uniformly by all bounded contexts using the subsystem.

3.6.10 Published language

Finally, the ***published language*** takes the open host service pattern to the next level by defining a formal common language for the service. An example of a published language in this context could be SQL or XML.

4
Defining the model

In this chapter I will define a model of the software that was specified in Chapter (2). In domain-driven design the model is comprised of both the code and all the documents that describe what the system does and which parts of the system are responsible for doing what. However, domain-driven design also emphasizes that the model must be understood by (typically) non-technical domain experts. Therefore, the aim of this chapter is to present the model from a point of view that is not overly implementation specific. I will use terms such as classes, interfaces and functions, but the implementation details will be left for the following chapters (5, 6 and 7). Apart from presenting the model, the aim of the following sections is also to illustrate the iterative process involved in domain-driven design. Therefore some sections may include a refactoring sub-section. You may argue that instead of having a refactoring sub-section I could have just performed the refactoring right away. This does, however, not illustrate the iterative process that is one of the cornerstones of domain-driven designas well, and therefore I have chosen the sub-section approach.

In the following I will refer to the software as a whole as "the IdP", and the person who administrates the IdP "the administrator" (or "an administrator" since there may be several). It is important to notice that in this context the administrator is a customer of the software, administrating a single IdP instance.

4.1. Figure notation

Any model is tightly coupled to the data that it represents, and for most applications and this one in particular, that data is stored in a database. Therefore, it seems natural to present the data schema together with the model. Since I have great focus on the ubiquitous language I will name my database tables and the classes that encapsulate each row in a given table (ie. the entity objects) the same.

Figure 16 shows an imaginary model concept called `SomeConcept` which

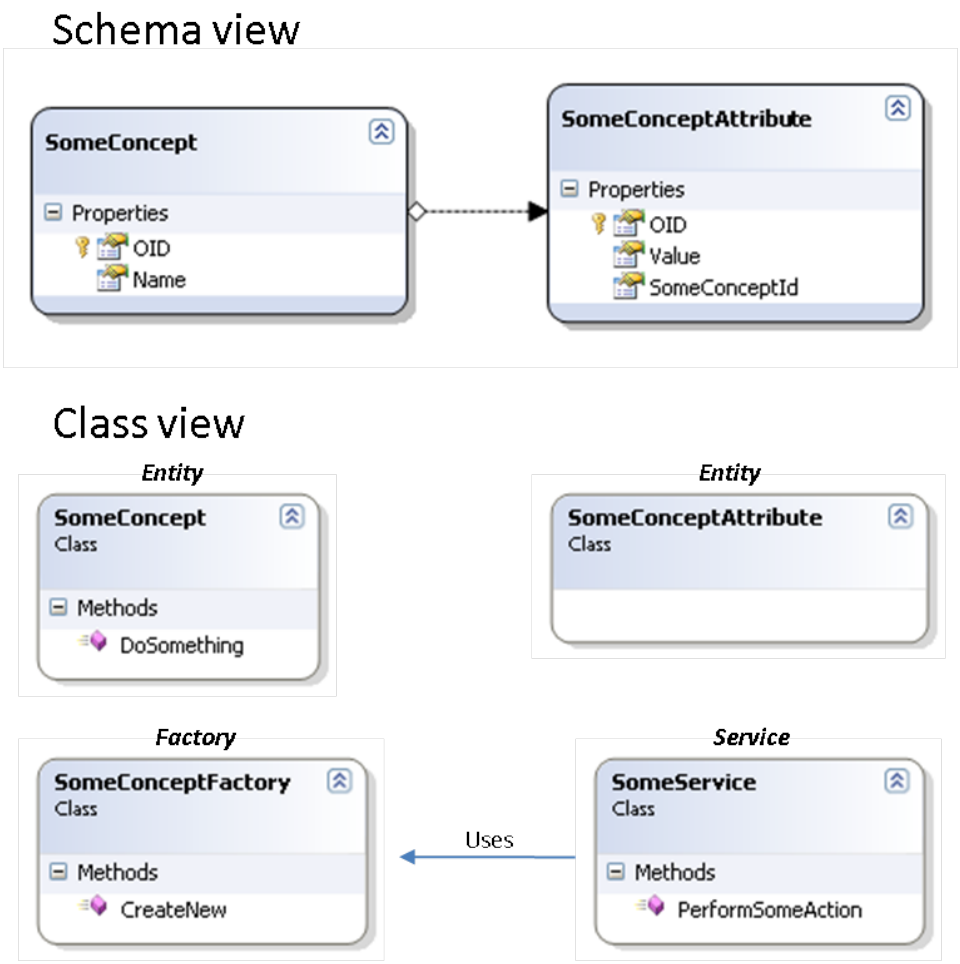


Figure 16. Figure explanation

has attributes called **`SomeConceptAttribute`**. The schema view shows shows all the data properties of each of these two concepts. It also shows the primary key, which is denoted by a small key symbol next to the property. Schema view also shows relationships between the classes. For example, the arrow in the figure denotes that a **`SomeConceptAttribute`** is coupled to a **`SomeConcept`**. In fact, the **`SomeConceptId`** is a reference to the **`OID`** of some instance of **`SomeConcept`**. I use a strict naming convention for these references, such that is should always be easy to infer which property references the **`OID`** of the other class. I use **`OID`** (short for object id) as a primary key for all entity objects. Apart from being naming convention that is easy to remember and easy to recognize, using this naming convention is useful in the imple-

mentation as we shall see in Chapter 5. Please note that I may chose not to show all relationships of a given table in schema view. For example, when introducing a new concept I will not show relationships to things that have not yet been introduced. Likewise, when introducing new concepts that expand something that has already been shown, I will not show relationships that have already been explained. I will generally explain the type and meaning of all the properties unless they are self-explanatory are identical to previously explained properties. The class view on the other hand shows additional functions, if any, of the classes. I will explain all methods and their parameters where I deem it necessary for understanding its purpose. Class view also shows what type of class each class is in terms of domain-driven design. This is denoted in italics above each class. In class view, you may assume that all the properties that were there in schema view are still there, but they are not shown in order to keep the figure simpler. There may sometimes be properties on the classes in class view, but if there is, these properties will denote some computed property that is not a part of the data schema. Also note that some classes are not part of the data schema at all, and therefore they may have non-computed properties. You may also assume that the relationships denoted by arrows in schema view still exist in class view. As a matter of fact, for Figure 16 you may assume that each **`SomeConcept`** class has a (possibly empty) list of **`SomeConceptAttribute`** instances (technically these are references to instances), and that each **`SomeConceptAttribute`** instance has an instance (reference to an instance) of a **`SomeConcept`** class. Finally, there may be blue arrows between the classes in class view denoting some kind of relationship between the classes. This nature of the relationship is explained with some text above or under the arrow, as exemplified in the figure.

Figure 17 shows the convention for interfaces, and abstract classes in class view. For example, **`ISomeInterface`** represents an interface (note the green color), while, **`SomeClass`** is a class that implements that interface, denoted by the circle and interface name above the class. An abstract class, such as **`SomeAbstractClass`** has dotted line around it, and the class, **`SomeOtherClass`** that inherits **`SomeAbstractClass`** has an arrow to class and a small arrow with the name of the base class next to it.

I have now introduced the figure notation that is going to be used throughout this chapter, and it is time to take a look at the core concepts of the model, which are:

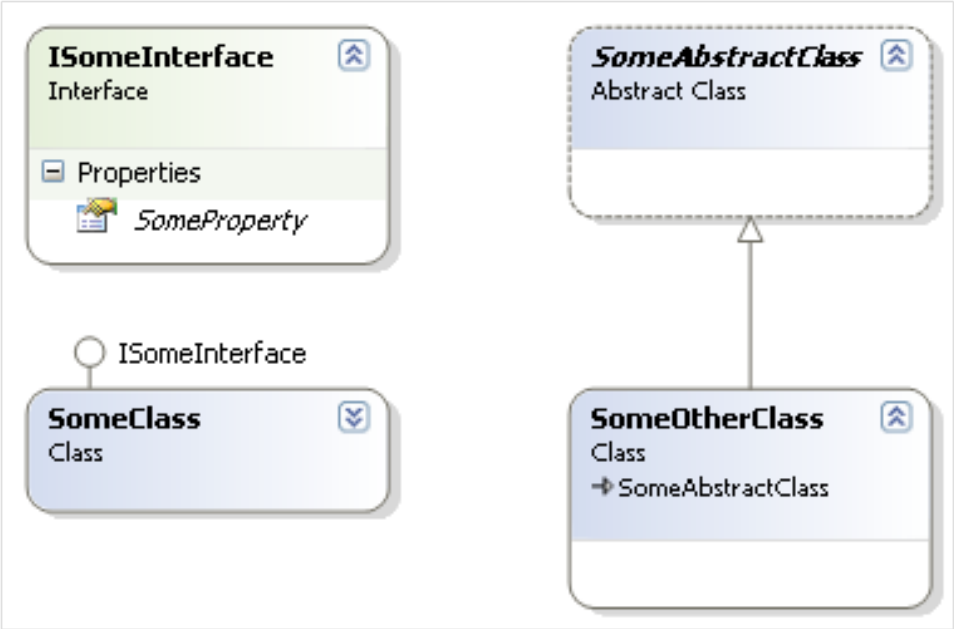


Figure 17. Figure explanation

- Plug-ins
- Users
- Configuration
- Certificates
- Credential providers
- Protocols
- Connections
- Claims
- The runtime
- Events

Please note that I have chosen to introduce reporting of events as one of the last things, since as a concept it is not as central as the others. Events in this context is logging of errors and other activities in the system. I will refer to logging and tracing before actually introducing the part of the model that describes them, which you can find in Section 4.12.

4.2. An extensible system

One of the most important properties of the system is that it must be extensible in two areas, namely in the way a user presents his credentials and in the protocols supported by the system. This will be dealt with through the concept of plug-ins. Current development visions for the IdP do not include any third-party plug-ins, however, letting the system be extensible in this way would definitely allow for this in the future. Another important benefit of using a plug-in structure is that extensions to the system can easily be deployed to existing installations, without the need of recompilation.

4.2.1 Plug-ins

A plug-in is an extension to the IdP. The core IdP framework itself does not implement any plug-ins, it only defines how such a plug-in should behave. A plug-in is created by coding a class that implements the **`IPlugin`** interface, and thus the IdP will not have to care about the how the plug-in is implemented, but only that it satisfies the contract defined by the interface. This technique is commonly used, and endorsed by [Gamma et al. 1995] with the motto "Program to an interface, not an implementation". Figure 18 shows two important interfaces used for creating plug-ins.

Figure 18. Class view: IPlugin and IEndpoint interfaces

Both the **`IPlugin`** and **`IEndpoint`** interfaces specify that properties called **`Name`** and **`Description`**. These properties are just descriptive strings. The

ProvidedLoggingSources property of the **IPlugin** interface returns a list of logging source names that the given plug-in uses when (or if) it creates log entries. During installation of a plug-in, the system must therefore create the logging sources by using the **LoggingSourceRepository**. However, the logging source repository definition that we saw earlier did not have an **Add** function, so that needs to be added. The **GetEndpoints** function defined by the **IPlugin** interface must return a list of **IEndpoint** implementations. Conceptually, an endpoint defines a (web-site absolute) path and some class that knows how to handle any requests for that path. The **Handler** property should return an instance of a class that can handle web requests. Every modern web development framework has extensibility points for such classes, and in the .Net framework it would simply be an implementation of the **IHttpHandler** interface.

The plug-in concept of our model is actually a general abstraction that will be specialized by the more specific credential plug-in and protocol plug-in as we shall see later on.

Each plug-in implementation will be a bounded context in domain-driven design terms. The bounded context comprised by each plug-in will have a conformist relationship to the IdP since the plug-in must adhere to the model defined in the IdP.

4.3. Users

One of the core concepts of the IdP is the ***user***. A user represents some person which is known by the IdP, and that, by definition, it knows something about.

Figure 19 shows the properties of a **User**. Apart from a unique id (the **OID**), the user contains a **UserName**, a **DateCreated** and a **LastLogin** property. These a quite self-explanatory. Lastly, a **User** has a **Password** and a **PasswordSalt** property. The **Password** is a computed hash of the user's plain text password, and the **PasswordSalt** is some random value used by the hashing function as entropy. Given the user's plain text password, the hashing function will only return the same computed hash as the one stored in the **User** object if the same salt value is used. So to verify a user's password, all that needs to be done is to compute the hash of the password entered by the user, using the original salt value, and comparing the result with the value

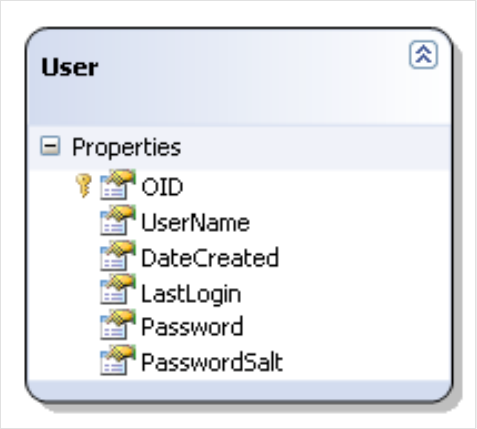


Figure 19. Schema view: User

stored in the **`User`** object. Using this approach disallows anyone with access to the database to see any of the users' passwords. Note that the **`User`** class shown in Figure 20 has a **`VerifyPassword`** function, that does just that.

4.3.1 User creation

The administrator must be able to manually create users. To achieve this, the classes shown in Figure 20 are needed. The **`UserFactory`** class has a **`CreateNew`** method that takes two arguments, namely a `userName` and **`clearTextPassword`**. When called, the **`CreateNew`** method computes the hashed password and password salt, sets the creation date and other relevant fields and returns a **`User`** instance. The **`User`** instance can then be persisted by passing it as a parameter to the **`Add`** function of the **`UserRepository`** class. Note that the **`UserRepository`** also has a **`ExistsByName`** function. This function takes a username and returns true if a user with the given name already exists, and false other otherwise. The IdP does not allow for more than user to have the same username, and it is therefore important to use the **`ExistsByName`** function to check that no user with that name already exists. The **`ExistsByName`** function in turn uses a specification (the **`ExactUserName`** specification) to compare the given username with that of existing users. Now, you may think that this is silly. Comparing usernames is just a matter string comparison right? The answer is no. The IdP has a rule that says that usernames are case insensitive, and the **`ExactUserName`** specification implements logic that compares usernames case insensitively.

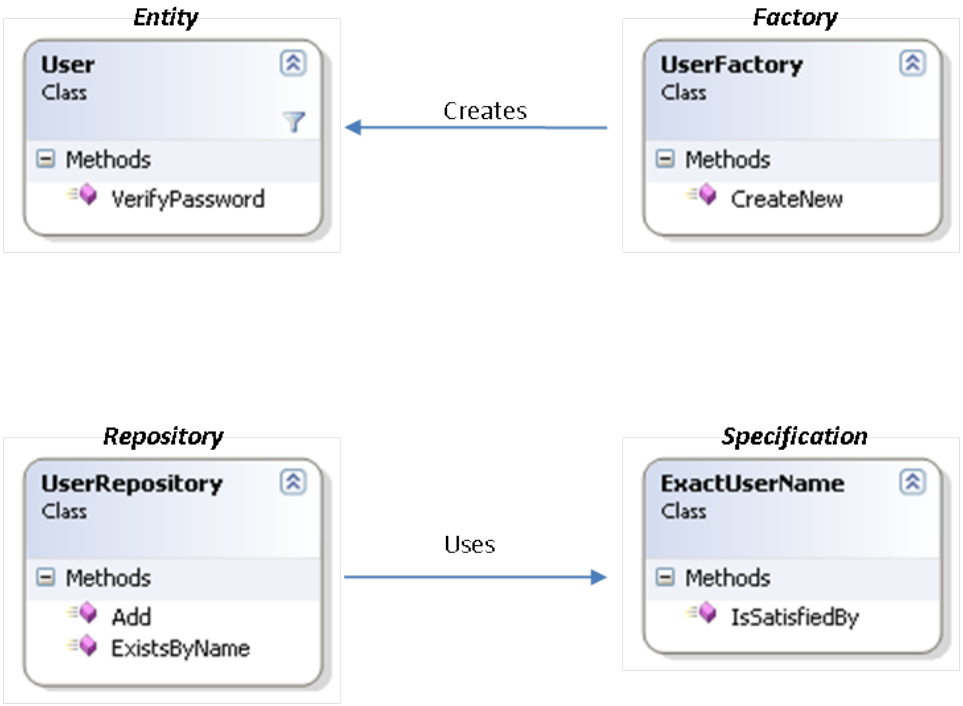


Figure 20. Class view: User creation

4.3.2 Displaying existing users

The administrator will need to be able to display a list of all existing users. To cater to this need, the **`UserRepository`** will need to have a **`FindAll`** function. It may also need to have a **`FindByFirstLetter`** function to easily fetch all users whose **`UserName`** starts with some specific letter. In fact, search functions will probably be needed in all the repositories. In Chapter 6 I will discuss how this can be achieved in a generic way. Until then, you may assume that all repositories have at least a **`FindAll`** function.

4.3.3 Inviting users

As you may remember from Chapter 2 an important feature of the system is that it must be possible to invite users to register at the the IdP. You can think of this as an easy way for the administrator to create users in the IdP, because instead of having to type in all the information about each

user, the invitation mechanism delegates this work to each user. The act of inviting a user merely entails sending an email to some email address with a verification code in it. The verification code is simply an automatically generated code that the IdP links the email address to which it was sent. By presenting his email address and the correct verification code, the user can prove that he was indeed invited. The user then chooses a user name and types in any additional information (for example values for his claims as we shall see in section 4.10) and he is then enabled in the system without further involvement of the administrator.

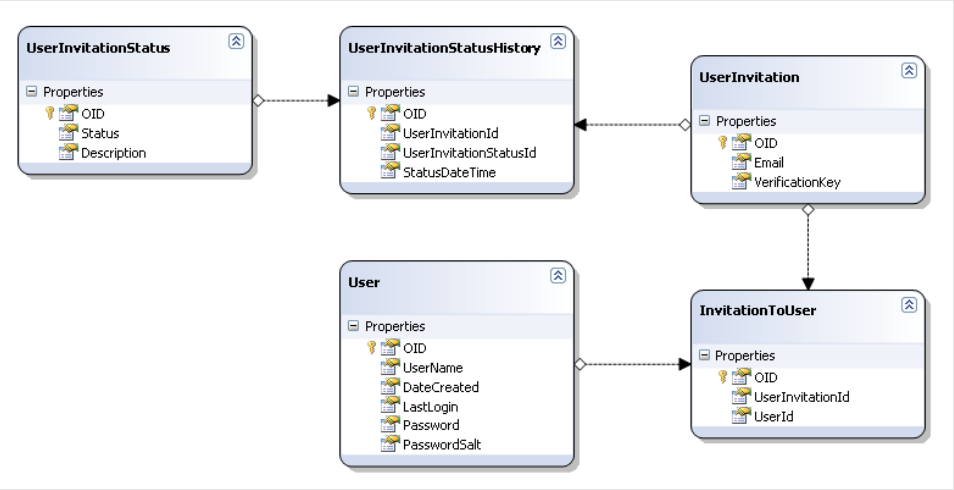


Figure 21. Schema view: User invitation

Figure 21 shows the schema involved in ***user invitations***. The `UserInvitation` class holds the `Email` and `VerficationKey` as just explained. Furthermore, each `UserInvitation` has a number of `UserInvitationStatusHistory` references, each of which reference a `UserInvitationStatus`. The `UserInvitationStatus` is used to define statuses, such as "email sent", "code verified" and "user created". The history table merely links a status to an invitation using the `StatusDateTime` property to record the date and time of the given status. When the user is created as an actual `User` in the system, an `InvitationToUser` object is created in order to be able to track which invitations correspond to which users.

Figure 22 shows the entities from Figure 21 in class view, together with all the other classes needed for user invitations. The class that orchestrates most things related to user invitations is the `UserInvitationService`

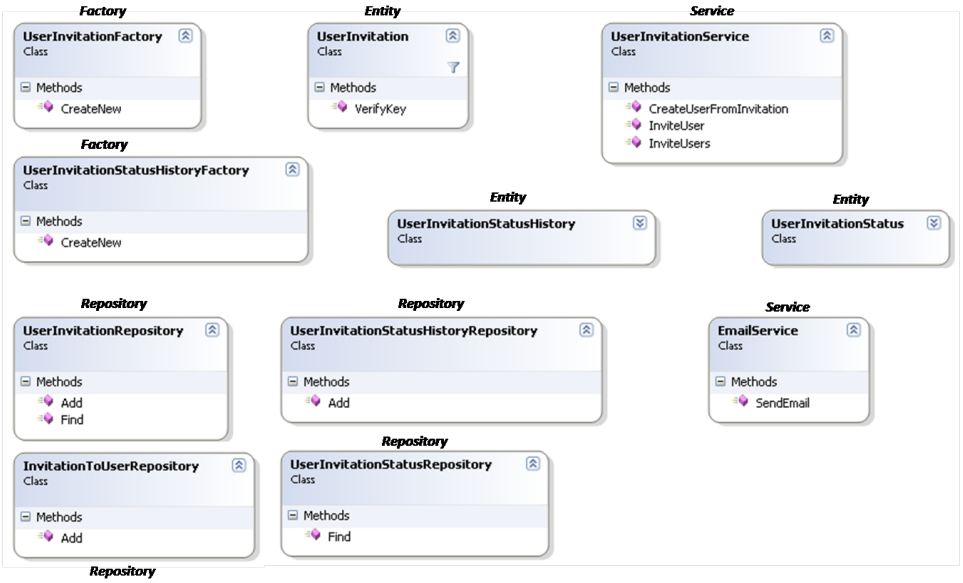


Figure 22. Class view: User invitation

class. The **`IniviteUser`** method takes a single parameter, an email address, and performs the following steps; First it calls the **`CreateNew`** method of the **`UserInvitationFactory`**, passing the email address as the only parameter to the method, to create a new instance of the **`UserInvitation`** class. The factory method internally generates the **`VerificationKey`** for the **`UserInvitation`** instance, and it also generates an instance of **`UserInvitationStatusHistory`** which it couples to the **`UserInvitation`** instance. Next, the **`UserInvitation`** instance is persisted by calling the **`Add`** method of the **`UserInvitationRepository`**. Finally, a mail message, including the verification key is composed, and the **`SendEmail`** method of the **`EmailService`** is called to send the email message. The **`InviteUsers`** method of the **`UserInvitationService`** takes a comma separated list of email addresses and calls the **`InviteUser`** method once for each email address. The **`CreateUserFromInvitation`** method takes an email address, a verification key, a user name and a password as parameters. It then finds the **`UserInvitation`** instance corresponding to the email address and verifies that the verification key is correct by calling the **`VerifyKey`** method on that instance. The **`VerifyKey`** method takes the verification key provided by the would-be user and compares it to the one it holds internally. If everything is ok, the invitation history is updated by creating the proper objects and calling the proper repositories. Finally a new **`User`** object is created and persisted, and finally an **`InvitationToUser`** instance is created and persisted.

And that concludes the user invitation, and the user is now ready to login with the IdP.

4.3.4 Self registration

Another way of user creation is through ***self registration***. Self registration has many similarities with user invitations, but differs in that the would-be user and not the administrator initiates the creation process. Furthermore, any self registration must be approved by an administrator.

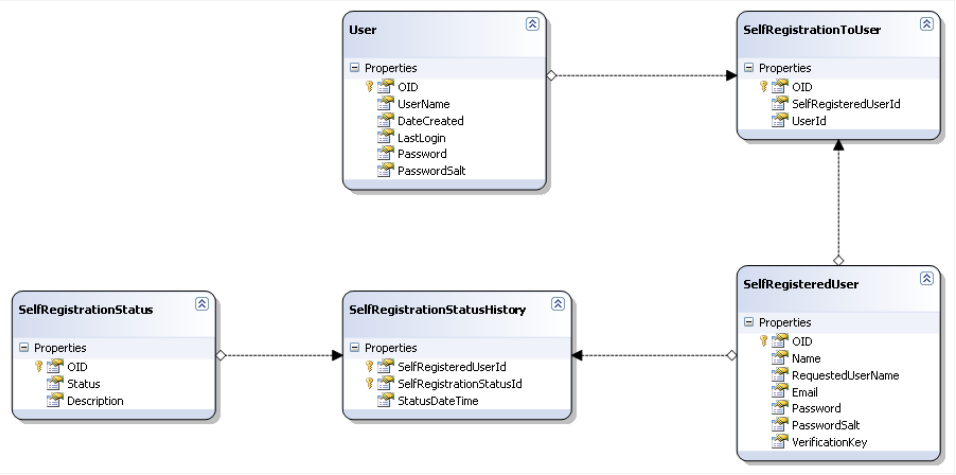


Figure 23. Schema view: Self-registration

Figure 23 shows the schema view of the classes needed for the support of self registration. You can probably recognize most of the concepts from user invitations. The main difference here is that the **`SelfRegisteredUser`** class contains all the properties needed for creating a **`User`** object, such as **`Password`**, **`PasswordSalt`**, etc. The **`SelfRegisteredUser`** class contains a property called **`RequestedUserName`**. The reason for this is that at the time of self registration a given user name may be available, but at the time of actual user creation, the name may have been taken by someone else.

Figure 24 shows the classes for self registration in class view. The classes are similar to those of user invitation, and therefore I will not explain them in detail. It is important though, to note that there are many more statuses involved in self registration than there are for user invitations. “Registered”

Figure 24. Class view: Self-registration

is the status for newly registered users. After registration an email is sent to the provided email address in order to verify the address. When the user confirms the email address, the status is changed to "Verified". When the address is verified an administrator must take some kind of action. The administrator can either approve the registration, which results in the status changing to "User created" and an email being sent to the user, confirming his registration. This email will contain the actual user name, which may have changed from what the user wanted, as described earlier. The administrator may also choose to deny registration, which results in a status of "Denied", and an email being sent to the user. Finally the administrator may choose to ignore the registration, which results in a status of "Ignored", but no email is sent. The **`UserRegistrationService`** is the class that provides all these status changes. Also notice that the **`UserFactory`** has been extended to include a method called **`CreateFromRegistration`** which takes an instance of a **`SelfRegisteredUser`** and an optional new user name, and returns an instance of a **`User`**.

4.3.5 Refactoring

The model thus far contains some implicit concepts regarding sending emails to users. Lets see how the model could benefit from having these concepts made more explicit. Instead of having the services, such as **`UserInvitationService`**, **`UserService`** and **`UserRegistrationService`** compose mail messages internally and sending them through the **`EmailService`**, these different email messages should be made explicit concepts in the model.

Figure 25 shows the classes involved in sending these email messages. First of all, the interface **`IEmailMessage`** defines three properties, **`Subject`**,

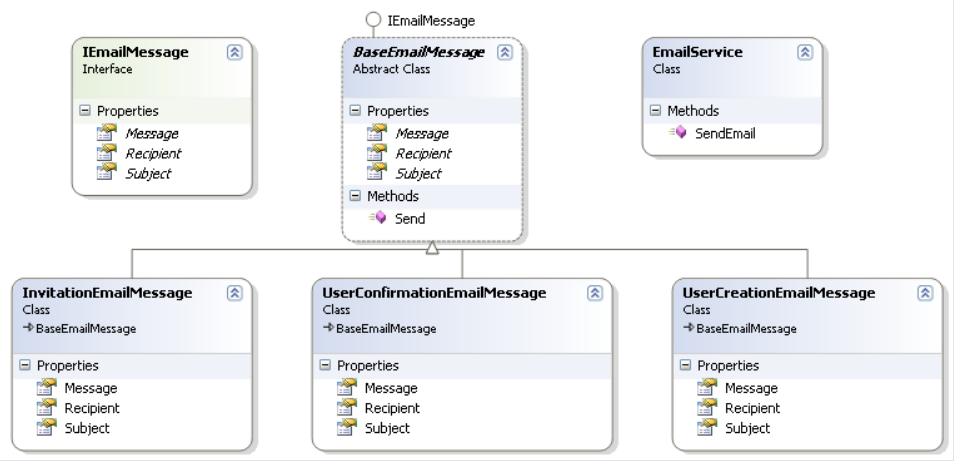


Figure 25. Class view: Emails

Message and **Recipient**. These are the properties that are needed by the **SendEmail** method of the **EmailService** class. So the signature of this method is changed to take an instance of an **IEmailMessage** as a parameter. Now, the **BaseEmailMessage** implements the **IEmailMessage** interface, but has all the properties from the interface as abstract members[4]. Furthermore, **BaseEmailMessage** implements a (non abstract) method called **Send**. This method calls the **EmailService**'s **SendEmail** method, passing itself as the parameter. This allows us to define explicit email messages such as the ones shown in the figure. For example, **InvitationEmailMessage** is a class that inherits the **BaseEmailMessage**, and takes an instance of an **UserInvitation** in its constructor. It implements the abstract properties **Subject**, **Message** and **Recipient** from the base class (and which originate from **IEmailMessage**). The **Subject** is simply a static string saying something like "You have been invited to join an identity provider". The **Message** property contains the message body, including the verification key (available through the **VerificationKey** property of the **UserInvitation** instance passed to the constructor) and a link to the verification page. Finally, the **Email** property is also read from **UserInvitation** instance. You may wonder how the correct address to the verification page is obtained. This address, and a lot of other parameters about the the IdP will be made available to all components in a way explained in Section 4.11.

A last concern is if the **EmailService** has been made obsolete. Could the

[4] Meaning that they must be implemented by any inheriting classes

logic implemented by the **SendEmail** method of the **EmailService** just as well have been put in the **Send** method of **BaseEmailMessage**? It definitely could have, but I do not think it is good idea. The main purpose of the **EmailService** is to know about which SMTP server to use and which credentials (if any) to present to that SMTP server. Furthermore, if at a later point I wished to implement a method called **CheckServerConnection**, it would definitely be more logical to have such a method on an **EmailService** class, than on a **BaseEmailMessage** class.

4.4. Configuration

Configuration of the IdP is essential in many ways. First of all, configuration of the IdP itself must be possible. Furthermore, each plug-in will rely on the ability to be configured in order to function correctly. This is also the case for connections, which are described later in this chapter. Therefore a configuration framework is called for. The configuration framework defines three basic types of data; ***text configuration element*** which hold string data, ***certificate configuration elements*** which hold references to certificates, and ***list configuration values*** which also hold string data, but which are logically grouped in lists (or tables).

Figure 26 shows the schema view of the configuration elements. Common for all the configuration elements is that they are identified by a **NameSpace** and an **ElementName**. The **NameSpace** is a string identifier of the owner of the configuration element. The namespace could be something like "idp.core" for all elements belonging to the IdP itself, or "protocol.saml2" for a some protocol implementation and finally "protocol.saml2.connection1" for some connection. Generally speaking, the namespace is just used to group configuration elements belonging to some part of the system. Therefore, the namespace used by any given part of the system must be unique. The **TextConfigurationElement** has a **Value** field which contains the actual value. The value is always stored as a string, but the **Type** field is used to indicate the type of data in the string. The **Type** could, for example, be "Boolean", and indicate thereby indicate that only the values "true" and "false" would be valid. The **CertificateConfigurationElement** contains references to certificates. The information contained in the **CertificateConfigurationElement** is to some extent specific for the way certificates are stored on machines running Windows. This does not make the model implementation specific, but

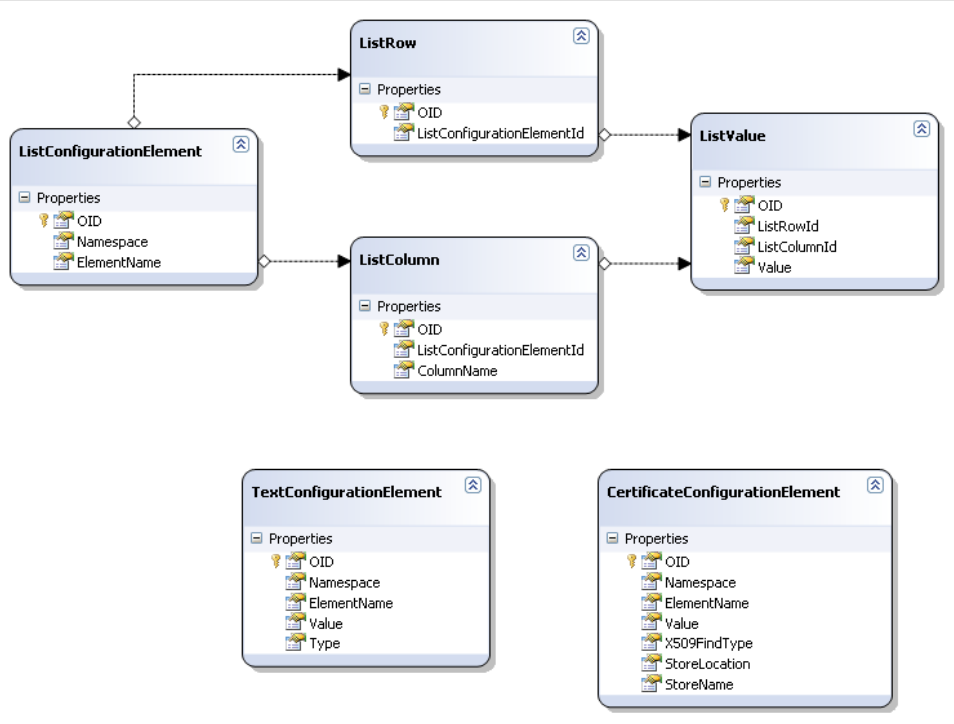


Figure 26. Schema view: Configuration elements

merely indicates that the bounded context that the model comprises, acts as a conformist to the environment where it is ultimately going to run.

The **`ListConfigurationElement`** is used to define configuration elements that have a tabular nature, and where zero or more rows of data may be called for. Figure 27 shows how a **`ListConfigurationElement`** could be presented. Generally, a single row of data contains several values, the meaning of which is defined by a column. So a **`ListValue`** is tied to both a row and a column, as shown in the figure.

The most important features of configuration elements are that they must be presentable to the administrator such that they can be read and edited. Furthermore, there must be a mechanism for validating the values typed in by the administrator. Another important concern is that in some cases, only certain values make sense for a given configuration element. The plug-in, or part of the system, that relies on a given configuration element must be able to provide a list of valid values for a given element, if such a list exists.

List ElementName

	Column 1	Column 2
	Col. ElementName	Col. ElementName
Row 1	Value (row 1, col. 1)	Value (row 1, col. 2)
Row 2	Value (row 2, col. 1)	Value (row 2, col. 2)

SAML protocol binding

Binding	Url
POST	https://...
Redirect	https://other...

Figure 27. Configuration lists

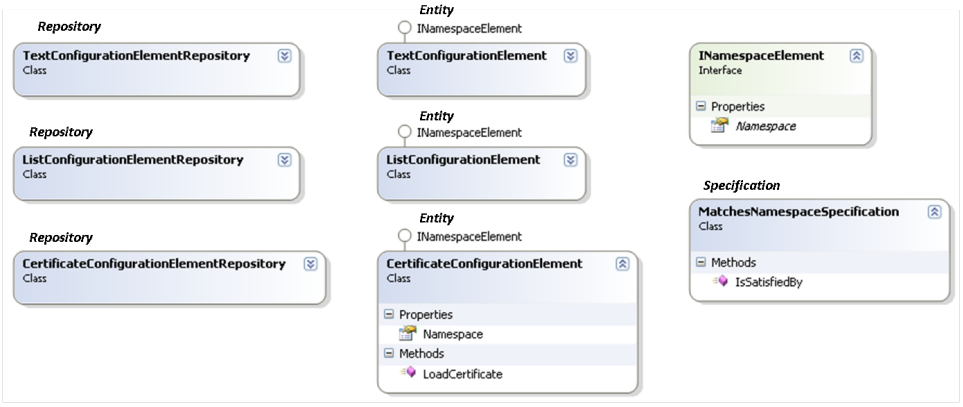


Figure 28. Class view: Configuration elements

Figure 28 shows the configuration elements and their repositories in class view. The important thing to notice here is the **`INamespaceElement`** interface, that is implemented by all the three entities. The interface merely defines that a single property called **`Namespace`**. Since all three entities

have such a property already, they can implement the interface without further ado. This is useful, because we can now define a specification called **`NamespaceMatchesSpecification`**, whose **`IsSatisfiedBy`** method takes an instance of a **`INamespaceElement`**. This way, all three repositories can use the same specification to find the elements of the respective type that matches a given namespace, thus making it straightforward to find all configuration elements belonging to a given component (identified by its namespace, of course). Also notice that the **`CertificateConfigurationElement`** defines a method called **`LoadCertificate`** that loads the actual certificate from the certificate store. This is useful because the **`CertificateConfigurationElement`** itself only contains information about where the certificate is stored. The method returns an instance of the framework class called **`X509Certificate2`**.

Any part of the system that depends on configuration values, be it the core IdP itself or any plug-in or connection, we will call a ***configurable component***. A configurable component provides the following functionality: It can provide a set of default values for all its configuration elements. It can provide a structure that logically groups its configuration values such that they can be displayed to an administrator in a way that makes sense in the context of the meaning of the configuration elements. It can validate the values of all configuration elements as a whole, and the value of a single configuration element individually. To provide this functionality a number of classes and interfaces are called for.

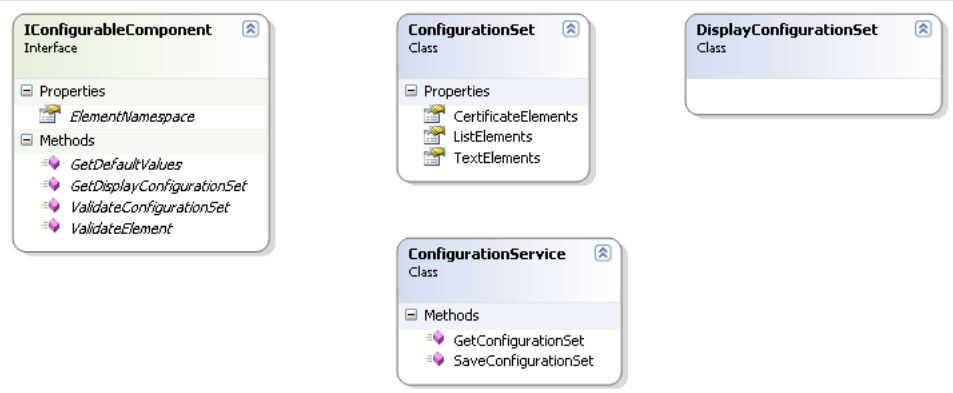


Figure 29. Class view: Configurable component

Figure 29 shows some of the concepts needed to support configurable components, in class view. A configurable component is represented by the **`IConfigurableComponent`** interface. The interface defines which namespace

the elements of the component has, through the **`ElementNamespace`** property. Furthermore, the interface defines a set of functions; **`GetDefaultValues`** is a function that returns an instance of a **`ConfigurationSet`**. This function is called during installation of a configurable component and its main purpose is to return all configuration elements such that they can be created in the database. A **`ConfigurationSet`** is merely a container for the configuration elements of the different types explained earlier. The **`GetDisplayConfigurationSet`** returns an instance of a **`DisplayConfigurationSet`**, a class that presents the configuration elements in a way suitable for showing in a user interface. The **`DisplayConfigurationSet`** class will be explained in greater details later. The **`ValidateConfigurationSet`** is a method that an instance of a **`ConfigurationSet`** as a parameter and returns an instance of a **`DisplayConfigurationSet`** (which will contain information about potential errors). Finally, the **`ValidateElement`** method, is an overloaded method that takes one of the three configuration element types as a parameter, and validates its value (or values, if it is a **`ListConfigurationElement`**). It returns a string, containing an error message if the value is not valid, or null if the validation succeeds.

The main purpose of the **`DisplayConfigurationSet`** is to display a configuration set in some UI, such as that shown in figure 30.

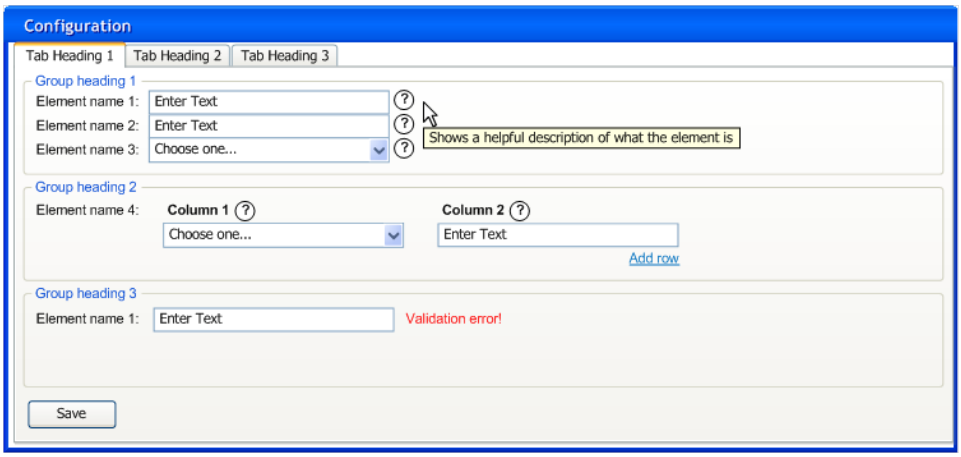


Figure 30. Visual conceptualization of configuration

Figure 31 shows the classes involved in configuration element display. The **`DisplayConfigurationSet`** is the aggregate root of all the classes, and as you may have noticed a few extra properties have been added since the class was introduced in Figure 29. These properties are the **`Caption`** property, which is a string containing a caption, or heading if you like, of the en-

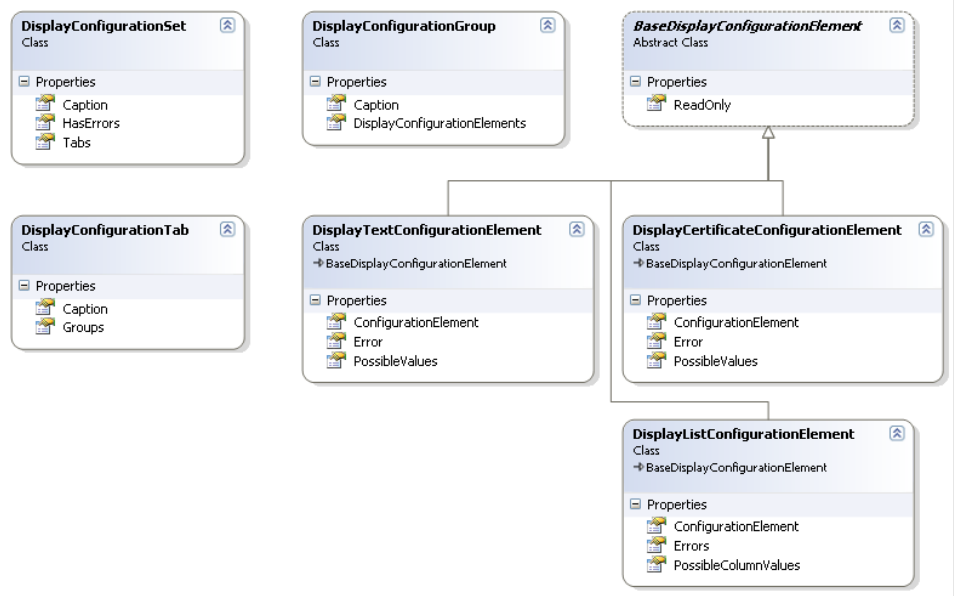


Figure 31. Class view: Display configuration elements

tire display configuration set. A caption value could for example be "Identity provider configuration" or "SAML 2.0 protocol configuration". The last property of this class is the **`Tabs`** property. This property returns a list of **`DisplayConfigurationTab`** instances. A **`DisplayConfigurationTab`** is a grouping container, intended to be displayed as a single tab in a tab control. A tab control is a common user interface control found in most applications. The **`DisplayConfigurationTab`** class also has a **`Caption`** property, and most importantly, it has a **`Groups`** property. The **`Groups`** property returns a list of **`DisplayConfigurationGroup`** instances. A **`DisplayConfigurationGroup`** is yet another grouping container, used to logically and visually group configuration elements. Apart from the **`Caption`** property it has an **`Elements`** property, which returns a list of instances of the abstract **`BaseDisplayConfigurationElement`**. The **`BaseDisplayConfigurationElement`** class serves the single purpose of being a common base class for the three display configuration elements. This is useful for the order of which the elements are displayed. The base class contains a single property, namely **`ReadOnly`** which is the only property the three display configuration elements have in common. The **`ReadOnly`** property indicates whether a given display configuration element is editable. **`DisplayTextConfigurationElement`** has the following properties; **`ConfigurationElement`** is of type **`TextConfigurationElement`** and represent the configuration ele-

ment that is being displayed. The **Error** property is a string containing a possible error message for the element. Finally, the **PossibleValues** is a list of possible values. The **PossibleValues** may be null, in which case the administrator can type in any value. However, if the **PossibleValues** is not null, the string values it contains will be rendered as a drop-down list, from where a value can be chosen. The other two display configuration elements are similar. They differ in the type returned by the **ConfigurationElement**. Furthermore, **DisplayCertificateConfigurationElement** differs in that its **PossibleValues** does not return a list of string values, but a list of **X509Certificate2** instances, from which the administrator can choose. The **DisplayListConfigurationElement** differs in that its **Errors** property is a matrix of error messages, corresponding each **ListValue** inside the **ListConfigurationElement** returned by the **ConfigurationElement** property. Finally, the **PossibleColumnValues** is similar to the **PossibleValues** property of the other two display elements, however it contains a list of lists, where each list corresponds to the possible values for a column in **ListConfigurationElement**.

4.4.1 Refactoring

There needs to be some way of communicating the meaning of a configuration element to the administrator. Sometimes the name of a configuration element may not be enough do explain what a configuration element is. Therefore a **Description** property is called for. Now, should the **Description** property be on the configuration element, or on its display counterpart? Since it is primarily a display related thing, I have chosen to extend the **BaseDisplayConfigurationElement** with a **Description** property.

4.5. Certificates

If you know a little about X.509 certificates, you know that most certificates are issued by other certificates. Those certificates that are not issued by another certificate are said to be self-issued, and are often referred to as ***root certificates***. In order for an application to deem some certificate to be valid, it must, amongst other things, trust the root certificate that issued that certificate. Quite a few companies world-wide make their living by issuing certificates. Such companies are called certificate authorities (CA's), and they use a very well guarded root certificate to issue certificates that

other companies or people can buy. The root certificates of these CA's are per default trusted by all computers world wide, because they ship with the operating system. Sometimes, when you do not wish to pay for a certificate that you are going to use for test purposes, you can create your own root certificate, and use this to issue other certificates. This will work fine on your own systems, because you can choose to trust that root certificate. However other systems will not trust your root certificate, but that does not really matter as long as you are only using it for test purposes, or if you can persuade your business partners to trust your root certificate.

We have already seen how certificates can be used in configuration of the system. However, the certificate configuration elements were only references to certificates in the certificate store. The administrator needs some means of inspecting (searching for) the certificates in the certificate store, creating new certificates, deleting existing certificates and so on.

Figure 32. Class view: Certificates

Figure 32 shows the classes involved in maintaining certificates. The most important class is the **`CertificateService`** class. The **`CreateRootCertificate`** method can be used to create a root certificate as explained above. The only parameter to the method is a so-called distinguished name, a string value that is used to identify the certificate later. The **`CreateCertificate`**

is used to create a certificate. As a parameter it also takes a distinguished name, but furthermore, it takes a root certificate that is used to issue the certificate. The root certificate parameter is passed to the method as an instance of the **`X509Certificate2`** framework class. The **`DeleteCertificate`** is used to delete a certificate, and it takes an instance of an **`X509Certificate2`** class its only parameter. The **`ExportCertificate`** method takes as its only parameter an instance of the **`X509Certificate2`** class, and returns a byte array representing the certificate's public key in a common format[5]. The **`ExportCertificateWithPK`** exports a certificate including its private key. This method takes an instance of an **`X509Certificate2`** class, together with a password to use to protect the certificate. It also returns a byte array, this time corresponding to both the public and private key of the certificate, and also in a common format[6]. The **`ImportCertificate`** method is an overloaded method. The first overload only takes a byte array (in CER format) and imports a certificate public key into the certificate store. The other overload, takes a byte array (in PKCS12 format) and a password, and imports the certificate private and public key into the certificate store. The **`FindCertificates`** method takes a list of certificate specification instances, and returns all the certificates that match those specifications. All the certificate specifications in the figure take an instance of the **`X509Certificate2`** in their respective **`IsSatisfiedBy`** methods. The specifications are; **`CertificateByDistinguishedName`** which compares a certificate to the distinguished name string parameter given in the specification's constructor. The **`CertificateIsValid`** checks if a certificate is valid, eg. issued by a trusted root certificate, and not expired. The **`CertificateIsInvalid`** does the opposite, and finally, the **`CertificateMatches`** compares one certificate to another. The **`FindRootCertificates`** does exactly the same as the **`FindCertificates`** method, only it searches in trusted root certificates store instead of in the standard location.

4.5.1 Refactoring

When deleting a certificate, it would be useful to have a feature that could check if the certificate is being used by a **`CertificateConfigurationElement`**, and warning the administrator if this was the case.

[5] A common format is the canonical encoding rules (CER) format which is supported on most platforms.

[6] A common format for exporting certificates with private key is the PKCS12 format. PKCS stands for Public-Key Cryptography Standards, and 12 is a version number.

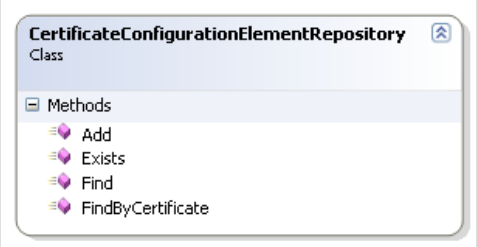


Figure 33. Class view: Refactoring the certificate configuration element repository

Figure 33 shows that the **`CertificateConfigurationElementRepository`** has been extended to include a method called **`FindByCertificate`**. It takes an instance of an **`X509Certificate2`** and returns a list of **`CertificateConfigurationElements`** that are currently the depending on that given certificate. If the list is empty, it is safe to delete the certificate. If the list is not empty, the administrator can be told which certificate configuration elements depend on that certificate, and possibly change those configuration elements to reference another certificate before deleting it.

4.6. Credential providers

Credential providers are specialized plug-ins that facilitate user logins. The IdP will offer several different credential providers, but it is the administrator who chooses which credential providers are enabled for his IdP. A concrete credential provider plug-in implementation is contained within a module (dll) on the IdP server, and registered in the database, such that the administrator can enable it.

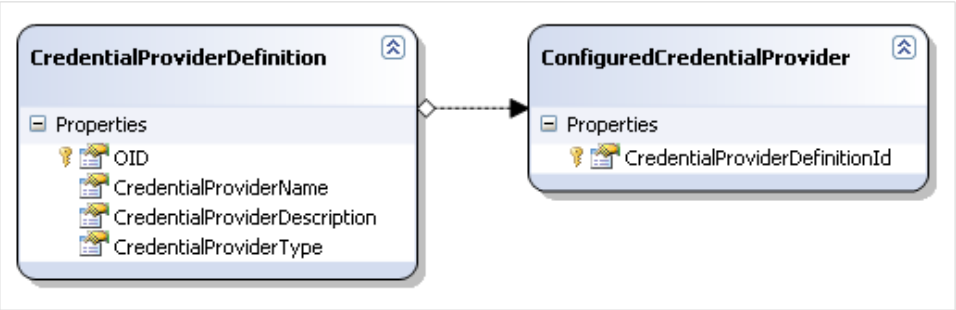


Figure 34. Schema view: Credential providers

Figure 34 shows the who main concepts involved in setting up credential providers. A **`CredentialProviderDefinition`** defines a credential provider that can be used to log users in. It has a **`Name`** and a **`Description`**, and most importantly it has a **`CredentialProviderType`**, which is a string that contains a fully qualified type name, that can be used to create an instance of the given credential provider using reflection. The type name must represent a type that implements a specific interface, which I will explain in detail below. The **`ConfiguredCredentialProvider`** is merely a pointer to a **`CredentialProviderDefinition`**, and indicates that the given **`CredentialProviderDefinition`** has been configured (is in use).

As mentioned in Chapter 2, the IdP must be able to accommodate credentials both in the form of username/password and via SAML 2.0 federation. The credential mechanisms are actually quite different, in that the username/password variant validates users through the local database, while the SAML variant uses federation with another identity provider to collect the user's credentials. The model only allows for one instance of each credential provider to be configured. This works well for the username/password scenario, as there is only one local user database anyway. But what about the SAML federation scenario. Here, it must be possible to setup several federations with several other IdPs. And this is indeed going to be possible, however not by configuring more than one instance of the credential provider, but by the concept of ***connections***, explained in Section 4.8.

Figure 35 shows the interface that must be implemented by plug-ins that are credential providers. The **`ICredentialProvider`** interface extends the **`IPlugIn`** interface, and adds two new properties. **`SupportsConnections`** is a boolean that indicates whether or not the credential provider supports connections. This is useful for displaying a user interface to the administrator where he can configure the connections for a given credential provider that supports connections. The **`DefaultEndpoint`** is a property that returns an instance of an **`IEndpoint`** (see Figure 18 on page 48). The default endpoint is the endpoint that initiates the login for a given user. A credential provider may have more endpoints for various purposes, so it is important that the IdP knows which is the default endpoint, such that it can initiate the login sequence properly.

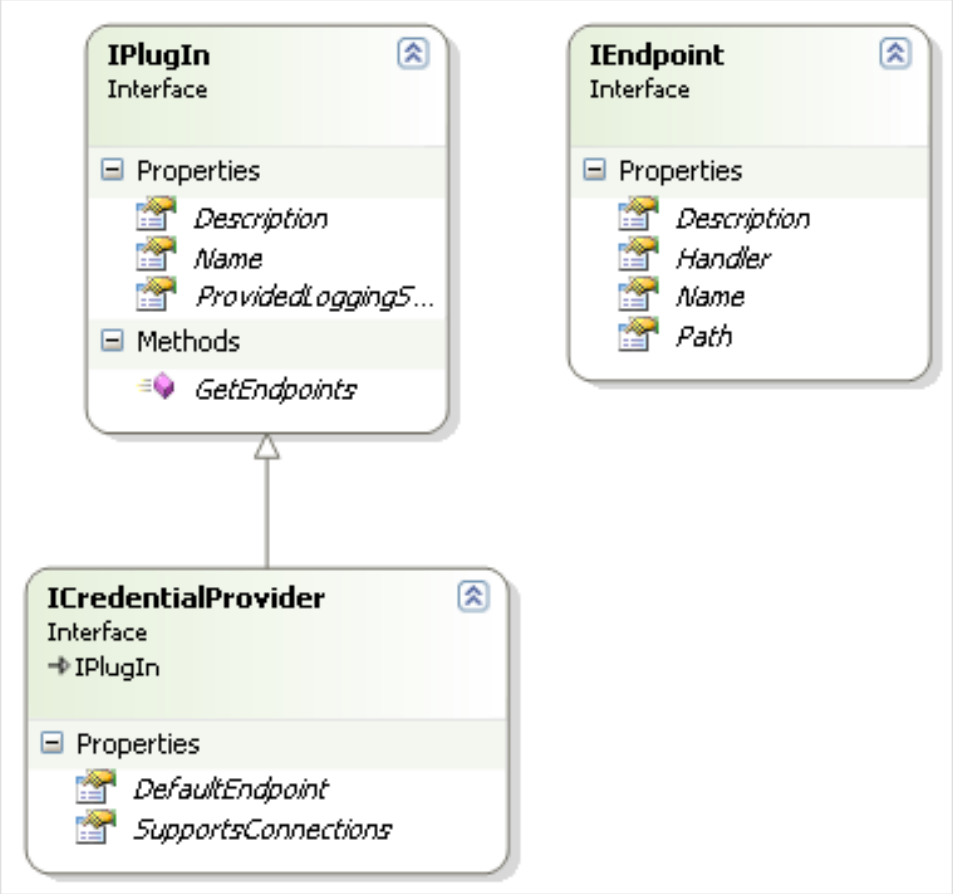


Figure 35. Class view: Credential provider interface

4.7. Protocols

Protocol plug-ins are specialized plug-ins that provide the implementation of some protocol for exchanging security related information. Such protocols include SAML, WS-Federation, OpenID and many more.

Figure 36 shows the schema view of the classes for defining and configuring protocols in the IdP. There is no real difference from the way credential providers where defined and configured, so I will not get into a lengthy explanation here. Notice however, that there are repositories for both concepts (`ProtocolDefinition` and `ConfiguredProtocol`) even though they are not shown here.

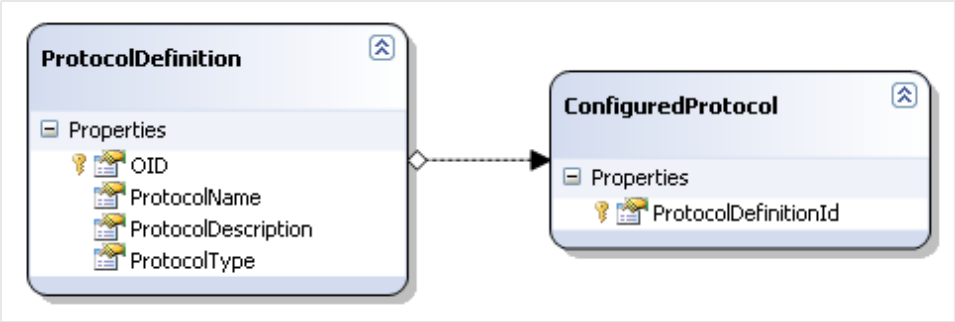


Figure 36. Schema view: Protocols

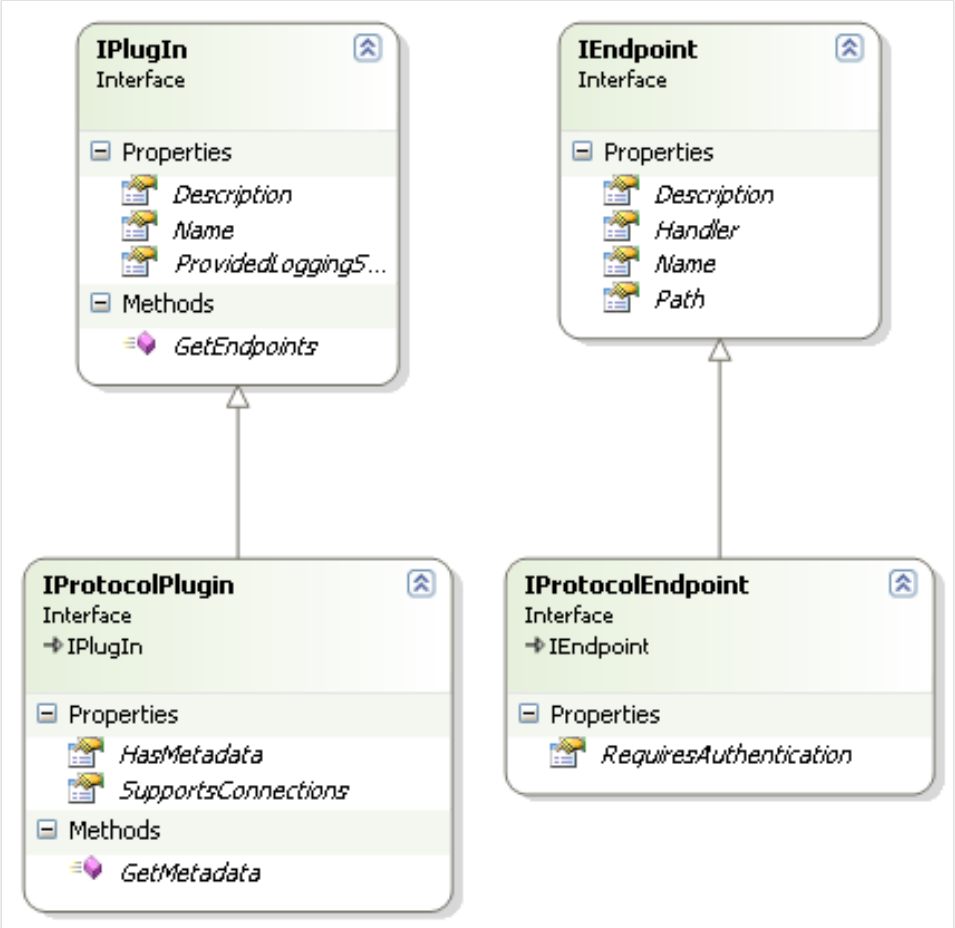


Figure 37. Class view: Protocol plug-in interfaces

Many, but not necessarily all, protocols rely on the exchange of metadata in order to establish a trust relationship between an identity provider and a service provider[7]. Our IdP has no way of knowing the metadata format used in different protocols, therefore the responsibility of creating the metadata must be delegated to the protocol plug-in implementation itself. Therefore, the `HasMetadata` property of the `IPotocolPlugIn` can be used by the implementor of a specific protocol to indicate whether or not the implementation provides (or has) metadata. When that is the case, the implementation of `GetMetadata` is supposed to return a string representing metadata for the given implementation. If `HasMetadata` returns false, the `GetMetadata` method will not be called. It does however still need to implemented, because otherwise the interface will not be implemented. Throwing an exception in the method implementation will be acceptable. The `SupportsConnections` property is identical to that of credential providers. This indicates that the property could be moved to the common interface `IPlugIn`, something that I will address below. The `IProtocolEndpoint` interface is an extension of the `IEndpoint` interface, which adds one important property, `RequiresAuthentication`. This property indicates whether a given protocol endpoint requires the user to be authenticated before delegating control to the endpoint. This is the case for all protocol endpoints that send user's identity to a service provider. The IdPwill check any protocol endpoint for this property, and make sure to authenticate the user (through one of the configured credential providers) before delegating control to that endpoint. I will explain this in greater detail in Section 4.11.

4.7.1 Refactoring

As mentioned before, the `SupportsConnections` property exists in both the `ICredentialProviderPlugIn` and `IProtocolPlugIn` interfaces. Therefore it can be moved to the `IPlugIn` interface. Furthermore, some credential providers also need to export metadata. This means that the `HasMetadata` property and the `GetMetadata` method could also be moved. This leaves the question of whether the two interfaces should be there at all. However I like the idea of having them both. They are good to have for the sake of of the possibility that the two may diverge in the future. Having them both also make them more explicit as concepts, which is important in terms of domain-driven design.

[7] Service provider is also sometimes referred to as "relying party".

4.8. Connections

As already mentioned, it is normal for an identity provider to require a configured connection before it wants to communicate with a service provider. A connection is merely some information about the other party. This information usually includes certificates, and a set of string values describing various aspects of the way communication between the two parties is carried out. In other words, a connection is a specialized configuration set. Given the fact that the data (the configuration set) that describes a connection is specific to a given protocol (the language the two parties "speak"), it must be responsibility of the protocol implementation to provide the default configuration set values, and the display configuration set. A credential provider may also need to configure connections, for example in the case of federated SAML2.0 login. In this case the credential provider acts as a service provider to some other IdP.

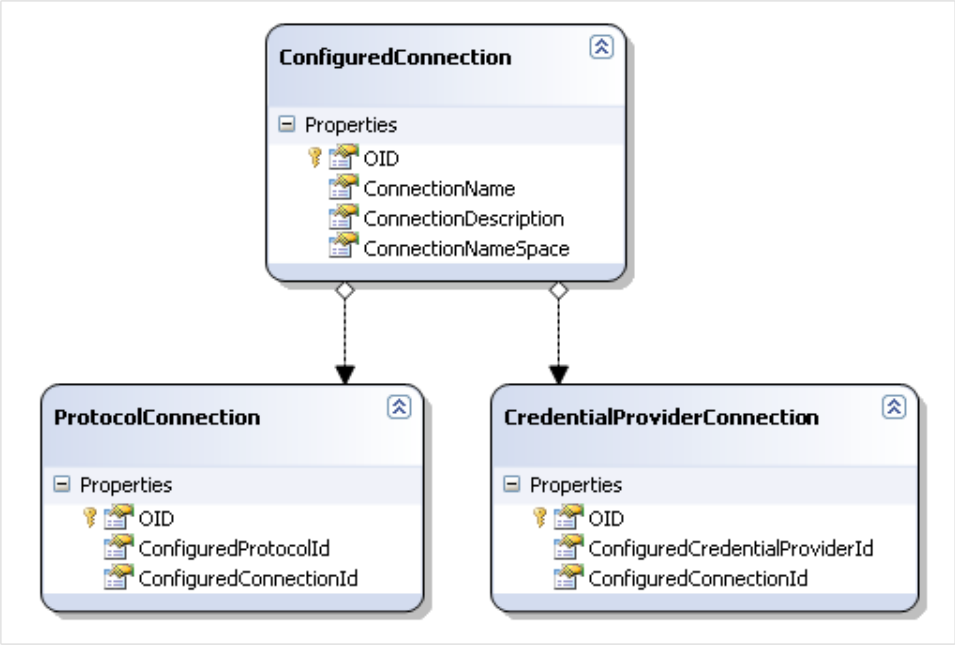


Figure 38. Schema view: Connections

Figure 38 shows the schema view of some of the classes needed to model connections. A `ConfiguredConnection` represents some configured connection. Besides having an object identifier, a configured connection has a name, a description, and a namespace, represented by the properties

`ConnectionName`, **`ConnectionDescription`** and **`ConnectionNamespace`**. The **`ConnectionNamespace`** is used to map a configured connection to a **`ConfigurationSet`**. As the figure also shows, a configured connection belongs to either a protocol or a credential provider. This mapping is achieved through the **`ProtocolConnection`** and **`CredentialProviderConnection`** classes, that map a configured connection to either a configured connection or a configured protocol. Please note that the **`ConfiguredProtocol`** and **`ConfiguredCredentialProvider`** classes are not shown in the figure.

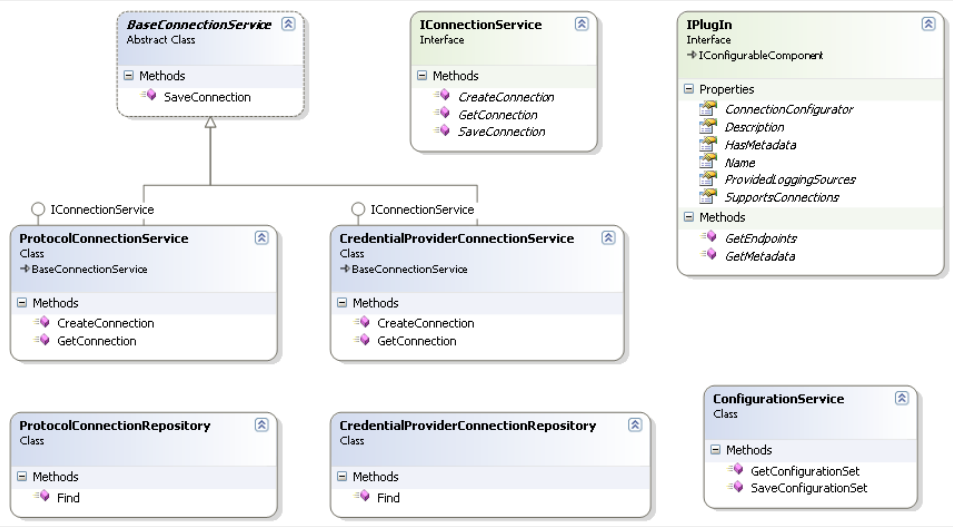


Figure 39. Class view: Connections

Now, in order to support working with the data classes shown in Figure 38 a set of services and repositories are called for. These are shown in class view in Figure 39. A ***connection service*** is a service that facilitates creating, updating and fetching of configured connections. The **`IConnectionService`** defines these functions. The **`CreateConnection`** method has three parameters, namely the parameters required to create a new instance of the **`ConfiguredConnection`** class. These are **`ConnectionName`**, **`ConnectionDescription`**, and **`ConnectionNamespace`**. The **`GetConnection`** method has a single parameter, **`OwnerId`** and it returns an instance of **`ConfiguredConnection`**. The **`OwnerId`** refers to the id of either a **`ConfiguredProtocol`** or a **`ConfiguredCredentialProvider`**. Finally, **`SaveConnection`** has two parameters; the first is an instance of **`ConfiguredConnection`**, and the second is an instance of a **`ConfigurationSet`**. There are two classes that implement the **`IConnectionService`** interface, namely **`ProtocolConnectionService`** and **`CredentialProviderConnectionService`**.

Both classes inherit the `BaseConnectionService`. The `BaseConnectionService` is an abstract class that contains the implementation for `SaveConnection`, since this method must do the same thing in the context of protocol connections and credential provider connections. The `SaveConnection` method updates (saves) the instance of `ConfiguredConnection` through the `ConfiguredConnectionRepository` (not shown in the figure), and likewise it saves the instance of `ConfigurationSet` through its corresponding service (`ConfigurationService` shown in the figure). The implementation of `GetConnection` and `CreateConnection` is different for the `ProtocolConnectionService` and `CredentialProviderConnectionService` classes. This is due to the fact that they interact with instances of the `ProtocolConnection` and `CredentialProviderConnection` respectively (shown in Figure 38). In the case of the `CreateConnection` method, it is the responsibility of the respective service implementation to create an instance of the correct connection class (either `ProtocolConnection` or `CredentialProviderConnection`) in order to associate the connection with either a protocol or a credential provider. Likewise, for the `GetConnection` method, it is the responsibility of the concrete implementation to use the `OwnerId` parameter to fetch the correct instance of either `ProtocolConnection` or `CredentialProviderConnection` (using the appropriate repository, again, not shown in the figure for brevity). It has also been necessary to expand the `IPlugIn` interface with a property called `ConnectionConfigurator`. This property is of type `IConfigurableComponent`, and is used to validate the values in the configuration set for either a protocol connection or credential provider connection. The concrete implementation of the `IConfigurableComponent` must of course be provided by the implementation of a protocol or credential provider, because only these implementations can know what the valid values are in their specific case.

4.9. Plug-in installation

Since the system is plug-in based, it is important that installation of new plug-ins is supported by the system. Most importantly, the installation of a new plug-in must not lead to multiple endpoints that handle the same path. It is also important to test that the new plug-in implements the correct interface, and that there are no endpoints within the plug-in that handle the same path.

Figure 40 shows the classes needed for plug-in installation. The central class is the `PluginInstallationService` that has two methods; one for in-

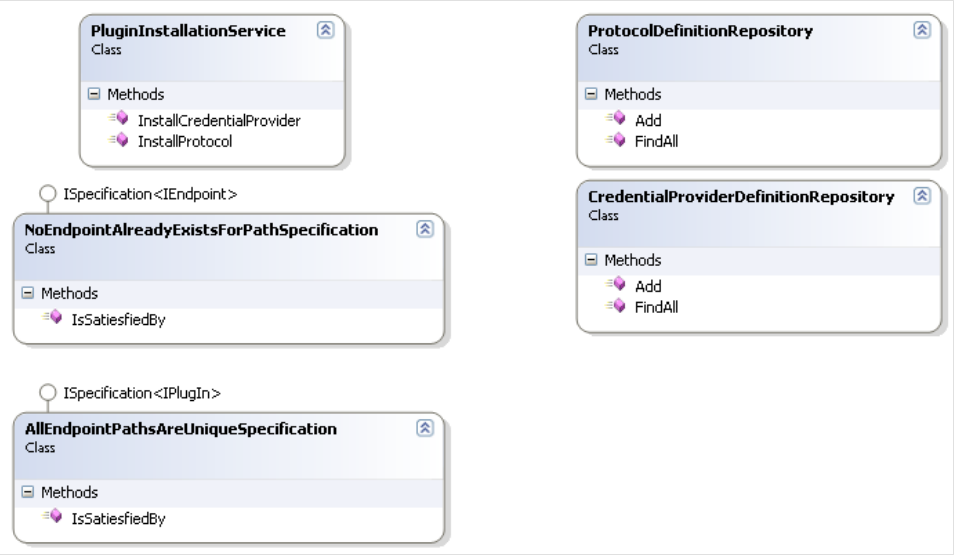


Figure 40. Class view: Plug-in installation

stalling protocol plug-ins and one for installing credential provider plug-ins. Each method has two parameters, namely the type string[8], and the assembly containing the plug-in, as a byte array. The two methods perform similar actions:

Step 1 Load the assembly and instantiate the class given in the type string through reflection.

Step 2 Test that the instance implements the correct interface (**`ICredentialProviderPlugin`** or **`IProtocolPlugin`**).

Step 3 Create an instance of the **`AllEndpointPathsAreUniqueSpecification`** and make sure that its **`IsSatisfiedBy`** method returns true. The **`IsSatisfiedBy`** method has an argument of type **`IPlugIn`**, and it iterates all endpoints returned by the **`GetEndpoints`** method and tests that no two endpoints have identical path properties.

Step 4 Create an instance of the **`NoEndpointAlreadyExistsForPathSpecification`** class. The constructor of this class takes a list of paths that are registered for those plug-ins that are already installed. In order to make this list, the two repositories shown are used get all installed plug-ins, which are then instantiated, and the paths of their handlers are appended to the list. Finally, the **`IsSatisfiedBy`** method is called for each endpoint

[8] Corresponding to the **`CredentialProviderType`** or **`ProtocolType`** properties of the **`CredentialProviderDefinition`** or **`ProtocolDefinition`** classes respectively.

in the plug-in that is going to be installed.

Step 5 If we have come this far, the plug-in is ready to be installed. An instance of either **`CredentialProviderDefinition`** or **`ProtocolDefinition`** is created and added to the corresponding repository. The name and description properties of the definition object are taken from the concrete instance instantiated in step 1. Since this instance implements the **`IPlugin`** interface (both Verb|ICredentialProviderPlugin| and **`IProtocolPlugin`** are extensions of **`IPlugin`**) these properties are available on the concrete instance.

4.10. Claims

The ability to define and work with claims is one of the core features of the IdP. The IdPdefines three conceptually different types of claims. The first type of claim is a claim that originates from some credential provider. In the simple case of username/password credentials, the only claim that originates from the credential provider is a username claim. However, in the case of federated login, a number of claims may be returned by the federation partner. This type of claim is called a **`CredentialClaim`**. The next type of claim is a claim that is defined in the IdP itself. This claim type is called an **`IdentityProviderClaim`**. Finally, the last type of claim is called **`IssuedClaim`**, and represents a claim that is issued by the IdP.

Figure 41 shows some of the classes related to claims, in schema view. The **`IdentityProviderClaimDefinition`** class is used to define an identity provider claim, in the sense that it does not provide a concrete value for the claim. It does however define every other aspect of the claim. The **`DisplayName`** property is a reader friendly name used for display in UI. The **`Name`** is the actual name of the claim used when sending the claim some security token. The **`NameFormat`** is used to describe the format of the **`Name`**. It is normal for different protocols to define name formats, that a name must comply to. A name format could, for example, specify that the name must be a well-formed URI. The **`ValueType`** defines the type of value that is acceptable for the claim. The value type is normally expressed as an xml type, such as ***xs:string*** or ***xs:int*** etc. The **`Description`** is merely a textual description, mainly used in UI. The **`DefaultValue`** is used to define a default value for the claim, in the case where a user has no explicit value for the given claim, or in the case where all users must get a predefined value for

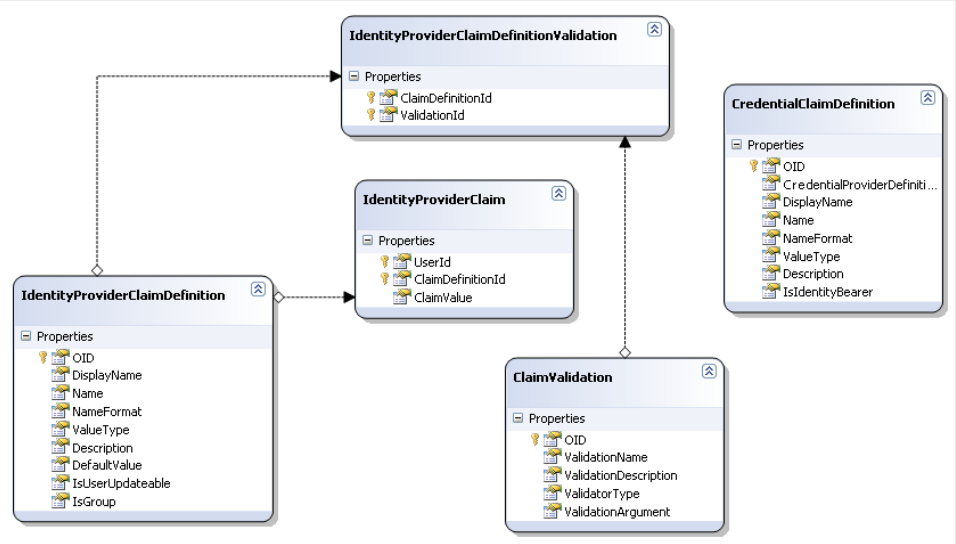


Figure 41. Schema view: Claims

the claim. The **`DefaultValue`** is, for example, used for defining groups. As specified in Chapter 2, it must be possible to assign users to groups. Adding a user to a group simply results in a claim that states that the user is part of some group. By creating an **`IdentityProviderClaimDefinition`** with a default value of for example ***Administrator***, and **`ClaimName`** ***idp:Group***, a conceptual group can be created. When the claim is issued by the IdP it looks like any other claim, but having the **`DefaultValue`** and **`IsGroup`** properties allows the IdP to display a user interface where the administrator can work with groups. Finally, the **`IsUserUpdateable`** boolean property is used to specify whether or not the value of a claim can be changed by a user. Again, in the case of groups, the **`IsUserUpdateable`** property will be set to false, whereas for other claims it could be set to true. The **`IdentityProviderClaim`** couples an **`IdentityProviderClaimDefinition`** to a **`User`**, and (optionally) assigns a **`ClaimValue`** for the given claim definition to the given user. The **`CredentialClaimDefinition`** class is similar to the **`IdentityProviderClaimDefinition`** class in that it also has **`DisplayName`**, **`Name`**, **`NameFormat`**, **`ValueType`** and `Description` properties. The **`CredentialClaimDefinition`** class does not have a **`DefaultValue`** property, since its value is not assigned by the IdP. For the same reason it does not have the **`IsGroup`** and **`IsUserUpdateable`** properties. It does however have an **`IsIdentityBearer`** property. This property is used to define that a given credential claim bears the identity of the user, and that its value can be used to find a

corresponding **User** instance. For this to work, the credential claim that is identity bearer must have a value that corresponds to a username of some given **User** instance. This solution definitely works theoretically, but it is probably not very practical, and therefore, this mechanism is a clear candidate for future refactoring. The Verb|ClaimValidation| class is used to define validation classes for the value of identity provider claims. If, for example, some claim dictates that its value must be an integer, or have a specific length or format, a **ClaimValidation** can be coupled to the **IdentityProviderClaimDefinition**. When an administrator or a user assigns a value to the claim, the validator is instantiated and evaluated against the value. The mechanism is very similar to that of plug-ins in that a **ClaimValidation** refers to a type string that can be used to create instances of the validator through reflection.

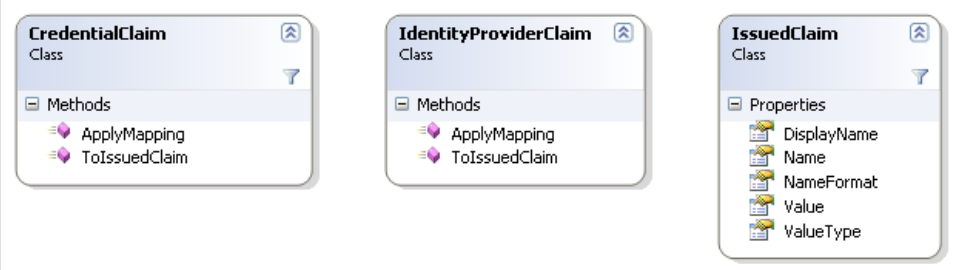


Figure 42. Class view: Claims

Figure 42 shows the three different claim types. The **CredentialClaim** is a class whose main purpose is to tie a **CredentialClaimDefinition** with a value. Instances of the **CredentialClaim** class are never persisted, and are created by a given **CredentialProvider** implementation. The **IdentityProviderClaim** class is the same as we have already seen in Figure 41. It is shown here again to emphasize the two methods it (and the **CredentialClaim** class) has. The **ToIssuedClaim** method takes no parameters and returns an instance of the **IssuedClaim** class. The **IssuedClaim** class is a simple data class that holds the values that are needed in the response of a given protocol implementation. A given protocol implementation can query the IdP for a list of instances of **IssuedClaim** for the current user, as we shall see in Section 4.11 on page 79.

4.10.1 Claim mapping

As mentioned in the specification, it must be possible to change various properties of a claim depending on who the receiver of that claim is. This concept is called ***claim mapping***. It is important to note that it is not the value of a claim that is changed, but only the descriptive properties, **`Name`**, **`NameFormat`**, and **`ValueType`**.

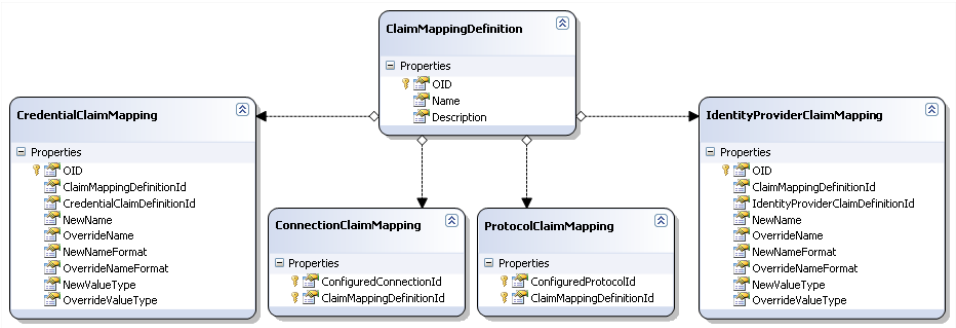


Figure 43. Schema view: Claim mapping

Figure 43 shows the entity classes used to model claim mapping, in schema view. The **`ClaimMappingDefinition`** is used to define a claim mapping. A claim mapping definition has a **`Name`** and a **`Description`** property, that describe the claim mapping. A claim mapping definition is made up of zero or more **`CredentialClaimMapping`**s and/or **`IdentityProviderClaimMapping`**s. These two classes define a number of Boolean values, **`OverrideName`**, **`OverrideNameFormat`**, and **`OverrideValueType`**. The semantics of these values are that if they have the value "true", the corresponding "New" value (**`NewName`**, **`NewNameFormat`**, or **`NewValueType`**) will be used to override the original value of the claim. By definition, a claim mapping belongs to either a **`Protocol`** or a **`Connection`**, and this fact is modelled in the classes **`ProtocolClaimMapping`** and **`ConnectionClaimMapping`**, which hold references to a **`ClaimMappingDefinition`** and a **`ConfiguredProtocol`** and **`ConfiguredConnection`** respectively.

As with every other aspect of the model, there are services and repositories to handle reading and writing claim mapping data. The actual mapping of claims is done by the **`ClaimMappingService`**, shown in Figure 44. The service has one method, `GetMappedClaimsForUser`, which has three parameters, a user name, a protocol id, and a connection id. In chapter 7 I will show the implementation of this method.

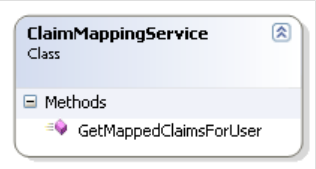


Figure 44. Class view: Claim mapping

4.11. The runtime system

During runtime, all the components of the model are orchestrated in a way that makes the system work. Figure 45 shows the sequence of events that happen when the IdP receives a request for a protocol endpoint.

Phase	Invoker	Executing component
BeginRequest	IIS	IdPCore
DetermineProtocol	IdPCore	IdPCore
ValidateRequest	IdPCore	Protocol implementation
AuthenticationCheck	IdPCore	Protocol implementation
DetermineCredentialProvider (*)	IdPCore	IdPCore
Authentication (*)	IdPCore	Crendential provider implementation
MapUser	IdPCore	IdPCore
ProcessRequest	IdPCore	Protocol implementation
ExtractIdPClaims (*)	ProtocolImpl	ClaimMappingService
SendResponse	-	Protocol implementation

Figure 45. Runtime request sequence

Each of the steps is described below.

BeginRequest In this phase the IdP initializes the ***IdPContext*** (explained in Section 4.11.1) and establishes a session. All request parameters will be saved in the IdPContext.

DetermineProtocol In this phase, the IdP uses the requested URI to determine which specific protocol implementation to reroute the request to. This protocol will be referred to as the "handling protocol". If the requested URI does not match any of the URIs exposed by the configured protocols an error is reported and a trace entry is created.

ValidateRequest During this phase, the IdP lets the handling protocol perform validation on, for example (but not limited to), the wellformedness of the request. If validation fails an error is reported and a trace

entry is created.

AuthenticationCheck During this phase, the IdP will ask the handling protocol if the requested endpoint is one that requires an authenticated user. If this is the case, the next phase will be performed; otherwise it will skip to the ProcessRequest phase. If the user is already authenticated, the IdP will skip to the ProcessRequest phase.

DetermineCredentialProvider (optional) During this phase the IdP will determine which credential providers have been configured. If more than one provider has been configured, the user will be presented with a list of possible providers to choose from. Otherwise there will be only one, and that one is chosen. The credential provider that ultimately collects the user's credentials will be referred to as the "handling credential provider".

Authentication (optional) In this phase the handling credential provider will collect the user's credentials and create the set of credential claims. The credential claims will stored in the IdPContext.

MapUser During this phase, the identity bearing credential claim will be mapped to a userid in the user store

ProcessRequest During this phase the handling protocol will process the request as specified by its implementation. Optionally the claims for the current user will be extracted.

ExtractIdPClaims (optional) During this phase the user's IdPClaims are extracted. If necessary, each claim is mapped as defined in the IdP configuration. First a protocol mapping is performed (if it exists) and then a service provider mapping is performed. Furthermore, all credential claims that must be reissued are promoted to IdPClaims

SendResponse During this phase the response it sent and the sequence ends.

4.11.1 IdPContext

The ***IdPContext*** is the IdP's main interface for plug-ins. It exists to make it easier for any plug-in to access the IdP's core functionality without having to know too many details about services and repositories, but it also serves as a placeholder for state information between http requests.

Figure 46 shows the `IdPContext` class in class view. The class does not have a public constructor, and instead it is implemented using the singleton pattern (as explained in [Gamma et al. 1995]). The singleton instance

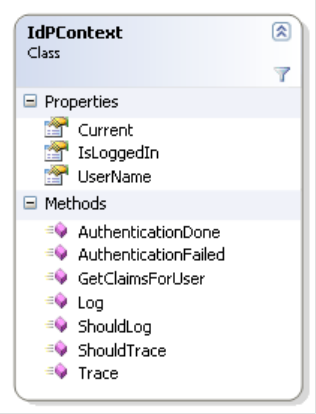


Figure 46. Class view: IdPContext

of the class is accessed through the **`Current`** property of the class. The **`AuthenticationDone`** method must be called by a credential provider, upon successful authentication of the user. The method has an argument whose type is a list of **`CredentialClaim`** instances. If, for some reason, the authentication fails, the **`AuthenticationFailed`** method must be called by the credential provider. The **`GetClaimsForUser`** method is the only method that a given protocol implementation must call. This method returns the mapped claims for the current user, given a protocol id and a connection id. The **`UserName`** property holds the username of the user, if the person that makes the request is currently logged in (indicated by the **`IsLoggedIn`** utility function). Finally, the **`ShouldTrace`** and **`Trace`** functions from the **`TraceService`**, and their log counterparts are being exposed here.

4.11.2 EndpointService

In general, when you request a URI, such as **`http://www.example.com/index.html`**, the last part of the URI, in this case **`index.html`**, corresponds to a file on the web server. On the IdP, however, no physical files are requested. Instead, what is requested are virtual endpoints that map to some implementation that knows how to respond. This makes it easier to upload new plug-ins, because a plug-in can be only a single file, namely the dll containing its implementation. Therefore, when the IdP receives a request, it must be able to find out which plug-in should handle the request. This is done though the **`EndpointService`**.

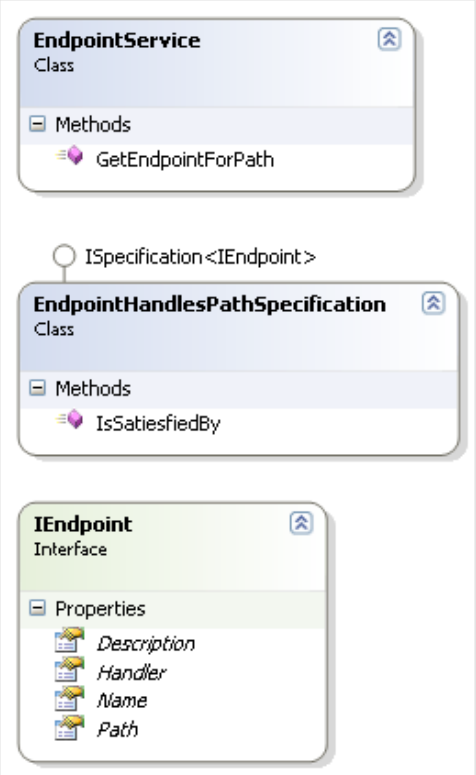


Figure 47. Class view: EndpointService

Figure 47 shows the **`EndpointService`** class and related classes. The **`EndpointService`** defines a single method, **`GetEndpointForPath`** which has a single argument, a path, and which returns an instance of **`IEndpoint`** that knows how to respond to requests for that path. The **`GetEndpointForPath`** method internally uses the **`ConfiguredProtocolRepository`** and **`ConfiguredCredentialProviderRepos`** to find all endpoints. It then uses the **`EndpointHandlesPathSpecification`** to determine which endpoint can handle the path.

4.12. Reporting of events

During runtime, the IdP must report events in a way that administrators of the system can see what is going on in terms of unexpected errors, configuration errors, but also in terms of actions taken by the users and the administrator or administrators themselves. Reporting of these events will be split into two different concepts; logging and tracing. Both log entries and

trace entries must be saved in the database because of the nature of the IdP being a software-as-a-service application. In such an application, the administrator will not have access to files on the machine where the application runs, and therefore writing these entries to a file on the disk is not as good an option as writing them to a table in a database. Having this information in a database enables rich query capabilities that are not offered by simple text files, something that will come in handy when needing to display the data in a graphical web user-interface.

4.12.1 Logging

As mentioned above, logging has to do with events that happen in the system as part of normal use. Logging such events will allow administrators of the system who has made changes to the system and when. Furthermore it allows him to see information about user activity. This is, for example, useful for corporations who need to comply to the Danish standard for information security (DS-484). A single unit of logging data will be called a ***log entry*** and its corresponding class will be called **`LogEntry`**. A log entry is always coupled to a ***logging source***. A logging source is used to define parts of the system that write log entries. This is useful for grouping log entries, and it can also be used to turn logging on and off for different parts of the IdP. the IdP will itself define a set of logging sources, but logging sources can also be defined for plug-ins as we shall see below in Section 4.2.1. The **`LogEntry`** and **`LoggingSource`** classes contains the following fields shown in Figure 48.

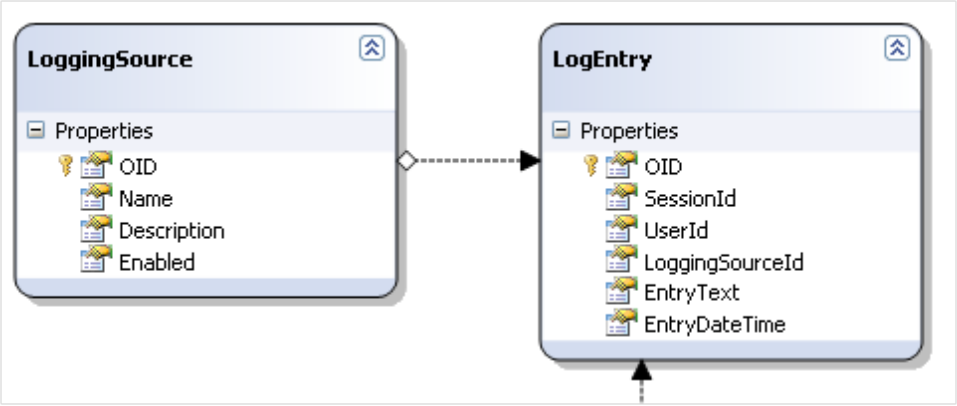


Figure 48. Schema view: LogEntry and LoggingSource

As the Figure 48 shows, each log entry contains a timestamp of when the log entry was created (**`EntryDateTime`**) and some message (**`EntryText`**). It also contains the id of the logging source it belongs to. Note that the relationship between the two is denoted by an arrow. You may have noticed another arrow pointing to the **`LogEntry`** class. That arrow represents the relationship to the **`User`** class, incurred by the **`UserId`** property. Finally, a log entry also contains a **`SessionId`**. A session is a well-known concept when working with web applications, and since the IdP is a web application it makes sense to log the session id, such that all log entries for a given session can be found. A logging source also has a unique id, a name and a description. Finally, it also has an **`Enabled`** flag, that is used to determine if logging for a given source is turned on or off. We have now defined what kind of data a log entry contains, but it is by no means obvious how it is to be used. In fact, how would a user of the class know what unique id to assign to the log entry, what the id of the current user is, or even what the unique id of the logging source is. We definitely need some way of facilitating logging. What we probably want is some kind of service class with a function that takes two parameters; the name of a logging source and some text. The **`LoggingService`** class would then have the responsibility of creating a new instance of a **`LogEntry`**, and fill in the missing parameters. But logging often entails building large text messages, which takes both memory and time. So what if logging has been turned off for the current logging source? In this case, the caller might decide not to build the message at all. Therefore our logging service needs to have a function to help the caller determine if logging is turned on for a given logging source. Figure 49 shows the **`LoggingService`** class.

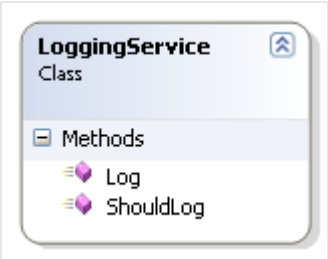


Figure 49. Class view: Logging service class

The **`ShouldLog`** function takes a single parameter, namely the name of a logging source, and returns true if logging is enabled for that logging source. In order to perform the correct logic, the **`LoggingService`** needs to interact

with other components of the system. Figure 50 shows which other components the **`LoggingService`** class interacts with. Note that the ***bold italic*** text over or below each component denotes what type of object it is in terms of domain-driven design.

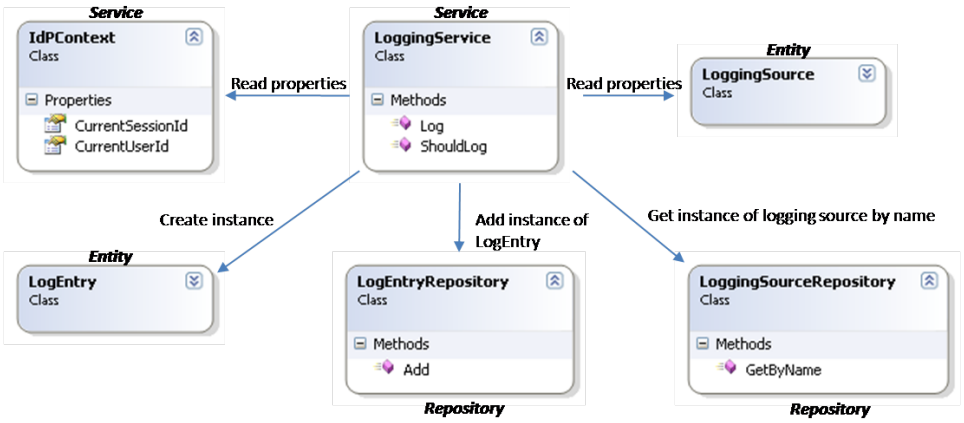


Figure 50. Class view: Logging service interactions

Lets start by considering the most simple of the two functions, namely the **`ShouldLog`** function. As mentioned this function takes a single parameter, the name of the logging source. When called, the **`ShouldLog`** function calls the **`FindByName`** function on the **`LoggingSourceRepository`**, which returns the corresponding instance of the **`LoggingSource`** class. The function can now check the **`Enabled`** property of the **`LoggingSource`** instance to determine what it should return. The **`Log`** function takes two parameters, namely a logging source name and a message. First of all, the function calls **`ShouldLog`** to determine if logging is turned on for the current logging source. If this is not the case, the function can return. This also means that callers, do not have to call the **`ShouldLog`** function, but that they may do so, if they wish. If the function determines that logging is enabled, starts by creating a new instance of the **`LogEntry`** class and fills in the **`LogEntryText`**, and the current time. It calls the **`LoggingSourceRepository`** to get the correct instance of the **`LoggingSource`** class from where it can retrieve the **`LoggingSourceId`**. Finally it reads the **`SessionId`**, and **`UserId`** properties the **`IdPContext`** class. The **`IdPContext`** class will be explained later, but for now just accept that it knows the current user's id and the session id. Finally, the function calls the **`Add`** function on the **`LogEntryRepository`** which takes an instance of a **`LogEntry`** as a parameter.

4.12.2 Tracing

Tracing has to with recording events that are related to errors and debugging the system. We define 4 levels of error tracing; critical, error, warning and information. A single unit of tracing data will be called a trace entry, and have a corresponding class called **`TraceEntry`**. Trace entries need to be written to the database for much the same reason that log entries needed to be written to the database. However, there is a small difference here. Since tracing is used to report errors, any trace entries with trace level critical will also be written to a file on disk. This is especially helpful when dealing with errors that have to do with missing database connectivity, but the file would of course only be accessible to maintainers of the entire software-as-a-service solution. It must be possible to turn tracing on and off, especially for warning and information level trace entries. It will also be possible to turn off errors, but it will however not be possible to turn off tracing of critical errors. Data regarding whether or not tracing is turned on for different levels is encapsulated in a concept called a trace profile. As before we create a service to facilitate adding trace entries. The **`TracingService`** and the classes it interacts with are shown in Figure 51.

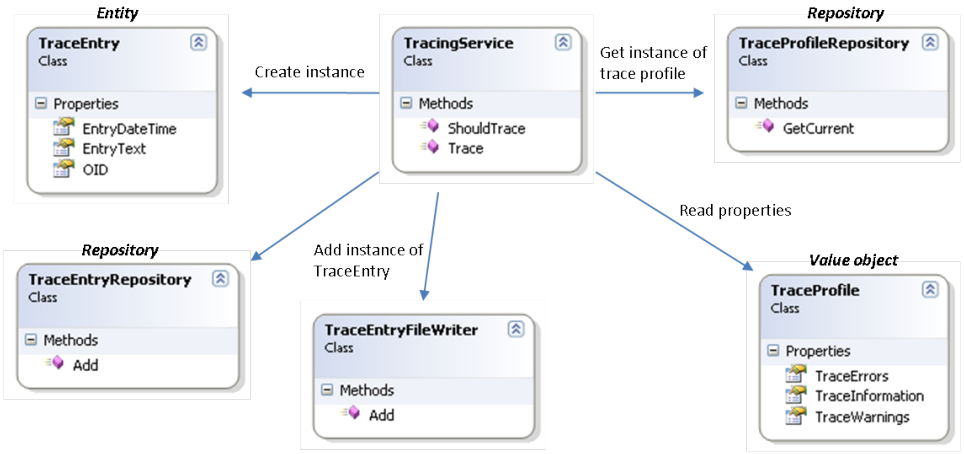


Figure 51. Class view: Tracing service interactions

Figure 51 shows that adding trace entries is in many ways similar to adding log entries. The most important things to notice is that the **`TraceProfile`** is a value object and not an entity objects. This is because a tracing profile does not have an identity, and any two instances of the class can be considered to be the same if their values are identical. Secondly, the **`TraceEntryFileWriter`**

is used to write trace entries to a file on disk. The `Trace` function of the `TracingService` takes as parameters a message and a trace level, and invokes the `Add` function on the `TraceEntryRepository` everytime, and the `Add` function on the `TraceEntryFileWriter` whenever the trace level is set to critical.

4.13. Ubiquitous language

The model described in this chapter has left us with a broad ubiquitous language, and can be summarized as follows: ***The IdP*** is an application that binds ***Claims*** to ***Users***. The IdP is extensible through ***plug-ins***. There are two kinds of plug-ins, ***credential provider*** plug-ins and ***protocol*** plug-ins. A credential provider plug-in can authenticate a user, and the result of an authentication is a set of ***Credential claims***. A protocol plug-in sends information about a user and his claims to some other party, called a ***service provider***. Both protocols and credential providers may rely on ***connections*** to describe how they communicate with other systems. When a protocol sends information about a user to a service provider, it sends a set of ***issued claims***. An issued claim originates from either a credential claim, or an ***identity provider claim***, which is a claim defined in the IdP. The set of issued claims that are sent may have undergone a ***claim mapping***. A claim mapping is a mechanism that alters any aspect of a set claim, except their value. There are two different types of claim mappings, namely ***protocol claim mappings*** and ***connection claim mappings***. The two are conceptually identical but the protocol claim mapping is always performed first. Users are created in the IdP through ***self registration***, ***import***, or ***invitation***. Every aspect of the IdP can be configured through ***configuration sets***, which contain different ***configuration elements***. When something in the IdP fails, a ***trace entry*** is created, and when an important event occurs, a ***log entry*** is created.

5 Object relational mappers

Most modern IT systems rely on databases to store application data, and the database schema used is indeed an important part of the domain model. As we saw in Chapter 3, the definition of a supple model is a model that can easily be changed. However, when using a traditional data access layer, changing the database schema would potentially require many changes in the data access classes and domain objects. Let consider the following simple example, where we have a **User** table, a **User** domain class, a **UserRepository** repository class and a **UserFactory** factory class. The table definition might look something like this:

User table

```
-- User table
CREATE TABLE [idp].[User](
        OID INT IDENTITY(1,1) NOT NULL,
        UserName NVARCHAR(128) NOT NULL,
        DateCreated DateTime NOT NULL DEFAULT GetDate(),
        [Password] NVARCHAR(128) NOT NULL,
        PasswordSalt NVARCHAR(128) NOT NULL,
        CONSTRAINT [PK_User] PRIMARY KEY CLUSTERED ([OID])
)
```

A **User** domain class could look like this:

User domain class

```
using System;

namespace IdP
{
    public class User
    {
        public int OID { get; set;}
        public string UserName { get; set; }
        public string Password { get; set; }
        public string PasswordSalt { get; set; }
        public DateTime DateCreated { get; set;}

        public User(int oid, string userName, DateTime dateCreated, string password, string passwordSalt)
        {
            OID = oid;
            UserName = userName;
            DateCreated = dateCreated;
            Password = password;
            PasswordSalt = passwordSalt;
        }
```

```
    }
}
```

A repository with methods to find a given user, either by name or by id, could look like the following:

User repository class

```
using System.Data.SqlClient;

namespace IdP
{
    public class UserRepository
    {
        public User GetUserById(int oid)
        {
            string sql = "select OID, UserName, Password, PasswordSalt, DateCreated from User
    ⇝   where OID = @oid";
            using (SqlConnection conn = new SqlConnection("a valid connection string"))
            {
                conn.Open();
                SqlCommand cmd = new SqlCommand(sql, conn);
                cmd.Parameters.AddWithValue("oid", oid);
                SqlDataReader reader = cmd.ExecuteReader();

                return UserFactory.FromReader(reader);
            }
        }

        public User GetUserByName(string userName)
        {
            string sql = "select OID, UserName, Password, PasswordSalt, DateCreated from User
    ⇝   where UserName = @UserName";
            using (SqlConnection conn = new SqlConnection("a valid connection string"))
            {
                conn.Open();
                SqlCommand cmd = new SqlCommand(sql, conn);
                cmd.Parameters.AddWithValue("UserName", userName);
                SqlDataReader reader = cmd.ExecuteReader();

                return UserFactory.FromReader(reader);
            }
        }
    }
}
```

And finally a factory which that is able to translate a record of a **SqlDataReader** to an instance of our **User** domain class, could look like this:

User factory class

```
using System;
using System.Data.SqlClient;

namespace IdP
{
    public class UserFactory
```

```
7       {
8           public static User FromReader(SqlDataReader reader)
9           {
10              if(reader.Read())
11              {
12                  int oid = (int) reader["OID"];
13                  string userName = reader["UserName"].ToString();
14                  string password = reader["Password"].ToString();
15                  string passwordSalt = reader["PasswordSalt"].ToString();
16                  DateTime dateCreated = (DateTime) reader["DateCreated"];
17
18                  return new User(oid, userName, dateCreated, password, passwordSalt);
19              }
20
21              return null;
22          }
23
24      }
25  }
```

The preceding classes show how we could implement data access for a the **User** table using the building blocks of domain-driven design. However, what if we were to add another field to the **User** table? Doing so would require updating the table, adding another field to the **User** domain class, updating the two select statements and adding another line of code to the factory method. Needless to say, if we had a large database schema, using this approach would not yield an especially supple design, since even small logical model changes would require quite a few changes to the code. In this chapter we shall see how we can leverage the power of object relation mappers to create a more supple and maintainable design. Object relational mappers are software tools that automatically map the tables of a database schema to classes in a programming language, complete with queries and update and delete methods. For .Net, several alternatives exist, the two most feature complete being NHibernate and LINQ to SQL. LINQ to SQL is part of the .Net 3.5 framework and has been developed in close cooperation with the SQL Server team for high performance when used with a SQL Server database. Since the specification requires me to use a SQL Server database, and I'm already using .Net framework 3.5, using LINQ to SQL seems like a natural choice.

5.1. LINQ

LINQ stands for language-integrated query and provides a set of standard query operators that can be used directly in any .Net language. Query expressions in LINQ benefit from compile time syntax checking, static typing and intellisense[9]. The standard query operators apply to **IEnumerable<T>** which is the interface implemented by every enumerator (array and specialized collections) in .Net over a generic type **T**. The following code sample shows the syntax for a standard LINQ query.

LINQ example

```
string[] names = { "Burke", "Connor", "Frank", "Everett", "Albert", "George", "Harris", "David" };

IEnumerable<string> query = from s in names
                            where s.Length == 5
                            orderby s
                            select s.ToUpper();

foreach (string item in query)
    Console.WriteLine(item);

```

There are several interesting things to note about the above code. First of all, the reverse syntax of the query, having the **from** keyword in the beginning of the expression. This has been necessary to make intellisense work. The most important thing to notice, however, is that the SQL like syntax in the example is merely syntactic sugar on top of a concept called extension methods. Extension methods are static methods that are declared outside of a the class they work on, and brought into scope by importing the namespace where they are declared. The following example shows how an extension method for the **User** class could be implemented.

User extension

```
namespace IdP
{
    public static class UserExtensions
    {
        public static int UserNameLength(this User u)
        {
            return u.UserName.Length;
        }
    }
}
```

[9] Intellisense is the auto-completion feature in the Visual Studio IDE

Note the special **this** keyword, that tells the compiler that the method applies to the type **User**. By importing the namespace where the extension method is defined, I can now call the **UserNameLength** function directly on the **User** class.

```
User u = UserRepository.GetByName("Klaus");
int len = u.UserNameLength();
```

So how does this apply to the LINQ query shown above? Well, as mentioned, the syntax of the above example is merely syntactic sugar, and actually translates to a set of extension methods on **IEnumerable<T>**. This means that the above query could be rewritten to this:

LINQ example 2

```
string[] names = { "Burke", "Connor", "Frank", "Everett", "Albert", "George", "Harris", "David" };

IEnumerable<string> query = names.Where(s => s.Length == 5).OrderBy(s => s).Select(s => s.ToUpper());

foreach (string item in query)
    Console.WriteLine(item);

Console.ReadLine();
```

The example shows how extension methods on **IEnumerable<T>** are used to express the query. Note that the result of applying a LINQ extension method to an **IEnumerable<T>** is itself an **IEnumerable<T>**, thus allowing us to keep applying extension methods as the example shows. The signature of the extension methods used is **this IEnumerable<T> source, Func<T, bool> predicate)**. The **Func** type is an expression from type **T** to **bool**, which is written as a lambda expression, such as **s => s.Length == 5** in the example. The full list of query operators defined by LINQ can be seen in Table 5.1.

The LINQ examples we have seen thus far have been good for illustratory purposes, however they have nothing to do with LINQ to SQL. As a matter of fact, LINQ to SQL is an extension of what we have seen so far, which is called LINQ to objects. Figure 52 shows the different flavours of LINQ that are offered by the framework.

Figure 52 shows that there are three different LINQ enabled data sources, LINQ to objects, LINQ to XML and the LINQ enabled ADO.Net. We have already seen how LINQ to objects works in the preceding examples. LINQ to XML provides language integrated query for XML documents, and LINQ enables ADO.Net provides language integrated queries for relational

Restriction	Where
Projection	Select, SelectMany
Ordering	OrderBy, ThenBy, Reverse
Grouping	GroupBy
Quantifiers	Any, All, Contains
Partitioning	Take, Skip, TakeWhile, SkipWhile
Sets	Distinct, Union, Intersect, Except
Elements	First, FirstOrDefault, ElementAt
Aggregation	Count, LongCount, Sum, Min, Max, Average, Aggregate
Conversion	ToArray, ToList, ToDictionary, AsEnumerable, ToLookup, OfType, Cast
Element	First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault, ElementAt, ElementAtOrDefault, DefaultIfEmpty

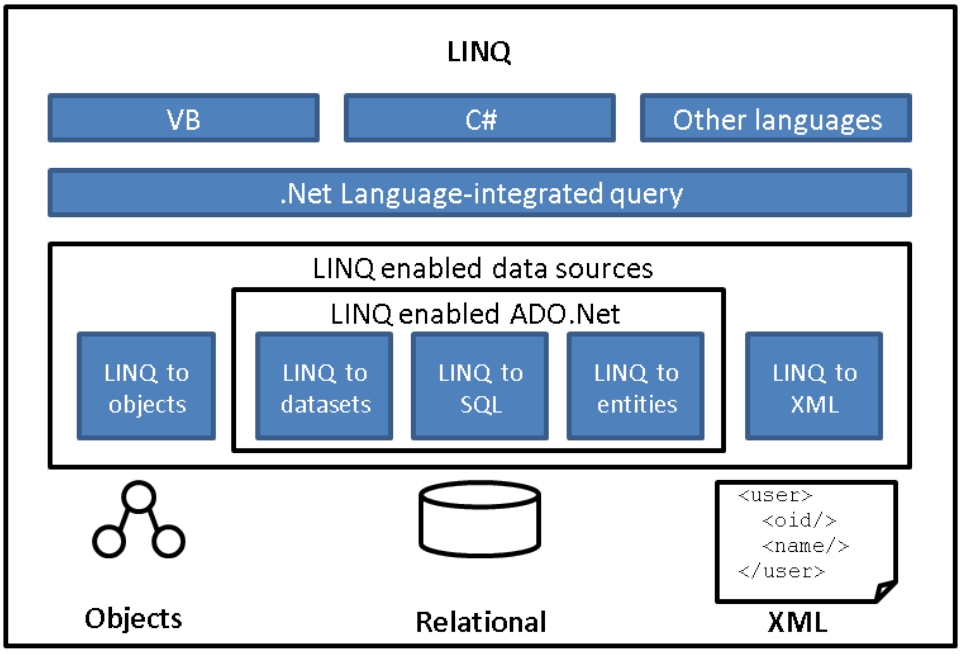


Figure 52. LINQ.

data stores. ADO.Net is the framework for manipulating relational data, and LINQ enabled ADO.Net consists of three different technologies, namely

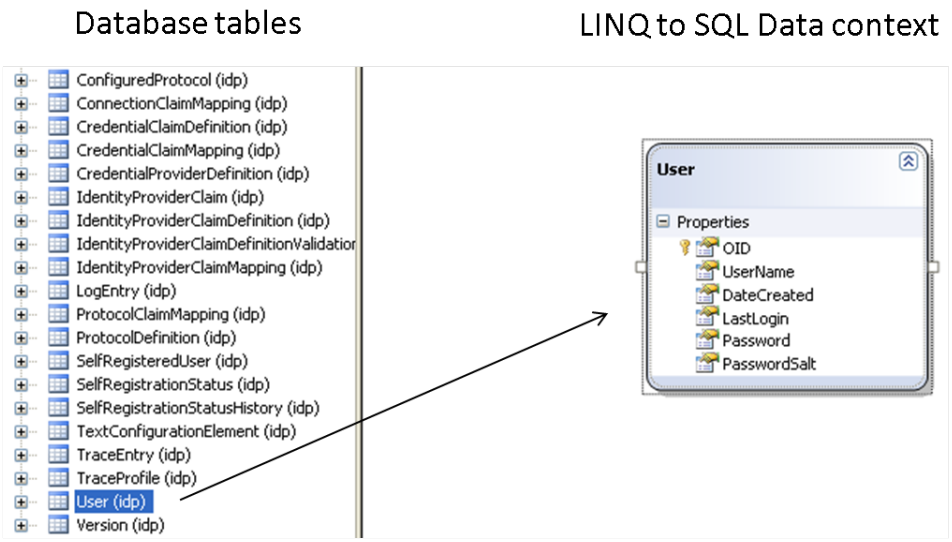


Figure 53. LINQ to SQL data context (designer view).

LINQ to datasets, LINQ to SQL and LINQ to entities. LINQ to datasets is an extension to the in memory dataset classes of the .Net framework, and LINQ to entities is the LINQ extension of the .Net "Entity framework", a framework for defining conceptual data schemas, called entity data models. LINQ to entities has still not proven to be mature, mainly because it lacks tool support in the IDE, but it is said that it will become the predecessor of LINQ to SQL.

5.2. LINQ to SQL

As with LINQ to objects, LINQ to SQL also supports compile time type checking. This is achieved through the what is called a "data context". A data context is a class that has several responsibilities. First of all, the data context contains information about which database to connect to. Furthermore, it contains definitions for the classes that represent the tables in the database, and finally it tracks in memory changes made to the instances of those classes.

Using the IDE we can create a new data context, and drag the `User` table onto the design surface, as shown in Figure 53.

We have now seen that we can easily drag tables onto our data context to generate the corresponding class. Now lets see what kind of code is generated behind the scenes.

LINQ data context

```
#pragma warning disable 1591
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.1433
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace IdP
{
        using System.Data.Linq;
        using System.Data.Linq.Mapping;
        using System.Data;
        using System.Collections.Generic;
        using System.Reflection;
        using System.Linq;
        using System.Linq.Expressions;
        using System.ComponentModel;
        using System;


        [System.Data.Linq.Mapping.DatabaseAttribute(Name="IdPDatabase")]
        public partial class IdPDataClassesDataContext : System.Data.Linq.DataContext
        {

                private static System.Data.Linq.Mapping.MappingSource mappingSource = new ⤳
    ⤳  AttributeMappingSource();

    #region Extensibility Method Definitions
    partial void OnCreated();
    partial void InsertUser(User instance);
    partial void UpdateUser(User instance);
    partial void DeleteUser(User instance);
    #endregion

                public IdPDataClassesDataContext() :
                                base(global::IdPTestApp.Properties.Settings.Default.⤳
    ⤳  SafewhereConnectionString, mappingSource)
                {
                        OnCreated();
                }

                public IdPDataClassesDataContext(string connection) :
                                base(connection, mappingSource)
                {
                        OnCreated();
                }

                public IdPDataClassesDataContext(System.Data.IDbConnection connection) :
```

```
51                                  base(connection, mappingSource)
52              {
53                      OnCreated();
54              }
55
56              public IdPDataClassesDataContext(string connection, System.Data.Linq.Mapping.⤳
    ⤳  MappingSource mappingSource) :
57                                  base(connection, mappingSource)
58              {
59                      OnCreated();
60              }
61
62              public IdPDataClassesDataContext(System.Data.IDbConnection connection, System⤳
    ⤳  .Data.Linq.Mapping.MappingSource mappingSource) :
63                                  base(connection, mappingSource)
64              {
65                      OnCreated();
66              }
67
68              public System.Data.Linq.Table<User> Users
69              {
70                      get
71                      {
72                              return this.GetTable<User>();
73                      }
74              }
75      }
76
77      [Table(Name="idp.[User]")]
78      public partial class User : INotifyPropertyChanging, INotifyPropertyChanged
79      {
80
81              private static PropertyChangingEventArgs emptyChangingEventArgs = new ⤳
    ⤳  PropertyChangingEventArgs(String.Empty);
82
83              private int _OID;
84
85              private string _UserName;
86
87              private System.DateTime _DateCreated;
88
89              private System.Nullable<System.DateTime> _LastLogin;
90
91              private string _Password;
92
93              private string _PasswordSalt;
94
95      #region Extensibility Method Definitions
96      partial void OnLoaded();
97      partial void OnValidate(System.Data.Linq.ChangeAction action);
98      partial void OnCreated();
99      partial void OnOIDChanging(int value);
100     partial void OnOIDChanged();
101     partial void OnUserNameChanging(string value);
102     partial void OnUserNameChanged();
103     partial void OnDateCreatedChanging(System.DateTime value);
104     partial void OnDateCreatedChanged();
```

```
105     partial void OnLastLoginChanging(System.Nullable<System.DateTime> value);
106     partial void OnLastLoginChanged();
107     partial void OnPasswordChanging(string value);
108     partial void OnPasswordChanged();
109     partial void OnPasswordSaltChanging(string value);
110     partial void OnPasswordSaltChanged();
111     #endregion
112
113                 public User()
114                 {
115                         OnCreated();
116                 }
117
118                 [Column(Storage="_OID", AutoSync=AutoSync.OnInsert, DbType="Int NOT NULL ⇝
     ⇝  IDENTITY", IsPrimaryKey=true, IsDbGenerated=true)]
119                 public int OID
120                 {
121                         get
122                         {
123                                 return this._OID;
124                         }
125                         set
126                         {
127                                 if ((this._OID != value))
128                                 {
129                                         this.OnOIDChanging(value);
130                                         this.SendPropertyChanging();
131                                         this._OID = value;
132                                         this.SendPropertyChanged("OID");
133                                         this.OnOIDChanged();
134                                 }
135                         }
136                 }
137
138                 [Column(Storage="_UserName", DbType="NVarChar(128) NOT NULL", CanBeNull=false⇝
     ⇝  )]
139                 public string UserName
140                 {
141                         get
142                         {
143                                 return this._UserName;
144                         }
145                         set
146                         {
147                                 if ((this._UserName != value))
148                                 {
149                                         this.OnUserNameChanging(value);
150                                         this.SendPropertyChanging();
151                                         this._UserName = value;
152                                         this.SendPropertyChanged("UserName");
153                                         this.OnUserNameChanged();
154                                 }
155                         }
156                 }
157
158                 [Column(Storage="_DateCreated", DbType="DateTime NOT NULL")]
159                 public System.DateTime DateCreated
```

```
160             {
161                     get
162                     {
163                             return this._DateCreated;
164                     }
165                     set
166                     {
167                             if ((this._DateCreated != value))
168                             {
169                                     this.OnDateCreatedChanging(value);
170                                     this.SendPropertyChanging();
171                                     this._DateCreated = value;
172                                     this.SendPropertyChanged("DateCreated");
173                                     this.OnDateCreatedChanged();
174                             }
175                     }
176             }
177
178             [Column(Storage="_LastLogin", DbType="DateTime")]
179             public System.Nullable<System.DateTime> LastLogin
180             {
181                     get
182                     {
183                             return this._LastLogin;
184                     }
185                     set
186                     {
187                             if ((this._LastLogin != value))
188                             {
189                                     this.OnLastLoginChanging(value);
190                                     this.SendPropertyChanging();
191                                     this._LastLogin = value;
192                                     this.SendPropertyChanged("LastLogin");
193                                     this.OnLastLoginChanged();
194                             }
195                     }
196             }
197
198             [Column(Storage="_Password", DbType="NVarChar(128) NOT NULL", CanBeNull=false
    ⇝ )]
199             public string Password
200             {
201                     get
202                     {
203                             return this._Password;
204                     }
205                     set
206                     {
207                             if ((this._Password != value))
208                             {
209                                     this.OnPasswordChanging(value);
210                                     this.SendPropertyChanging();
211                                     this._Password = value;
212                                     this.SendPropertyChanged("Password");
213                                     this.OnPasswordChanged();
214                             }
```

```
215                         }
216                 }
217
218                 [Column(Storage="_PasswordSalt", DbType="NVarChar(128) NOT NULL", CanBeNull=⇝
    ⇝  false)]
219                 public string PasswordSalt
220                 {
221                         get
222                         {
223                                 return this._PasswordSalt;
224                         }
225                         set
226                         {
227                                 if ((this._PasswordSalt != value))
228                                 {
229                                         this.OnPasswordSaltChanging(value);
230                                         this.SendPropertyChanging();
231                                         this._PasswordSalt = value;
232                                         this.SendPropertyChanged("PasswordSalt");
233                                         this.OnPasswordSaltChanged();
234                                 }
235                         }
236                 }
237
238                 public event PropertyChangingEventHandler PropertyChanging;
239
240                 public event PropertyChangedEventHandler PropertyChanged;
241
242                 protected virtual void SendPropertyChanging()
243                 {
244                         if ((this.PropertyChanging != null))
245                         {
246                                 this.PropertyChanging(this, emptyChangingEventArgs);
247                         }
248                 }
249
250                 protected virtual void SendPropertyChanged(String propertyName)
251                 {
252                         if ((this.PropertyChanged != null))
253                         {
254                                 this.PropertyChanged(this, new PropertyChangedEventArgs(⇝
    ⇝  propertyName));
255                         }
256                 }
257         }
258 }
259 #pragma warning restore 1591
```

LINQ data context

The main data context class begins on line 25. It is decorated with a **DatabaseAttribute** which defines the name of the database it connects to. Also note that it is declared as **partial**, which means that it can extended. Partial classes are especially useful when one part of the partial class is autogenerated by a tool, because it means that we can define other meth-

ods and fields of that same class in an other file, which will not be overwritten if we run the code generation tool again. The class inherits the **`System.Data.Linq.DataContext`** class, which contains most of the logic for tracking object changes, connecting to the database, committing changes etc.

On lines 32-35 we find a set of extensibility methods, again declared **`partial`**. These methods allow us to implement extra logic whenever **`User`** objects are inserted, updated or deleted.

Lines 38-66 contain various overloaded constructors. On line 68 we can see a public property called **`Users`**. This is the field that we can use to perform LINQ queries, as we shall see in an example shortly. The return type of the property is **`Table<User>`**. The generic **`Table<T>`** type is an integral part of the LINQ to SQL framework and implements the translation from LINQ expressions to actual SQL statements. The **`Table`** class implements the **`IEnumerable<T>`** interface and can thus make use of the extension methods mentioned earlier. However, even though they can be used, the LINQ to objects extension methods should not be used with **`Table<T>`** classes. This is because the LINQ to objects extension methods are executed immediately on the underlying **`IEnumerable<T>`** instance. If you remember the example from page 92, we applied three extension methods to an array. If we were to do this with the data from a SQL database table, we would effectively fetch all rows into memory and do the selection logic on the in memory data representation. This would potentially cripple the application's performance, and in general it would be a complete misuse of the rdbms's capabilities. This problem has of course been overcome by the designers of LINQ to SQL. The way it has been done, is that the **`Table<T>`** class implements another interface, called **`IQueryable<T>`**, for which all the same extension methods have been defined, but with another internal implementation, of course. In LINQ to SQL, any query on a **`Table<T>`** is not executed until you iterate over the result. This behaviour allows us to compose a complex query of several less complex parts without executing the actual query before we iterate over the result. When we start iterating over the result, a SQL query is automatically generated behind the scenes and executed against the database.

On line 77 we have the definition for the **`User`** class. The **`User`** class has a **`Table`** attribute that includes information about the actual table name in the database. Furthermore, the **`User`** class implements the **`INotifyPropertyChanging`** and **`INotifyPropertyChanged`** interfaces, which allow LINQ to SQL to track

in-memory data-changes. Apart from that, the **User** class has all the expected properties, such as **OID**, **UserName** etc., and all these properties are decorated with attributes that tell something about the database column name, database column type etc. The example below shows how we can use the data context to query the user table.

LINQ to SQL example

```
using(var ctx = new IdPDataClassesDataContext())
{
    //Find users with a UserName of length 5

    var result = from u in ctx.Users
                 where u.UserName.Length == 5
                 orderby u.OID
                 select u;

    //or

    var result2 = ctx.Users.Where(u => u.UserName.Length == 5).OrderBy(u => u.OID).Select(u ⇝
     ⇝  => u);

    //we can further refine the result by adding an extra where clause specifying that OID ⇝
     ⇝  must be higher than 100
    var result3 = result2.Where(u => u.OID > 100).Select(u => u);

    //The query is not performed against the database before we iterate over the result
    foreach(var user in result3)
    {
        Console.WriteLine(user.UserName);
    }
}
```

Note that the **var** keyword is a shorthand that can be used where the compiler can infer the type by looking at the type of what is assigned to variable. In this case, all three variables declared using the **var** keyword are actually of type **IQueryable<User>**.

A very nice feature of LINQ to SQL is its ability to translate generic expressions to SQL statements. We can leverage this ability to declare our domain-driven designspecifications as LINQ expressions, thus enabling us to have specifications that can be used both in code and in the database. In Chapter 3 we saw an example of a specification called **RecentlyCreated**, which contained the definition for a recently created user. We could extend the **User** class generated by the data context to contain a method that could tell us if a user was recently created:

Extending the User class

```
using System;

namespace IdPTestApp
```

```
{
    public partial class User
    {
        public static Func<User, bool> spec = (user => user.DateCreated > DateTime.Now.⇝
     ⇝  AddDays(-10));

        public bool IsRecentlyCreated()
        {
            return spec.Invoke(this);
        }
    }
}
```

Note the static variable called **spec**. This variable can be used in a LINQ to SQL query too. So if we want to find all recently created users in the database, we can reuse that same LINQ expression, as shown in the following example.

Using specification to query the database

```
using (var ctx = new IdPDataClassesDataContext())
{
    //Find recently created users

    var result = ctx.Users.Where(User.spec).Select(u => u);

    foreach (var user in result)
    {
        Console.WriteLine(user.UserName);
    }
}
```

In this elegant way, we can ensure that our domain logic is only defined in one single place, thus making maintenance a lot easier.

6
A domain-neutral component

In Chapter 5 we saw how LINQ to SQL works, and in this chapter we shall explore how we can leverage the capabilities of LINQ to SQL to create a domain-neutral component that can support development using the domain-driven design concepts.

In Chapter 3 we saw that repositories should be used to store and retrieve entity objects, thus creating the illusion that these objects are all in memory, when in reality they are being retrieved from some external storage, such as a relational database. Given its long life-span, an entity object always has an identity, and in a relational database, this identity usually translates to a primary key. We can define an interface for entity objects as shown below.

IEntity.cs

```
1 namespace Safewhere.Core.Domain
2 {
3     public interface IEntity<TPrimaryKey>
4     {
5         TPrimaryKey OID { get; }
6     }
7 }
```

The **IEntity** interface shown above is a generic interface with a type parameter, **TPrimaryKey**, which specifies the type of the primary key. The interface specifies that every object that implements the interface must have a readonly property called **OID** (acronym for object id).

We will sometimes need to manipulate several different entity objects from different repositories within a transaction. The word transaction is well-known from the world of relational databases. A transaction is an encapsulation of a series of operations that are co-dependent, meaning that if one of the operations fail, the result of any of the operations should not be stored. In the domain-driven design community, and in [McCarthy 2008] and [Harding 2008] in particular, this transaction concept is referred to as a "unit of work". I presume that this term has been chosen instead of "transaction" because of the fact that the word transaction is so tightly coupled to a specific storage technology, namely relational databases. It is, of course, not the responsibility of a repository to know when it is used within a unit

of work. But a repository must have the ability to be enrolled in a unit of work. The interface for an abstract unit of work could be defined as follows:

IUnitOfWork.cs

```
using System;

namespace Safewhere.Core.Domain
{
    public interface IUnitOfWork : IDisposable
    {
        T Create<T>() where T : IUnitOfWorkElement, new();

        void Complete();
    }
}
```

The **IUnitOfWork** interface defines two functions, **Create** and **Complete**. The interface also inherits the **IDisposable** interface, an interface that can be implemented to release unmanaged resources during garbage collection when working in .Net. LINQ to SQL resources are not unmanaged , but should we wish to use some other data storage in the future, including the **IDisposable** interface is a safe choice. The **Complete** method is used to complete the unit of work, an equivalent to committing when working with relational databases. The **Create** function is a generic function that will create an instance of any class that implements the **IUnitOfWorkElement** interface and has a default constructor (eg. a constructor that takes no arguments. This is denoted by the **new()** restriction on the generic type **T**). The **IUnitOfWorkElement** interface has the following definition:

IUnitOfWorkElement.cs

```
namespace Safewhere.Core.Domain
{
    public interface IUnitOfWorkElement
    {
        IUnitOfWork UnitOfWork { set; }
    }
}
```

The **IUnitOfWorkElement** interface defines a write-only property called **UnitOfWork** of type **IUnitOfWork**. Any implementor of the **IUnitOfWorkElement** interface will thus expose a way of setting the associated **IUnitOfWork** instance, and will be used by the **IUnitOfWork**'s **Create** method, as we shall see in the following concrete implementation of a unit of work:

LinqToSqlUnitOfWork.cs

```
using System;
using System.Data.Linq;
using System.Transactions;

```

```
5  namespace Safewhere.Core.Domain
6  {
7      public class LinqToSqlUnitOfWork : IUnitOfWork, IRepositoryImplementationHelper
8      {
9          DataContext _dataContext;
10         TransactionScope _txScope;
11
12         public LinqToSqlUnitOfWork(DataContext dataContext) : this(dataContext, null){}
13
14         public LinqToSqlUnitOfWork(DataContext dataContext, Transaction transactionToUse)
15         {
16             if (dataContext == null)
17                 throw new ArgumentNullException("dataContext");
18             _dataContext = dataContext;
19             _txScope = transactionToUse == null ? new TransactionScope() : new ⇝
       ⇝  TransactionScope(transactionToUse);
20         }
21
22         public T Create<T>() where T : IUnitOfWorkElement, new()
23         {
24             return new T {UnitOfWork = this};
25         }
26
27         public void Complete()
28         {
29             _dataContext.SubmitChanges();
30             _txScope.Complete();
31         }
32
33         public void Dispose()
34         {
35             _dataContext = null;
36             _txScope.Dispose();
37         }
38
39
40         #region IRepositoryImplementationHelper Members
41
42         IEntityContainer<T> IRepositoryImplementationHelper.GetEntityContainer<T>()
43         {
44             return new LinqToSqlEntityContainer<T>(_dataContext);
45         }
46
47         #endregion
48     }
49 }
```

The **LinqToSqlUnitOfWork** class has two constructors. The first one takes a LINQ **DataContext** as discussed in the previous chapter, and calls the other constructor with a **null** value for the **transactionToUse** parameter. The second constructor saves the **DataContext** in a private variable and sets the private **_txScope** variable based on wether or not the **transactionToUse** parameter is null. A **TransactionScope** is used to implicitly enlist a block of

code in a transactions in .Net, as described in [Microsoft 2007].

The **Create** method is implemented by calling the default constructor[10] of the type **T** and setting the unit of work instance to the current **LinqToSqlUnitOfWork** instance.

The **Complete** method is implemented by calling **SubmitChanges** on the private **_dataContext** object, and then calling **Complete** on the **TransactionScope** instance. The **SubmitChanges** method on a **DataContext** will make the **DataContext** persist the changes made in memory to the database.

Finally, the class implements the **IRepositoryImplementationHelper** interface by implementing the **GetEntityContainer** method. The **IRepositoryImplementationHelper** interface is defined as follows:

IRepositoryImplementationHelper.cs

```
1  namespace Safewhere.Core.Domain
2  {
3      public interface IRepositoryImplementationHelper
4      {
5          IEntityContainer<T> GetEntityContainer<T>() where T : class;
6      }
7  }
```

The **IRepositoryImplementationHelper** interface is going to be used by our repository implementation, and defines a single method called **GetEntityContainer**, which returns an **IEntityContainer** of a given class **T**. The Verb|IEntityContainer| has the following definition:

IEntityContainer.cs

```
1  using System.Linq;
2
3  namespace Safewhere.Core.Domain
4  {
5      public interface IEntityContainer<T> : IQueryable<T>
6      {
7          void Add(T element);
8          void Remove(T element);
9      }
10 }
```

The **IEntityContainer** interface is a generic interface that inherits from the generic built-in **IQueryable** interface. It defines the two methods **Add** and **Remove**, which should be used to add an remove entity objects from a

[10] Note the use of the object initializer syntax that is new in .Net 3.5, and allows you to assign values to public properties during the call to the constructor. See [Microsoft 2009] for an in-depth explanation.

container. The following class is an implementation of the **IEntityContainer** interface.

LinqToSqlEntityContainer.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.Linq;

namespace Safewhere.Core.Domain
{
    internal class LinqToSqlEntityContainer<T> : IEntityContainer<T> where T : class
    {
        Table<T> _table;

        internal LinqToSqlEntityContainer(DataContext dataContext)
        {
            _table = dataContext.GetTable<T>();
        }

        #region IEntityContainer<T> Members

        public void Add(T element)
        {
            _table.InsertOnSubmit(element);
        }

        public void Remove(T element)
        {
            _table.DeleteOnSubmit(element);
        }

        #endregion

        #region IEnumerable<T> Members

        public IEnumerator<T> GetEnumerator()
        {
            return _table.GetEnumerator();
        }

        #endregion

        #region IEnumerable Members

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
        {
            return _table.GetEnumerator();
        }

        #endregion

        #region IQueryable Members

        public Type ElementType
        {
            get
```

```
54              {
55                  var q = _table as IQueryable<T>;
56                  return q.ElementType;
57              }
58          }
59
60          public System.Linq.Expressions.Expression Expression
61          {
62              get
63              {
64                  var q = _table as IQueryable<T>;
65                  return q.Expression;
66              }
67          }
68
69          public IQueryProvider Provider
70          {
71              get
72              {
73                  var q = _table as IQueryable<T>;
74                  return q.Provider;
75              }
76          }
77
78          #endregion
79      }
80  }
```

LinqToSqlEntityContainer.cs

The first thing to notice about the **LinqToSqlEntityContainer** is that it is marked **internal**. This means that it can only be used from within the assembly where it resides. The reason that this class is made **internal** is that it is not supposed to be used directly by any client, but only as an internal datastorage-specific container for entities. Since this entity container uses LINQ to SQL, its internal implementation is based on an instance of a **Table**, as described earlier, for the generic type **T**. The constructor takes a LINQ **DataContext** which is uses for retrieving the correct **Table** for the generic type **T**. It implements all the interface methods from **IEntityContainer** by calling the suitable methods on the **Table** instance **_table**.

We have now seen the definition of most of the classes needed for our generic repository implementation, and we can now define the interface for the repository as follows:

IRepository.cs

```
1  using System;
2  using System.Linq;
3  using System.Linq.Expressions;
4
5  namespace Safewhere.Core.Domain
6  {
7      public interface IRepository<TEntity, TPrimaryKey>
```

```
8           : IQueryable<TEntity>, IUnitOfWorkElement
9           where TEntity : IEntity<TPrimaryKey>
10      {
11          void Add(TEntity element);
12          void Remove(TEntity element);
13
14          TEntity this[TPrimaryKey primaryKey] { get; }
15
16          IQueryable<TEntity> Find(Expression<Func<TEntity, bool>> expr);
17          IQueryable<TEntity> Find(IBooleanExpressionHolder<TEntity> spec);
18
19          TEntity FindFirst(Expression<Func<TEntity, bool>> expr);
20          TEntity FindFirst(IBooleanExpressionHolder<TEntity> spec);
21
22          IQueryable<TEntity> FindAll();
23
24          bool Exists(Expression<Func<TEntity, bool>> expr);
25          bool Exists(IBooleanExpressionHolder<TEntity> spec);
26
27      }
28  }
```

The **IRepository** interface is a generic interface of type **TEntity** and **TPrimaryKey**, where **TEntity** is the type of the entity object that the repository serves, and **TPrimaryKey** is the type of the primary key of that entity type. Furthermore, the interface imposes the restriction that the **TEntity** type must implement the **IEntity** interface with using the same type, **TPrimaryKey**, as a primary key. The repository must also implement the **IQueryable** interface for type **TEntity**, and the **IUnitOfWork** interface.

The **IRepository** interface defines **Add** and **Remove** functions for adding and removing entity objects. It also defines an indexer function (in line 14) that allows finding a single instance of an entity object based on its unique identifier. Lines 16-22 define different finder functions. The two **Find** functions return a collection of entity objects based on some search criterion. The first function takes a parameter of type **Expression<Func<TEntity, bool>>**, which is the lambda expression that was described earlier in this chapter. The second **Find** function takes an **IBooleanExpressionHolder**, an interface that we will use to encapsulate our specifications, as we shall see further down. The two **FindFirst** functions take the same parameters as the **Find** functions, but only return the first result instead of returning a collection. The **FindAll** function returns all the entity objects in the repository. Finally, the interface defines two **Exists** functions that determines if one or more entity objects exist for a given lambda expression or **IBooleanExpressionHolder**. The **IBooleanExpressionHolder** interface is defined as follows:

IBooleanExpressionHolder.cs

```
using System;
using System.Linq.Expressions;

namespace Safewhere.Core.Domain
{
    public interface IBooleanExpressionHolder<T>
    {
        Expression<Func<T, bool>> Expression { get; }
    }
}
```

The **IBooleanExpressionHolder** is a generic interface for a generic type **T** and contains a single property of type **Expression<Func<T, bool>>**. The purpose of this interface is to hold an expression that will evaluate to a Boolean value for an instance of a given generic type **T**. When used in conjunction with a repository, an implementation of **IBooleanExpressionHolder** can be used to express a search criterion for use with the **Find**, **FindFirst** or **Exists** functions. The most important class in the component that implements the **IBooleanExpressionHolder** interface is the **LambdaSpecification** class, that is defined as follows:

ISpecification.cs

```
using System;
using System.Linq.Expressions;

namespace Safewhere.Core.Domain
{
    public abstract class LambdaSpecification<T> : ISpecification<T>, ⇝
     ⇝  IBooleanExpressionHolder<T>
    {
        Expression<Func<T, bool>> _expr;
        Func<T, bool> _compiledExpr;

        protected LambdaSpecification(Expression<Func<T, bool>> specExpression)
        {
            _expr = specExpression;
        }

        public Expression<Func<T, bool>> Expression
        {
            get { return _expr; }
        }

        public Func<T, bool> CompiledExpression
        {
            get
            {
                if (_compiledExpr == null)
                {
                    _compiledExpr = _expr.Compile();
                }
                return _compiledExpr;
```

```
30              }
31          }
32
33          public bool IsSatiesfiedBy(T element)
34          {
35              return CompiledExpression.Invoke(element);
36          }
37      }
38  }
```

ISpecification.cs

The **LambdaSpecification** class is a generic abstract class that implements both **ISpecification** (described further down) and the **IBooleanExpressionHolder** interface. The class is abstract since, even though it could be possible, we do not want anyone to use instances of the class directly. This is because in domain-driven design a specification should be explicitly named and be a concept of it's own. The **LambdaSpecification** class does however implement most of the logic for creating those explicitly named specifications. The class contains two properties, **Expression** and **CompiledExpression**. The value of the **Expression** property is the same that the class gets in the constructor. The **CompiledExpression** is that same expression, only compiled internally for better performance. Lastly, the implementation of the **ISpecification** interface is implemented by the **IsSatisfiedBy** function. The method takes an argument of an instance of the class' generic type, and invokes the compiled expression on this instance. The **ISpecification** interface is defined as follows:

ISpecification.cs

```
1  namespace Safewhere.Core.Domain
2  {
3      public interface ISpecification<T>
4      {
5          bool IsSatiesfiedBy(T element);
6      }
7  }
```

The **ISpecification** interface is a generic interface that has a single function, **IsSatisfiedBy**, just like the example presented earlier. We can now define a repository in the following way:

Repository.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Linq.Expressions;
5
6  namespace Safewhere.Core.Domain
7  {
8      public class Repository<TEntity, TPrimaryKey>
9          : IRepository<TEntity, TPrimaryKey>
```

```
10          where TEntity : class, IEntity<TPrimaryKey>
11      {
12          protected IUnitOfWork _unitOfWork;
13          protected IEntityContainer<TEntity> _entityContainer;
14
15          protected IEntityContainer<T> CreateEntityContainer<T>() where T: class
16          {
17              var repHelper = (IRepositoryImplementationHelper) UnitOfWork ;
18              return repHelper.GetEntityContainer<T>();
19          }
20
21          #region IRepository<TEntity, TPrimaryKey> Members
22
23          public void Add(TEntity element)
24          {
25              _entityContainer.Add(element);
26          }
27
28          public void Remove(TEntity element)
29          {
30              _entityContainer.Remove(element);
31          }
32
33          public TEntity this[TPrimaryKey primaryKey]
34          {
35              get {
36                  return _entityContainer.SingleOrDefault(entity => primaryKey.Equals(entity.⇝
    ⇝  OID));
37              }
38          }
39
40          public IQueryable<TEntity> Find(Expression<Func<TEntity, bool>> expr)
41          {
42              return _entityContainer.Where(expr);
43          }
44
45          public IQueryable<TEntity> Find(IBooleanExpressionHolder<TEntity> spec)
46          {
47              return _entityContainer.Where(spec.Expression);
48          }
49
50          public TEntity FindFirst(Expression<Func<TEntity, bool>> expr)
51          {
52              return _entityContainer.FirstOrDefault(expr);
53          }
54
55          public TEntity FindFirst(IBooleanExpressionHolder<TEntity> spec)
56          {
57              return _entityContainer.FirstOrDefault(spec.Expression);
58          }
59
60          public bool Exists(Expression<Func<TEntity, bool>> expr)
61          {
62              return _entityContainer.Count(expr) > 0;
63          }
64
```

```
65          public bool Exists(IBooleanExpressionHolder<TEntity> spec)
66          {
67              return _entityContainer.Count(spec.Expression) > 0;
68          }
69
70          public IQueryable<TEntity> FindAll()
71          {
72              return _entityContainer.AsQueryable();
73          }
74
75          #endregion
76
77          #region IEnumerable<TEntity> Members
78
79          public IEnumerator<TEntity> GetEnumerator()
80          {
81              return _entityContainer.GetEnumerator();
82          }
83
84          #endregion
85
86          #region IEnumerable Members
87
88          System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
89          {
90              return _entityContainer.GetEnumerator();
91          }
92
93          #endregion
94
95          #region IQueryable Members
96
97          public Type ElementType
98          {
99              get { return _entityContainer.ElementType; }
100         }
101
102         public Expression Expression
103         {
104             get { return _entityContainer.Expression; }
105         }
106
107         public IQueryProvider Provider
108         {
109             get { return _entityContainer.Provider; }
110         }
111
112         #endregion
113
114         #region IUnitOfWorkElement Members
115
116         public IUnitOfWork UnitOfWork
117         {
118             set
119             {
120                 _unitOfWork = value;
```

```
                _entityContainer = CreateEntityContainer<TEntity>();
            }

            get
            {
                return _unitOfWork;
            }
        }

        #endregion
    }
}
```

Repository.cs

The **Repository** class, of course, implements the **IRepository** interface presented earlier. The most important thing to notice is in the implementation of the **IUnitOfWorkElement** interface. When the **UnitOfWork** instance is set on the class, the private **_entityContainer** variable is instantiated by calling the **CreateEntityContainer** function. The **CreateEntityContainer** function in turn casts the **UnitOfWork** to a **IRepositoryImplementationHelper** which contains the **GetEntityContainer** function. The rest of the functions in this class use this **EntityContainer** internally to perform the work.

6.1. Preparing for test

As mentioned in Chapter 3, one important reason for using repositories is that a concrete repository implementation, for example one that uses an SQL server backend, can be interchanged with another one. This is very useful for those unit tests whose purpose is not to test the data storage system, but only to test the domain functionality. By adding a few extra classes to the domain neutral component, we can implement an in-memory data store. For this we need, amongst other things, an **InMemoryUnitOfWork** and an **InMemoryEntityContainer**.

InMemoryUnitOfWork.cs

```
using System;

namespace Safewhere.Core.Domain
{
    public class InMemoryUnitOfWork : IUnitOfWork, IRepositoryImplementationHelper
    {
        readonly IInMemoryDataContainer _container;

        public InMemoryUnitOfWork(IInMemoryDataContainer container)
        {
            if (container == null)
                throw new ArgumentNullException("container");
```

```
            _container = container;
        }

        #region IUnitOfWork Members

        public T Create<T>() where T : IUnitOfWorkElement, new()
        {
            return new T { UnitOfWork = this };
        }

        public void Complete()
        {
            //this method does nothing
        }

        #endregion

        #region IDisposable Members

        public void Dispose()
        {
            //this method does nothing
        }

        #endregion

        #region IRepositoryImplementationHelper Members

        public IEntityContainer<T> GetEntityContainer<T>() where T : class
        {
            return new InMemoryEntityContainer<T>(_container);
        }

        #endregion
    }
}
```

InMemoryUnitOfWork.cs

The **InMemoryUnitOfWork** class is a lot like its LINQ counterpart. The most important difference is the argument to its constructor, an instance of an **IInMemoryDataContainer**, instead of the **DataContext** instance that is used by the LINQ version. Furthermore, this class does not make use of the **TransactionScope** class. This is solely by decision, since it easily could, but because its purpose is to facilitate testing without an underlying database, the class can be kept simple. The **GetEntityContainer** function uses an in-memory version of an entity container.

InMemoryEntityContainer.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Safewhere.Core.Domain
{
```

```
7   internal class InMemoryEntityContainer<T> : IEntityContainer<T> where T : class
8   {
9       readonly List<T> _data;
10
11      internal InMemoryEntityContainer(IInMemoryDataContainer container)
12      {
13          _data = container.GetData<T>();
14      }
15
16      #region IEntityContainer<T> Members
17
18      public void Add(T element)
19      {
20          _data.Add(element);
21      }
22
23      public void Remove(T element)
24      {
25          _data.Remove(element);
26      }
27
28      #endregion
29
30      #region IEnumerable<T> Members
31
32      public IEnumerator<T> GetEnumerator()
33      {
34          return _data.GetEnumerator();
35      }
36
37      #endregion
38
39      #region IEnumerable Members
40
41      System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
42      {
43          return _data.GetEnumerator();
44      }
45
46      #endregion
47
48      #region IQueryable Members
49
50      public Type ElementType
51      {
52          get
53          {
54              IQueryable<T> q = _data.AsQueryable();
55              return q.ElementType;
56          }
57      }
58
59      public System.Linq.Expressions.Expression Expression
60      {
61          get
62          {
```

```
63                  IQueryable<T> q = _data.AsQueryable();
64                  return q.Expression;
65              }
66          }
67
68          public IQueryProvider Provider
69          {
70              get
71              {
72                  IQueryable<T> q = _data.AsQueryable();
73                  return q.Provider;
74              }
75          }
76
77          #endregion
78      }
79  }
```

InMemoryEntityContainer.cs

The implementation of the **InMemoryEntityContainer** class also resembles its LINQ counterpart, differing only in that its internal data-holder is a generic **List** (the **_data** variable). By using the LINQ to objects extensions defined in **System.Linq** the class can perform its logic on that list just in the same way as the LINQ counterpart does with its **Table** instance. The **IInMemoryDataContainer** provides the data, and is defined as follows:

IInMemoryDataContainer.cs

```
1  using System.Collections.Generic;
2
3  namespace Safewhere.Core.Domain
4  {
5      public interface IInMemoryDataContainer
6      {
7          List<T> GetData<T>();
8      }
9  }
```

The **IInMemoryDataContainer** defines a single generic function that provides data of the correct type.

7 Implementation

In the previous chapter we saw how we could use LinqToSql to create a domain-neutral framework to help us with the implementation of repositories, entity objects and specifications. In this chapter I will present the implementation of some of the classes described in Chapter 4. Since the entire implementation is too large to be presented here, I will focus on the parts that I find most interesting, of course including some repository implementations using the framework described in the preceding chapter. If I have left out some part that is of your particular interest, please feel free to look at the implementation of that part on the enclosed disk media. The enclosed disk media contains a Visual Studio 2008 solution file and all the corresponding files. Figure 54 shows an analysis of the solution file.

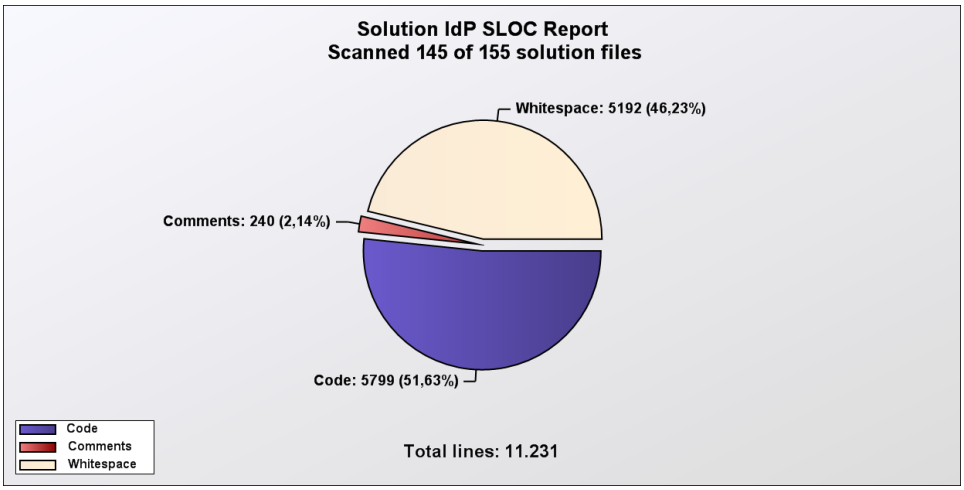


Figure 54. Source lines of code.

As you can see the solution contains 155 files, 10 of which do not contain source code. As you may have noticed the percentage of lines of comments is quite low, but this is natural since the system is not fully implemented, and therefore not fully documented either.

Figure 55 shows the structure of the solution. As you can see the solution

structure and file names adhere strictly to the ubiquitous language.

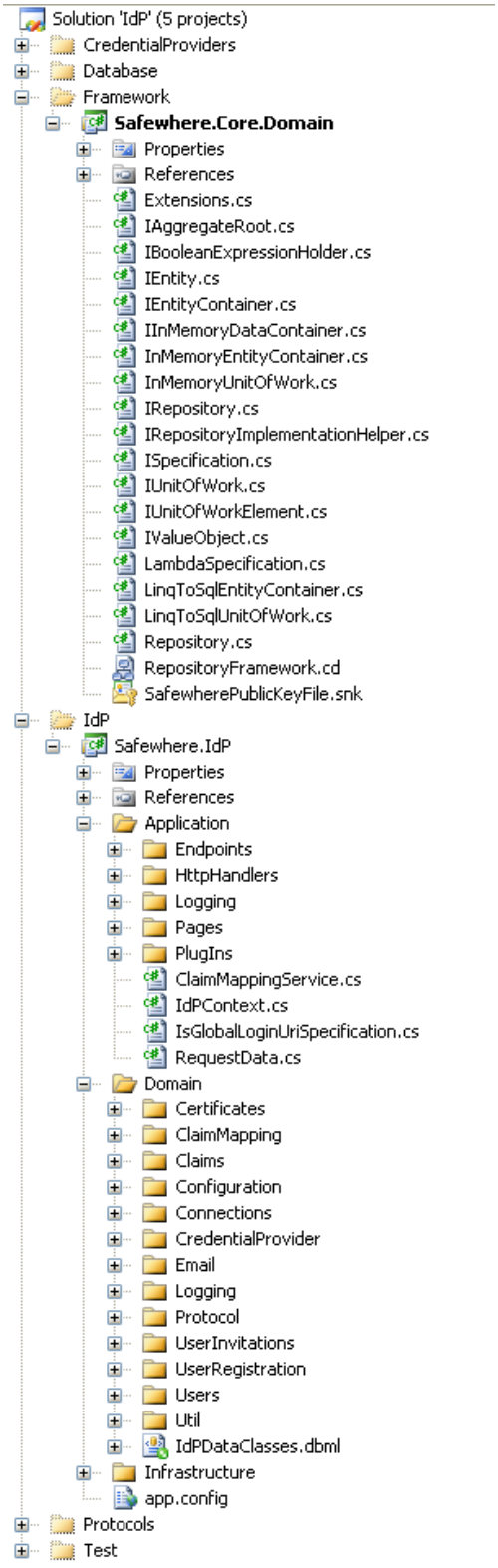


Figure 55. Solution file structure.

7.0.1 Runtime system

The runtime system is particularly interesting, since it shows how the IdP delegates the actual work to the different plug-ins. The class that handles every request and determines what to do is the **IdPEndpointHandlerFactory** class, shown below.

IdPEndpointFactory.cs

```
using System.Web;
using System.Web.SessionState;
using Safewhere.IdP.Application.Endpoints;
using Safewhere.IdP.Application.Pages;
using Safewhere.IdP.Domain.Logging;
using Safewhere.IdP.Properties;

namespace Safewhere.IdP.Application.HttpHandlers
{
    public class IdPEndpointFactory : IHttpHandlerFactory, IRequiresSessionState
    {
        public IHttpHandler GetHandler(HttpContext context, string requestType, string ⇝
     ⇝  virtualPath, string path)
        {
            var loginSpec = new IsGlobalLoginUriSpecification();

            if(loginSpec.IsSatiesfiedBy(context.Request.Url))
                return new IdPLoginPage();

            IEndpoint endpoint = EndpointService.GetEndpointForPath(virtualPath);

            if (endpoint != null)
            {
                return endpoint.Handler;
            }

            string errorMessage = string.Format(IdPErrorMessages.EndpointNotFound, ⇝
     ⇝  virtualPath);

            IdPContext.Current.Trace(TraceLevel.Warning, errorMessage);

            return new ErrorHandler(errorMessage);

        }

        public void ReleaseHandler(IHttpHandler handler)
        {
            return;
        }
    }
}
```

The **IdPEndpointHandlerFactory** class implements the **IHttpHandlerFactory** framework interface and is hooked into the web server configuration file. Hereafter, for every request that the web server receives, it will call the

GetHandler method of our class to get an appropriate handler. In the **GetHandler** method, a check is performed to see if the request is for the global login page, and if this is the case, an instance of the **IdPLoginPage**[11] is returned. Otherwise, the **EndpointService**'s **GetEndpointForPath** method is called, to find the endpoint that serves the given path. If an endpoint is found, its handler is returned. Otherwise an error message is traced, and a generic error page is returned.

EndpointService.cs

```
using Safewhere.Core.Domain;
using Safewhere.IdP.Domain;
using Safewhere.IdP.Domain.Credentials;
using Safewhere.IdP.Domain.Protocol;
using Safewhere.IdP.Infrastructure.Util;
using System.Linq;

namespace Safewhere.IdP.Application.Endpoints
{
    public class EndpointService
    {
        /// <summary>
        /// Gets an implementation of IEndpoint that can handle a given path.
        /// Looks for both protocol and credential endpoints. Protocol endpoints
        /// take precedence over credential endpoints if two should exist
        /// with the same path (eventhough this is considered an error).
        /// </summary>
        /// <param name="path">The path.</param>
        /// <returns>An instance of a class that implements IEndpoint,
        /// or null if no suitable implementation is found.</returns>
        public static IEndpoint GetEndpointForPath(string path)
        {
            var ep = GetProtocolEndpointForPath(path);
            if (ep != null)
                return ep;

            ep = GetCredentialEndpointForPath(path);

            return ep;
        }

        private static IEndpoint GetProtocolEndpointForPath(string path)
        {
            using (var uoo = new LinqToSqlUnitOfWork(new IdPDataClassesDataContext()))
            {
                var cpr = new ConfiguredProtocolRepository { UnitOfWork = uoo };
                var all = cpr.FindAll();
                foreach (var cp in all)
                {
                    var plugin =
                        ActivatorUtil.GetInstance<IPlugIn>(cp.ProtocolDefinition.ProtocolType
    ↪ );
```

[11] The **IdPLoginPage** is the page that shows the user the different configured credential providers and lets him choose which one to use for authentication.

```
42
43                      if (plugin == null) continue;
44
45                      plugin.PluginId = cp.OID;
46
47                      var spec = new EndpointHandlesPathSpecification(path);
48
49                      var endpoint = plugin.GetEndpoints().FirstOrDefault(spec.IsSatiesfiedBy);
50
51                      if (endpoint != null)
52                          return endpoint;
53                  }
54              }
55
56              return null;
57          }
58
59          private static IEndpoint GetCredentialEndpointForPath(string path)
60          {
61              using (var uoo = new LinqToSqlUnitOfWork(new IdPDataClassesDataContext()))
62              {
63                  var ccpr = new ConfiguredCredentialProviderRepository { UnitOfWork = uoo };
64                  var all = ccpr.FindAll();
65                  foreach (var ccp in all)
66                  {
67                      var plugin =
68                          ActivatorUtil.GetInstance<IPlugIn>(ccp.CredentialProviderDefinition.⇝
      ⇝  CredentialProviderType);
69
70                      if (plugin == null) continue;
71
72                      plugin.PluginId = ccp.OID;
73
74                      var spec = new EndpointHandlesPathSpecification(path);
75
76                      var endpoint = plugin.GetEndpoints().FirstOrDefault(spec.IsSatiesfiedBy);
77
78                      if (endpoint != null)
79                          return endpoint;
80                  }
81              }
82
83              return null;
84          }
85      }
86  }
```

The **EndpointService** looks for protocol endpoints and credential provider endpoints, and tries to find one that matches the **path** given as an argument. First it calls the **GetProtocolEndpointForPath** method, which uses the **ConfiguredProtocolRepository** to find configured protocols. It uses reflection to instantiate every configured protocol implementation, and it then iterates over the endpoints provided by the implementation to see if any

endpoint matches the requested path. Something very similar is done in the **GetCredentialEndpointForPath** method.

ConfiguredProtocolRepository.cs

```
using Safewhere.Core.Domain;

namespace Safewhere.IdP.Domain.Protocol
{
    public class ConfiguredProtocolRepository : Repository<ConfiguredProtocol, int>
    {

    }
}
```

The **ConfiguredProtocolRepository** is very simple, because the **FindAll** method is implemented by our generic **Repository** class.

7.0.2 The username/password credential provider

Let us see how the username/password credential provider plug-in is implemented.

UsernamePasswordPlugin.cs

```
using System.Collections.Generic;
using Safewhere.IdP.Application.Endpoints;
using Safewhere.IdP.CredentialProviders.UserNamePassword.Application.Endpoints;

namespace Safewhere.IdP.CredentialProviders.UserNamePassword.Application
{
    public class UsernamePasswordPlugin : IPlugIn
    {
        public string Description
        {
            get { return "Provides login via username/password"; }
        }

        public List<IEndpoint> GetEndpoints()
        {
            return new List<IEndpoint> {new UsernamePasswordEndpoint(PluginId)};
        }

        public int PluginId { get; set; }
    }
}
```

The **UsernamePasswordPlugin** class implements the **IPlugin** interface, which is pretty straightforward. The most important thing to notice is the **GetEndpoints** method, which in this case returns a list with only one element, an instance of the **UsernamePasswordEndpoint** class.

UsernamePasswordEndpoint.cs

```
using System.Web;
using Safewhere.IdP.Application.Endpoints;
using Safewhere.IdP.CredentialProviders.UserNamePassword.Application.Pages;

namespace Safewhere.IdP.CredentialProviders.UserNamePassword.Application.Endpoints
{
    public class UsernamePasswordEndpoint : IEndpoint
    {
        private int _pluginId;

        public UsernamePasswordEndpoint(int pluginId)
        {
            _pluginId = pluginId;
        }

        public string Path
        {
            get { return "unpwdlogin.idp"; }
        }

        public string Description
        {
            get { return "Endpoint for username/password login."; }
        }

        public IHttpHandler Handler
        {
            get { return new UsernamePasswordPage(_pluginId); }
        }

        public string Name
        {
            get { return "UsernamePasswordEndpoint"; }
        }
    }
}
```

The **UsernamePasswordEndpoint** class implements the **IEndpoint** interface, and most importantly it returns an instance of the **UsernamePasswordPage** class in its **Handler** property.

```
ï ż £using System.Collections.Generic;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using Safewhere.Core.Domain;
using Safewhere.IdP.Application;
using Safewhere.IdP.Application.Pages;
using Safewhere.IdP.CredentialProviders.UserNamePassword.Properties;
using Safewhere.IdP.Domain;
using Safewhere.IdP.Domain.Claims;
using Safewhere.IdP.Domain.Credentials;
using System.Linq;
```

```
13  using Safewhere.IdP.Domain.Logging;
14
15  namespace Safewhere.IdP.CredentialProviders.UserNamePassword.Application.Pages
16  {
17      /// <summary>
18      /// A page that collects username and password through a form.
19      /// </summary>
20      public class UsernamePasswordPage : CredentialProviderBasePage, IRequiresSessionState
21      {
22          private Label _usernameLabel;
23          private Label _passwordLabel;
24          private TextBox _username;
25          private TextBox _password;
26          private Button _submit;
27          private Panel _buttonPanel;
28          private RequiredFieldValidator _usernameValidator;
29          private RequiredFieldValidator _passwordValidator;
30          private Panel _messagePanel;
31
32          public UsernamePasswordPage(int credentialProviderDefinitionId) : base(⇝
        ⇝  credentialProviderDefinitionId)
33          {
34
35          }
36
37          protected override void OnLoad(System.EventArgs e)
38          {
39              return;
40          }
41
42          protected override void CreateChildControls()
43          {
44              _usernameLabel = new Label { Text = Resources.UsernameLabelText };
45              _usernameLabel.Width = 100;
46              _content.AddControl(_usernameLabel);
47
48              _username = new TextBox{Width = 150};
49              _username.ID = "username";
50              _content.AddControl(_username);
51
52              _usernameValidator = new RequiredFieldValidator
53                                       {
54                                           ControlToValidate = _username.ID,
55                                           Display = ValidatorDisplay.Dynamic,
56                                           ErrorMessage = Resources.UsernameValidationText
57                                       };
58              base._content.AddControl(_usernameValidator);
59
60              base._content.AddControl(new LiteralControl("<br/>"));
61
62              _passwordLabel = new Label { Text = Resources.PasswordLabelText };
63              _passwordLabel.Width = 100;
64              base._content.AddControl(_passwordLabel);
65
66              _password = new TextBox { Width = 150, TextMode = TextBoxMode.Password };
67              _password.ID = "password";
```

```
68              _content.AddControl(_password);
69
70              _passwordValidator = new RequiredFieldValidator
71              {
72                  ControlToValidate = _password.ID,
73                  Display = ValidatorDisplay.Dynamic,
74                  ErrorMessage = Resources.PasswordValidationText
75              };
76              base._content.AddControl(_passwordValidator);
77
78              base._content.AddControl(new LiteralControl("<br/>"));
79
80              _buttonPanel = new Panel { Width = 255 };
81              _buttonPanel.Style.Add("text-align", "right");
82              _submit = new Button {Text = Resources.SubmitButtonText, CausesValidation = true}
    ;
83              _submit.Click += _submit_Click;
84              _buttonPanel.Controls.Add(_submit);
85              base._content.AddControl(_buttonPanel);
86
87              _messagePanel = new Panel {Visible = false};
88
89              base._content.AddControl(_messagePanel);
90          }
91
92          void _submit_Click(object sender, System.EventArgs e)
93          {
94              if(IsValid)
95              {
96                  var user = UserService.GetUser(_username.Text);
97
98                  if (user == null || !user.VerifyPassword(_password.Text))
99                  {
100                     LoginError();
101                     return;
102                 }
103
104                 RetrieveCredentialClaims(user);
105
106             }
107         }
108
109         private void RetrieveCredentialClaims(User user)
110         {
111             var claims = new List<CredentialClaim>();
112             using (var uoo = new LinqToSqlUnitOfWork(new IdPDataClassesDataContext()))
113             {
114                 var ccpr = new ConfiguredCredentialProviderRepository {UnitOfWork = uoo};
115                 var ccp = ccpr.FindByPrimaryKey(CredentialProviderDefintionId);
116                 foreach(var ccd in ccp.CredentialClaimDefinitions)
117                 {
118                     if (ccd.IsIdentityBearer)
119                         claims.Add(CredentialClaim.FromDefinition(user.UserName, ccd));
120                 }
121             }
122
```

```
123             if(claims.Count > 0)
124             {
125                 IdPContext.Current.AuthenticationDone(claims.AsQueryable());
126             }else
127             {
128                 IdPContext.Current.Trace(TraceLevel.Error, "No credential claims found for ⇝
      ⇝  user " + user.UserName);
129             }
130         }
131 
132         private void LoginError()
133         {
134             _messagePanel.Visible = true;
135             _messagePanel.Controls.Add(new LiteralControl("Wrong username and password ⇝
      ⇝  combination"));
136         }
137     }
138 }
```

The **UsernamePasswordPage** inherits the **Page** class from the framework (which in turn implements the **IHttpHandler** interface). This page basically displays two text fields where the user can enter his username and his password. The most interesting method is the **_submit_Click** method. This method uses the **UserRepository** class to get an instance of the **User** class corresponding to the name entered by the user. It then calls the **VerifyPassword** method on the **User** class instance, to see if the password entered by the user corresponds to the one stored in the **User** instance. If this is not the case, an error message is displayed, and the user can try again. Otherwise, the user's credential claims are extracted and the **AuthenticationDone** method of the **IdPContext** class is called, thus finalizing the authentication.

UserRepository.cs

```
1  using System.Linq;
2  using Safewhere.Core.Domain;
3  using Safewhere.IdP.Domain.Users.Specifications;
4  
5  namespace Safewhere.IdP.Domain.Repositories
6  {
7      public class UserRepository : Repository<User, int>
8      {
9          public IQueryable<User> FindUsersBeginningWith(string beginsWith)
10         {
11             var spec = new UserNameBeginningWith(beginsWith);
12             return Find(spec);
13         }
14 
15         /// <summary>
16         /// Checks if a user with the given userName already exists.
17         /// The check is performed case insensitively
18         /// </summary>
19         /// <param name="userName">the userName.</param>
20         /// <returns>True if the user exists, false otherwise</returns>
```

```
21          public User FindByName(string userName)
22          {
23              return FindFirst(new ExactUserName(userName));
24          }
25
26          public bool ExistsByName(string userName)
27          {
28              return Exists(new ExactUserName(userName));
29          }
30      }
31  }
```

The **UserRepository** class also has a simple implementation, even though a few extra methods have been added. The **FindUserByName** method used by the **_submit_click** method described above simply uses an instance of the **ExactUserName** specification class to find the corresponding **User** instance.

7.0.3 A dummy protocol

Although no production-ready protocol implementation has been developed as part of this thesis, I will show the following implementation of a dummy protocol. The dummy protocol responds on a single URI. The dummy protocol displays all the claims for the user that logs in.

TestProtocolPlugin.cs

```
1  using System.Collections.Generic;
2  using Safewhere.IdP.Application.Endpoints;
3  using Safewhere.IdP.Protocols.TestProtocol.Application.Endpoints;
4
5  namespace Safewhere.IdP.Protocols.TestProtocol.Application
6  {
7      public class TestProtocolPlugin : IPlugIn
8      {
9          public string Description
10         {
11             get { return "Simple test protocol"; }
12         }
13
14         public List<IEndpoint> GetEndpoints()
15         {
16             return new List<IEndpoint> {new TestEndpoint(PluginId)};
17         }
18
19         public int PluginId{get; set;}
20     }
21 }
```

The **TestProtocolPlugin** class is very similar to the `UsernamePasswordPlugin` presented above.

TestEndpoint.cs

```
using System.Web;
using Safewhere.IdP.Application.Endpoints;
using Safewhere.IdP.Protocols.TestProtocol.Application.Handlers;

namespace Safewhere.IdP.Protocols.TestProtocol.Application.Endpoints
{
    public class TestEndpoint : IEndpoint
    {

        private readonly int _protocolId;

        public TestEndpoint(int protocolId)
        {
            _protocolId = protocolId;
        }

        public string Path
        {
            get { return "test.idp"; }
        }

        public string Description
        {
            get { return "Test endpoint"; }
        }

        public IHttpHandler Handler
        {
            get { return new TestHandler{Path = Path, ProtocolId = _protocolId}; }
        }

        public string Name
        {
            get { return "testendpoint"; }
        }
    }
}
```

The most important feature of the **TestEndpoint** class is that it returns an instance of the **TestHandler** class, which is shown below.

TestHandler.cs

```
using System.Collections.Specialized;
using System.Web;
using Safewhere.IdP.Application;
using Safewhere.IdP.Application.HttpHandlers;

namespace Safewhere.IdP.Protocols.TestProtocol.Application.Handlers
{
    public class TestHandler : IdPHttpHandlerBase
    {
        private NameValueCollection _requestParams;

        public override bool ValidateRequest(NameValueCollection requestParams)
```

```
13          {
14              _requestParams = requestParams;
15              return true;
16          }
17
18          public override bool RequiresAuthentication()
19          {
20              return true;
21          }
22
23          public override bool ForceReauthentication()
24          {
25              return false;
26          }
27
28          public override void SendResponse(HttpContext context)
29          {
30              string username = IdPContext.Current.UserName;
31              string connectionName = _requestParams["connection"];
32              context.Response.Write("<html><head><title>Claims for user: ");
33              context.Response.Write(username + "</title></head><body>");
34              context.Response.Write("<h1>" + username + "</h1>");
35
36              var claims = ClaimMappingService.GetMappedClaimsForUser(username, ProtocolId , ⇝
    ⇝  connectionName);
37              string claimStr = "<claims>\n";
38              foreach(var claim in claims)
39              {
40                  context.Response.Write("<b>" + claim.DisplayName + "</b><br/>");
41                  context.Response.Write("Name: " + claim.Name + "<br/>");
42                  context.Response.Write("Value: " + claim.Value + "<br/>");
43                  context.Response.Write("NameFormat: " + claim.NameFormat + "<br/>");
44                  context.Response.Write("ValueType: " + claim.ValueType + "<br/><br/>");
45                  claimStr += "<claim>\n";
46                  claimStr += "<name>" + claim.Name + "</name>\n";
47                  claimStr += "<value>" + claim.Value + "</value>\n";
48                  claimStr += "<nameformat>" + claim.NameFormat + "</nameformat>\n";
49                  claimStr += "<valuetype>" + claim.ValueType + "</valuetype>\n";
50                  claimStr += "</claim>\n";
51              }
52              claimStr += "</claims>\n";
53              context.Response.Write("<form><textarea cols=100 rows=50>" + claimStr);
54
55              context.Response.Write("</textarea></form>");
56              context.Response.Write("</body></html>");
57          }
58      }
59  }
```

You can see that this handler participates in the runtime request sequence described in Section 4.11 on page 79. It does so by overriding the **`ValidateRequest`**, **`RequiresAuthentication`**, **`ForceReauthentication`** and **`SendResponse`** methods from its parent class. Since the **`RequiresAuthentication`** method returns **`true`**, the core IdP makes sure that the user is authenticated before

calling the **`SendResponse`** method. Therefore, this method can make use of **`IdPContext.Current.UserName`** and call the **`ClaimMappingService`** to get the user's claims, which it then just outputs as html.

8
Design validation

The purpose of this chapter is to evaluate the design that I have presented in this thesis, and not least, to assess the practicability of domain-driven design. Before looking into what has been learned, I wish to present the background for introducing a new design paradigm in Safewhere.

Safewhere is a relatively new company with only three years of existence. As with many small companies, Safewhere started on a good idea and a prototype written by the founders. In the beginning of the company's life money was scarce, and in order to attract venture capital having an actual product to show for was priority number one. So, given the circumstances, not much time was spent on software design and modelling. Instead, a few programmers were hired, myself included, and each programmer was given a few high level features to implement. as it turned out however, each programmer had his own individual coding style and understanding of the business domain. In the beginning, this did not seem to cause that big of a problem, and version one of the product was delivered, fully functional. After the initial release the need for new features grew rapidly, and these were added on an ad hoc basis. The product was working well, but it became increasingly difficult to add new features and refactor existing features, as the code base grew. Despite of established coding guidelines, extensive test suites and a generally open communication culture, there was something missing in our development method. One of the ways this manifested itself was that we had several classes in different assemblies with similar names but with dissimilar meaning. It was also difficult to explain the architecture to new members of the team, since the code was not organized in a heterogeneous manner. In retrospect, the problem clearly was that we did not have either a model (and model framework) nor a ubiquitous language. One day, most of the developers of Safewhere attended a seminar where we were first introduced to domain-driven design. Even though domain-driven design was not the main topic of the seminar we were all left with the impression that it was a smart thing to do. Therefore we decided to try out domain-driven design in practice. Since I was about to write my thesis it was decided that my current project could be a guinea-pig project for introducing domain-driven design in Safewhere. Upon writing this thesis, I presented my work to my

colleagues. The first purpose of this was to have them assess the usability and completeness of the model, and secondly to make them give me their opinion of domain-driven design, and whether they would support the introduction of domain-driven design as a design standard on future projects in Safewhere. I interviewed[12] three people, and below is what I found.

8.1. Model evaluation

Based on the interviews, I can conclude that the following is true about the model:

- **It solves the business problem well.**
- **It represents a concise and expressive ubiquitous language, which makes it easy to talk about.**
- **It is flexible and has a clear distribution of responsibilities between components, making it easy to refactor.**
- **It will lead to a stable and mature end-product.**
- **It will reduce total cost of ownership of the software.**

So overall my work has been very well received, and there is no doubt that domain-driven design will be the future design paradigm in Safewhere. That said, there were also a number of less positive things, which I will address below.

First of all it was noted that a few terms could have had better names. These include the different types of claims[13]; `CredentialClaim`, `IdentityProviderClaim` and `IssuedClaim`. The distinction between the three types of claims is more important than their names, and even though the names may not be the most elegant, they do convey important information. Another concept which could have been named better is the `Logging` concept. The word "logging" is overloaded and has potentially different meanings to different people. A better name for this model concept could have been "business activity monitoring" which is both more expressive and unambiguous. As for the "tracing" model-concept, it should probably not have been made part of the model since it is an infrastructure concept and has no real relevance to the business problem.

[12] The interviews can be found in full length in the Appendix B.

[13] Only one interviewee though that these names could have been better. Another interviewee, Peter, found them really good.

If anything it could have been part of the domain-neutral component, but it is not clear if it even belongs here.

It has also been pointed out that there is an excessive use of factories in the model. There is however a good reason why using a lot of factories is a good convention for this model. The factories are mainly used to construct entity objects. The problem is that the LinqToSql framework generates parameter-less constructors. It is of course still possible to add more constructors with more parameters. However if constructors instead of factories were used throughout the code, it could potentially lead other developers to the impression that the parameter-less constructors are also an option, which they are not. So by using factories to create entity objects throughout the code, I have created a convention that others can follow, and hopefully avoid unintended use of the parameter-less constructors. Unfortunately this convention is implicit and I have not been able to come up with a way to express it explicitly in the model. This is definitely a weakness in the model, and I hope that a better solution can be found in the future.

Another weakness of the model has to do with the plug-in nature of the system. Refactoring of the plug-in system, mainly the interfaces, could lead to breaking backwards compatibility with third party plug-ins. I do not think that this is a flaw in the model per se, but rather a natural consequence of any plug-in architecture. There are no current plans to allow third party plug-ins in the IdP, so this is no great concern at the moment, since breaking changes for self-developed plug-ins can be handled at compile time and during testing.

9 Conclusion

This chapter contains my conclusion on what has been learned throughout the process of writing this thesis.

9.1. The model

All in all I think the model that has been developed is good and thorough, and solves the business problem defined in the specification well, a fact that is substantiated by the interviews that were carried out. Now, is the model perfect? No, it is not. And that is exactly one of the points of domain-driven design. You will never get your model one hundred percent right the first time, and that is why it is so important that the model is flexible so that future refactorings can be carried out with ease. I have achieved this flexibility to a high degree through distribution of responsibilities and by having made the model storage-agnostic. Also, when it comes to the domain-neutral component, or model framework, I am sure that this will contribute to shorter development times on future projects because it is highly reusable.

9.1.1 Future perspectives

Hosting software in the cloud is most likely going to be become very common in the future. Cloud hosting services are being offered by many of the big players on the market, with products such as “Google App Engine”, “Microsoft Azure”, “Amazon Elastic Cloud” and more. The benefit of these services is that they provide instant on-demand scaling of the applications they host. It would be natural to host a product such as the IdP in the cloud to make sure that even extremely high loads on the application could be handled. However, the storage mechanisms used in these cloud services are very dissimilar to relational databases. Therefore, the model will face its greatest test if at some point it is decided to host the IdP in the cloud.

9.2. Domain-driven design

I can only say that this, my first encounter with domain-driven design, has been very positive. Domain-driven design is based on some very sound principles that can be applied to any software design task that I can think of. Textbooks about domain-driven design often use examples from business domains that are commonly understood, such as hotel reservation systems, order systems or the like. The business domain that has been modelled here is of a very technical nature, and at the beginning of this project I was anxious to find out if the domain-driven design principles could be applied to this kind of system as well. I think that it is safe to conclude that it can. Even though there was no sharp distinction between domain experts and developers on this project, having a broad ubiquitous language that is reflected in the model and code has made the system so much easier to talk about and discuss. Furthermore, a lot of time has been saved by the ability to refactor in the model instead of refactoring code that has taken a long time to write. And using the domain-driven design concepts such as entities, repositories, services, specifications, etc. makes the code much more recognizable to newcomers to the team.

Using domain-driven design on a project requires commitment from all developers. In the modelling phase, the developers involved must have strong modelling skills, and a thorough understanding of domain-driven design. In the development phase, when the model has already been lined out, it is still important that every developer on the team has knowledge about domain-driven design such that all the concepts used in the model can be understood by everyone. I have not had the opportunity to try domain-driven design on bigger scale since this project has not involved other developers than myself. I am however convinced that the benefits of using domain-driven design will turn out to be even bigger when working on large teams. The larger the teams, the more important it becomes to have a common ubiquitous language and a strong and flexible model.

Mastering domain-driven design is not something that can be learned from a single project, but my work with it so far has definitely encouraged me to keep using it, and I look forward to becoming even more adept at applying its principles. For me personally, being able to design software very well is a goal that I strive to achieve because it will give me greater professional satisfaction, and by practicing my domain-driven design skills I am sure that

I will be able to reach that goal.

Appendix A
C# language elements

This aim of this appendix is to give a short introduction to the C# language, enabling readers without prior knowledge of the C# language to understand the code samples in the thesis. The contents presented herein are based on the C# Language Specification [Microsoft 2009], and should by no means be regarded as complete. Please refer to [Microsoft 2009] for the complete reference.

A.1. Program structure

The key organizational concepts in C# are ***programs***, ***namespaces***, ***types***, ***members***, and ***assemblies***. C# programs consist of one or more source files. Programs declare types, which contain members and can be organized into namespaces. Classes and interfaces are examples of types. Fields, methods, properties, and events are examples of members. When C# programs are compiled, they are physically packaged into assemblies. Assemblies typically have the file extension .exe or .dll, depending on whether they implement applications or libraries.

A.2. Types

C# programs use type declarations to create new types. A type declaration specifies the name and the members of the new type. Five of C#'s categories of types are user-definable: ***class types***, ***struct types***, ***interface types***, ***enum types***, and ***delegate types***.

Class, struct, interface and delegate types all support ***generics***, whereby they can be parameterized with other types.

There are two kinds of types in C#: ***value types*** and ***reference types***. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects.

With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other

A.2.1 Classes

A class type defines a data structure that contains data members (fields) and function members (methods, properties, and others). Class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.

Members

The members of a class are either static members or instance members. Static members belong to classes, and instance members belong to objects (instances of classes). The following table provides an overview of the kinds of members a class can contain.

Member	**Description**
Constants	Constant values associated with the class
Fields	Variables of the class
Methods	Computations and actions that can be performed by the class
Properties	Actions associated with reading and writing named properties of the class
Indexers	Actions associated with indexing instances of the class like an array
Events	Notifications that can be generated by the class
Operators	Conversions and expression operators supported by the class
Constructors	Actions required to initialize instances of the class or the class itself
Destructors	Actions to perform before instances of the class are permanently discarded
Types	Nested types declared by the class

Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are five possible forms of accessibility. These are summarized in the following table.

Accessibility	**Meaning**
public	Access not limited
protected	Access limited to this class or classes derived from this class
internal	Access limited to this program
protected internal	Access limited to this program or classes derived from this class
private	Access limited to this class

Inheritance

A class inherits the members of its direct base class type. Inheritance means that a class implicitly contains all members of its direct base class type, except for the instance constructors, destructors and static constructors of the base class.

Base access

A base-access consists of the reserved word ***base*** followed by either a "." token and an identifier or an expression-list enclosed in square brackets:

```
base-access:
base   .   identifier
base   [   expression-list   ]
```

A base-access is used to access base class members that are hidden by similarly named members in the current class or struct. A base-access is permitted only in the block of an instance constructor, an instance method, or an instance accessor.

This access

A this-access consists of the reserved word ***this***.

```
this-access:
this
```

A this-access is permitted only in the block of an instance constructor, an instance method, or an instance accessor. Within an instance constructor or instance function member of a class, this is classified as a value. Thus, while this can be used to refer to the instance for which the function member was invoked, it is not possible to assign to this in a function member of a class.

Constructors

The **`this(...)`** form of constructor initializer is commonly used in conjunction with overloading to implement optional instance constructor parameters.

```
class Text
{
        public Text(): this(0, 0, null) {}
        public Text(int x, int y): this(x, y, null) {}
        public Text(int x, int y, string s) {
                // Actual constructor implementation
        }
}
```

In the above example, the first two instance constructors merely provide the default values for the missing arguments. Both use a this(...) constructor initializer to invoke the third instance constructor, which actually does the work of initializing the new instance.

Likewise, a constructor can invoke the constructor of its base class by using base-access as illustrated below:

```
using System;

class A
{
        protected string theString;
        public A(string aString){
                this.theString = aString;
        }
```

```
9  }
10
11 class B : A
12 {
13         private int theInt;
14
15         public B(string aString, int anInt) : base(aString) {
16                 this.theInt = anInt;
17         }
18
19         public void PrintIt(){
20                 Console.WriteLine("The string: " + theString + ". The int: " + theInt);
21         }
22 }
```

A.2.2 Structs

A struct type is similar to a class type in that it represents a structure with data members and function members. However, unlike classes, structs are value types and do not require heap allocation. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type **object**.

A.2.3 Interfaces

An interface defines a contract that can be implemented by classes and structs. An interface can contain methods, properties, events, and indexers. An interfaces does not provide implementations of the members it defines, it merely specifies the members that must be supplied by classes or structs that implement the interface. A class or struct that implements an interface must provide implementations of the interface's function members. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

A.2.4 Delegates

A delegate type represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages,

but unlike function pointers, delegates are object-oriented and type-safe. The following example declares and uses a delegate type named **Function**.

```
using System;
delegate double Function(double x);
class Multiplier
{
        double factor;
        public Multiplier(double factor) {
                this.factor = factor;
        }
        public double Multiply(double x) {
                return x * factor;
        }
}
class Test
{
        static double Square(double x) {
                return x * x;
        }
        static double[] Apply(double[] a, Function f) {
                double[] result = new double[a.Length];
                for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
                return result;
        }
        static void Main() {
                double[] a = {0.0, 0.5, 1.0};
                double[] squares = Apply(a, Square);
                double[] sines = Apply(a, Math.Sin);
                Multiplier m = new Multiplier(2.0);
                double[] doubles =  Apply(a, m.Multiply);
        }
}
```

An instance of the **Function** delegate type can reference any method that takes a double argument and returns a double value. The Apply method applies a given Function to the elements of a **double[]**, returning a **double[]** with the results. In the **Main** method, **Apply** is used to apply three different functions to a **double[]**. A delegate can reference either a static method (such as **Square** or **Math.Sin** in the previous example) or an instance method (such as **m.Multiply** in the previous example). A delegate that references an instance method also references a particular object, and when the instance method is invoked through the delegate, that object becomes this in the invocation. Delegates can also be created using anonymous functions, which are "inline methods" that are created on the fly. Anonymous functions can see the local variables of the surrounding methods. Thus, the multiplier example above can be written more easily without using a **Multiplier** class:

```
double[] doubles =  Apply(a, (double x) => x * 2.0);
```

An interesting and useful property of a delegate is that it does not know or care about the class of the method it references; all that matters is that the referenced method has the same parameters and return type as the delegate.

A.2.5 Partial types

A type declaration can be split across multiple partial type declarations. The type declaration is constructed from its parts by following the rules in this section, whereupon it is treated as a single declaration during the remainder of the compile-time and runtime processing of the program.

A.2.6 Extension methods

When the first parameter of a method includes the this modifier, that method is said to be an extension method. Extension methods can only be declared in non-generic, non-nested static classes. The first parameter of an extension method can have no modifiers other than **this**, and the parameter type cannot be a pointer type. The following is an example of a static class that declares two extension methods:

```
public static class Extensions
{
        public static int ToInt32(this string s) {
                return Int32.Parse(s);
        }
        public static T[] Slice<T>(this T[] source, int index, int count) {
                if (index < 0 || count < 0 || source.Length - index < count)
                        throw new ArgumentException();
                T[] result = new T[count];
                Array.Copy(source, index, result, 0, count);
                return result;
        }
}
```

An extension method is a regular static method. In addition, where its enclosing static class is in scope, an extension method can be invoked using instance method invocation syntax, using the receiver expression as the first argument. The following program uses the extension methods declared above:

```
static class Program
{
        static void Main() {
                string[] strings = { "1", "22", "333", "4444" };
                foreach (string s in strings.Slice(1, 2)) {
                        Console.WriteLine(s.ToInt32());
                }
        }
}
```

The **Slice** method is available on the **string[]**, and the **ToInt32** method is available on string, because they have been declared as extension methods. The meaning of the program is the same as the following, using ordinary static method calls:

```
static class Program
{
        static void Main() {
                string[] strings = { "1", "22", "333", "4444" };
                foreach (string s in Extensions.Slice(strings, 1, 2)) {
                        Console.WriteLine(Extensions.ToInt32(s));
                }
        }
}
```

Appendix B
Interviews

B.1. Mikkel Christensen — Developer at Safewhere

Me What is your general impression of the model?

Mikkel I think the model solves the business requirements presented in the specification well, and in a flexible way.

Me After having read this thesis, is it obvious for you how you could implement a new plug-in.

Mikkel Yes. I would have to implement the `IPlugin` interface. Or rather, either the `ICredentialProviderPlugin` or `IProtocolPlugin`.

Me Do you think that the model is flexible or good enough to make refactoring easy.

Mikkel Well, both yes and no. I think the plug-in structure introduces a few issues with backwards compatibility, with regards to refactoring. However, I do acknowledge that these issues will always be present in systems that use a plug-in structure, and that it is not a problem with the model per se. As a matter of fact the model probably does make it easier, because of the way it clearly distributes responsibilities. I guess that having a well-defined model like this, you can actually perform a lot of refactoring on the model itself, before you begin programming.

Me That is correct, and I have already done that many times.

Me Do you think that this specific system benefits from being modelled using the domain-driven paradigm?

Mikkel Yes. I think that DDD allows the model to be described in a short and concise manner. I think the readability of this model is better than many other standard-OO models I have seen in the past.

Me What do you think about domain-driven design's focus on a ubiquitous language? Does it make sense?

Mikkel It makes a lot of sense. I think you avoid the disconnect between the developers and the business. It makes developers understand the relevance of all classes, and they don't have to spend time to come up with good class names. That makes them more productive, I think.

Me You seem like a fan of domain-driven design already. Don't you think that there is anything negative about it?

Mikkel Well, if I have to say something it would be that it requires quite skilled developers in the modelling phase, to bridge the gap between the business and the code. But apart from that, no.

Me In the modelling phase only?

Mikkel Yes. When the system has been modelled, I think that it should be a piece of cake to implement, since all the decisions have been made already. Past the modelling phase, developers would still have to be familiar with domain-driven design, but they would not have to great business insight to implement the model, I think.

Me So, would you use domain-driven design on your next project?

Mikkel Based on what I have read in your thesis, it seems effective. I can't conclude that it is the solution to everything, but I am definitely ready to try it out in practice.

B.2. Mark Seemann — Senior Developer at Safewhere

Me What is your impression of the model?

Mark My overall impression is that it is good. I think the distinction between different claim types is good, however I think the names could have been better.

Me Do you have a better suggestion for the names?

Mark Hehe, no, not really.

Me Ok, anything else?

Mark I don't like the naming for "Logging". I think it is an overloaded word and that it doesn't describe the exact meaning. I think that "business activity monitoring", or something like that would have been better.

Me Good point. I agree.

Mark I am also not sure that tracing actually belongs to the domain. In my understanding it is more of an infrastructure thing.

Me I wouldn't say that I disagree. The only reason I have chosen to make it part of the domain model is that I want to expose services that allows inspection of trace messages in the administrator UI.

Mark Also, you have quite a few factories. I appreciate that the factory pattern is often useful, but it seems that in this model, everything that these factories do could have just as well been done in constructor functions.

Me A valid point. The factories are mainly used to construct entity objects. The problem is that the LinqToSql framework generates parameter-less

constructors. It is of course still possible to add more constructors with more parameters. However if I used constructors instead of factories throughout the code, it could potentially lead others to the impression that the parameter-less constructors are also an option, which they are not. So by using factories to create entity objects throughout the code, I create a convention that others can follow, and hopefully avoid unintended use of the parameter-less constructors.

Mark Ok, that makes sense. That is a good convention then.

Me Yes, and I admit that it is a minor flaw in the model. But I think that the extra productivity provided by the or-mapper is worth making this sacrifice for.

Me So what is your overall impression of domain-driven design?

Mark I think that domain-driven design is a very sound approach. The "domain model" pattern was actually introduced by Martin Fowler a long time ago. In his book, it was presented on 10–15 pages. However I like how Eric Evans has written an entire book about it, because it really is an essential pattern that requires that kind of attention.

Me So do you think there is anything negative to say about domain-driven design?

Mark Well, not really. But using domain-driven design requires strong modelling skills. And maturity. Both technical maturity for the programmer, but also a mature organization.

Me Mark, you are big advocate for test-driven development (TDD). You even have your own blog about TDD (`http://blog.ploeh.dk`). Do you think that a model such as this renders itself well to TDD?

Mark Well, TDD is all about rapid feedback. You want to write tests that cover the entire system, and run these tests over and over again, to make sure that you are not making breaking changes. For a system such as the one you have modelled, it would not be unnatural to have around 1000 unit-tests. Therefore, I am concerned with the tight coupling to the relational database that you have presented in your model. You see, for TDD to be effective you really have to run your tests often. And running all, say, 1000 tests often, require them to be execute really fast. And here, database access really becomes a bottle-neck. If your tests do not run in a few seconds, you just stop running them, and effectively stop exercising TDD.

Me Well, I agree that unit tests should not rely on database access. Therefore, the domain neutral component has an in-memory equivalent to the database access component. This in-memory equivalent was made

exactly with unit testing in mind.

Mark Oh, ok. I must have missed that. In that case I think the model renders itself well to TDD.

B.3. Peter Haastrup — Technical director of Safewhere

Me Do you think that the model covers the specification?

Peter Haastrup I think that is generally covers specification well, however I don't think that it covers the multi-tenancy aspect at all.

Me That is correct. The model is multi-tenancy unaware, and the entire multi tenancy issue is supposed to be handled at installation time, by using separate database schemas, separate web applications in IIS, and separate machine accounts for execution.

Me Do you think that model presented herein represents a good ubiquitous language?

Peter Haastrup Yes, I think that the model has made a lot of concepts explicit, making the model easy to talk about on a very detailed level.

Me One of the other interviewees found that the claim names could have been better. Do you agree with this.

Peter Haastrup No, not at all. I am very happy with those names, and think that they convey important information about what they represent.

Me Do you think that using the domain-driven design paradigm and the domain-neutral component presented here will make development faster?

Peter Haastrup I think that it is positive that a lot of decisions about how to structure the code have been made already. It will be easier to maintain the code, and programmers will be able to recognize the concepts used from project to project.

Me What impact do you think that using domain-driven design for this IdP project will have on the total cost of ownership for the product?

Peter Haastrup Although introducing a new design paradigm, such as domain-driven design, seems to require more preparation time before coding can start, I am sure that this extra cost is made up for by faster development times. As a matter of fact I think that it is made up for already by the time of the first release, and that having such a good model will make it much easier and faster to develop future releases. Furthermore, I think that using domain-driven design the way you have here, will result in a much more stable and mature product.

Bibliography

References

Daarbak, Torben. 2008. Hver tredje CRM-løsning er SaaS-baseret i 2012. `http://www.computerworld.dk/art/49054`.

Evans, Eric. 2004. *Domain Driven Design*. Addison-Wesley.

Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Handy, Alex. 2009. Domain-driven design through Eric Evans' eyes. `http://www.sdtimes.com/content/article.aspx?ArticleID=33357`.

Harding, Kim. 2008. Domain-Driven Design course. `http://www.kimharding.com`.

McCarthy, Tim. 2008. *.NET Domain-Driven Design with C#*. Wiley Publishing Inc.

Microsoft. 2007. Implementing an Implicit Transaction using Transaction Scope. `http://msdn.microsoft.com/en-us/library/ms172152.aspx`.

Microsoft. 2009. C# Language Specification 3.0. `http://go.microsoft.com/fwlink/?LinkId=64165`.

OpenSAML. 2009. OpenSAML website. `http://www.opensaml.org`.