Security in Domain-Driven Design

Author: Michiel Uithol

Supervisors:
dr.ir. M.J. van Sinderen
dr.ir. L. Ferreira Pires
drs. T. Zeeman (Sogyo)
E. Mulder Ba (Sogyo)

Abstract

Application development is a process that becomes increasingly complex depending on the intricacy of the application being developed. Development techniques and methodologies exist to manage and control the complexity of this development process. Amongst the techniques introduced to manage the complexity of the application development process is Domain-driven design (DDD).

DDD prescribes a specific application of separation of concerns to the application model into a domain model and DDD-services.
This Masters assignment investigates how to handle issues concerning the modelling and implementation of authorization and authentication functionality in an application developed according to the DDD principle of separating domain-related functionality from domain-independent functionality. This means an application where security functionality is located in a DDD-service.

The goal of this Masters assignment is to find design options to separate security from domain-related functionality and provide guidelines for usage of these design options. To find the best design options, the problem is explored and the levels of coupling between DDD-services and the domain implementation are clarified.
Criteria for the application design phase are that the design should comply with the DDD principle of separating domain-related functionality from domain-independent functionality, and that the design should provide usability and efficiency when implemented.

Two prototypes use Aspect-Oriented Programming (AOP) to separate security logic into a DDD-service. AOP allows for cross-cutting concerns to be declared in a single place called an aspect. This aspect is then applied in all the places this functionality is needed.

The implementation resulted in three prototypes:

1. A prototype with intermediary adapters that regulate interactions with the domain implementation.
2. A prototype with AOP point-cuts aimed at intermediary adapters.
3. A prototype with AOP point-cuts aimed at the domain implementation.

After reviewing the prototype implementations, the recommendation is made for the AOP implementation with point-cuts towards the domain implementation. This choice is made based on the application structure and compliance with the criteria.

Table of contents

Chapter 1
Introduction

This chapter starts by introducing the context of the investigation performed to analyse a problem occurring in Domain-driven design (DDD).
After introducing the background information concerning DDD this chapter presents the problems related to security in DDD. Based on the problem description the goals for this investigation are clarified. Next, this chapter describes how these goals are achieved within this Masters assignment.
This chapter's role is to describe the context of this Masters assignment. This context is used to clarify the objectives and approach.

1.1 Background

Application development is a process that becomes increasingly complex as the intricacy of the application being developed increases. Development techniques and methodologies have been defined to manage and control the complexity of this development process. These methodologies and techniques are often based on specific application models and the fact that models should facilitate the understanding, implementation and maintenance of the application being developed. Amongst the various techniques introduced to manage the complexity of the application development process is Domain-driven design [EVAN03]. A short overview of other methodologies and techniques is found later in this report.

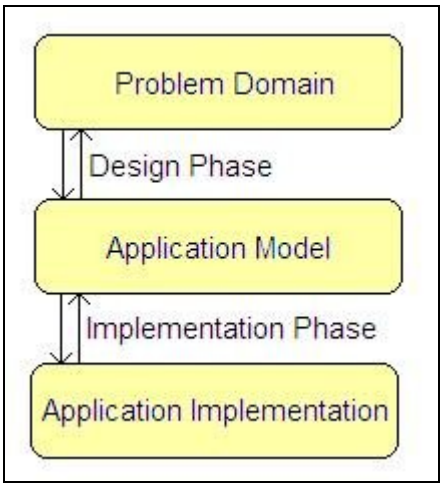

Figure 1: The development process

Understanding of the terms used in the *development process* seen in *Figure 1* is vital to provide a clear description of DDD.
At the top level is the *problem domain*, which initially encompasses only a rough idea on what the final application should accomplish. This rough idea is refined by producing a

document describing the set of requirements. This requirements specification is needed before the *design phase* is started.
The *design phase* transforms these requirements into the *application model* that meets the defined requirements. During this phase the requirements are adjusted according to insights gained by producing the *application model*.
The second level of abstraction consists of the specification of the *application model* that is used to represent the problem domain.
During the *implementation phase* an implementation is produced with functionality that corresponds with the *application model*.
The third level of abstraction is the *application implementation*, which represents the code structure and behaviour of the actual implementation.

DDD originates from the notion that the heart of the complexity of software development lies within the problem domain itself. DDD presents guidelines for applying separation of concerns to an application model and implementation. This separation aims to move the focus in software projects to the deep issues of the problem domain, namely the domain logic [DDDO] [EVAN03] [SDWZ07]. The domain logic is also commonly referred to as business logic.
During the design phase, collaboration between developers and domain experts is needed to attain the goal of creating a coherent application model expressed in a ubiquitous language defined by the team itself [EVAN03 p.32]. In DDD, ubiquitous language means ensuring the domain experts and software specialists communicate in the same common language, thus enabling efficient and correct communications between these team members.
In DDD the transformation from the problem domain into an application model results in a specific layout in the application model. At this level the application model consists of a domain model, containing domain-related functionality, and domain-independent functionality. Domain-independent functionality is represented by different services that facilitate the usability of the domain model. Therefore, the term 'service' has a special meaning in DDD when compared to the use of this term in other architectures like service-oriented architecture (SOA), as services represent domain-independent functionality.
In the application implementation structure the separation between domain-independent (service implementation) and domain-related functionality (domain implementation) is maintained. On the implementation level this means the classes in the domain implementation have no references to the classes and interfaces in the service implementations.

Both transformations during the design and implementation phase are conducted in an iterative manner. Changes made to the application implementation structure are also reflected in the application model.

DDD introduces a level of robustness to requirement changes. Requirements changes are far less likely to impact the domain logic. An example of this robustness is seen in a banking application. A requirements change is unlikely to affect the domain structure, consisting of accounts and how transfers between accounts are made.

Services however, are more likely to be the target of requirement changes. An example is a new web-based front end that allows access to the domain implementation or another type of database that is used to store the state of the domain implementation.

1.2 Problem Description

In some cases, the separation of concerns prescribed by DDD is difficult to achieve, namely when functionality is considered which is domain-independent but which is intrusive on and closely related to domain-related functionality.
These problems apply to security. Security is conceptually domain-independent functionality, thus security should be modelled as a service in DDD. But security is hard to separate from domain-related functionality in the implementation.
For the security service to function properly it needs to be intrusive on domain-related functionality.

Two main problems arise when separating security into a service:

1. The security service must authorise actions that affect the implementation of domain classes. This applies when these actions are initiated by other services. The security service must be aware of these actions and be able to regulate them.
2. When reviewing security as a service within DDD, it is clear that when access control is only based on the role of a user, security functionality can be located in a security service with low coupling to the domain implementation.
   However, when the state of the domain implementation affects access control, decoupling security functionality becomes more problematic. An example of this problem is a domain concerning contract management. A user must have the role of manager in order to close a contract, this constraint can be enforced by the security service.
   However, if more complicated conditions need to be checked, like 'Hiring contracts can only be closed by HR-managers and sales contracts may only be closed by Sales-managers', the security service needs domain knowledge and could be required to retrieve information, from the domain implementation to enforce the security rules. Such a service is hard to decouple from the domain implementation it secures.

From these problems the question arises whether it is feasible to separate security into a service and which possibilities exist to implement this service. Strong coupling between a service and the domain implementation contradicts the aim of DDD to have a low coupling between services and the domain implementation, while maintaining high cohesion between the objects inside the domain implementation or a service. Regular solutions for the problem of strong coupling like using events or observer and observable pattern are not feasible in the case where security is simply modelled as a service, because the security service needs to authenticate users and authorise the actions from other services before they can interact with the domain implementation.

1.3 Goals

This Masters assignment investigates how to handle issues concerning the modelling and implementation of authorization and authentication functionality as a security service in a DDD environment. The separation of security into domain-independent functionality should be performed in such a manner that the benefits of DDD are maintained.
For this Masters assignment the scope of security (containing audit, privacy, authentication, trust, authorization etc.) is narrowed to the authentication and authorization of users.

The first goal is to provide insight into the coupling of services in DDD and how services are classified according to their behaviour.
The second goal is to find the best design options to separate security functionality from domain-related functionality. The best design options are tested by creating a prototype implementation. Prototype implementations should comply with the principles of DDD while still providing usability and efficiency when implementing an application.
If multiple design options are found, the best design options must be implemented.
Evaluation of the design options is performed by reviewing the prototype implementations that demonstrate the possibilities for assigning authentication and authorisation functionality to a security service decoupled from the domain implementation.
For these prototypes, guidelines are presented describing which implementation is applicable for which situation.

1.4 Approach

The approach to reach the goals for this Masters assignment consists of four phases: investigation, design, implementation and evaluation.

To clarify how DDD is applied in practice, we have investigated the terminology, inner workings and benefits of DDD.
The knowledge gained from the investigation is used to describe a problem domain for a case study. This case study has been used to compare the different design options.
The case study illustrates how DDD works in a practical environment. The case study has also been used as a starting point to be able to compare new design and implementation options.
From this case study, requirements for the design options have been derived.

In the design phase, possible design options to cope with the security concerns identified in the case study have been derived. The design options describe the composition of the application model. The application model describes how the domain-related and domain-independent functionality cooperate. This phase has been partly a creative process as the number of possible design options was not known in advance. Each design option has specific benefits and concerns because of different manners of cooperation between the domain-related and domain-independent functionality in the design.

The design phase is followed by the implementation phase. The implementation phase consists of two parts.

1. If there were more than three design options, the number has been reduced based on the ranking of the design options. This ranking has been performed by using criteria specified at the introduction of the case study. It was not feasible to implement more than three variants of this case in the limited time of this assignment.
2. The best design options were implemented for the case study described in the investigation phase.

After the implementation phase, the implemented case studies were evaluated. The different implementations have been compared on the compliance with the DDD target situation in the case study as presented in the investigation. Measurements to determine the complexity and clarity of the design and implementation experiences are also factored into the evaluation. The evaluation phase also contains a comparison between the different implementations by measuring the coupling between the security service and the domain implementation.
Because each implementation solves the problems in a different way, the evaluation could result in different prototypes options being recommended in different situations and environments.

1.5 Report structure

The report structure reflects the four phases from the approach: investigation, design, implementation and evaluation.

Chapter 2 describes a more detailed view of DDD. It explains the origin of DDD and its relation to some other development techniques. It elaborates on the coupling between the domain implementation and service implementations. It also contains an analysis of the problems that occur when separating security into a service is presented.

Chapter 3 presents a case study that is used to clarify the implementation phase of the development process. A reference implementation of the case is presented to illustrate the usage of this case study. Criteria for the design options are defined based on the case study.

Chapter 4 provides an overview of design options that can be used to structure the application model. Differences in the design options occur due to use of different techniques or due to a different choice for separation between domain-related and domain-independent functionality. These design options are treated as different scenarios, each with individual characteristics and trade-offs.

Chapter 5 combines the case from *Chapter 2* with the design options presented in *Chapter 4*. This chapter describes the implementation phase and identifies possibilities and problems with the design options. Before creating implementing the prototype

implementations, a selection is made in case more than three design options are defined in *Chapter 4*.

Chapter 6 evaluates the measurements performed on the prototype implementations from *Chapter 5*. The case study implementations are compared to the reference implementation from *Chapter 3*. The results of these measurements and comparison is analysed and the relations between the results are explained. Based on these results guidelines for the usage of the scenarios are presented.

Chapter 7 contains the conclusion. This chapter presents a summary of the results of the Masters assignment. A reflection on the total development process (investigation, design, implementation and evaluation) is given in this chapter.

Chapter 2
Domain-driven design and Security

This chapter presents the background and terminology used in the application of DDD. This chapter's role is twofold. In the first place this chapter provides detailed information about Domain-driven design and how it is applied. The second role is to analyse the current problem with separating security into a service.

2.1 DDD

The content of this section is based on various DDD sources. The references for the content are [DDDO] [EVAN03] [EVAN06] [SDWZ07] [WOLT05] [NILS06] [YDDD].

2.1.1 Principles

Application development is a process that becomes increasingly complex depending on the intricacy of the application being developed. Amongst the various techniques introduced to manage the complexity of the application development process is Domain-driven design.

To more efficiently deal with requirements changes, complexity in the application needs to be manageable. According to [BROO86], essential and accidental complexities exist. Brooks argues that while essential complexity cannot be reduced because it is inherent to the problem being solved, accidental complexity can be managed by design and implementation choices. To manage the accidental complexity in the application model, DDD uses the principle of separation of concerns. This principle is used to separate the application model into a *domain model*, consisting of domain-related functionality, and domain-independent functionality, which is represented by different *services* that facilitate the usability of the domain model. Therefore, the term service has a special meaning in DDD, when compared to the use of this term in other architectures like service-oriented architecture (SOA). In DDD services represent domain-independent functionality.
The separation of domain-related and domain-independent functionality in DDD is based on the idea of applying separation of concerns to software development, which dates back to the early 1970s [PARN72] [DIJK74].
At its core, DDD can be characterized as a specific style of applying OO programming. The principle of separation of concerns is also related to the foundation of Object-Oriented (OO) programming [WBMC03]. In the late 1970s OO was applied in Smalltalk [SMAL08] by considering that "everything is an object", in this way associating properties and behaviour to individual classes.

During the design phase, collaboration between developers and domain experts is needed. To ensure that domain experts and software specialists communicate in the

same terminology, a ubiquitous language is defined by the team. Ubiquitous language in DDD stands for getting all team members, software specialists and domain experts, to use the common language in diagrams, writing and speech. This ubiquitous language enables the creation of a clear and understandable application model for all team members.

In DDD, the key to controlling the complexity of the application model is to use a correct and complete domain model. The creation of this correct and complete model is the objective in DDD during the design phase. *Correct* means that the team members are satisfied with the functionality that is incorporated and *complete* means that all domain-related functionality is present in the domain model. Complete does not mean that the domain model is a fixed, rigid or finished product after the design phase. Changes introduced by decisions during the development phase are reflected in the domain model. This means that the domain model remains centred and consistent with the domain implementation during the entire development process.
Although not required, DDD is perfectly suited for performing the design and implementation phases in an iterative fashion.

Applying the domain model correctly should result in a complex development effort becoming more dynamic and more focused on the core issues of the problem domain. This focus on the domain models results in increased separation of concerns in the application model, illustrated by the loose coupling of services to the domain model. This increased separation yields an application implementation that is more flexible, making it easier to facilitate the addition of new requirements or features.
The aforementioned separation of domain-independent functionality into services is an attempt to ensure the independence of the domain model from technology. Technology means specific versions or types of databases or different UIs. Especially services like UIs are susceptible to changes in technology and requirements.

Determining the separation between domain-independent functionality and domain-related functionality is not a trivial task. The decision can be difficult because a lot of functionality does not fit clearly into either category.
Deciding where functionality should be placed in the model is based on knowledge, experience and common sense.
Thus, whether a specific piece of functionality should reside in a service or in the domain thus varies, depending on the person making the decision.

2.1.2 Models

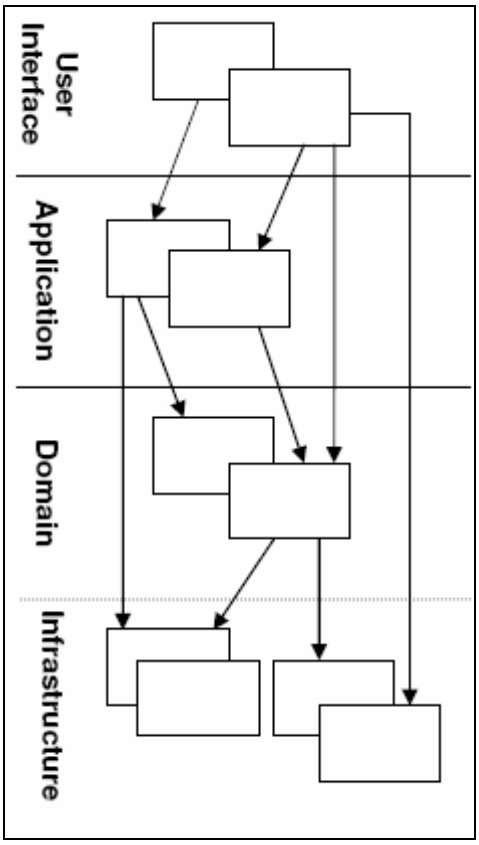

Figure 2: DDD Layer model by Eric Evans [EVAN03 p.68] [EVAN06 p. 29]

In [EVAN03] a layered application model is used to show how DDD can be projected onto a layered model (see *Figure 2*). The goal of this layered model is to concentrate the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code. Services can be present in any of these three layers.
Each of these layers has a special purpose. In contrast to regular layered models where layers are only allowed to communicate with the layers directly above and below, the layers in this model are allowed to interact in the direction of the arrows without intervention of the intermediate layers. The arrows show the direction of the interactions between the different components in the model. This means that the components in the Domain layer are unaware of the existence of the layers above the Domain layer and have no references to those components.

The four layers identified in *Figure 2* are:

- The User Interface (or Presentation) layer is responsible for showing information to the user and interpreting user commands.
- The Application layer is an optional layer that helps translate actions in order to use the Domain layer. It does not contain any business rules or knowledge. This layer is kept thin and has no state.
- The Domain layer represents the domain implementation. It contains the concept of business information, business rules and the state of the current business situation.
- The Infrastructure layer provides generic technical capabilities that support the higher layers. Examples are data persistency and message exchange.

The layered model is unable to represent more services clearly than are currently depicted. New services would be placed in the Infrastructure layer or User Interface layer. More complex designs lose a lot of clarity when depicted in this layer model, because different services would reside in one layer while their function could vary distinctively.

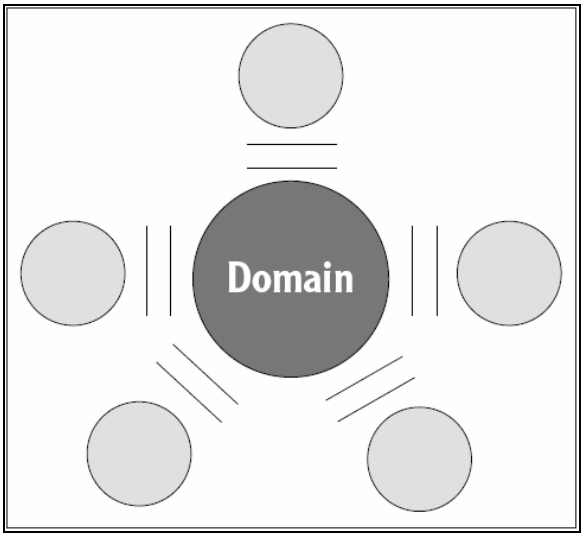

Figure 3: DDD application model by Sogyo

Within Sogyo the aim is to use a so-called 'sunflower' model, as depicted in *Figure 3*, where the domain-related functionality is unaware of the services outside the domain model. In contrast to the layer model in *Figure 2* the domain implementation is also unaware of the services in the infrastructure layer.
The sunflower model depicts the same elements organised in another way, where all elements can be mapped to the layered model. The difference is that the separate services are not depicted in one layer but in separate entities surrounding the domain model and the domain model is unaware of the elements from the infrastructure layer.

The domain model is depicted in the centre of the application model to stress its importance. The various services are depicted around the domain model. Choices are made during implementation whether the services have references to surrounding services and the domain classes. There is no clear consensus in the DDD community on whether services are allowed to only have references to other services, thus not knowing the implementation of domain classes.
The domain model remains the foundation of the design. The domain implementation does not require any service to operate, the domain implementation is independent and is able to operate standalone. Operating standalone has the advantage that for example no persistency service is required to test and run the domain.
The double lines between the services and the domain model represent a 'glue' layer. This glue layer is used to translate actions in order to use the domain classes. The glue layer it is considered equivalent to the application layer in *Figure 2*.

2.1.3 Goals and benefits

Domain-driven design is not a goal in itself, but aims to provide the means to facilitate easier maintenance, speed up development, and add flexibility and robustness to requirement changes. The reasons behind these positive effects are related to the separation between the domain model and the services that facilitate the use of the domain model. During development, requirement changes are far less likely to impact the domain model. New functionality is mostly contained in a few methods to be added to the implemented domain classes. Services are more likely to need new functionality and entire new classes or even a new service are introduced to provide new functionality. Since changes to services impact a smaller part of the total development process, this results in a decrease of the total impact of a requirement change.
Changes to business rules are more easily introduced because they are centralized in the domain model. This centralisation inherently introduces a drawback, since changes in the domain implementation can have an impact on all service implementations. All services that interact with the changed domain class have to be checked for correctness.

Thus changes to the domain implementation can propagate throughout the services and have a large potential impact.
Another benefit of applying the DDD principles to a project is increased testability of the domain implementation. Increased testability is achieved because the domain implementation operates independent from services. This independence facilitates the testing of methods with unit tests. The independence from services also eliminates the need for mock databases.

2.2 State of the art

This section describes technologies and techniques related to DDD and techniques that are used within this thesis. Different development methodologies that are related to DDD are discussed to be able to place DDD better within its context. Furthermore, implementation techniques and measurement metrics that are relevant to this thesis are clarified.

2.2.1 Development methodologies

DDD is not the only development technique or methodology that is defined to manage and control the complexity of the development process. Examples of other related techniques include Model-Driven Architecture (MDA) [MDA04] [HAYW04], Model-Driven Development (MDD) [IBMR06] and Table-Driven Design (TDD) [FOWL02]. These techniques are shortly described here to indicate how they work and how they influence the development process. These techniques are discussed because they are related to DDD.
MDA is the Object Management Group (OMG) [OMG08] implementation of MDD. The MDA concept is implemented by a set of tools and standards that can be used within an MDD approach to software development. MDD describes concepts related to the modelling of an application that are also applicable within DDD. However, in DDD the model is closer to the code than in MDD.
TDD is a domain logic pattern described by Martin Fowler in [FOWL02]. TDD is used in applications that are centred around databases and with data table representations that are easily mappable on business objects.

Model-Driven Architecture and Model-Driven Development
Model-Driven Architecture (MDA) describes a set of viewpoints for defining models, such as the Platform-Independent Model (PIM) and Platform-Specific Model (PSM). Models from these viewpoints can be packaged as reusable assets. If there is a situation in which some recurring business concepts can be applied across multiple applications and implementation technologies, it can be valuable to invest the effort to create the PIM and PSM models and any associated transformations, each of which may be a reusable asset and stored in an asset repository [IBMR06](p541).
Objectives of MDA include separating business requirements and analysis from technology, by introducing the PIM and PSM. Another objective of MDA is to put the focus back on modelling. Ideally, if the model is detailed enough, only little time is spent in the actual implementation.

DDD and MDD are compatible. The main difference is that MDD is more aimed at the translation of models into code, while DDD is more aimed at defining a correct domain model. Within a MDD-based project there are no problems to apply the DDD principles. Model-Driven Development (MDD) is the approach corresponding with MDA, as MDD focuses on models as the primary artefacts. Model-Driven Development recognizes the necessity of having several kinds of models to represent the system as it progresses from early requirements through final implementation. These models may represent different aspects of the system (e.g., structural or behavioural), or they may represent the system at varying levels of abstraction (e.g., an analysis model or a design model) [DDMD].

Table-Driven Design
Table-Driven Design (TDD) organises business logic around objects that can be directly mapped to data tables. This method is applied in the current Sun blueprint for EJB implementations.
The application of TDD addresses the problems that occur when mapping objects in OO models to relational databases. In these situations an O/R-mapping is needed to persist the business objects. The retrieval and instantiation of these objects from a database is a fault prone task in an object oriented mindset [ORM97]. Despite improvements made concerning databases that can handle objects, these solutions are still not fully satisfactory, as described in [COOK06].
Table-Driven Design is built around table representations of business objects. For each type of object, a table is available for storage, which is a flat and simplistic representation of a real world business entity. Objects like datasets are used to exchange information back and forth between UIs and databases [SDWZ07]. Instead of the dataset objects, an active record pattern can be used to link the objects to corresponding tables. This pattern places the data access logic in the objects themselves [FOWL02].

2.2.2 Implementation techniques

Within this Masters assignment the concept of Aspect-Oriented Programming (AOP) [YASE07] is used. This is an important aspect of some design options in this Masters assignment.

Aspect-Oriented Programming
Aspect-Oriented Programming (AOP) is a concept to deal with cross-cutting concerns efficiently. Cross-cutting concerns are functionality of a program that affects other functionality [KICZ96]. These concerns often cannot be cleanly separated from the rest of the functionality.
To deal with cross-cutting concerns, AOP defines aspects, advices, join points and point-cuts [YASE07] [LADD02].

- Aspects represent a specific piece of functionality that was identified as a cross-cutting concern.
- The aspect consists of advices that define what actions should be executed in specific situations, like a method entry or method exit.
- Join points represent locations, like method entries or property declarations, in the source code.

- Point-cuts exist to link the join points to the advices that should be executed. Point-cuts define to which join points a specific advice should be applied.

This structure enables AOP to modularize the cross-cutting functionality in aspects. According to [EADD07] this modularisation of cross-cutting concerns improves software quality.
AOP is applied to the source code by weaving the aspects into the source code. The programming language is important for weaving because special compilers are required to weave AOP aspect code into source code. Examples of these weavers are: AspectJ [ASPJ] for Java [JAVA08] or PostSharp [POST] for .NET [MSFT08].
Other .NET options the Policy Injection Application Block [PIAB07] in Enterprise Library 3, Aspect.NET and LOOM.NET.

The choice for PostSharp over the other available .NET AOP implementations is based on recent research into applicability of AOP in the .NET environment by [SRIN07]. In this research, PostSharp is recommended over Aspect.NET and LOOM.NET.
A drawback of PostSharp is that point-cuts are declared at the target location, this means that the point-cuts would reside in the domain implementation if domain classes are targeted. This is unwanted behaviour in DDD because then a dependency on the aspects is introduced in the domain implementation.

2.2.3 Metrics for measurements

In this Masters assignment, metrics are used to measure coupling and complexity in implementations.

Coupling Between Objects
The Coupling Between Objects (CBO) [CHID94] metric is frequently used to measure the amount of coupling between object classes. An object is coupled to another object if the object uses the other object.

A higher CBO value between classes means a stronger coupling. A stronger coupling is not beneficial for changes in the development phase, since with stronger coupling changes in a class have more impact on other classes. A class with a high CBO figure is more difficult to maintain and test. Stronger coupling also reduces the possibility for code reuse.

Coupling consists of two factors, namely fan-in and fan-out:

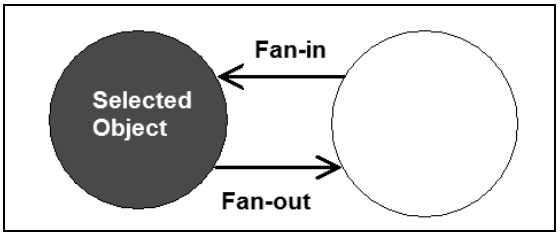

Figure 4: Coupling Between Objects

- **Fan-in** indicates how many references other objects have to the selected object.
- **Fan-out** indicates how many references the selected object has to another object.

The CBO is calculated by counting of the number of references in the code. This count is meaningful under the assumption that all references are actually used when the system executes. Another important factor is how often the reference is actually used during program execution. This can not be measured with CBO, because this CBO is calculated based on the static environment.

When measuring the fan-in and fan-out of larger applications, counting the references can be performed on a selected group of objects. Since a service is a collection of classes with an interface, instead of measuring the coupling between two objects at a time, the coupling of a whole service to the domain classes can be measured.

Cyclomatic Complexity
Cyclomatic Complexity (CC) [CCMB89] is a method to measure the complexity of software. The CC value of a programming method is measured by counting the number of possible paths through the method. This means the CC of a class is measured by counting the number of methods and then adding the CC value of each method.

The higher the CC value the more complex the method is. A complex method is potentially harder to understand, harder to test and has higher maintenance costs.
A method with a lower CC value is easier to test because there are fewer possible paths through the method, which means fewer test cases are needed.

The CC measurement yields three values:
- Total Cyclomatic Complexity
- Cyclomatic Complexity per method
- Number of Decision Points (DP)

Because the total CC value is less suitable for comparison between implementations with largely varying numbers of classes, the CC per method value is also used in this thesis. The CC per method value presents the average CC per method in the measured class. CC value displayed as an average per method provides a better interpretation of how complex a class implementation actually is than the total CC measurement.

Decision points represent the number of expressions in the source code where different paths are possible. In the DP value the ‘while’, ‘for’ and ‘for each’ loops count for 2 decision points, because these cycles make the code more difficult to understand. The DP value indicates the number of paths that can be taken in the code and thus the actual complexity of a class. For example a class with 1 method containing 20 decision points is harder to understand than a class with 10 methods containing 2 decision points per method.

2.3 Coupling of services

In order to understand why security poses difficulties, it is necessary to identify different types of services that can occur in a DDD application model. When examining services

more closely, considering the type of coupling with the domain implementation, the services can be divided into three categories:

1. A service that queries the domain implementation
2. A service that alters the state of the domain implementation
3. A service that requires the domain implementation to interact with this service

This means there are three fundamentally different types of services that interact with the domain implementation.

Differences between these types of services can be found by defining the amount and type of coupling between the service and the domain implementation.

2.3.1 A service that queries the domain implementation

If a service only reads information from the domain classes, interactions with the domain classes take place that do not alter the state of the domain classes.
Examples of services that display this behaviour are persistency and logging. In DDD the persistency and logging services take the initiative to retrieve this information from the domain implementation after they have been notified. This behaviour is facilitated by the observer and observable pattern, where the domain classes implement the observable role and the service the observer role. Another option is that a service responds to events originating from the domain classes.
The coupling between a service that only queries the domain implementation and the domain classes consists of an amount of fan-in to the domain classes. The fan-in consists of the references needed for the retrieval of information by the services.

2.3.2 A service that alters the state of the domain implementation

This type of service consists of services that call methods in the domain. Methods called in the domain classes can alter the state of the domain implementation or values contained by classes. By changing the state of the domain implementation, reactions in other services inside the application can be triggered like the persistency service.
A typical example for this type of service is a user interface.
A user interface either responds to events originating from the domain classes by updating views, or initiates interactions with the domain classes.
The coupling between a service that alters the state of the domain implementation and the domain classes consists of an amount of fan-in to the domain classes. This fan-in consists of the references needed for the retrieval of information and the alteration of the domain state by the services.

2.3.3 A service that requires the domain implementation to interact with this service

This type of service contains information that is required by the domain classes before actions are executed by these domain classes.
These services exist whenever the domain class requires verification of input data (input validation) or permission to execute methods. Essential information from this service is needed in order to continue execution.

A service that displays this behaviour is a security service. Input from services that interact with the domain classes, like user interfaces, requires checking before the actions are executed by the domain classes.
Other services, like a user interface, initiate an action. Then the domain classes have to check the permission for this action before executing the action, which means that the user interface has delegated the initiative for permission checking to the domain classes. This results in the domain classes having to consult a service and producing fan-out from the domain classes.
The coupling between a service that requires the domain implementation to interact with this service and the domain classes consist of both fan-in and fan-out.

2.4 Security service

Implementations designed with security as a normal service in DDD suffer from the problems described in section *1.2*.
These problems are specific for the interactive type of service security belongs to, as discussed in section *2.3.3*. The result is that the security service has a strong coupling to the domain classes and that the domain classes need to be aware of the existence of the security service.

The scope of what encompasses security has, for the purposes of this Masters assignment, been narrowed to authentication and authorisation.

- Authentication consists of proving that the user is a specific person (or role), who is allowed to access functionality in the application.
- Authorisation consists of proving that the specific person (or role), who has been authenticated, is authorised to access specific functionality in the application.

The function of the security service is to regulate actions performed by other services by implementing authentication and authorisation functionality. Especially interactions with either the domain classes or other services originating from user interfaces must be checked. If the security service is placed on the same level as a normal service in DDD, the security service is unable to perform its task properly.
This is the case because in DDD services are coupled to the domain classes, but domain classes have no references to the services. However, services cannot be trusted to perform the security checks themselves. In a default DDD situation the security service does not know about the interactions between the user interface and the domain classes.

To perform authentication, the security service needs to receive credentials and check these credentials against stored credentials. Authentication is a crosscutting concern because every time authorisation is needed, the authentication of the issuer has to be checked. Authentication is either performed only once, after which a session bound object is created to store the login, or it is performed each time an authorisation is needed. In either case, the authentication is checked at every authorisation.

Authorisation is a crosscutting concern since multiple methods in different services and the domain need to be checked for authorisation. Two different kinds of checks can be

performed during authorisation: check if an interaction is allowed for this role or user, and check if an interaction is allowed under certain conditions for this role or user. These conditions are usually related to the state of the domain classes. If an interaction is allowed, some filtering might still need to be applied to the results based on the authorisation of the user.

The type of the security service, as discussed in section *2.3*, indicates that it is difficult to separate security from the domain implementation in the way other types of services are separated. The crosscutting nature of authentication and authorisation of security add up to the problem resulting in a service with high coupling to the domain implementation concerning both fan-in and fan-out.

Chapter 3
Case Study

This chapter presents the design and a reference implementation of the case study. This chapter also provides the criteria to determine the usability of new designs to implement this case study.
The role of this chapter is to present the case study, in order to compare new design options and review these possible improvements.

3.1 Case study overview

To describe the problems and possible solutions, the 'Bank case study' [BANK07] is chosen because this is a case where multiple levels of security are easily and naturally represented. In this case study the effect of adding security functionality is clearly visible and easily comprehendible.
A case study is needed to illustrate the concepts behind the separation of security as a service. The reference implementation of the case study provides the possibility to measure the amount of coupling introduced by the security service.
The reference implementation of the case study has the security functionality built into the domain classes. This implementation will be used as a reference implementation and provides reference measurements that can be compared to new implementations of the case study.
The design of the reference implementation is provided to be able to compare new designs with the reference design. Because at the design phase only the application model exists and no actual code is available, only estimations of the coupling between different objects are available.

The bank case study models how a bank is run and how clients and managers can interact with the accounts. Clients and managers each have their own specific possibilities and restrictions.
The bank case study allows clients and managers to perform actions with certain limitations, as shown in *Table 1*. Clients are allowed to manage their own accounts up to certain limits. Clients can withdraw up to 1000 euros a day, deposit an unlimited amount of money and transfer money up to 1000 euros a day. Clients are also allowed to view their own account status.
Managers can perform the same actions as a client, but they are also allowed to create clients and accounts. Actions of a manager are also restricted based on their role as a Junior or Senior manager.

Roles / Rights	Client	Junior Manager	Senior Manager
Deposit	Unlimited	500/Client/day	2000/Client/day
Withdraw	1000/day	2000/Client/day	5000/Client/day
Transfer	1000/day	2000/Client/day	5000/Client/day
Add client	Not Allowed	Not Allowed	All clients
Alter client name	Not Allowed	Not Allowed	All clients
Alter client address	Own address	All clients	All clients
Create account	Not Allowed	For clients that are less than 1000 euros in dept	All clients
View client details	Own details	All clients	All clients

Table 1: Roles and rights in the bank case

Both managers and clients interact with the domain implementation via the same user interface. The difference between these roles in the user interface is made by the credentials used to login.

The bank case study is designed according to a DDD application model. The domain model consists of the interfaces IBank, IClient, IAccount, ITransaction. The domain implementation contains these interfaces and classes that implement these interfaces. The Bank class implementation contains lists of clients and accounts and class is able to create new clients and accounts. The Client class implementation contains information about a client and related accounts. The Account class implementation contains the account number, owner, current balance and a collection of all transactions performed related to the account. The Transaction class implementation contains information about a transaction. The interfaces and their relations are depicted in *Figure 5*. An implementation of the bank case study in C# was provided by Sogyo and is the basis for the case study implementations discussed in this report.

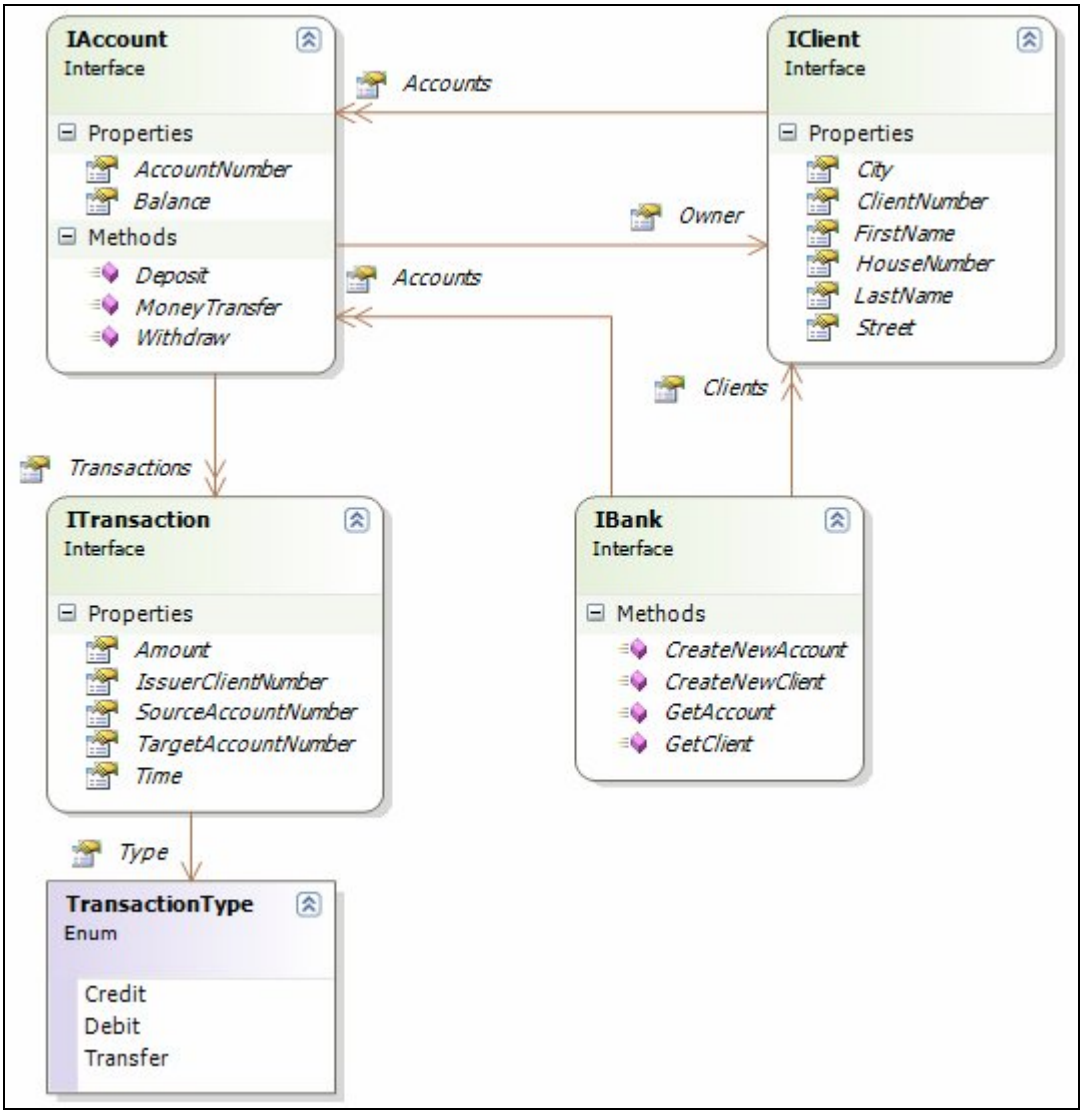

Figure 5: The relations between the interfaces in the domain model in the Bank case study

Outside of the domain model, the bank case study contains services. A repository containing references to the Bank class implementation and to services is available to provide access to the domain objects and services. The application model contains a UI service that provides accessibility to a user. In the application model, multiple instances of the UI service can be coupled to the domain model to allow multiple users to use the bank concurrently. Another service in the application model is an IO service, which performs batch file processing. This service allows the user to execute a series of actions, which can consist of all the possible actions of the current user. The IO service also contains classes that can create reports about the state of the domain model, containing client and account information. The IO service thus allows users to check the current amount of money on their account.

3.2 Reference implementation: Security embedded in the domain model

Currently when security is applied to a problem domain, the security functionality is integrated into the domain model as seen in *Figure 6*. This implementation structure with security functionality in the domain implementation is often found in projects at Sogyo.

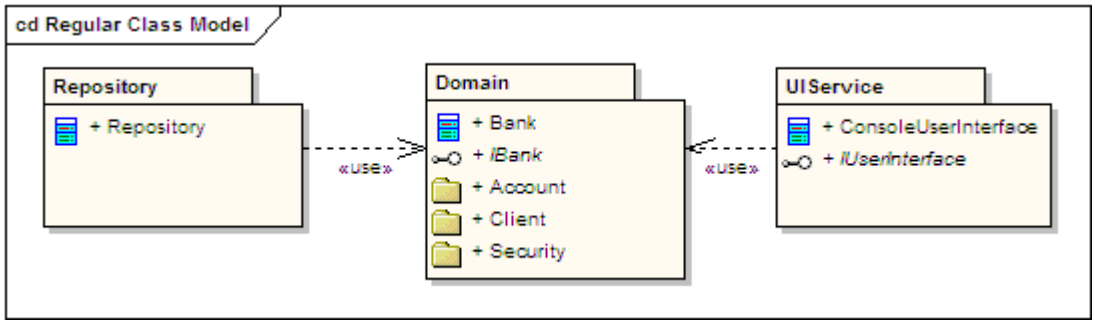

Figure 6: Class diagram of the application of security in a DDD application

The implementation only provides accessibility for one concurrent instance of the UI service, so the ability to facilitate concurrent UI service instances is only visible in the model.

For the implementation, Role Based Access Control (RBAC) is used because this is a better solution than authorising individual users. RBAC is preferred over authorisation of individual users because it is easier to configure and maintain [COMP03].

3.2.1 Usability and Features

Due to the integration of the security functionality in the domain implementation, the security functionality is only consulted in the domain model when information is being accessed that needs authorisation. Authentication and authorisation functionality are both centralised in the domain model and are checked by the domain classes when needed.

In this scenario, the UI communicates directly with the domain model and no restrictions are imposed on the methods it can access.
The domain class implementation is responsible for checking the UI service's authorisation. An example of the security check in the Client class implementation is shown in *Code Fragment 1*, here the authentication of the user is checked to determine whether the user requesting the information is authorised to retrieve it. In this example the security implementation has already stored the user's credentials in a User object. The individual methods check the authorisation level corresponding with the method call with the security functionality in the domain implementation.

```
public override string ToString()
{
    if (!security.user.RoleSet.IsPrintClientDetailsPermitted(this))
    {
        security.GenerateSecurityException(
                "You are not allowed to view this clients details.");
    }
    return String.Format("{0} {1} {2} - {3} {4}, {5}",
                this.clientNumber, this.firstName, this.lastName,
                this.street, this.houseNumber, this.city);
}
```

Code Fragment 1: Security check in the Client implementation

Code Fragment 1 is executed when the client's ToString() method is called. First the authorisation of the current user is checked. The current user is retrieved in the security functionality and then the corresponding Roleset with the user's authentication is retrieved. According to the Roleset of the user this action is allowed or disallowed. In case the action is disallowed, the calling object receives an exception and the code execution in the client class is stopped. In case the action is allowed, the calling object receives a string representation of the client class. This behaviour is shown in the sequence diagram in *Figure 7*.

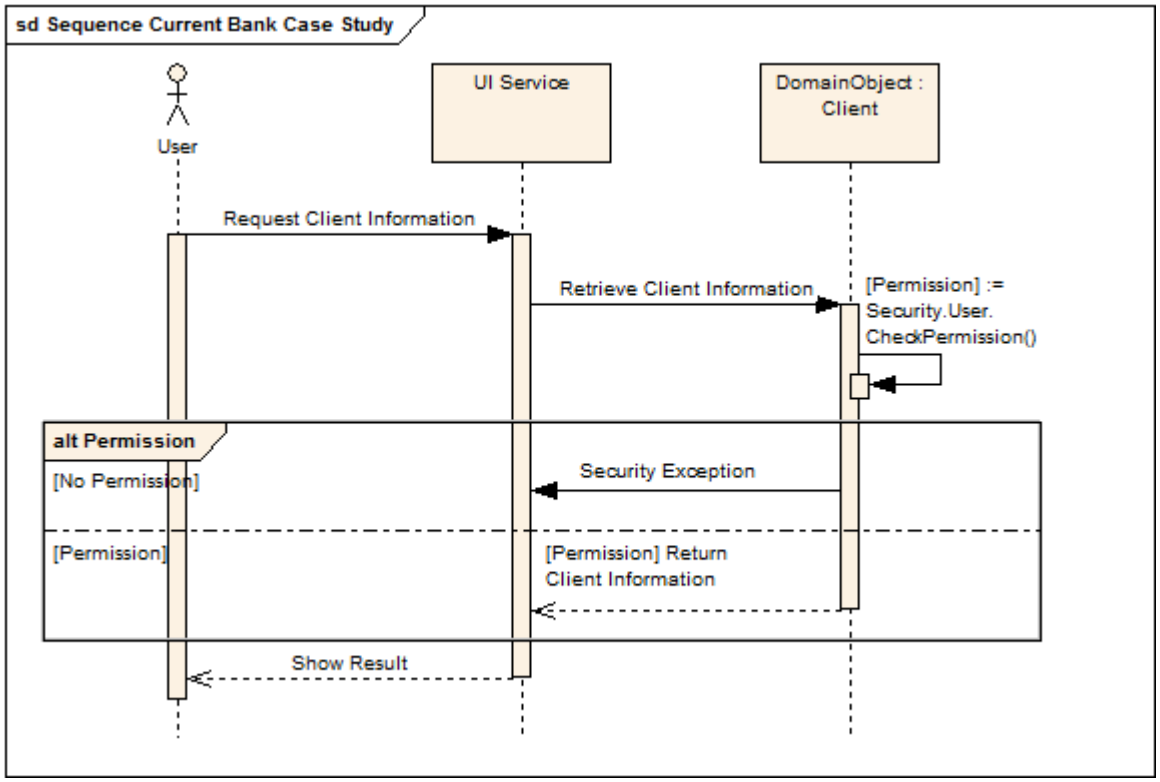

Figure 7: Sequence diagram of the security checks in the Client implementation

This high cohesion between the domain model and security is practical and efficient to use. Guards and access impairments to the domain methods are immediately clear to developers.

3.2.2 Concerns

The high cohesion between security and the domain model poses a problem for separation of concerns. The coupling results in security functionality that cannot be separated from the domain functionality in this scenario.

3.2.3 Conclusion

Within this scenario no problems arise concerning authentication and authorisation. This implementation does not apply the DDD idea, where security should be a service, as it is domain-independent functionality. This should be improved in new scenarios.

The coupling between security classes and the other domain classes is strong because of the cohesion between the individual methods in the Client, Bank and Account classes and the security functionality. The domain class implementations were found to have a fan-in of 10 references and a fan-out of 17 references to the security functionality, as the domain classes have a reference to the security functionality in almost every method.

3.3 Criteria for case study designs

Selection criteria have been defined to determine the usability of the scenario designs. These criteria are the design and implementation level goals that should be attained. To what extent the scenarios comply with these criteria indicates their usability as a solution.

A design option complies with the target DDD situation, in this case study, if it complies with these requirements:

- The security functionality is placed in a service with low coupling between the security service and domain implementation.
- The security functionality is implemented as a separate service and only fulfils the tasks of authentication and authorisation.
- The security service has knowledge and control over all interactions with the domain implementation.
- The security service and the other services are 'regular' services. A 'regular' service has access to both the domain implementation and the other services. This means that other services are still allowed to interact with the domain implementation and not only with the security service.

Six criteria have been defined. These criteria are used to judge the potential of designs. Criteria for the new case study designs are:

1. At the design level an estimation of the coupling in the application model introduced by the security implementation can be made. Without actual implementation of the classes, known metrics as described in [MOOD95], [CHID94] cannot be used because these metrics involve the analysis of source code. At the implementation level this coupling can be quantified by using the CBO measurements. The coupling is compared to the coupling generated by the security implementation in the scenario described in section 3.2.

2. The extent to which the scenario complies with the principles of separation of concerns. Is the security functionality separated from the domain implementation? Does the security functionality only handle the concerns of correct authentication and authorisation?
3. The usability and efficiency for developers. How much code is needed to implement the security? Can, a part of, this code be generated? The scalability of the solution proposed in the scenario should be considered here. In the bank case study this means the model can be effectively used in an environment containing multiple instances of the UI service.
4. The extent to which the scenario yields the same level of security as the design in section *3.2*. This level of security indicates whether the UI service has possibilities to bypass the authentication and authorisation functionality.
5. The degree to which the scenario complies with the target DDD situation as explained in this section and the DDD definition as presented in *Chapter 2*. If the security functionality is implemented in a service, does it follow the general service definition?
6. The transparency and logical structure of the idea behind the design, thus making it easy to understand. This indicates whether developers can easily understand and modify the code without breaking functionality. Usage of common patterns increases clarity in the application model. This also includes instrumentation, allowing the developer to accurately assess how the application works and the implications of changes in the source code.

Chapter 4
Security service design scenarios

This chapter presents the alternative scenarios developed to implement a security service in a DDD environment.
This chapter's role is to create a set of platform-independent models that describe the design alternatives to the problems faced when designing security as a service in DDD. The goal is to create security in a service while maintaining ease of use and authentication and authorisation correctness. In this chapter the assumption is made that the UI service is running on the computer of a user, and thus cannot be trusted.

If in this chapter a comparison is made with section *3.2*, this comparison is made between the design of the case study implementation described in section *3.2* and the design of the scenario.
The scenarios in this chapter are reviewed to determine the possibilities and features the model provides, but also to determine concerns about the restrictions introduced by the model structure. Finally a conclusion is made considering the usability of the design.

The scenario conclusion reflects on the design taking into account both the advantages and disadvantages of the design. The usability of the design indicates the usability for developers to work with the design is reviewed. The usability is determined by the compliance to the criteria that are summarized in section *3.3*.

4.1 Scenario 1:
Security service as regular service

This scenario presents the possibilities that are available to developers when security functionality is implemented as a regular service.
This means the security service only has knowledge about the services and the domain implementation and is not able to intercept communications or enforce rules on other services or on domain classes.

4.1.1 Usability and Features

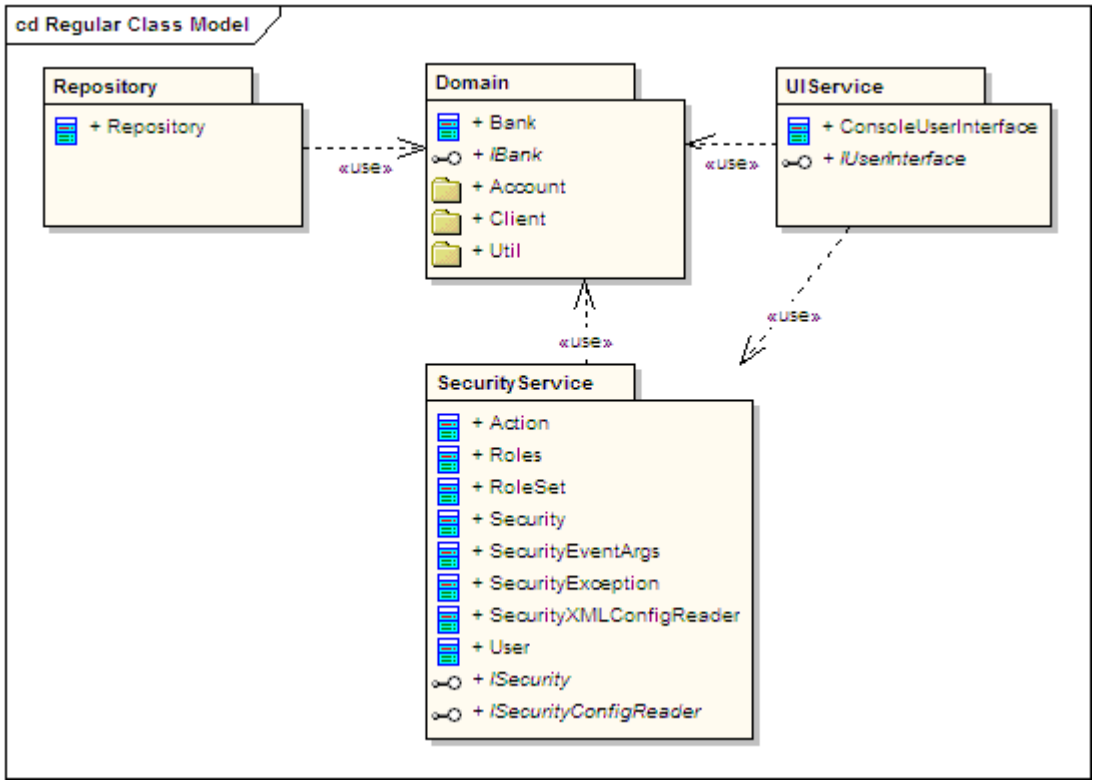

Figure 8: Class diagram of security as a regular service

In this scenario security is coupled to the domain model in the same way as a regular service. Separating the service resolves the issue concerning domain-integrated security. Because the security service is not able to impose rules on other services in this scenario, the UI service is allowed to interact with either the security service or directly with the domain classes.

The first part of security involves the authentication of the user. When interacting with the domain implementation the UI service first authenticates the user with the security service. The security service keeps track of the correct credentials for the UI. After authentication the UI service is able to request authorisation for actions.

The second part of security involves authorising actions of the user. The credentials for the user should be already available by authentication. The UI service is authorized according to its authentication level or roles. After authorisation the security service can perform the actions and return possible results or leave this interaction to the UI service.

If the UI service performs these actions the interactions with the domain classes is safe, this behaviour is shown in *Figure 9*.

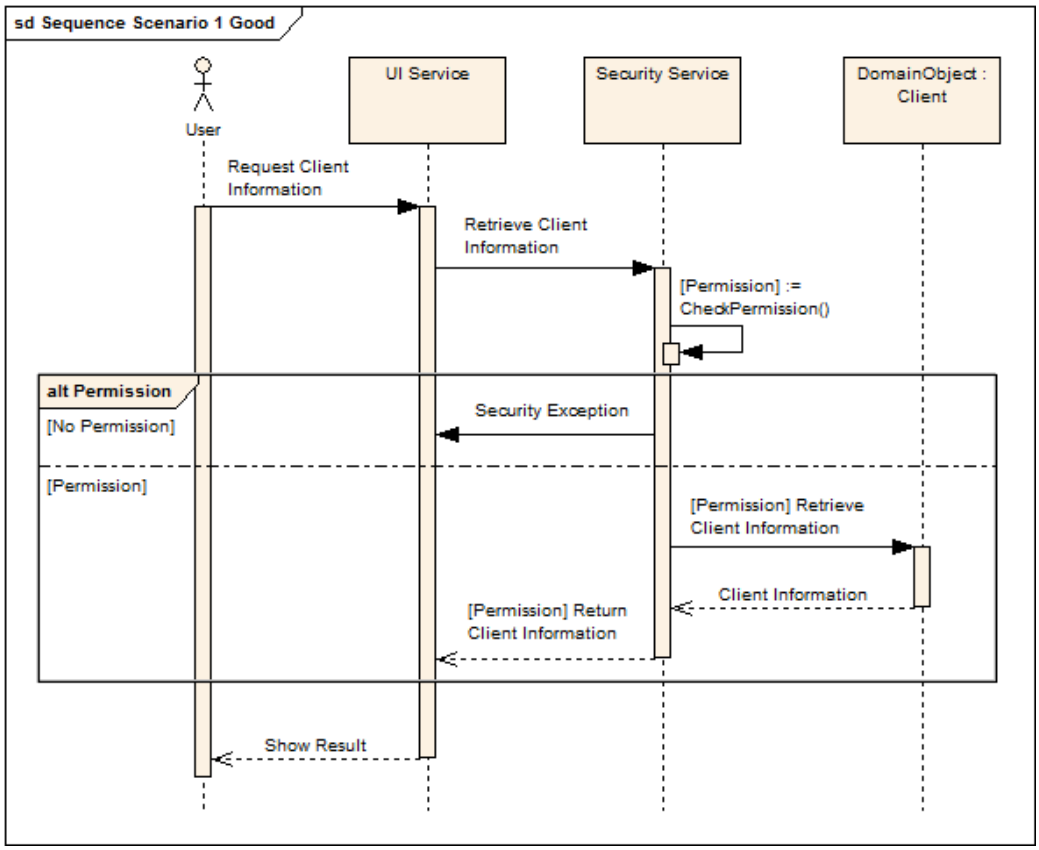

Figure 9: Sequence diagram of successful security checks in the Security service

If the UI service interacts with the domain classes directly problems occur because these interactions are not checked, this behaviour is shown in *Figure 10*.

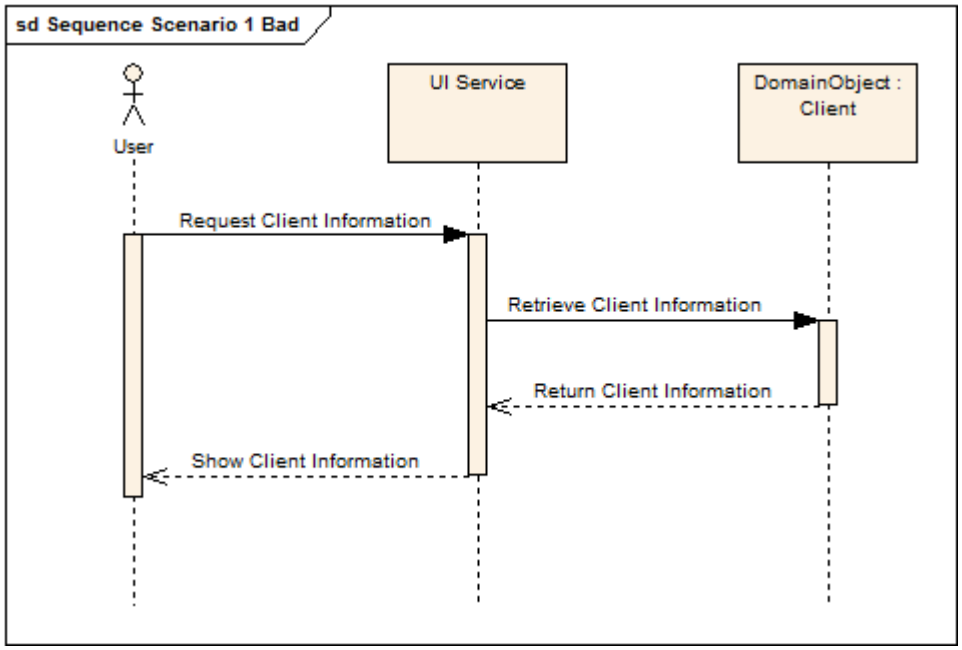

Figure 10: Sequence diagram of failed security checks in the Client implementation

4.1.2 Concerns

Because the security service is implemented as a regular service, it is unable to regulate all interactions performed by the UI service. The UI service is able to interact with the domain classes without the security service being aware of this interaction. This interaction is shown in *Figure 10*. The UI service accesses the domain class without the security functionality being aware of this interaction.
When the UI service interacts with the methods in domain classes a problem arises, namely the identity of the user behind the UI service is not verified by the domain classes. Here authentication is not performed. The same problems occur in authorisation. In some way the domain classes need to assure that the UI service has been authenticated and is authorised to perform the action. This represents a problem in the DDD environment. The authorisation of the UI service can not be checked if the domain model is not aware of any service.

If the UI service interacts with the domain classes directly the remaining options are insecure:

- The event raising method is unsafe because there are no guarantees about which service responds to these events, if any.
- If a method is reserved to be called by the security service, this rule can not be enforced by the domain classes.

4.1.3 Conclusion

This design results in a lower coupling between the domain model and security. But this change is obtained in an insecure way.

This scenario is not usable because the UI service cannot be trusted. The direct interaction between the UI service and the domain model allow the UI service to perform arbitrary actions.

4.2 Scenario 2: Security embedded in the UI

This scenario places the security functionality in the UI service. Moving the security functionality into the UI service could solve the problems with security seen in *Scenario 1*.
The UI service should consult the rules as specified in these security classes before starting an interaction with the domain implementation.

4.2.1 Usability and Features

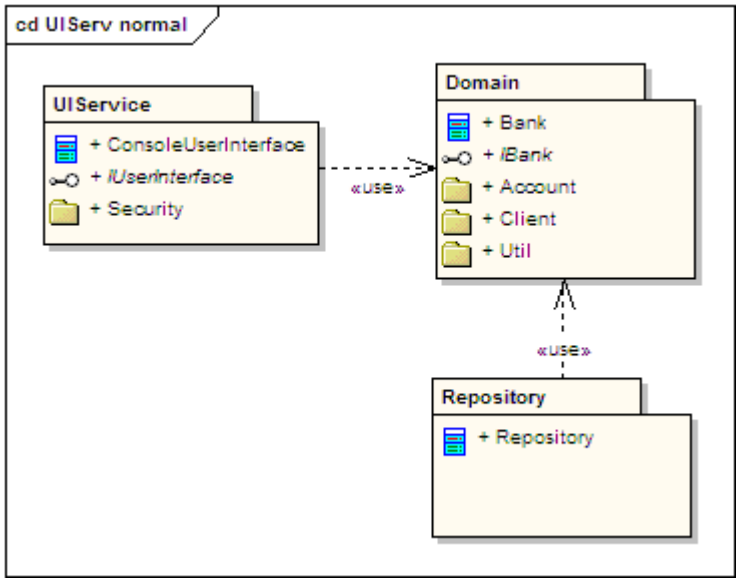

Figure 11: Class diagram of integrated security in the UI

In this scenario, the UI service embeds and implements all required features for authentication and authorisation. When this scenario is used in a problem domain, the UI service authenticates the user by checking the provided credentials. After a successful authentication, the UI service checks all the actions to authorise them, before sending the actions to the domain model. The self regulating behaviour is efficient to program because the security classes are only used when needed. The authorisation code for an action can be coded in the corresponding method or delegated to a security class inside the UI service.

The authorisation and authentication stages of security are performed by the UI service. Before starting to interact with the domain classes the UI service authenticates itself and the UI service authorises every interaction the user performs before forwarding this interaction to the domain classes. This behaviour is shown in *Figure 12*.

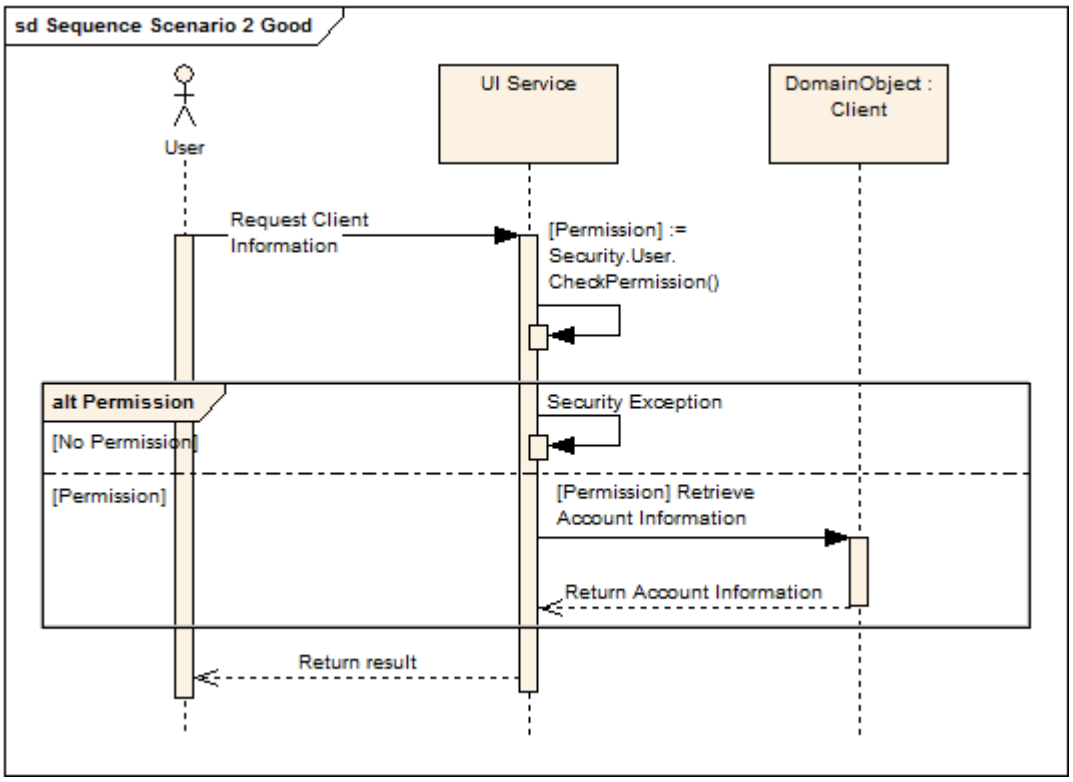

Figure 12: Sequence diagram of successful security checks in the UI service

By placing the security features into the UI service, the domain classes are unaware of the security service implementation. The domain model has no pollution into the source code and does not implement any security features.

This scenario works in compliance with the DDD paradigm concerning the domain-related functionality and separation of security into a service. However, integrating different domain-independent functional components (the user interface and security) is not compliant with the DDD target situation as presented in section *3.3*.

4.2.2 Concerns

The domain classes have to assume all actions are authorised. The possibility arises that the UI service bypasses the security functionality as shown in *Figure 13*. The UI service can have access to information and actions it should not have. Because the domain classes do not know about security rules, it assumes that the requests received from the UI are in accordance to the security rules.

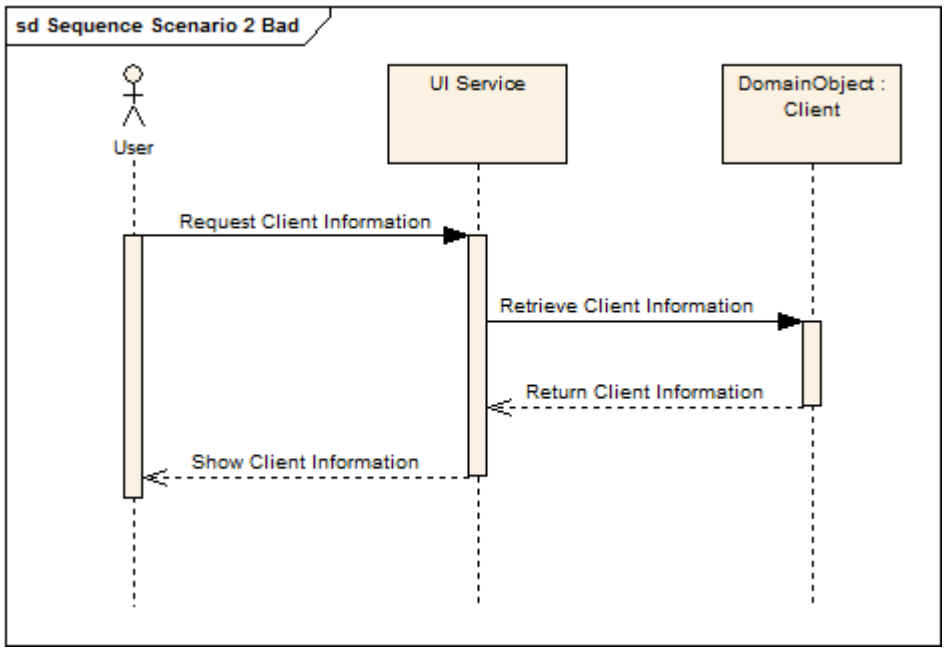

Figure 13: Sequence diagram of bypassed security checks in the UI service

If a different UI is required, the complete security code needs to be re-written in the new UI. This causes code duplication and extra work when expanding the application. Code duplication refers to the fact that any change to the security rules needs to be done in multiple places. When not all UIs are updated correctly, bugs are introduced in the application. Code duplication conceals this if testing does not obtain full coverage.

4.2.3 Conclusion

Integrating security functionality into the UI service is a bad choice for the separation of concerns. Different domain-independent functionality should not be located in a single service. This combination of functionality creates a dependency between the UI and the security implementation, any change in security policy involves recalling all UI implementations and updating them.

Glitches in the security implementation in the UI service represent a problem of the application that should be considered. If security methods are not checked, the UI service has a free passage into restricted domain functionality as shown in *Figure 13*. This scenario is not tamperproof; if the UI service methods are altered the domain classes do not know that executed actions are not allowed.

4.3 Scenario 3: Security service encapsulating the domain model

This scenario is related to the service layer architecture as described in [FOWL02]. By moving the UI service and other services into the model, the security service regulates all interactions with the domain model. This architecture has traits that resemble a proxy or façade pattern. The security service functions as a proxy towards, or façade to, the domain implementation.

This scenario provides for secure interactions between the services and the domain classes by intercepting all interactions originating from services. This should solve the issues with security described in *Scenario 1* and *Scenario 2*.

4.3.1 Usability and Features

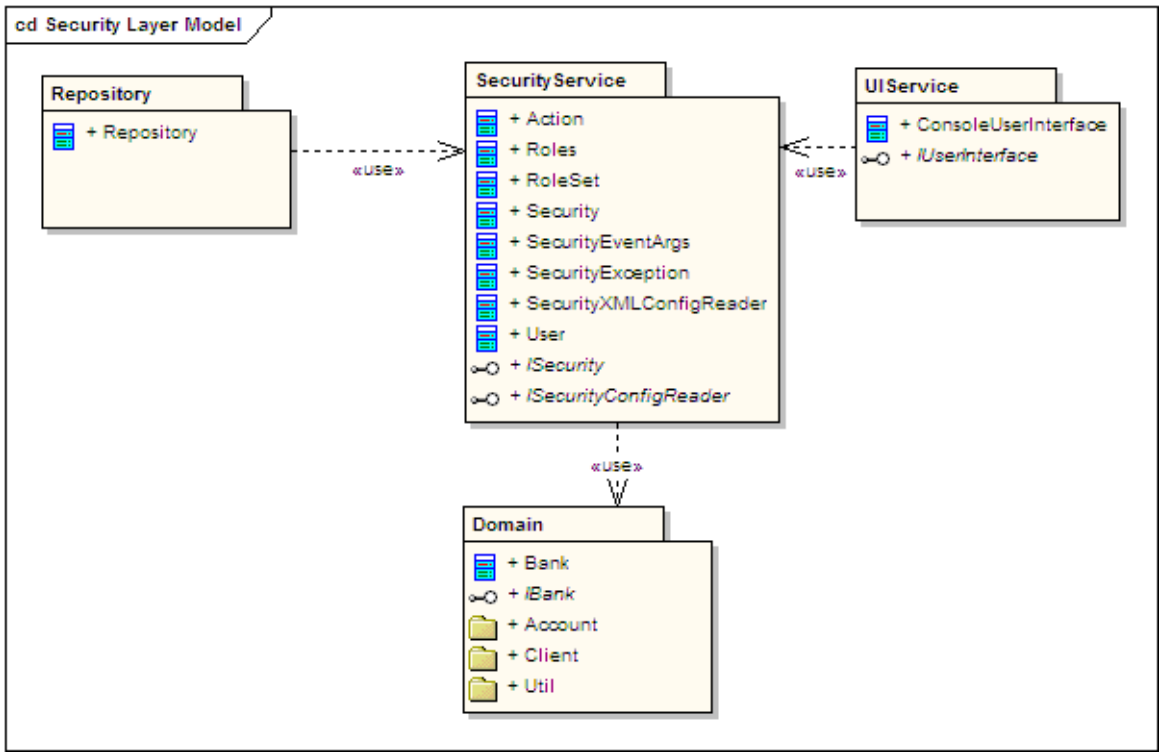

Figure 14: Class diagram of security service as a layer above the domain model

In this scenario all services in the application are coupled to the security service. The security service relays all actions for the domain classes. This scenario allows imposing rules on other services and regulating their activities for a maximum security level.

When one of the regular services performs an action in the domain implementation, the security service checks the authentication and authorisation level of the caller. After the check the security service has to perform a number of different actions. Depending on the action and authorisation of the caller, the security service has to:

- Tunnel the function and procedure calls into the correct domain classes.
- Tunnel the function and procedure calls into the correct domain classes and filter the feedback afterwards.
- Call another, more restricted, function or procedure in the domain classes.
- Block the function or procedure call and raise an exception.
- Relay events from the domain model towards other services.

Authentication and authorisation are both handled by the security service. It is not possible for the UI service to bypass these security checks, because the UI service has no direct access the domain classes. The authorisation sequence is shown in *Figure 15*.

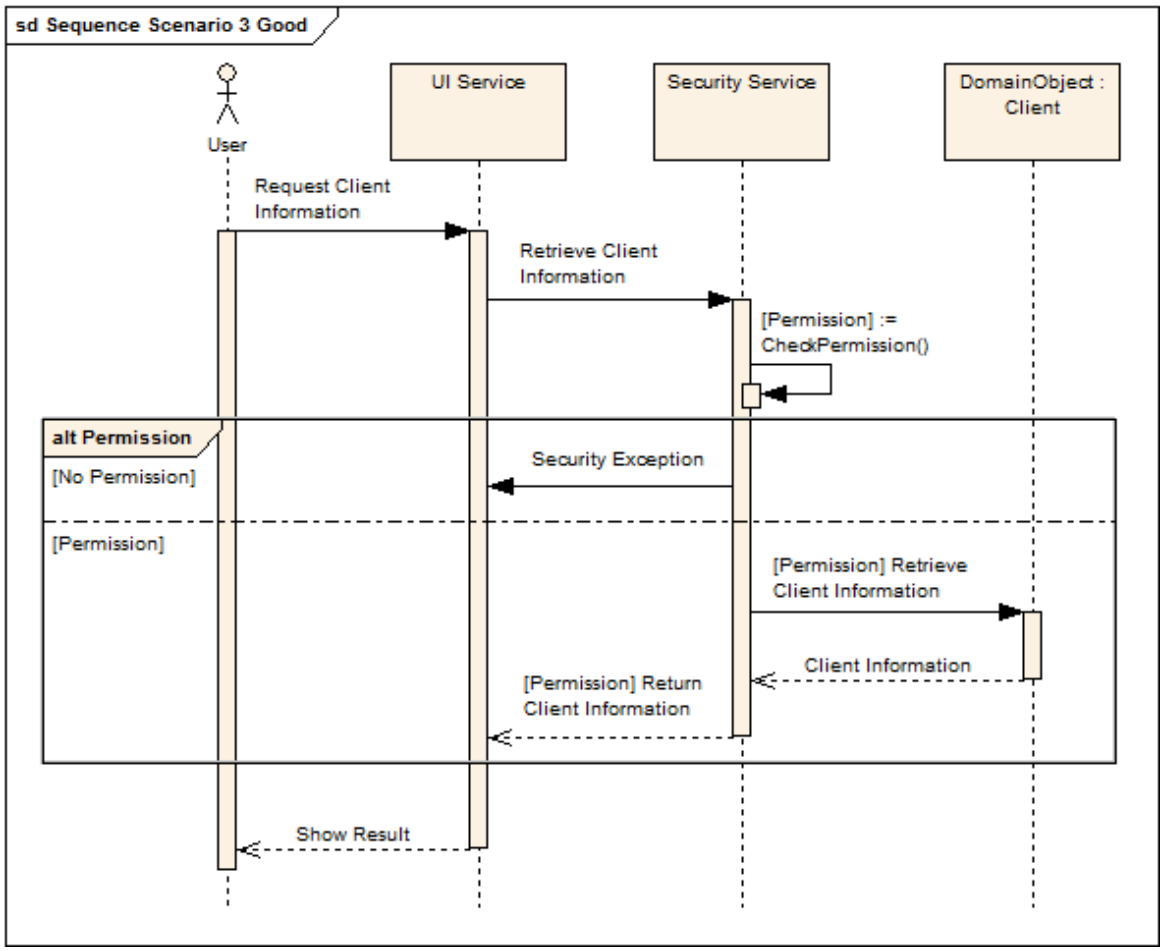

Figure 15: The authorisation sequence with the security service as a layer

This scenario works in compliance with the DDD paradigm concerning the domain-related functionality and the separation of security functionality into a separate service. However, the security service does not only handle the authentication and authorisation, but also functions unrelated to security. These extra functions in the security service are not compliant with the DDD target situation as presented in section *3.3*.

4.3.2 Concerns

In this scenario the security service acts like a broker of all interactions with the domain model. Managing all these interactions should not be the responsibility of the security service.
The security service contains a lot of methods mainly aimed at relaying information and requests to the other services.

Other effects of this design include that the security service needs to distinguish different services. A persistency service requesting a complete list of all users is completely different from an UI requesting the same data. In this case the persistency service needs an authorisation level. This is not practical to implement because the main concern should be to authorise user actions.

Concerns regarding event handling arise; if services require reception of events originating from the domain, these events need to travel through the security layer. An

example is to notify the persistency service that a domain object is changed and needs to be stored. In this scenario the event or observable pattern should be intermediated by the security service, which is inefficient.

4.3.3 Conclusion

Compared to the implementation from section *3.2* this scenario has a lower coupling between the security implementation and the domain model. The remaining coupling is caused by checks performed by the security service that are based on the state of the domain model and the relaying of method invocations from all services.

In this scenario security is tight, as all interactions from all services towards the domain classes are checked. Authentication and authorisation work well when determining the security level between the UI service and the domain model. In order to implement this scenario, roles should be defined for services. Defining roles for specific services is more difficult. Each method invocation should be performed at the level of the lowest rights in the calling chain. Thus for different calls of the same method different roles must be provided by the service.

The security service has to perform too much tunnelling and intermediary work. Multiple tasks are performed by the security service. These tasks, security and performing intermediary work, should be separated.

4.4 Scenario 4: Security service as a gateway for the UI

This scenario is based on the decision to let the security service only handle input from the UI service. By only authenticating and authorising the actions performed by the UI, other services do not have to be authorised and intermediated. This scenario is a refinement of *Scenario 3*, in this scenario the security functionality is not required to handle all communication with the domain classes. Only interactions originating from the UI service are checked.

4.4.1 Usability and Features

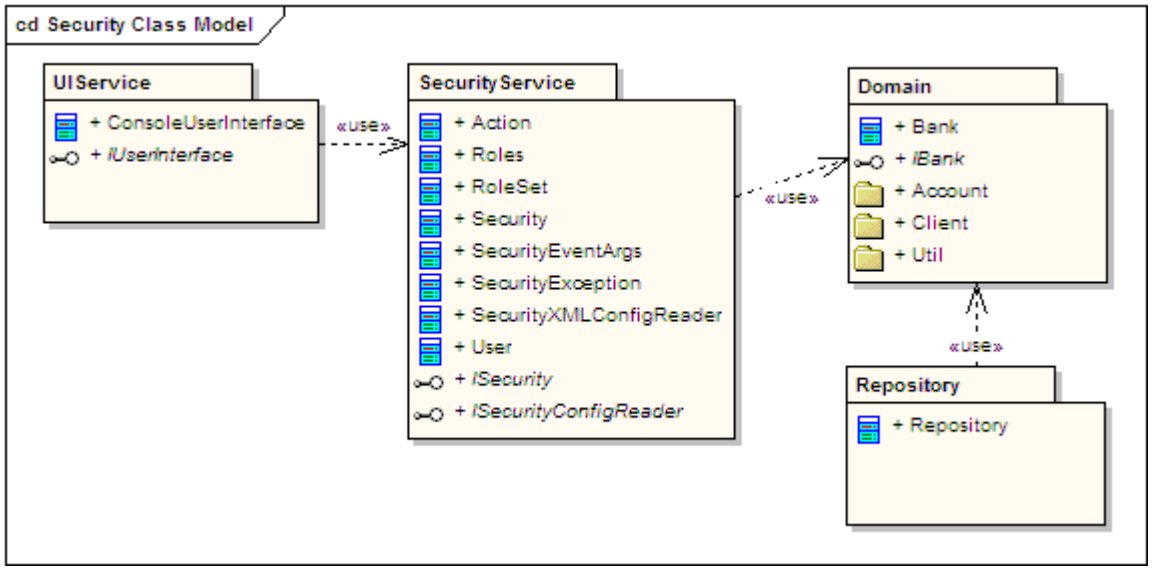

Figure 16: Class diagram of the security service as a gateway for the UI

In this scenario the UI is only connected to the security service, all function and procedure calls are directed at the security service. The input from the UI is transferred and checked by the security service. When the UI starts an interaction it needs to authenticate using the security service. Based on this authentication further actions initiated by the UI can be authorised. If another UI needs to be coupled to the domain, it can be linked to the already available security service, thus ensuring that the same restrictions apply to all the interfaces.

The location of the security service is based on the need to check the communication from the UI service with the domain model. Communication between the domain classes and the persistency service do not have to be checked because that service is only reacting to changes in the domain classes and no outside parties (like an UI) are able to directly access the persistency service.

When the security service receives a request from the UI it checks the authorisation level. After the check the Security service has to perform four different actions.

- Tunnel the function and procedure calls into the correct domain classes.
- Tunnel the function and procedure calls into the correct domain classes and filter the feedback afterwards.
- Call another, more restricted, function or procedure in the domain classes.
- Block the function or procedure call and raise an exception.

The security service also has to relay events from the domain classes towards the UI.

Authentication and authorisation are both handled by the security service. To ensure the application security, either all services have to be coupled to the security service so their requests can be checked. The other option is to regulate requests from the UI service towards the other services. The sequence diagram concerning the authorisation process is the same as in the previous scenario, shown in *Figure 15*.

This scenario works in compliance with the DDD paradigm concerning the domain-related functionality and the separation of security functionality into a separate service. However, the security service does not only handle the authentication and authorisation, but also functions unrelated to security. These extra functions in the security service are not compliant with the DDD target situation as presented in section *3.3*.

4.4.2 Concerns

The security service checks and relays all requests from the UI service; this includes requests unrelated to any security check. By handling all requests from the UI service the security service implements methods that only relay information between the domain model and the UI service. This is not a job that should be performed by the security service. Scalability problems arise, if the domain model is more complex, many methods are needed in the security service. A proxy or adapter pattern could be implemented to reduce this effect. With an adapter or proxy pattern, the adapter is responsible for the handling the tunnelling functions and the security functionality is only accesses when an interaction requires authorisation.

If the UI service is completely separated even from other services, the security service must also relay requests from the UI service towards other services. This also applies vice versa, i.e. events from other services must be redirected back to the UI service.

If another service requires authorisation checks, all interactions from this service must be handled by the security service to prevent unauthorised access.

4.4.3 Conclusion

Compared to the implementation from section *3.2* this scenario has a lower coupling between the security implementation and the domain model. The remaining coupling is caused by checks the security service performs that are based on the state of the domain model and the relaying of method invocations from the UI service.

This design presents a good step towards separation of concern by intercepting only the necessary method invocations. Only the pass-through functions should be separated from the security functionality, which enables better reuse of the security functions and classes.

4.5 Scenario 5: Security service as an adapter for the UI

This scenario is an improvement on the decision to let the security service only handle input from the UI service. By only authenticating and authorising the actions performed by the UI service, actions of other services do not have to be handled by the security service.
The improvement consists of moving the functionality needed to relay and check method calls from the UI service into the adapters [GOF95]. For simplicity this scenario only references adapters to illustrate the concept. Using a proxy or façade pattern in the

same way this would yield the same functional result, as these are similar structural patterns [GOF95].
In its current form, this scenario leaves a choice to the designer. Either all services should be coupled to security adapters so their requests can be checked, or requests from the UI service towards the other services should be intercepted and checked. Checking requests towards other services means that adapters must be defined for interactions with these services.

4.5.1 Usability and Features

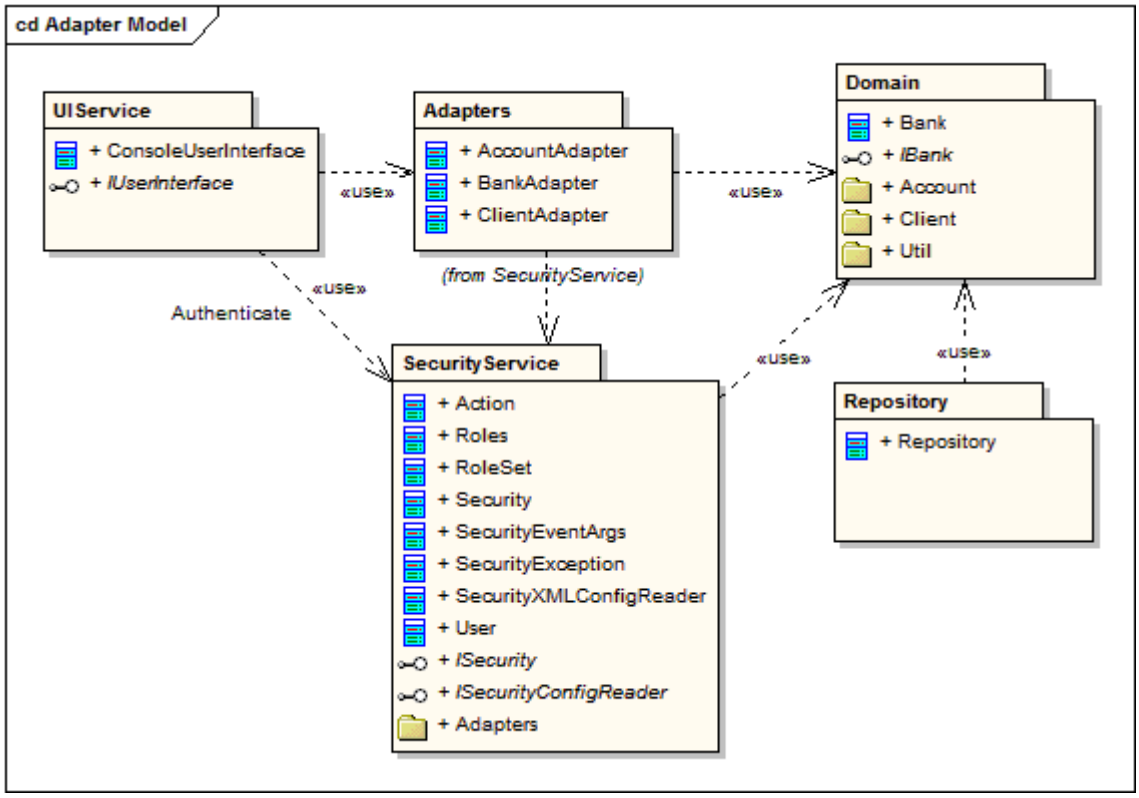

Figure 17: Class diagram of an adapter implementation

In this scenario the UI service interacts with the domain adapters and other services. The other services used by the UI services should also interact through security adapters. The security adapter checks all the method calls from the service. In these methods the clearance of the UI service is checked using the classes in the security service. In this scenario the checking is performed with regular method invocation. No security logic is present in the security adapter, since it is a slim intermediary. The sequence of the authorisation process is shown in *Figure 18*.

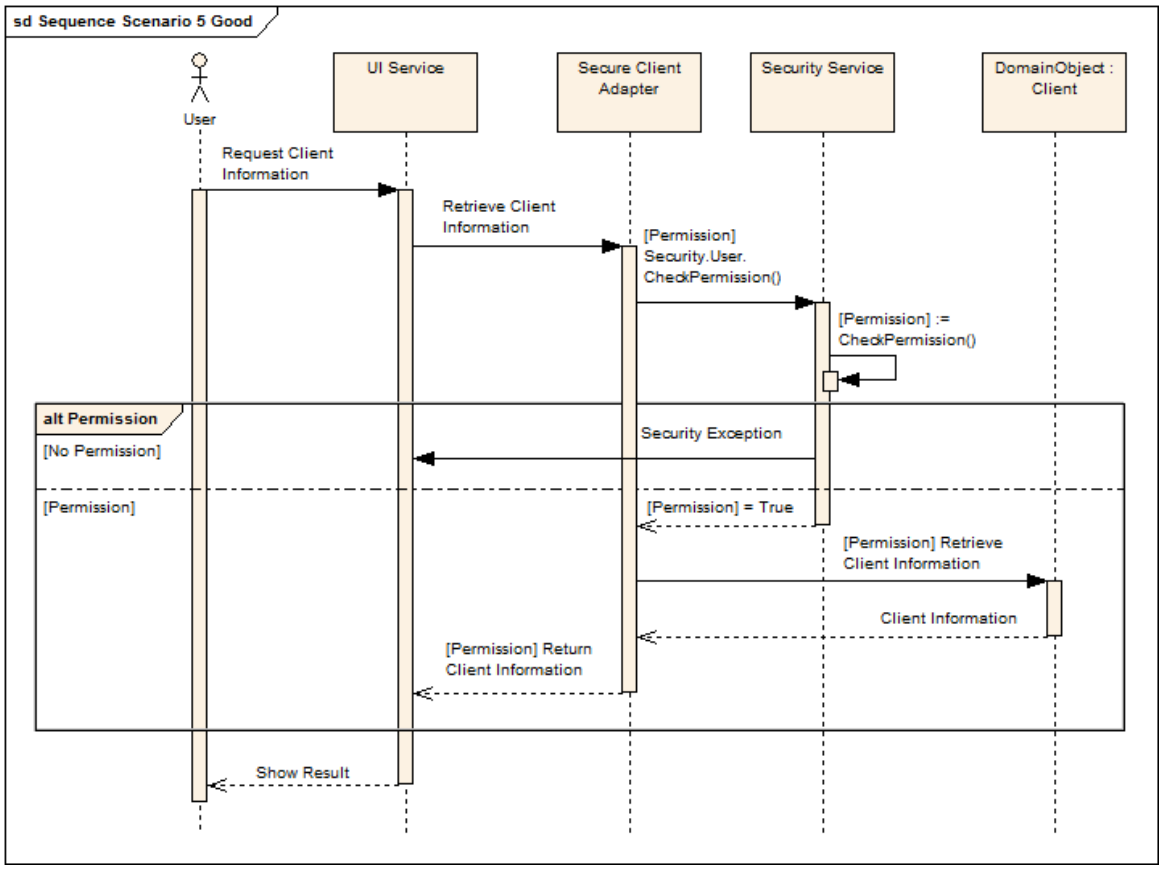

Figure 18: The authorisation sequence with the secure Client Adapter

Multiple services can use the same secure adapter if restricted access to the domain classes is needed. Security logic is separated into a separate service and all security rules are stored in the security service. This means there is one point where the security policy is defined.
If the repository is accessed by the UI service it should also return the security adapter objects instead of the domain objects.

By separating the security functionality into a standalone service, its focus retains the original purpose. The security service does not tunnel requests to the domain classes, the security functionality is only used when a method requires restricted access.
If the security adapter can not perform security checks because the security service is unavailable, the adapter blocks all restricted calls to the domain classes.

Only security adapters need to know the domain classes and generate coupling. In this scenario the security service still has to interact with the domain classes when security decisions are made that depend on the state of the domain classes.

The main benefit from this approach is that the base implementation of the security adapters can be generated. Points where security is applied are defined after this generation.
Expandability is facilitated by more adapters to allow the UI service to communicate with other services or these other services can check the UI service access rights with the security service implementation.

This scenario works in compliance with the DDD target situation concerning the domain-related functionality and the separation of security functionality into a separate service. Depending on the placing of the adapters in the model this scenario is compliant with the DDD target situation as presented in section *3.3*.
The adapters can not be placed inside the domain model. If the adapters are placed in the security service, the UI service does not interact directly with the domain implementation.

4.5.2 Concerns

In order to function properly, assurances are needed that the UI service has no direct access to the domain classes.
Other services used by the UI service should also have no direct access to the domain classes, but only to the secure adapters.

If it is required that the UI service is completely separated even from other services, the security adapters must also relay requests from the UI service towards other services. This also applies vice versa, i.e. events from other services must be redirected back to the UI service.

4.5.3 Conclusion

Compared to the implementation from section *3.2* this scenario has a lower coupling between the security implementation and the domain model. The remaining coupling is caused by checks the security service performs that are based on the state of the domain model.

By moving the pass-through functions from the security service into adapters the actual security rules and logic are separated from the relaying of information. The security service regains the original concern of security.

4.6 Scenario 6: Security service integrated by AOP with adapters

This scenario introduces Aspect-Oriented Programming (AOP) [AOP08] to the problem domain. By using AOP the UI service, adapters and the domain model do not have to be aware of the security service for authentication purposes. The UI service only requires knowing the security service for authentication and the definition of security exceptions. Aspect oriented programming defines aspects as pieces of code that have to be inserted in the source code at specific points. These aspects signify a crosscutting concern (aspect) of all the targeted methods in the source code.
In this scenario, AOP is used in conjunction with the scenario from section *4.5*. The combination results in the point-cuts intersecting with join points in the intermediary adapter. AOP removes the security logic from the adapters into aspects, an added benefit is that the adapters can be generated by an Integrated Development Environment (IDE) and do not have to be edited to apply security.

AOP defines join points, point-cuts and advices to represent these places in the code [YASE07]. Join points represent locations, like method entries or property declarations, in the source code where AOP can hook in. Point-cuts define to which join points a specific advice should be applied. An advice is a part of a specific aspect which defines what actions should be executed. Examples are onEntry or onExit of the method call, which apply before and after the method defined by the join point.

4.6.1 Usability and Features

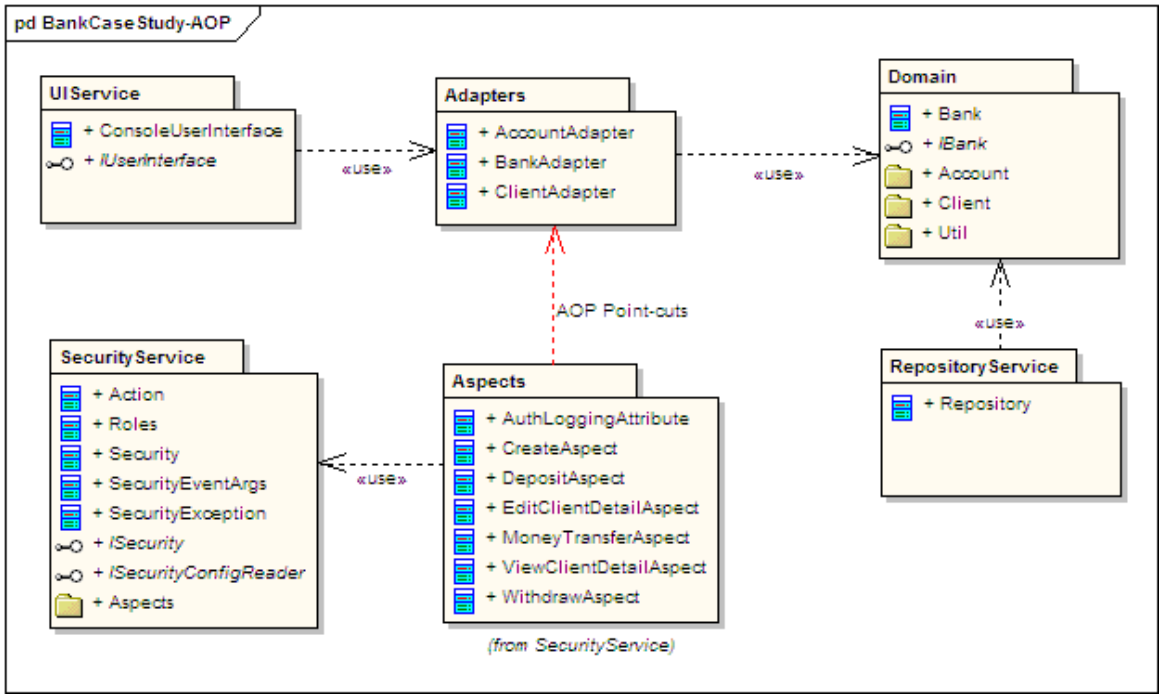

Figure 19: Class diagram of an AOP security service implementation with adapters

In this scenario the security service is located and coded in a standard service location outside of the domain model. The UI service has no direct links towards the security service for authorisation.
After the advices are weaved into the adapter code, the security code is executed every time the functions are referenced. The compiled code containing the weaved advices behaves the same as if the security functionality had been integrated in the regular security adapters.

When AOP is applied to the application model the aspects yield the same end result as seen in the scenario with the adapters providing access to the domain model. The usage of aspects for crosscutting concerns save developers lines of code, because the crosscutting concerns are handled in one location in the code instead of many locations.

Authentication is handled by the security service and authorisation is woven into the adapters by the aspects. The sequence of authorisation after the aspects are woven in the source code is shown in *Figure 20*. The sequence highlighted with 'critical AOP' is handled by code specified in the aspects in the security service.

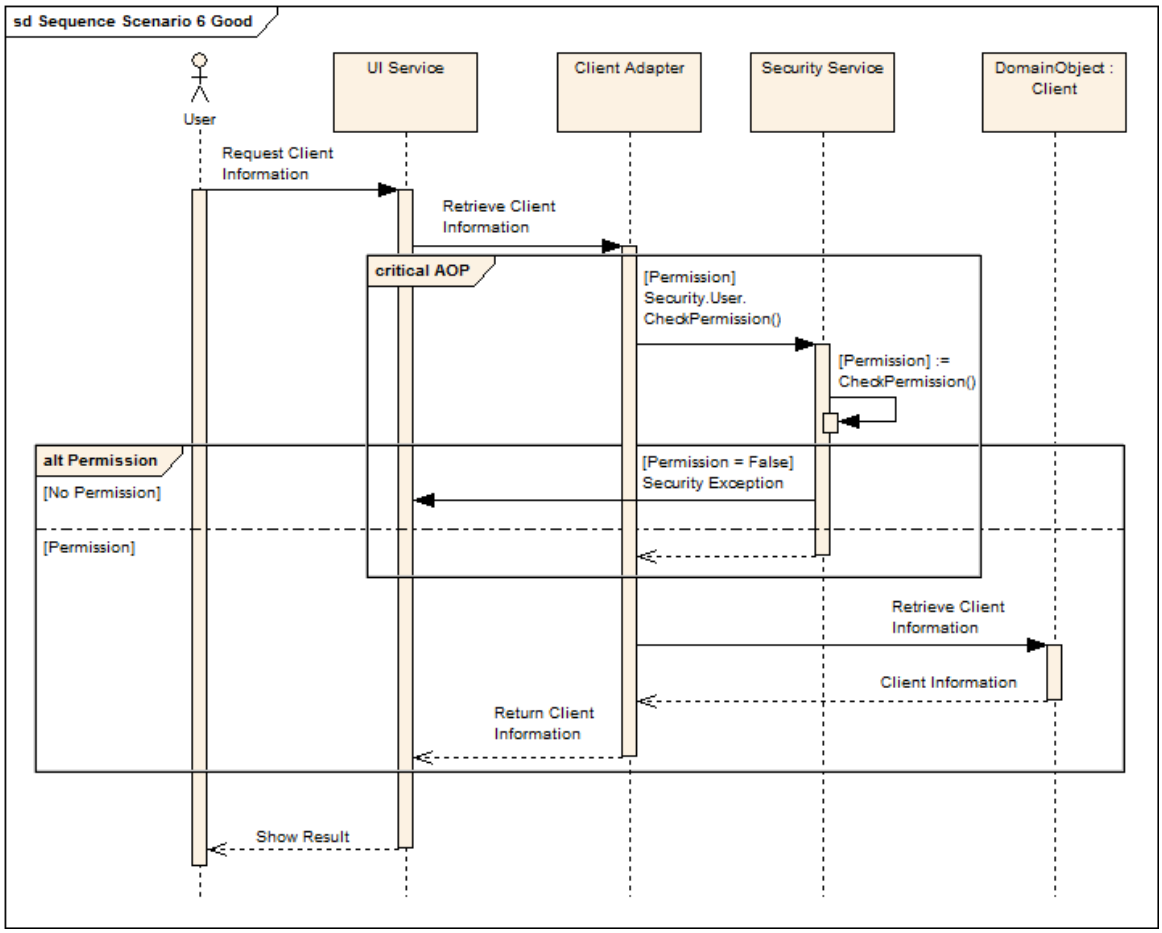

Figure 20: The authorisation sequence with the Client adapter and AOP

Weaving ensures that the advices are inserted at the join points designated by the point-cuts. The advices make sure that corresponding methods for authorisation are executed in the security service. Different implementations of AOP enable different kinds of weaving that can be performed by AOP at compile time or at runtime.

This scenario works in compliance with the DDD target situation concerning the domain-related functionality and the separation of security functionality into a separate service. Depending on the placing of the adapters in the model this scenario is compliant with the DDD target situation as presented in section *3.3*.
The placing of the adapters influences the compliance with the DDD target situation because the adapters can not be placed inside the domain model. If the adapters are placed in the security service, the UI service does not interact directly with the domain implementation.

4.6.2 Concerns

When using aspects some problem areas like clarity of code, changes to the code and overuse of AOP must be considered.

Clarity of code becomes a concern because the point-cuts of advices are not immediately clear in a standard IDE. Specific plug-ins are needed to correctly display where point-cuts hook into join points in the application implementation. This is difficult to apply to

the domain model, since the least possible pollution of source code is wanted in the domain classes. Despite adding little information to the domain model it should be perfectly clear where point-cuts apply. Especially for security it is vital to be sure every possible path is covered.

When using AOP, it becomes more hazardous to change in the source code because of methods that are targeted by point-cuts.
While the application is under development it is impossible to make no changes to methods targeted by point-cuts, because changing the domain model during the implementation phase is at the heart of developing an application. With incorrect tooling, changes to the source code are always a risk. Problems consist of point-cuts not matching the correct join points any more, or point-cuts that inadvertently apply or do not apply to new methods.

AOP is meant to be a solution for crosscutting concerns. Clear limits and boundaries need to be posed on the use of aspects. The cases where it is applied have to be studied to prevent overuse of aspects in places where they are not necessary. This is needed to prevent business logic from being migrated to advices outside of the domain model. Otherwise there is a possibility that an anaemic domain model is created that does not contain the business logic.
In the bank case study, business logic like the prevention of negative deposits on an account could be placed in an aspect. This would remove responsibility from the domain implementation that should remain in the domain implementation.

Furthermore the order of multiple point-cuts on one join point is not guaranteed or known. If other aspects besides security aspects are used, i.e. to create transaction objects, these could have precedence over the security aspects and thus result in unpredictable behaviour. This could also result in the domain state becoming invalid. Different implementations of AOP offer optional priorities or the possibility to declare precedence of aspects. However, this is only a partial solution to the problem because collisions between these priorities can still occur.
Concerning the different types of advices, no standardized advices are defined for AOP in general, meaning that the types can be different for each implementation.

4.6.3 Conclusion

Compared to the implementation from section *3.2* this scenario has a lower coupling between the security implementation and the domain model. The remaining coupling is caused by checks the security service performs that are based on the state of the domain model.

AOP provides great possibilities especially within DDD, since coupling can be easily reduced for crosscutting concerns. These improvements are achieved with some drawbacks that should not be overlooked.

4.7 Scenario 7: Security service integrated with AOP

In this scenario AOP is applied directly to the domain classes. It explores another possibility for applying AOP in an application. By applying AOP to the domain model intermediary adapters are not necessary and the design is simplified.

4.7.1 Usability and features

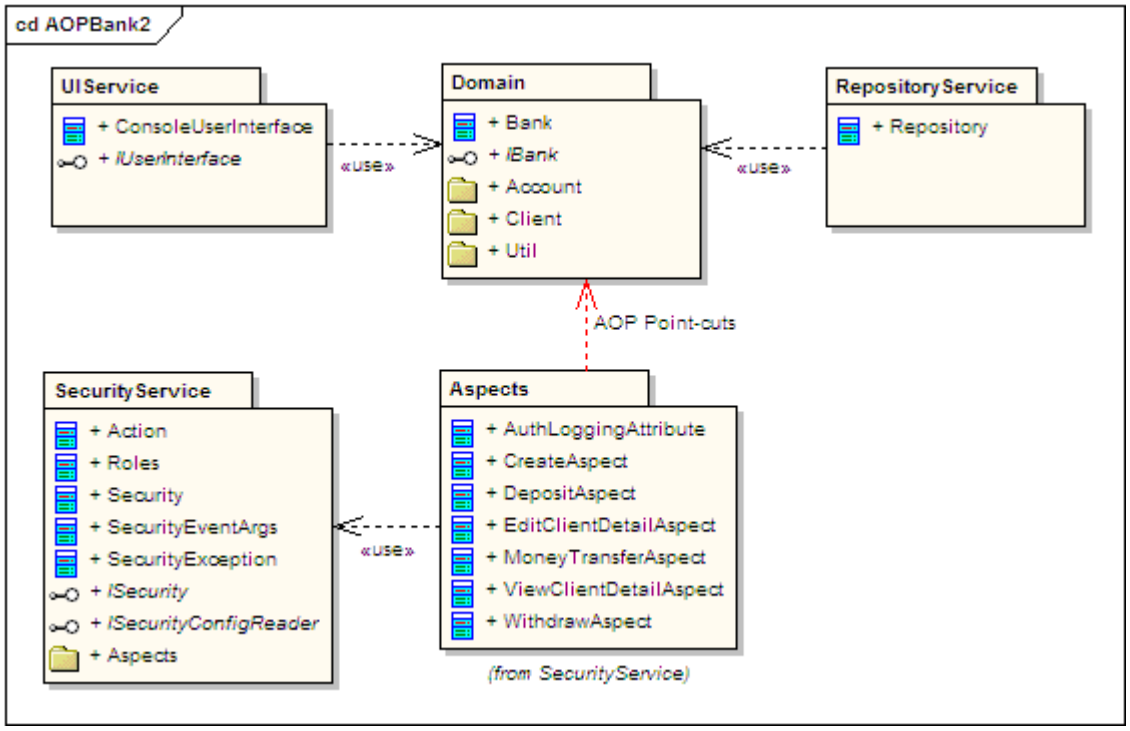

Figure 21: Class diagram of an AOP security service implementation

In this scenario the security service, containing the security logic is located and coded in a regular service location outside the domain model. The UI service has no direct links towards the security service for authorisation.
This scenario looks very much like the scenario with security as a regular service in section *4.1* except that Aspect-Oriented Programming (AOP) is used. After the advices are weaved into the domain class code, the security code is executed every time the functions are referenced. In weaved code the application behaves like the security service is integrated into the domain model as seen in section *3.2*.

When AOP is applied to the application model the aspects yield the same end result as seen in scenario *3.2*. The usage of aspects for crosscutting concerns save lines of code and reduces the complexity of methods. Using AOP means there is no pollution in the source code of the domain model originating from the security service.

Authentication is handled by the security service and authorisation is woven into the domain model by the aspects. In principle this is sufficient and it can be assumed that the aspects are correctly woven into the source code. The sequence of authorisation after the aspects are woven in the source code is shown in *Figure 22*.

The UI service interacts directly with the domain class. The UI service is unaware of the security screening in this situation. The sequence highlighted with 'critical AOP' is caused by code specified in the aspects in the security service.

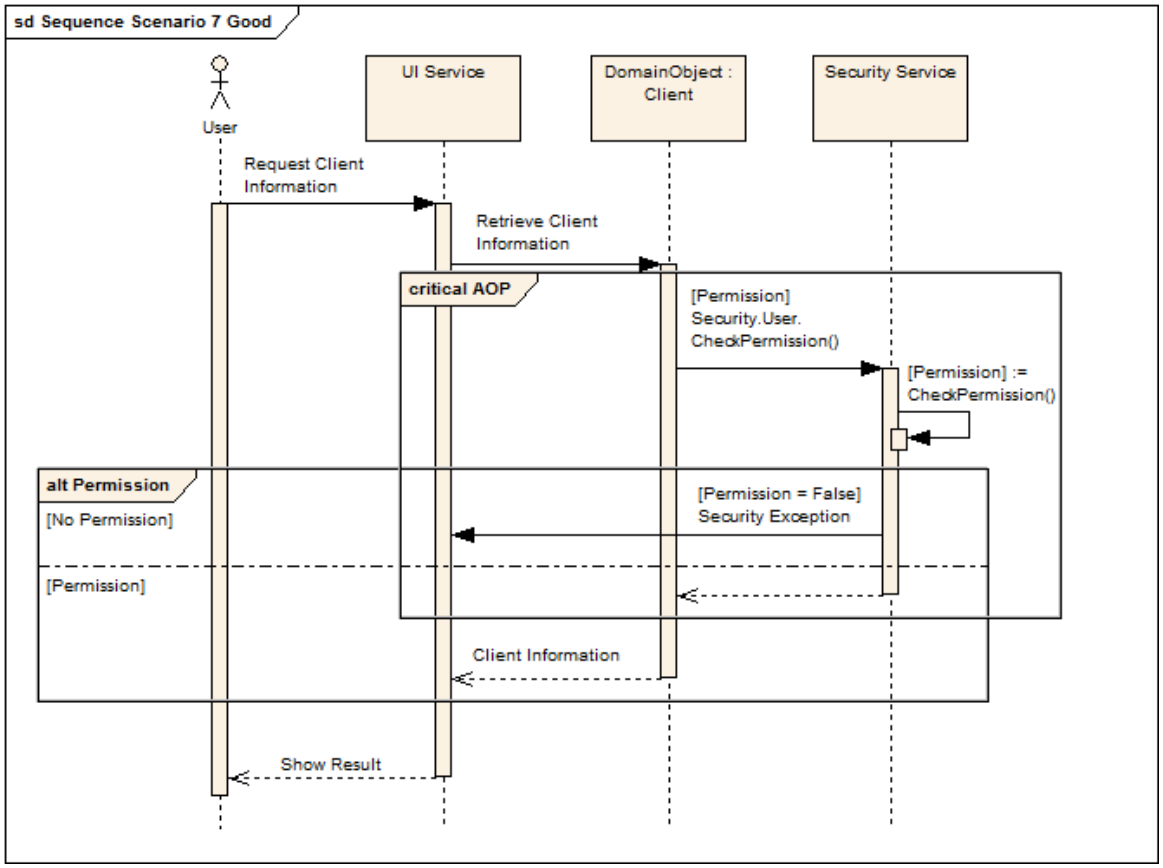

Figure 22: The authorisation sequence with the Client class and AOP

Weaving ensures that the advices are inserted at the join points designated by the point-cuts. These advices make sure that corresponding methods for authorisation are executed in the security service.

The usage of aspects in programming complies with the DDD target situation. No security code is added to the source code of the domain model. The UI service now has a role as a regular service and can perform actions directly in the domain model.

4.7.2 Concerns

The same concerns about the usage of AOP as described in *section 4.6.2* apply to this scenario.

A problem that must be considered in this design is how method invocations in the domain implementations are connected to a corresponding UI service. The domain implementation does not know about the existence of a UI service, so the identity and role of the UI service must be stored elsewhere.

4.7.3 Conclusions

Compared to the implementation from section *3.2* this scenario has a lower coupling between the security implementation and the domain model. The remaining coupling is caused by checks the security service performs that are based on the state of the domain model.

AOP provides great possibilities especially within DDD, since coupling can be easily reduced for crosscutting concerns. These improvements are achieved with some drawbacks that should not be overlooked. Deliberation is needed to apply AOP successfully in an application implementation.

This scenario has a very clean look and complies with the DDD target situation as presented in section *3.3*.

Chapter 5
Security design implementations

This chapter presents the scenario implementations. A selection is made of designs from *Chapter 4*. The selected designs are implemented for the case study design as described in *Chapter 2*.
The role of this chapter is to present the selection process and to determine the usability of the scenarios according to the criteria from section *3.3*. Because the selection is made based on scenarios on the model level without actual source code to measure, metrics that quantify coupling are not available. After the selection of the most usable designs these are implemented in the bank case study.
This implementation reveals the possibilities and problems the scenarios pose to implement security functionality as a service in the DDD environment.

5.1 Scenario selection

As described in section *3.3* the criteria for scenario selection are:

1. Estimated coupling Security and Domain: An estimation of the coupling reduced by the security implementation in comparison with the reference design from section *3.2*.
2. Separation of Concerns: The extent to what the scenario complies with the principles of separation of concerns.
3. Usability and efficiency: The extent to which the scenario is usable and efficient.
4. Functional result as required: The extent to what the scenario yields the same level of security as seen in section *3.2*.
5. Accordance with DDD definition: The degree to what the scenario complies with the DDD definition as presented in *Chapter 2* and the DDD target situation presented in section *3.3*.
6. Transparent and Logical: Is the idea behind the structure of the scenario transparent and logical?

Because the scenarios only present models, the score for the criteria is based on the intermediary conclusions from *Chapter 4*.
In *Table 2* the scenarios presented in *Chapter 4* are weighed according to the criteria. For each of the criteria the scenario is awarded a rating on a scale of 1 to 5, which are displayed as --/-/o/+/++, with -- meaning poor performance and ++ excellent performance concerning this area.

Criteria / Scenarios	Estimated coupling Security and Domain	Separation of Concerns	Usability / Efficiency	Level of security as required	Accordance with DDD target situation	Transparent / Logical	Final Score
1: Regular Service	+	+	--	-	++	+	N/A
2: Integrated in UI	+	-	-	-	o	+	N/A
3: Security Layer	o	o	-	++	o	+	(2) Poor
4: Gateway	+	o	+	++	+	++	(8) Average
5: Adapter / Proxy	+	+	++	++	+	++	(10) Good
6: AOP w/ Adapter	+	++	++	++	+	+	(10) Good
7: AOP w/ Domain	+	++	++	++	++	o	(11) Good

Table 2: Usability overview of scenario from *Chapter 4*

The most important criterion of *Table 2* is whether the same level of security is attained as in the reference implementation. Without this level of security the scenario is not suitable. The accordance with the DDD target situation is more important than the remaining criteria, so 2 points are rewarded for every + in this category, the other criteria result in 1 point per + and -1 point for every -.

Scenario 1 illustrates the initial idea about security as a service. Here security is presented as a regular service. The problems as described in section *1.2* become immediately clear.
Due to the problems concerning the security functionality this scenario is not suitable and should not be implemented.

Scenario 2 demonstrates the option of integrating security into the UI service.
Due to the problems concerning the security functionality this scenario is not suitable and should not be implemented.

Scenario 3 introduces the principle of moving security into a service that works like a layer surrounding the domain model. A low mark is received on the 'Usability and efficiency' because the security service has to handle all communication between services and the domain model in the application. Consequently this is an inefficient implementation with low performance.
The 'Functional result' of this scenario is good because of checking of all communication, the design is easy to understand because the layer model is well known.

Scenario 4 retains a high 'Functional result' with the security being able to check all interactions with the UI service. This scenario has a higher score for 'Usability and efficiency' than scenario 3, because only interactions originating from the UI service are checked. Only performing checks on UI service interactions reduces coupling to the domain model. Different implementation options exist for other services, which are

accessed through the security service by the UI service or are allowed to perform actions in a manner resembling the Model-View-Controller model (MVC) [GOF95]. These services (acting as View) can query the domain classes (acting as the Model) and directly receive events. Interactions with the domain model, however, can be routed through the security service (acting as Controller).

Scenario 5 improves the 'Usability and efficiency' of the case study by separating the security functionality from the tunnelling functionality. Tunnelling and interception logic concerning the authorisation is moved into security adapter implementation. This scenario is transparent because well known patterns of adapter/proxy/façade are used in conjunction with the application of the MVC model. In this MVC model the adapters implement the Controller function.
In this scenario either adapters are created for the domain implementation and all services use these adapters, or the UI service also communicates with an adapter for each specific service.

Scenario 6 adds Aspect-Oriented Programming to the case study. The introduction of AOP results in decoupling of the authorisation checks from the domain functions. Also the coupling between the adapters and the security service is removed in the application model, which results in a better separation of concerns. This scenario has a high 'Usability and efficiency' because the crosscutting concern of security is handled more efficiently than in the reference scenario. AOP does not add to the transparency of the implementation because most developers have no experience with AOP and do not know the implications of applying AOP in practice.

Scenario 7 moves AOP point-cuts from the adapters to the domain model. Moving the AOP point-cuts to the domain model eliminates the need for the adapter implementation. Therefore this scenario has a complete conformance with the DDD goal of separating security into a service. This scenario has a high 'Usability and efficiency' because the crosscutting concern of security is handled more efficiently than in the reference scenario. AOP does not add to the transparency of the implementation because most developers have no experience with AOP and do not know the implications of applying AOP in practice.

Selection
Only the best scoring scenarios from *Table 2* are implemented. Deriving from the main goals as presented in *Chapter 1*, the most important criteria involve the application of correct working security and compliance with the DDD principles.

It is not useful to implement scenarios that do not offer the same level of security as the reference design. This disqualifies scenario 1 and 2 from implementation.

As seen in *Table 2*, scenario 5, 6 and 7 offer the most promising solutions to the problem of introducing security as a service in DDD and have been implemented.

5.2 General implementation choices

Before describing the implementation phase for the three selected scenarios, some general implementation choices are discussed.

As described before the already available case study was implemented in the C# language, this means the .NET environment is used for the implementation scenarios. However, the methods and techniques used in the scenarios are generic and can be implemented in other programming languages than C#.

The application consists of five packages: the domain implementation package and four packages containing one service each. The domain implementation package is where the implementation of the domain model is located. The classes contained in the domain implementation are depicted in *Figure 5.*

Services surrounding the Domain package are:

- The repository service, which is used to create and find instances of objects in the different packages.
- The IO service, which is responsible for creating reports for users to display the current state of the domain package and to handle batches of input commands.
- The UI service, which presents an interface through which the user can interact with the application. This service also handles user input.
- The Security service, which is responsible for regulating authentication and authorisation in the application implementation.

The package structure is very similar for the three scenario implementations. The main difference between implementations is of the UI and IO services being connected to the adapters instead of the domain classes.
Besides these changes, the contents of the repository, UI, IO service and domain implementation are the same in the three implementations in this chapter. The security service is kept the same as much as possible, but changes the most.

Different user roles are included in the bank case implementation. These roles provide an example for multiple access levels. An overview of the different roles available in the bank case study was already discussed in section *3.1*.

5.3 Scenario 5 implementation: Security service as an adapter for the UI

The implementation of security adapters in the bank case result in restricting the actions of the UI service. The UI service is not allowed to interact with the domain implementation directly, but is only allowed to interact with the security adapters and the other services in the application model.
Concerning the security service, other services are able to interact with the Security class through the ISecurity interface. This interface allows access to the authentication functionality. Within the security service, the Action class is known to provide the action

enumeration, which allows the UI service to identify the interactions it requests. The SecurityException class must be known by other services to recognize the exceptions raised by the security functionality. The other classes are used internally by the security service to retrieve and represent users of the application and their roles, these classes are private classes and thus cannot be accessed by classes outside the security service.
Other services used by the UI service are also required to communicate with the adapters in the security package, as the function of the adapters is also to perform the Controller function in this MVC model-like structure.

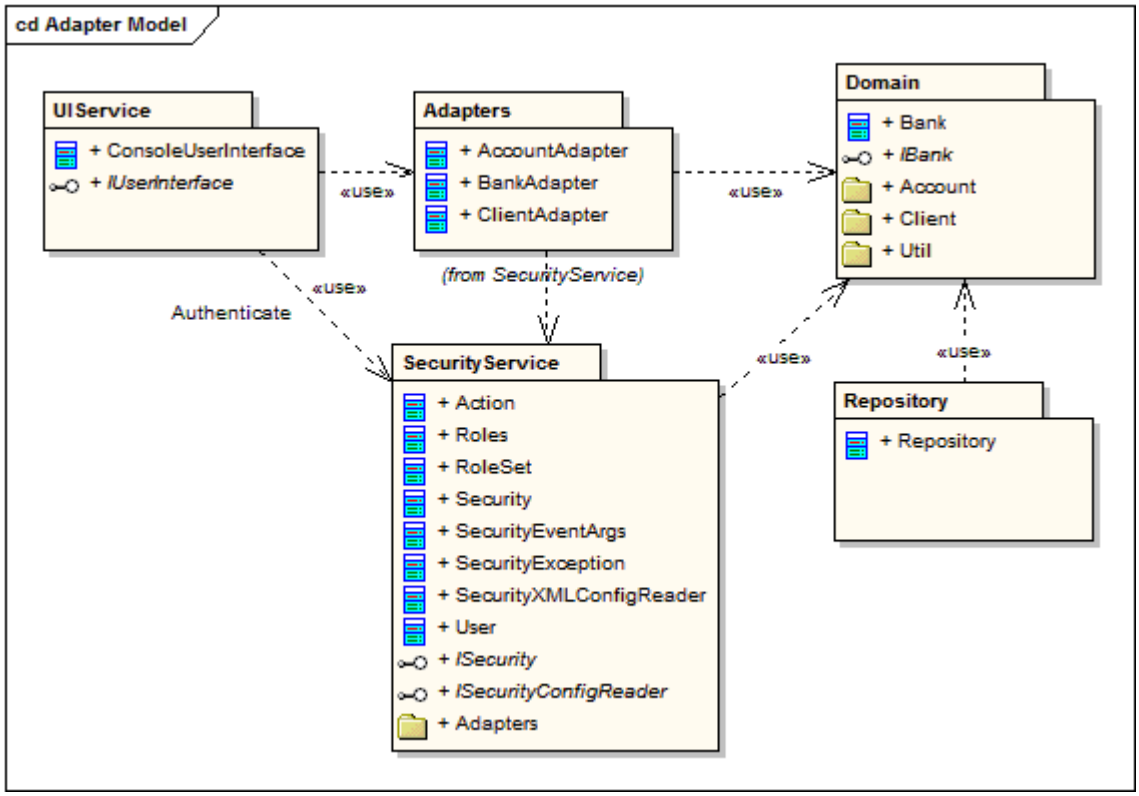

Figure 23: Class diagram of the adapter implementation

To perform actions in this application, the UI service authenticates with the security implementation. Based on the authentication credentials supplied by the UI service, a corresponding user object is created with an appropriate role set. This user representation is saved and used by the security service to authorise actions of the UI service.
Interactions with the domain implementation originating from the UI service are handled by the security adapters. The adapter receives a request from the UI service and checks if the action is allowed by consulting the permissions of the user's role set. If any restriction posed by the security functionality is violated, a security exception is raised as illustrated in *Code Fragment 2*.

```
Public override string ToString()
{
   if(!security.user.RoleSet.IsPrintClientDetailsPermitted(this.client))
   {
        security.GenerateSecurityException(
           "You are not allowed to view the client details.");
   }
   return this.client.ToString();
}
```

Code Fragment 2: Raising security exceptions in the ClientAdapter implementation

By raising a SecurityException the code execution in the ToString() method is stopped and the exception is sent to the calling class.

The inner working of the adapters is straightforward, since methods from the domain objects are mostly reflected in the adapters. Information contained in the domain objects is retrieved by interacting with the adapters.
The security adapter implementations must ensure that no instances of domain objects are returned, but only their adapter representations.
This wrapping of the domain objects is shown in *Code Fragment 3*.

```
Public List<AccountAdapter> GetAccounts()
{
   List<AccountAdapter> accountList = new List<AccountAdapter>();

   foreach (IAccount account in this.client.Accounts.Values)
   {
       accountList.Add(new AccountAdapter(account));
   }
   return accountList;
}
```

Code Fragment 3: Returning adapters in the ClientAdapter

The usage of adapters allows the adapter to be responsible for session management. In this way it is always known which user is responsible for the performed action.

5.4 Scenario 6 implementation: Security service integrated by AOP with adapters

For AOP there are multiple implementation options available depending on the implementation platform that is selected. The programming language is important because special compilers are required to weave AOP aspect code into source code. Examples of these weavers are: AspectJ [ASPJ] for Java or PostSharp [POST] for .NET. As discussed in section 2.2.2 the PostSharp weaver will be used in this thesis.

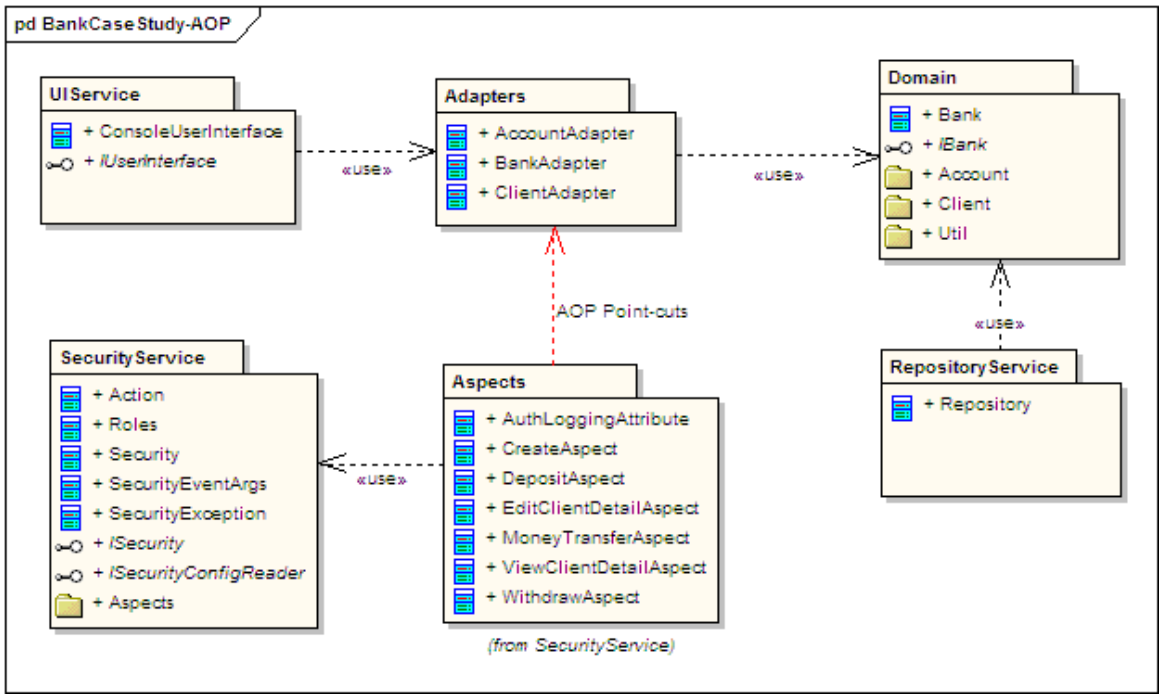

Figure 24: Class diagram with AOP point-cuts on join points in the adapters

If adapters are already present in an implementation the choice can be made to apply the point-cuts to these adapters.
To perform actions the UI service authenticates with the security class implementation. Based on the authentication credentials supplied by the UI service a corresponding user object is created with the appropriate role set. This user representation is saved and used by the security service to authorise new actions of the UI service.
Interactions with the domain implementation originating from the UI service are handled by the security adapters.

In PostSharp defines several base types of aspects, one of these is the OnMethodBoundary aspect [PDOC]. The OnMethodBoundary aspect overrides both OnEntry, and OnExit methods. On entry of the method, the call is intercepted and the code specified in the OnEntry method in the aspect is executed. On exit of the method, the return value or the exception is received in the OnExit method, which allows altering or filtering of the return values to match the user's authorisation level. Other types of aspects can be used to apply AOP at different locations and for different uses. Examples are aspects that react to exceptions or that are invoked with getters and setters of properties in a class.
The OnMethodBoundary aspect is useful in our case, because the aspect can regulate whether or not to proceed with a method invocation and modify the response or throw an exception.
The OnEntry advice is used to test the security conditions when a method is invoked. Throwing an exception in this code segment protects access to the domain methods. This behaviour is shown in *Code Fragment 4*.

```
Public class EditClientDetailAspect : OnMethodBoundaryAspect
{
    private readonly Action action;

    public EditClientDetailAspect(Action action)
    {
        this.action = action;
    }

    public override void OnEntry(MethodExecutionEventArgs eventArgs)
    {
        ClientAdapter clientAdapter = (ClientAdapter)eventArgs.Instance;

        if (!clientAdapter.user.RoleSet.IsEditClientPermitted(action,
               clientAdapter.client))
        {
           security.GenerateSecurityException(
              "You are not allowed to alter the client details.");
        }
    }
}
```

Code Fragment 4: Aspect advice in EditClientDetailAspect

Code Fragment 4 displays the aspect called EditClientDetailAspect. This aspect is invoked with an action, in this case either EditClientName or EditClientAddress, which is used to check permission.
On invocation the OnEntry method retrieves the instance of the ClientAdapter in which the method is inserted. This instance of the ClientAdapter has a reference to the user object that represents the authentication of the user that is responsible for this action. The role set of the user is checked and results in a security exception if the action is disallowed. In case the action is allowed, the execution continues to the method that was originally invoked. This sequence in the code is also shown in *Figure 25*.

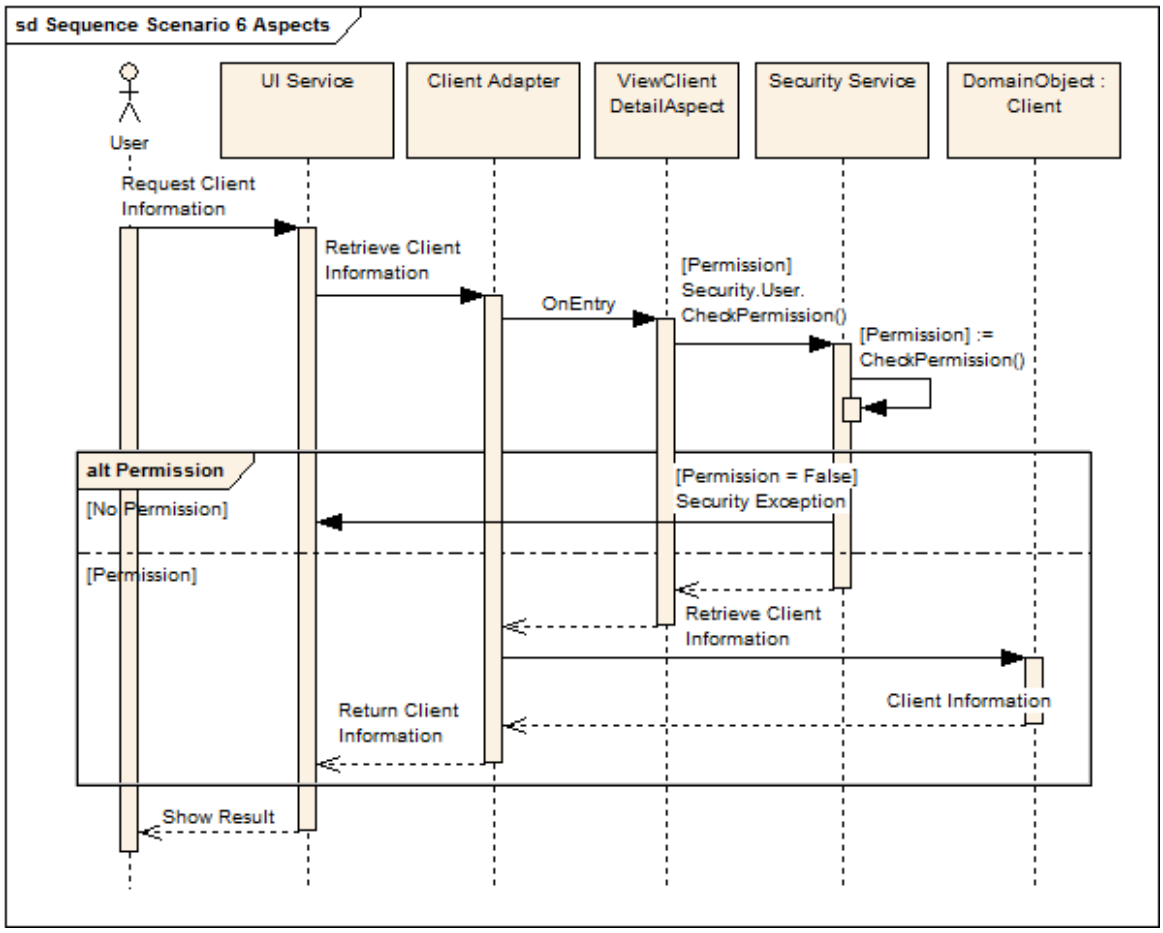

Figure 25: Sequence diagram of the OnEntry advice

In order for the aspect advice to be applied, AOP requires point-cuts in the source code as shown in *Code Fragment 5*.

```
[EditClientDetailAspect(Action.EditClientName)]
public void SetName(string firstName, string lastName)
{
    this.client.FirstName = firstName;
    this.client.LastName = lastName;
}

[EditClientDetailAspect(Action.EditClientAddress)]
public void SetAddress(string houseNumber, string street, string city)
{
    this.client.HouseNumber = houseNumber;
    this.client.Street = street;
    this.client.City = city;
}
```

Code Fragment 5: Join points with point-cuts in the ClientAdapter

Above the method SetName in *Code Fragment 5* the point-cut declares that the advice EditClientDetailAspect is applied. This aspect is applied to the action of editing of a client's name. The second example method, SetAddress, has the same aspect applied to the action of editing a client's address.

In the implementation the choice is made for explicit point-cuts in the source code so the security aspect interceptions are clearly visible to the developer (see *Code Fragment 5*). By using point-cuts in this way it is immediately clear to developers that both methods are checked by security.
These point-cuts can also be specified in another way by using the assembly. This method offers less clarity because it is not immediately clear what aspects apply at the method level (see *Code Fragment 6*). By declaring the point-cuts using the assembly the point-cut can be applied to several methods with one declaration.
In both examples in this fragment the Deposit method is targeted in the AccountAdapter class.

```
[assembly: DepositAspect(
    AttributeTargetTypes = "Adapters.AccountAdapter",
    AttributeTargetMembers = "Deposit")]
[DepositAspect]
public void Deposit(decimal amount)
```

Code Fragment 6: Different ways of defining a point-cut in the AccountAdapter

Due to implementation specifics of the case study, not all aspects enjoy the same versatility as the aspect that restricts access to editing or printing of client details, which can be applied to multiple methods. An example of a less versatile aspect is the DepositAspect shown in *Code Fragment 7*. This aspect is so specific it can only be applied to one method in the adapters.

```
Public class DepositAspect : OnMethodBoundaryAspect
{
    public override void OnEntry(MethodExecutionEventArgs eventArgs)
    {
       AccountAdapter accountAdapter =
            (AccountAdapter)eventArgs.Instance;

       if (!accountAdapter.user.RoleSet.IsDepositPermitted(
              (Decimal)eventArgs.GetArguments()[0],
              accountAdapter.account))
       {
           security.GenerateSecurityException(String.Format(
              "You are not allowed to deposit {0} on this
              account.", eventArgs.GetArguments()[0]));
       }
   }
}
```

Code Fragment 7: Aspect advice in DepositAspect

The DepositAspect in *Code Fragment 7* checks if the amount deposited is allowed. The amount passed to the Deposit method in the AccountAdapter is retrieved using eventArgs.GetArguments()[0].
This aspect is specifically written to accommodate checking of the deposit method in the AccountAdapter. The problems that prevent application to multiple methods are the aforementioned retrieval of the amount and the method used in the RoleSet class. The RoleSet class contains a specific method to handle a deposit transaction. If the RoleSet class was structured in a way more adhering to the OO principles, this could have been prevented.

This example shows the difficulties that are encountered to define one aspect that covers crosscutting concerns. Already in the design phase the developer needs to consider versatility and flexibility of the methods. If the RoleSet is designed with a method IsTransactionPermitted one aspect can be created that handles all transactions: withdrawals, transfers and deposits. In this implementation, this is separated in three aspects.

Correctly and efficiently dealing with crosscutting concerns by using AOP is difficult on the architectural level. Authorisation as a whole is a crosscutting concern, so in order to deal efficiently with this concern one aspect can be defined that is applied to every method that requires authorisation. The problem is that so many methods and properties need to be checked, all with largely varying method signatures and with different fail criteria. The diversity in these methods results in an aspect that probably contains a large if/else structure or a large switch case on the action that is performed. This is a sign of bad application of OO principles. This aspect can be split in multiple aspects that deal with specific sub-sets of actions. A drawback is that if a sub-set becomes too small (i.e. targets one method) the efficiency gain of using AOP is reduced.

5.5 Scenario 7 implementation: Security service integrated by AOP

This scenario applies AOP as described in section *4.7*. The point-cuts are aimed at the domain implementation. This AOP variation results in different problems and options in the implementation.

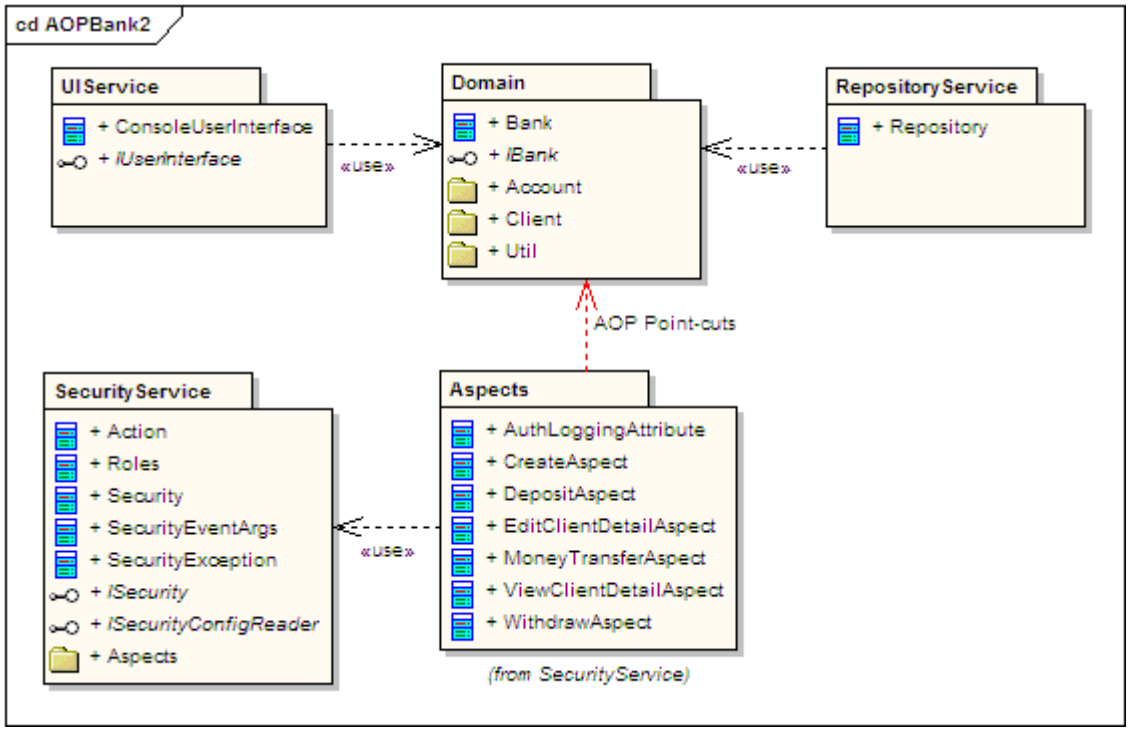

Figure 26: Class diagram with AOP point-cuts on join points in the domain implementation

Differences between the implementation with point-cuts aimed at the domain implementation instead of the adapters lie in the retrieval of the user object by the aspects. Determining which user is responsible for actions in the domain implementation is harder in a multi UI service environment without altering the domain implementation. This is because the domain implementation does not contain a definition of users and authorisation. The issuers of the interactions with the domain implementation are not known. In this case study implementation the connection between an instance of the UI service and an interaction is quite easy because there is only one simultaneous user. Multi UI service environments need a location to perform session management. This could be, for example, performed with Windows Authentication in the .NET environment.

Choosing PostSharp introduces some oddities in the development structure. With the current PostSharp implementation as it is, point-cuts need to be declared in the domain implementation. In the previous scenario, where point-cuts were defined in the adapters this did not pose a problem as the adapters were already aware of the security implementation.
If the domain implementation and the aspects are in different assemblies this has the effect that a circular dependency is created between the aspects that use the domain interfaces, and the domain classes being targeted by the aspects. To prevent this circular dependency, the domain interfaces can be separated into a different assembly from the domain implementation, or the aspects can be placed within the same assembly as the domain implementation.
If this implementation would be made in Java using AspectJ this problem would not occur because point-cuts are declared in the aspects.
Using PostSharp also has the drawback that it complicates debugging. Because weaving is performed after the Visual Studio compiler has compiled the source code, the Visual Studio debugger does not always accurately display the correct stack trace for aspect code. Some exceptions thrown in the aspect code seem to have a different stack origin.

For this implementation, the aspects are placed in the same assembly as the domain implementation because of the problems related to PostSharp.

Except for the difference that the aspects use the domain interfaces instead of the adapters, this implementation contains the same AOP aspects as seen in the 'AOP with adapters' implementation in section *5.4*.

```
public class EditClientDetailAspect : OnMethodBoundaryAspect
{
    private readonly Action action;

    public EditClientDetailAspect(Action action)
    {
        this.action = action;
    }

    public override void OnEntry(MethodExecutionEventArgs eventArgs)
    {
        IClient client = (IClient)eventArgs.Instance;
        ISecurity security = Repository.Get<ISecurity>();
```

```
        if (!security.user.RoleSet.IsEditClientPermitted(action,
              client))
        {
           security.GenerateSecurityException(
              "You are not allowed to alter the client details.");
        }
    }
}
```

Code Fragment 8: Aspect advice in EditClientDetailAspect

Code Fragment 8 shows the functionality corresponding EditClientDetailAspect in this implementation. Compared to the adapter implementation as seen in *Code Fragment 4*, the only difference is that the user object is retrieved from the security interface instead of from the ClientAdapter.

```
[EditClientDetailAspect(Action.EditClientName)]
public void SetName(string firstName, string lastName)
{
    this.FirstName = firstName;
    this.LastName = lastName;
}

[EditClientDetailAspect(Action.EditClientAddress)]
public void SetAddress(string houseNumber, string street, string city)
{
    this.HouseNumber = houseNumber;
    this.Street = street;
    this.City = city;
}
```

Code Fragment 9: Join points with point-cuts in the Client class

As seen in *Code Fragment 9* the point-cuts are moved from the adapter implementation to the domain implementation. This introduces some unwanted pollution in the source code of the domain implementation.
Besides these indicated differences the case study implementation works the same as discussed in section *5.4*.

Chapter 6
Measurements

This chapter presents and evaluates the measurements derived from the case study implementations. These measurements provide a possibility to determine the usability of the scenarios for developers.

The measurements taken for the new scenarios are compared with the measurements performed on the reference implementation from section *3.2* to determine to which extent these scenarios improve the reference implementation.
The measurements of the CBO and CC values in the implementations are performed by using NDepend [NDEP08].
In the measurements of the CC values only the relevant classes are taken into account. Relevant classes are the classes that have different implementation in each of the scenarios. The relevant classes consist of the Domain, Aspect and Adapter classes. NDepend is unable to correctly display measurements in case AOP aspects are used, since it takes measurements after the aspects have been woven into the point-cuts. In the scenarios that apply AOP a manual correction is applied to retain the correct measurement.

First both CC and CBO measurement results are presented, after which both measurements results are discussed in the concluding section.

6.1 Cyclomatic Complexity

To perform the CC measurements, NDepend will be used. In NDepend, the CC of a class is the sum of the number of methods in this class and the number of specific expressions seen in *Code Fragment 10*. By counting these expressions, the number of possible paths through the method is counted.

```
If | while | for | foreach | case | default | continue |
goto | && | || | catch | ternary operator ?: | ??
```

Code Fragment 10: List of code expressions counted in CC [NDEP08]

Reference bank case study
The CC values calculated in the reference bank case study implementation discussed in section *3.2* are shown in *Table 3*.

Domain	CC	CC/Method	DP
Client	25	/14=1.786	11
Account	18	/10=1.8	8
Bank	9	/7=1.286	2
Total	52	**Avg.** 1.624	21

Table 3: The CC values of the reference implementation

Bank case study Adapters

The measurements derived from the bank case study with the security service implementation using adapters discussed in section *5.3* are shown in *Table 4*.

Domain	**CC**	**CC/Method**	**DP**
Client	14	/14=1	0
Account	14	/10=1.4	4
Bank	7	/7=1	0

Adapters	**CC**	**CC/Method**	**DP**
ClientAdapter	18	/11=1.636	8
AccountAdapter	16	/10=1.6	6
BankAdapter	13	/9=1.444	6
Total	83	**Avg.** 1.347	24

Table 4: The CC values of adapter implementation

Bank case study AOP adapters

The measurements derived from the bank case study with the security service implementation using AOP with adapters discussed in section *5.4* are shown in *Table 5*.

Domain	**CC**	**CC/Method**	**DP**
Client	14	/14=1	0
Account	14	/10=1.4	4
Bank	7	/7=1	0

Adapters	**CC**	**CC/Method**	**DP**
ClientAdapter	12	/11=1.091	2
AccountAdapter	10	/10=1	0
BankAdapter	11	/9=1.222	4

Aspects	**CC**	**CC/Method**	**DP**
DepositAspect	2	/2=1	1
EditClientDetailAspect	3	/2=1.5	1
CreateAspect	5	/2=2.5	3
ViewClientDetailAspect	4	/2=2	3
WithdrawAspect	2	/2=1	1
MoneyTransferAspect	2	/2=1	1
Total	86	**Avg.** 1.309	20

Table 5: The CC values of the AOP adapter implementation

Bank case study AOP

The measurements derived from the bank case study with the security service implementation using AOP with the domain classes discussed in section *5.5* are shown in *Table 6*.

Domain	CC	CC/Method	DP
Client	14	/14=1	0
Account	14	/10=1.4	4
Bank	7	/7=1	0

Aspects	CC	CC/Method	DP
DepositAspect	2	/2=1	1
EditClientDetailAspect	3	/2=1.5	1
CreateAspect	5	/2=2.5	3
ViewClientDetailAspect	4	/2=2	3
WithdrawAspect	2	/2=1	1
MoneyTransferAspect	2	/2=1	1
Total	53	**Avg.** 1.378	14

Table 6: The CC values of the AOP implementation

6.2 Coupling Between Objects

Because of the relative small scale of the bank case study, the domain implementation only consists of three classes that are used by services. The CBO measurements are performed on the method level instead of the class level. The number of methods that fan-in to the measured class or namespace gives a better indication of the degree of coupling to the other objects in this relatively small case study.

The amount of coupling concerning the domain implementation is measured by counting the number of methods from other specified classes that interact with the domain implementation.

Reference bank case implementation
The reference bank case study implementation from section *3.2* contains a Bank/Client/Account implementation in the domain implementation that has Fan-in from other services and the security functionality. The Bank/Client/Account implementation also has Fan-out towards the security functionality in the domain implementation.

Fan-out from all services towards the Bank/Client/Account classes

Methods from all services directly using domain classes	22

Security functionality Fan-out towards Bank/Client/Account classes

Security methods using domain classes	10

Security Fan-in from Bank/Client/Account classes

Domain methods directly using security functionality	17

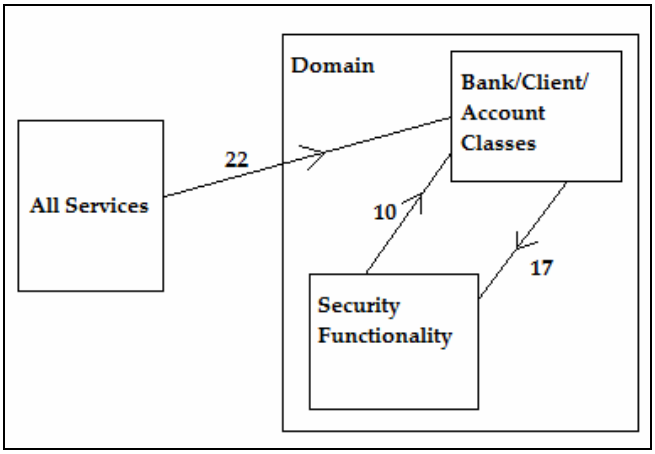

Figure 27: Coupling in the reference implementation

Adapters bank case implementation

The bank case study with the security service implementation using an adapter as seen in section *5.3*, has a clear and measurable fan-out from the security service towards the domain implementation.

Other coupling to measure is the coupling between the adapters and the rest of the security service. The security service classes are implemented in the same way as in the reference implementation.

Security service Fan-out to domain implementation

Security service (including adapters) methods directly using domain classes	32

Security service Fan-in from the adapters

Adapters to other security service classes	13

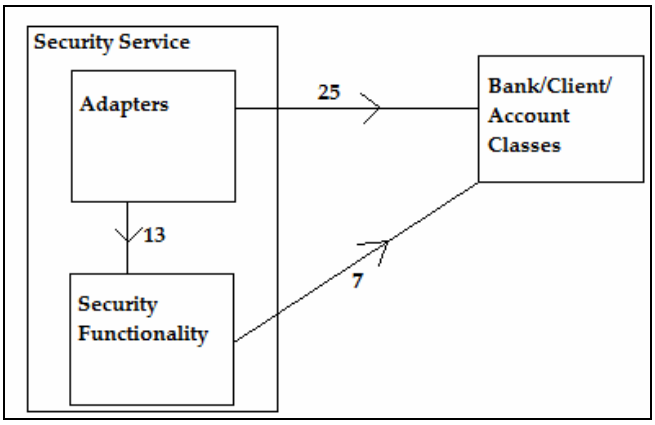

Figure 28: Coupling in the adapter implementation

AOP adapters bank case implementation

The measurements derived from the bank case study with the security service implementation using AOP with adapters as seen in section *5.4*.
Here the adapters are moved out of the security service so their coupling is measured separately.

Security service Fan-out to the domain implementation

Methods directly using domain classes (Domain state based checks)	7

Security service Fan-in from the adapters

Adapter methods directly using security functionality	3

Adapter method Fan-out to the domain implementation

Adapter methods directly using domain classes	25

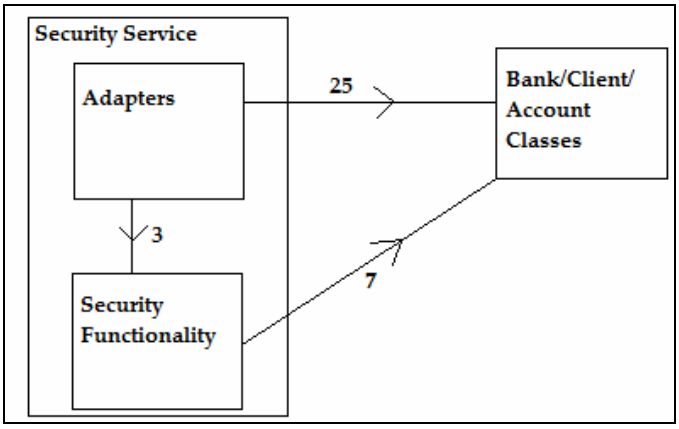

Figure 29: Coupling in the AOP adapter implementation

Bank case study AOP

The measurements derived from the bank case study with the security service implementation using AOP to the domain implementation as seen in section *5.5*. Here adapters are removed from the coupling overview and the coupling from other services to the domain implementation is reintroduced.

Fan-out from all services towards the domain implementation

Methods from all services directly using domain classes	22

Security service Fan-out towards the domain implementation

Methods directly using domain classes (Domain state based checks)	7

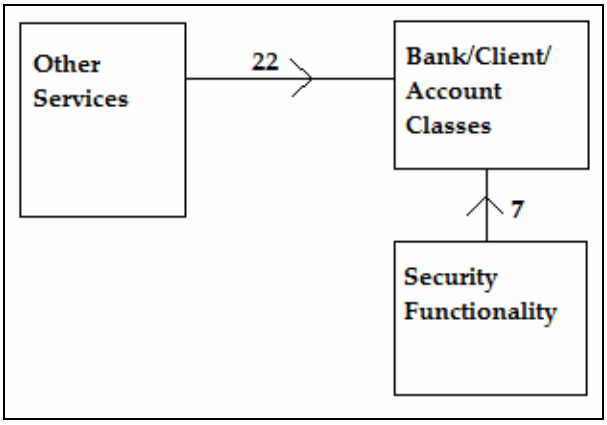

Figure 30: Coupling in the AOP implementation

6.3 Measurement comparison

The measurements presented in the previous two sections provide the possibility to compare the complexity and coupling for the implemented scenarios. This also provides the possibility to compare scenarios with the reference implementation from section *3.2*.

6.3.1 Cyclomatic Complexity

The CC measurements indicate how easy it is to implement and maintain the classes in the scenarios.

	Total CC	CC per method	DP
Reference scenario	52	1.63	21
Adapter scenario	83	1.35	24
AOP adapters scenario	86	1.31	20
AOP Domain classes	53	1.38	14

Table 7: Overview of the results of the Total CC, CC per method and DP measurements

The reference scenario domain implementation has a CC value of 52, which means circa 1.6 CC per method. In total the domain classes contain 21 Decision Points.

The regular adapter implementation reduces the CC of the domain implementation to 35, at the cost of increasing the total CC to 83. The average CC per method of the domain and adapters is lowered to circa 1.3. A slight increase in DP from 21 to 24 is caused by specifics of the adapter implementation. The positive effect of reducing the complexity in the domain is partially compensated by added complexity in the adapters. As the DP value shows, the overall complexity is not reduced, but is merely shifted.

In both AOP implementations, the point-cut annotations do not influence the CC values in the application because they do not influence the number of DP's.

The AOP implementation with adapters retains the lowered complexity in the domain implementation. By introducing AOP complexity is removed from the adapters and placed in the aspects. Total CC is higher with an increase to 86, average CC per method is slightly reduced to a lower region of 1.3. The impact of efficiently dealing with

crosscutting concerns is partially negated by the adapter implementation and results to a reduction of DP from 21 to 20.

The AOP implementation with point-cuts aimed towards the domain implementation eliminates the need for the intermediary adapters. The total CC amounts to 53, which is approximately the same as seen in the reference scenario with 52. The CC per method remains in the region between 1.3 and 1.4 compared to 1.6 in the reference scenario. The implication of using AOP to reduce complexity of crosscutting concerns is clearly reflected in this scenario because the DP value is reduced to 14. This is a large decrease when compared with the reference scenario where 21 DPs are found.

6.3.2 Coupling Between Objects

The CBO values indicate to what extent and in which manner the security functionality is separated from the domain-related functionality.

Separation of the security functionality into a DDD service is achieved in all three new case study implementations.
The remaining coupling between the security service and the domain implementation is measured using the CBO method.

	Domain Fan-in	Domain Fan-in from security adapters	Domain Fan-in from security	Domain Fan-in from other services	Domain Fan-out to security	Coupling security and security adapters
Reference scenario	32	0	10	22	17	0
Adapter scenario	32	25	7	0	0	13
AOP adapters scenario	32	25	7	0	0	3
AOP domain classes scenario	29	0	7	22	0	0

Table 8: Overview of the results of the CBO measurements

The reference scenario presents the implementation where the security implementation is located in the domain implementation. A strong coupling between the bank/client/account implementations and the security implementation exists. The fan-in consists of 17 methods from the domain classes that use a part of the security implementation. The fan-out from outside the domain classes consists of 32 methods that use a domain object (this includes 10 calls from the security implementation).
All three new implementations with security as a service succeed in eliminating the fan-in from the domain implementation.

The regular adapter implementation introduces the adapters together with a strong coupling to the security service. The fan-out from the security service towards the domain implementation stays the same with 32 methods that use domain objects. The

adapter implementations are coupled to the security implementation by 13 methods using objects in the security implementation.

The AOP implementation with adapters mostly eliminates the coupling between the adapters and the security service. Only 3 adapter methods still make use of the security implementation objects. In this implementation a clear separation is found between adapter methods using the domain implementation with 25 methods and the methods used by the security implementation with 7 methods that use domain objects for domain state based checks.

The AOP implementation with point-cuts aimed towards the domain implementation has no need for adapters and only the 7 methods that use domain classes for domain state based checks remain. Besides these security methods the other services now interact with the domain implementation and cause 22 methods to use the domain classes, the same number as seen in the reference implementation.

6.4 Results

The scenarios are compared on the areas defined in our goals:

- Separating security functionality and domain-related functionality
- Usability and efficiency
- DDD principle compliance

Separating security functionality and domain-related functionality
In all prototype implementations the fan-out from the domain implementation is removed from the application. This means the security functionality is separated from the domain-related functionality in the domain implementation.
Concerning CBO the AOP implementation with point-cuts towards the domain implementation causes the lowest coupling towards the domain implementation.

Usability and efficiency
In all implementations the average CC per method becomes lower than seen in the reference implementation. The CC per method drops from 1.6 per method to around 1.3 per method. This indicates that the complexity in the relevant parts of the implementation is reduced. All implementations result in approximately the same value for complexity per method.
The DP measurements indicate that the AOP implementation with point-cuts towards the domain implementation achieves the lowest DP value in the measured classes.

DDD principle compliance
The AOP implementation with adapters results in removing most of the coupling between the adapters and the security service.
The implementation with adapters does not comply fully with the DDD target situation. As it is implemented, the adapters are part of the security service. This is not a strict separation of concerns, because forwarding the requests from other services is not the

concern of the security service. In case the adapters are relocated into a glue layer the adapter scenarios do comply with the DDD target situation.

The AOP implementation with point-cuts in the domain implementation removes the need for intermediary adapters. This design results in a simple layout that resembles the sunflower model depicted in *Figure 3*.
This AOP implementation complies with the requirements stated in the goals. However, a drawback is introduced by the implementation in .NET (PostSharp). This implementation requires that the point-cuts are defined in the targeted class or assembly. This causes the point-cuts towards the domain implementation to be known in the domain implementation, which is undesirable behaviour because the point-cuts are a part of the security functionality. This means the domain implementation has dependencies on a service. In AspectJ (for Java) the point-cuts are declared in the aspects, this means this unwanted behaviour does not appear.

Conclusion
Concerning the criteria defined in our goals it becomes clear that the two AOP implementations are the most compliant with the DDD target situation as presented in section *3.3*.
Both AOP implementations come closest to the DDD target situation, although both have minor problems. The prototype implementation with AOP point-cuts in the domain implementation is the most elegant solution, as it is closest to the DDD target situation and provides the highest level of decoupling from the domain implementation and a reduction in application complexity.

Chapter 7
Conclusion

This chapter presents the conclusions of this Masters assignment. The results from the measurements from *Chapter 6* are compared to the initial goals from *Chapter 1*. This comparison results in guidelines to the implemented cases. The guidelines are presented to recommend the application possibilities.

7.1 Guidelines and recommendations

This thesis has shown that a security service that complies with the DDD target situation, as proposed in section 3.3, is possible in the bank case study. The compliance with the DDD target situation is achieved by using AOP. Achieving the separation of the security functionality from domain-related functionality is difficult without AOP.

As a result of this research the scenario with AOP point-cuts towards the domain implementation is recommended over the other scenarios.
AOP with point-cuts towards the domain implementation provides the best compliance with the DDD principles while achieving a reduction in the complexity of the application.

In situations where AOP point-cuts in the domain implementation are unwanted, the scenario with AOP in the adapters is a feasible solution. This scenario is preferred over the regular adapter implementation because an increased separation of concerns is achieved. This scenario is also usable with already existing adapters.

Considering the goals of this Masters assignment the adapter scenario is the least recommended. This scenario has the benefit that it is usable with current techniques and current adapters. The usage of adapters has the benefit that developers do not need the knowledge, training or technology that is required for the AOP solutions. This scenario has the drawback that the same result is achieved with more source code, which

Recommendations for integrating the security service AOP
The principle of AOP point-cuts in the domain implementation complies with the DDD principles. The point-cuts in the domain implementation introduce the risk of aspect overuse. When aspects are overused, they could implement domain logic. In DDD the domain objects should incorporate their behaviour and domain logic.
An added issue in this scenario is user identity, since actions in the domain implementation need to be correlated with a user.
Point-cuts to the domain implementation yield a significant reduction in the number of decision points because of the crosscutting aspect of security.

When AOP is chosen, it should be used efficiently. For AOP to work efficiently the application must facilitate the usage of AOP. An application layout adhering to Object Oriented principles facilitates efficient usage of AOP.
Correctly and efficiently dealing with crosscutting concerns by using AOP is difficult on the architectural level. The problem is that so many methods and properties need to be checked, all with largely varying method signatures and with different fail criteria with different error messages. When this variation concerns one crosscutting concern, it should ideally be handled by one aspect.
This one aspect can be split in multiple aspects that deal with specific sub-sets of actions. A drawback is that if a sub-set becomes too small (i.e., targets one method) the efficiency gain of using AOP is largely reduced.

During the implementation of the AOP scenarios we found that aspects are also suitable to enforce business rules in the domain implementation. But according to the DDD principles the business logic should remain in the domain classes. An example from the case study is the checking for negative deposits, this is a domain logic issue and should not be handled by an aspect.
The goal of AOP should be about efficiently removing crosscutting concerns out of domain implementation, not removing business logic.

7.2 Evaluation

The main goals of this Masters assignment was to provide insight into the coupling of services in DDD and to find the best design options to separate security functionality from domain-related functionality.

Clarifications about DDD, and insight into the coupling of services in DDD resulted in using the CBO metric to illustrate the different kinds of services in a DDD environment. This insight provided the possibility to determine an approach to separate a service that requires the domain implementation to interact with it from the domain implementation. The security functionality has to be aware of interactions with the domain classes, before the interactions reach the domain implementation.

To find the most suitable design options a case study has been chosen where the problems with separating the authentication and authorisation functionality from the domain implementation were clear. Based on this situation a target DDD situation has been presented with different criteria. The formalisation of these requirements resulted in a reference design to which new designs have been compared. Different designs have been presented, each with their own advantages and disadvantages. After creating the designs, a selection had to be made due to the number of options. This selection was difficult because there are no metrics for measuring coupling on the model level. Without actual implementation of the classes, known metrics as described in [MOOD95], [CHID94] cannot be used because these metrics involve the analysis of source code. In this Masters assignment the model only consisted of a package structure with an outline of the possible classes within these. It modelled how the interactions between these packages would take place.

The most promising design scenarios have been selected for implementation. The implementation resulted in more clarity about the usability of the scenarios, and revealed problems with defining the location of point-cuts in PostSharp.
After the implementation, the measurements provided the knowledge about the actual impact and reduction of coupling in the case study application.

Based on the knowledge gained in the scenario implementation, the AOP design can be applied to the placing of security within DDD. The solutions to the problems in the case study are also applicable for other DDD implementations. By using AOP the security functionality has knowledge about the interactions with the domain implementation and the first main problem stated in section *1.2* is resolved. The second main problem stated in section *1.2* was that separation becomes problematic when the state of the domain implementation plays a role in the access to the domain implementation.
Because the state of the domain must be considered for authorisation, the security functionality has domain specific checks embedded in the functionality. This still reduces the portability of the security functionality. The domain state checks are essential complexity introduced by access control.

The advice is to implement security with AOP because it provides benefits by separating the concern of security into a service. Instead of security scattered throughout the domain implementation.
The PostSharp problem is specific for the PostSharp implementation. Other AOP implementations doe not have the problem that point-cuts should be declared in the domain implementation.

7.3 Future work

Session management is still an open issue when AOP point-cuts are used in the domain implementation. This issue is not resolved within this Masters assignment.

This Masters assignment was restricted to solutions that are platform-independent. Interesting topics related to this Masters assignment are platform-specific solutions. For both .NET and Java specific security frameworks and features exist.

Code Access Security (CAS) in the .NET environment can be used in the code to regulate access based on Windows user authentication. This can be a solution for authentication in an environment with windows clients.

In the Java environment opportunities are provided by Spring Security [ACEG08] and JBoss Seam [SEAM07].
Spring security was formerly known as Acegi security. Spring security is a solution for the Java EE platform and can be used as a part of the Spring framework. Its compliance with the DDD principles is unknown. Spring security by default provides the security infrastructure for a web-based architecture. The Spring security implementation also makes use of AOP.

JBoss Seam is an application framework for EJB 3.0. JBoss Seam combines the two frameworks Enterprise JavaBeans (EJB3) and Java Server Faces (JSF). It greatly reduces the amount of glue code needed when working with both frameworks.
JBoss Seam security is a part of the application framework and provides the security functionality needed in this framework. For this security functionality and the JBoss Seam functionality, the compliance with the DDD principles is also unknown.

In this Masters assignment, issues related to web deployments were not included. Further research can be performed by reviewing how the result of this Masters assignment applies to web-deployed applications.
When a web-deployed application is used the authentication procedure can be coupled to web-server sessions.
If the AOP scenario is reused in a web deployment, it is unknown what the restrictions are on the AOP implementation.
A final question concerning web-deployed applications could be how to map the application model onto a three-tier model [ECKE95], which is widely used for web-deployments.

References

[ACEG08]	Interface21, *"Acegi Security - Acegi Security System for Spring"*, accessed March 2008, http://www.acegisecurity.org/
[AOP08]	Markus Voelter, *"Aspect-Oriented Programming in Java"*, accessed March 2008, http://www.voelter.de/data/articles/aop/aop.html
[ASPJ]	The Eclipse Foundation, *"The AspectJ Project"*, accessed March 2008, http://www.eclipse.org/aspectj/
[BANK07]	Sogyo, *"Bank case"*, August 2007, Code & documentation
[BROO86]	Fred Brooks, *"No Silver Bullet – Essence and Accident in Software Engineering"*, Proceedings of the IFIP Tenth World Computing Conference, pp. 1069-1076, 1986
[CCMB89]	Thomas J. McCabe / Charles W. Butler, *"Design Complexity Measurement and Testing"*, Communications of the ACM, 32, pp. 1415-1425, December 1989.
[CHID94]	Chidamber / Kemerer, *"A Metric Suite for Object-Oriented Design"*, IEEE Transactions on Software Engineering, vol. 20 no. 6, June 1994, pp. 476-493. http://web.cs.wpi.edu/~gpollice/cs562-s05/Readings/CKMetrics.pdf
[COMP03]	Trey Guerin / Richard Lord, *"How role-based access control can provide security and business benefits"*, November 2003, http://www.computerworld.com/securitytopics/security/story/0,10801,86699,00.html
[COOK06]	William R. Cook / Ali H. Ibrahim, *"Integrating Programming Languages & Databases: What's the Problem?"*, September 2006, http://www.odbms.org/experts.html#article10
[DDDO]	Domain Language Inc., *"Domain-Driven Design: What Is It?"*, accessed March 2008, http://www.domaindrivendesign.org
[DDDO03]	OOPSLA, *"OOPSLA Domain Driven Design"*, 2003 / 2006 / 2007, http://www.oopsla.org/oopsla2003/files/tut-32.html http://www.oopsla.org/2006/submission/tutorials/domain-driven_design:_putting_the_model_to_work.html http://www.oopsla.org/oopsla2007/index.php?page=sub/&id=76
[DDMD]	Eric Evans, *"Domain-Driven / Model-Driven"*, February 2004, http://domaindrivendesign.org/discussion/blog/evans_eric_ddd_and_mdd.html
[DIJK74]	E. W. Dijkstra, *"On the role of scientific thought"*, Selected writings on Computing: A Personal Perspective, New York, NY, USA: Springer-Verlag New York, Inc., pp. 60–66
[EADD07]	Marc Eaddy / Alfred Aho / Gail C. Murphy, *"Identifying, Assigning, and Quantifying Crosscutting Concerns"*, International Conference on Software Engineering: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques p. 2 , 2007

[ECKE95]	Wayne Eckerson, *"Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications."*, Open Information Systems 10, 1 (January 1995): 3(20)
[EVAN03]	Eric Evans, *"Domain Driven Design: Tackling Complexity in the Heart of Software"*, Addison-Wesley Professional 2003, **ISBN-13:** 978-0321125217
[EVAN06]	Eric Evans / edited by Abel Avram and Floyd Marinescu, *"Domain-Driven Design Quickly"*, December 2006, http://www.infoq.com/minibooks/domain-driven-design-quickly
[GOF95]	Erich Gamma / Richard Helm / Ralph Johnson / John M. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley Professional 1994, **ISBN-13:** 978-0201633610
[FOWL02]	Martin Fowler, *"Patterns of Enterprise Application Architecture"*, Addinson Wesley Professional 2002, **ISBN-13:** 978-0321127426, http://martinfowler.com/eaaCatalog/
[HAYW04]	Dan Haywood, *"MDA: Nice idea, Shame about the…"*, TheServerSide, May 2004, http://www.theserverside.com/tt/articles/article.tss?l=MDA_Haywood
[IBMR06]	IBM Research, *"Model-Driven Software Development"*, IBM Systems Journal *"MDSD"* Volume 45 Number 3 2006, http://www.research.ibm.com/journal/sj/453/hailpern.html p.451 http://www.research.ibm.com/journal/sj/453/balmelli.html p.569
[JAVA08]	SUN, *"Java.com: Java + You"*, accessed March 2008, http://www.java.com/en/
[KICZ96]	G. Kiczales / J. Irwin / J. Lamping / J.-M. Loingtier / C. V. Lopes / C. Maeda / A. Mendhekar, *"Aspect-oriented programming"*, ACM Computing Surveys, 28(4es):154, 1996.
[LADD02]	Ramnivas Laddad, *"Separate software concerns with aspect-oriented programming"*, Javaworld.com, January 2002, http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html?page=1
[MDA04]	Alan Brown, *"An introduction to Model Driven Architecture"*, IBM Developer Works, February 2004, http://www.ibm.com/developerworks/rational/library/3100.html
[MOOD95]	Fernando Brito e Abreu, *"Design Quality Metrics for Object-Oriented Software Systems"*, ERCIM News No.23 – October 1995
[MSFT08]	Microsoft Corporation, *"Microsoft .NET Framework"*, accessed March 2008, http://www.microsoft.com/NET/
[NDEP08]	SMACCHIA.COM S.A.R.L, *"NDepend"*, accessed March 2008, http://ndepend.com/
[NILS06]	Jimmy Nilsson, *"Applying Domain Driven Design and Patterns"*, Addison-Wesley Professional 2006, **ISBN-13:** 978-0321268204
[ORM97]	Scott W. Ambler, *"Mapping Objects to Relational Databases"*, 1997 modified in 2000, http://www.crionics.com/products/opensource/faq/docs/mappingObjects.pdf
[OMG08]	Object Management Group Inc., *"Object Management Group"*, accessed March

	2008, http://omg.org/
[PARN72]	D. L. Parnas, *"On the Criteria To Be Used in Decomposing Systems into Modules"*, Communications of the ACM, 15(12):1053–1058, 1972.
[PDOC]	Gael Fraiteur and the postsharp.org Community, *"PostSharp 1.0 Online Documentation"*, accessed March 2008, http://doc.postsharp.org/
[PIAB07]	Tom Hollander, *"Announcing the Policy Injection Application Block"*, Microsoft Corporation, accessed March 2008, http://blogs.msdn.com/tomholl/archive/2007/02/23/announcing-the-policy-injection-application-block.aspx
[POST]	Gael Fraiteur, *"PostSharp - Bringing AOP to .NET"*, accessed March 2008, http://www.postsharp.org/
[SDWZ07]	E. van Dillen / R. Wolter / T. Zeeman, *"Domain Driven Design: Achtergronden en ervaringen uit de praktijk"*, Software Release magazine April 2007, http://www.sogyo.nl/wp-content/uploads/Publicaties/DDD_SD_April_2007.pdf
[SEAM07]	Red Hat Middleware LLC, *"JBoss.com - JBoss Seam"*, accessed March 2008, http://jboss.com/products/seam
[SMAL08]	Smalltalk.org and Active Information Corporation, *"Smalltalk.org"*, accessed March 2008, http://www.smalltalk.org/
[SRIN07]	Srinivasan Tharmarajah, *"Aspect Oriented Programming in Domain Driven Design"*, August 2007, http://homepages.cwi.nl/~paulk/thesesMasterSoftwareEngineering/2007/SrinivasanTharmarajah.pdf
[WBMC03]	Rebecca Wirfs-Brock / Alan McKean, *"Object Design: Roles, Responsibilities, and Collaborations"*, Addison-Wesley Professional November 2002, **ISBN-13:** 978-0201379433
[WOLT05]	Ralf Wolter, *"Thesis"*, 2005, Sogyo
[YDDD]	Yahoo Domain-Driven Design discussion group, *"domaindrivendesign : Domain-Driven Design"*, accessed March 2008, http://tech.groups.yahoo.com/group/domaindrivendesign/
[YASE07]	Yasser EL-Manzalawy, *"Aspect Oriented Programming"*, accessed March 2008, http://www.developer.com/design/article.php/3308941