Exploring for Architecture
“Beyond the module viewtype”

Pieter Hartman
Exploring for Architecture: “beyond module viewtype”

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Pieter Hartman
born in Dordrecht, the Netherlands

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
WWW.EWI.TUDELFT.NL

AEGON Nederland NV
Service Center Pensioen
Aegonplein 50
the Hague, the Netherlands
WWW.AEGON.NL
Exploring for Architecture: “beyond module viewtype”

Author: Pieter Hartman
Student id: 1047388
Email: P.A.C.Hartman@gmx.net

Abstract

Keywords: reverse engineering, component and connector, views, viewtypes, Symphony, software architecture, OfferteNet

The software architecture description is the most important document for software engineers. AEGON experienced some problems that could not be solved without a good description of their software.

TU Delft currently developed Symphony, a framework for reporting reconstruction approaches. Because Symphony results in a uniform description it is even possible to evaluate reconstruction approaches. This method is used in a case study to recover the information to solve the problems.

In this reconstruction process we leave the classic module viewtype and we explore more dynamic structures such as the Component and Connector viewtype. Component and Connector views define models consisting of elements that have some runtime presence. In short a component and connector view is a picture of runtime entities and potential interactions.

In this report we introduce software architecture, especially the component and connector viewtype. Later on we present the reconstruction results and finally we will give some recommendations for improving OfferteNet.

Thesis committee:

Chair: Prof. Dr. A. van Deursen, Faculty of EEMCS, TU Delft
University Supervisor: Dr. Ing. L. Moonen, Faculty of EEMCS, TU Delft
Company Supervisor: Drs. H. Brouwer, Service Center Pensioen, AEGON NV
Committee Member: Dr. K. van der Meer, Faculty of EEMCS, TU Delft
Preface

September 1994, a youngster goes to high school. He has one goal, becoming an engineer. For him no such lessons as history, geography but mathematics and physics.

September 2000, a young man for the first time at Technical University Delft. He studies Computer Science, in order to become an excellent software engineer. The faculty mission is stated on their website:

“The focus of Software Engineering is on the development of software within technical and economical boundaries. This includes all aspects of software engineering (software architecture, requirements specification, design, implementation, validation and verification) of (large scale) software systems.”

Four years of study have not changed the young man’s goal. But the course Software Architecture Recovery and Modelling changed his goal. During that course the young man learned techniques and methods for architecture recovery of complex software systems. Furthermore he learned how recovered software architectures can be used for system exploration, in order to improve program understanding and to check a system’s conformance to a reference architecture. He wants to become a reverse engineer.

As you will understand, the above mentioned person is the author of this thesis. During my master thesis I recovered a software system of AEGON. Reverse engineering helped me to explore secrets of the past. One of these secrets was the easter egg, with background music “Mission Impossible”. By recovering facts from the existing software system, I was able to improve future.

I want to thank Leon Moonen for giving me the insight that reverse engineering is more than only rewriting system documentation. Besides I want to thank Hein Brouwer, my supervisor within the company, for his support during my thesis project. Last but not least I want to thank you, as reader, for the interest shown in my work.

Enjoy reading,

PIETER HARTMAN
# Table of Contents

Preface ............................................................................................................................................. iii

Table of Contents ............................................................................................................................ v

List of figures and tables ............................................................................................................... ix

1. Introduction ............................................................................................................................... 1
   1.1. Environment ...................................................................................................................... 1
   1.2. Problem Statement ........................................................................................................... 1
   1.3. Current situation ............................................................................................................... 1
   1.4. Approach ......................................................................................................................... 3
   1.5. Document overview ....................................................................................................... 3

2. Documenting Software Architecture ...................................................................................... 5
   2.1. Importance of Software Architecture .......................................................................... 5
       2.1.1. Software Architecture ........................................................................................... 5
       2.1.2. Views ...................................................................................................................... 6
   2.2. Component and Connector viewtype ............................................................................... 7
       2.2.1. Elements ................................................................................................................. 8
       2.2.2. Relations and Properties ....................................................................................... 10
       2.2.3. Connector Abstractions ...................................................................................... 10
   2.3. Component and Connector notation ............................................................................. 11
       2.3.1. Rapide ................................................................................................................... 12
       2.3.2. Wright ................................................................................................................... 12
       2.3.3. Acme ..................................................................................................................... 13
       2.3.4. Unified Modelling Language .............................................................................. 14
       2.3.5. Conclusion ........................................................................................................... 15

3. Reconstructing with Symphony .............................................................................................. 17
   3.1. Views in Symphony .......................................................................................................... 17
   3.2. Symphony Steps .............................................................................................................. 18

4. Reconstruction OfferteNet ....................................................................................................... 19
   4.1. Problem Elicitation .......................................................................................................... 19
   4.2. Concept determination ................................................................................................. 20
   4.3. Exploration of OfferteNet ........................................................................................... 21
       4.3.1. Metrics of OfferteNet ........................................................................................... 21
       4.3.2. Source mapping ...................................................................................................... 22
       4.3.3. Discussion ............................................................................................................... 24
   4.4. First Iteration – Execution view ..................................................................................... 25
       4.4.1. Target, Source viewpoint and mapping rules ....................................................... 25
       4.4.2. Data gathering ........................................................................................................ 25
       4.4.3. Information Interpretation .................................................................................... 26
       4.4.4. Discussion ............................................................................................................... 26
   4.5. Second Iteration – Generate Decomposition View ....................................................... 27
       4.5.1. Target, Source viewpoint and mapping rules ....................................................... 27
List of figures and tables

Figures
Figure 1: Current situation ................................................................. 2
Figure 2: Target situation ................................................................. 3
Figure 3: Viewtypes – Styles – Views .................................................... 7
Figure 4: Structure of Component and Connector .................................. 8
Figure 5: Replacement Connector 1 ...................................................... 11
Figure 6: Client Server System with properties .................................... 13
Figure 7: Interaction during Reconstruction Design ............................... 18
Figure 8: Part of XML file (sourceview.xml) ........................................ 23
Figure 9: OfferteNet application overview .......................................... 24
Figure 10: Entities and relations .......................................................... 25
Figure 11: Part of first iteration ............................................................ 26
Figure 12 Part of decomposition view .................................................. 27
Figure 13: Example of a model ............................................................. 28
Figure 14: Fastlane Reader Pattern ....................................................... 29
Figure 15: Structure of components ....................................................... 29
Figure 16: FormulierBouwer getAlgemenenValidaties() .......................... 31
Figure 17: Overview of FormulierBouwerSLSB .................................... 32
Figure 18: getSectield() example ........................................................ 33
Figure 19: Abstracted sequence diagrams ............................................. 37
Figure 20: Hierarchical structure of OfferteNet packages ....................... 38
Figure 21: Screenshot OfferteNet with data mapping ............................ 41
Figure 22: Part of the OfferteNet data model ........................................ 42
Figure 23: Part of the OfferteNet data model ........................................ 44
Figure 24: Screenshot OfferteNet .......................................................... 44
Figure 25: Positioning Calculation kernel ............................................ 47
Figure 26: JCF architecture overview .................................................... 47
Figure 27: Example of Assignmentstep ................................................ 49
Figure 28: Example of Conditionstep .................................................... 49
Figure 29: Example of Datareferencestep ............................................. 50
Figure 30: Example of Formulastep ..................................................... 50
Figure 31: Example of Includestep ....................................................... 51
Figure 32: Throughput EJB implementations ....................................... 54
Figure 33: Part of unparsed document ................................................. 55
Figure 34: Naming of methods ............................................................ 56
Figure 35: Error handling in OfferteNet .............................................. 56
Figure 36: Use of error handling in normal flow ................................... 56
Figure 37: Use of isolated kernel in OfferteNet ..................................... 58
Figure 39: Two independent decisions ................................................ 61

Tables
Table 1: Architecture documentation and stakeholder communication needs ........ 6
Table 2: Summary of C&C viewtype ..................................................... 8
Table 3: Conclusions about notation style ............................................ 15
Table 4: Candidate view list ............................................................... 21
Table 5: Selected views .................................................................... 21
Table 6: Metrics .............................................................................. 22
Table 7: Mapping rules .................................................................... 23
Table 8: Used classes of beans and servlets ........................................ 34
1. Introduction

Internet technology has become more and more important. During the information technology hype at the end of the 19th century many web applications were developed. AEGON developed OfferteNet within two months. The software was finished, but a good software architecture description is until now not available. This is where software architecture reconstruction comes in. Software recovery is one of the research interests of the SoftWare Evolution Research Lab at Delft University of Technology. In this chapter we will describe the environment of AEGON, we will define our problem statement and finally we present our approach for this thesis project.

1.1. Environment

The practical work for this thesis project is done at AEGON in The Hague. AEGON the Netherlands is part of one of the largest insurance groups in the world. Their businesses focus on life insurance, pensions, savings and investment products. Furthermore AEGON sells accident, health and general insurance products and has limited banking activities. The United States, the Netherlands and the United Kingdom are the three major markets. But the group is also active in many other countries such as Canada, China, Hungary, Spain and Taiwan. The AEGON Group head office is in The Hague. Worldwide AEGON is the employer of 28,000 people [URL1].

In the Netherlands AEGON is divided into four marketing and sales organisations. These organisations are served by five professional service centres delivering new products, and services and doing the administrative work for the marketing and sales organisations. This master thesis project is done at Service Center Pensioen (SCP). SCP has about 500 employees of which about 30 people work at the front- and midoffice department.

1.2. Problem Statement

At SCP there are three software systems running; OfferteNet/AEBOT, BeheerNet and AEGON Pensioen Site. Currently these systems are J2EE (Java 2 Enterprise Edition) web applications that cannot interact with other systems (see figure 1). In the future these systems should better share common functionality. This means that there will be three presentation layers, which use all common services that provide shared functionality and data. (see figure 2).

The goals of this thesis are determining the possible improvements for the current situation, and create a mapping to the proposed architecture. This is performed with help of information we gathered during reconstruction. The results will not only be an advice for AEGON, but also a case study of Symphony.

1.3. Current situation

OfferteNet is a J2EE application for importing, calculating and printing AEGON pension proposals. OfferteNet is meant for employees and for employees of intermediaries. Users can access the application by means of a web-interface. For administrator tasks there is also an admin-interface.

A special OfferteNet characteristic is the data-driven product configuration. This enables a very short development time for new pension product user interfaces. At this moment there is no physical coupling with other systems. The OfferteNet application is not well documented. Currently there are some general documents, and a few technical business rule specifications. General architectural information is not available.
OfferteNet is hosted on a dedicated web server. This web server is installed with IBM WebSphere 4, which is based on J2EE 1.3. Furthermore there is a separate Oracle 9 database server. The servers are shared by all Service Centres.

Figure 1 Current Situation
1.4. **Approach**

We will recover the architecture of OfferteNet using Symphony. Symphony is a view driven framework. A view is a representation from a particular viewpoint. One of the first steps in this process is establishing an overview of the required views. After that we can reconstruct the selected views and make a comparison with the available documentation. Finally we will give some recommendations for OfferteNet improvement. One of the improvements should be the possibility to map OfferteNet on the target architecture.

1.5. **Document overview**

This report consists of two parts. In the first part we will first discuss the importance of software architecture. Next, the component and connector viewtype will be discussed and finally we will present some alternatives for its documentation. We focus on the component and connector viewtype, because in many software architecture descriptions nothing is written about system runtime behaviour. However, since this information is important it gives us answers to questions such as: How many instances of an object are instantiated? Which objects are really used by the system? What happens during software system execution?

The second part of this report is about architecture reconstruction, using the Symphony method. We will describe Symphony and then we will explain the steps we have taken to recover OfferteNet. The iteration process and results will be discussed in chapter 3. A detailed investigation of the complex system components can be found in chapter 6. The results of this reconstruction will lead to some recommendations in order to improve OfferteNet. These results are discussed in the last chapter.

---

**Figure 2 Target Situation**
2. **Documenting Software Architecture**

This chapter contains three sections. First we will describe the importance of software architecture. We will describe why we need software architecture and why we need more then one view. In section 2.2 we describe the component and connector viewtype. Finally we will discuss several notation styles for the component and connector viewtype.

### 2.1. Importance of Software Architecture

**2.1.1. Software Architecture**

The term software architecture has multiple definitions; no universal definition exists. The book “Software Architecture in Practice” defines it as follows: “The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them [CBB+02].” Perry and Wolf compare software architecture with building architecture, because there are many overlaps between these two. Both paradigms need for example multiple views and an architectural style. They argue that an architecture is a set of elements, which have a particular form. Their proposed model is:

\[ \text{Software Architecture} = \{ \text{Elements, Form, Rationale} \} \]

They defined three types of elements. First, there are the processing elements, which transform the data elements. Data elements contain the information that could be transformed. And the “glue” between data and processing elements are the connecting elements. An example of a connecting element is a procedure call or a message that is transmitted. The form describes the properties and relationships of a system. And the rationale is an explanation of the various choices made during architecture design [PW92].

But why do we need software architecture? Software engineering and especially software maintenance is an expensive task. Two factors cause high software costs. Firstly, software is dominated by a continuing process of changes. Secondly, software is adapted to new uses and functional requirements. This is called software evolution. In all changes a good system description is important. A software engineer cannot adapt the system without a software architecture description [MKM+97]. Perry and Wolf investigated what should be characteristics of an architectural description [PW92]:

- The architectural views should be presented to the desired level of detail – this means that we for example should consider how general / particular a description should be.
- It should be clear what are “must haves” and what is just luxury.
- Present the architectural information in multiple views, different aspects should be presented in separated views.
- It should be possible to do dependency and consistency analysis. For example the requirement logging is in contradiction with the performance requirement. Furthermore should it be possible to check consistency between architectural styles, between styles and architecture and between architectural elements.

This summarization gives us information about the content of an architecture description. Of course the architecture description itself has also several purposes. First, it has its educational value. New employees should be able to understand system’s behaviour within a short time. During development the architecture prescribes what should be true and afterwards it describes what is true. An architecture description proves its value also during system analysis. And finally it is used for communication with the stakeholders.
The stakeholders involved have all different communication needs and the architecture description should address those needs (table 1) [CBB+02].

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architect and requirements engineers who represent the customer(s)</td>
<td>A forum for negotiating and making trade-offs among competing requirements.</td>
</tr>
<tr>
<td>Architect and designers of the constituent parts</td>
<td>To resolve resource contention and to establish performance and other kinds of runtime resource consumption budgets. To provide inviolable constraints and exploitable freedoms on downstream development activities.</td>
</tr>
<tr>
<td>Implementers</td>
<td></td>
</tr>
<tr>
<td>Testers and integrators</td>
<td>To specify the correct black-box behaviour of the pieces that must fit together.</td>
</tr>
<tr>
<td>Maintainers</td>
<td>A starting point for maintenance activities, revealing the areas a prospective change will affect.</td>
</tr>
<tr>
<td>Designers of other systems with which this one must interoperate</td>
<td>To define the set of operations provided and required and the protocols for their operation. Basis for forming development teams corresponding to the work assignments identified, work breakdown structure, planning, allocation of project resources, and tracking of progress by the various teams.</td>
</tr>
<tr>
<td>Product line managers</td>
<td>To determine whether a potential new member of a product family is in or out of scope and if out, by how much.</td>
</tr>
<tr>
<td>Quality assurance team</td>
<td>Basis for conformance checking for assurance that implementations have in fact been faithful to the architectural prescriptions.</td>
</tr>
</tbody>
</table>

2.1.2. Views

As already mentioned in the previous section, a software architecture description is meant for several purposes and stakeholders. Each stakeholder requires different information about the system. For example a security analyst is only interested in parts of the system that interact with the environment, the data flow in the system, and the parts that handle authentication and authorisation. Every stakeholder should have information in a suitable view. That is why a software architecture description consists of many views. "A view is a representation of a whole system from the perspective of a related set of concerns [IEEE00]."

In building architecture several views are created before building starts. For example an electricity plan, a cross section of the building or a front view etc. Software architecture should also consist of multiple views. Each view should highlight a related set of architectural aspects. The definition of Clements et al is based on this knowledge: "An architecture is a set of relevant views and some documentation that applies to more than one view [CBB+02]."

Several models are developed to create complete software architectures. In 1995 Kruchten presented his "4+1" view model for software architectures. It consists of four main views and one fifth view. This fifth view is a set of scenarios that holds the other four together. The other views are [KRU95]:

- **Logical view** – describes the as designed object model. As alternative it is possible to use an entity relationship diagram for data driven systems.
- **Process view** – describes the concurrency and synchronization aspects of the design
- **Physical view** – this is the view that shows the mapping between software and hardware. Furthermore it describes all distributed aspects of the system.
- **Development view** – this view describes how the software modules, libraries and packages are organised in the file system
In the same period Soni, Nord en Hofmeister came up with a model to describe software architectures which has become known as the Siemens Four View Model and it consists of the following four views [SNH95]:

- **Conceptual structure** – describes all elements of a system and the relation between them.
- **Module structure** – this structure describes functional decomposition and layers. Functional composition is a description of the logical decomposition. It describes how modules, subsystems and abstract program units are decomposed. Secondly, the layers describe the interface and the interface constraints of the components.
- **Execution structure** – describes some runtime aspects of a system.
- **Code structure** – this view is almost the same as the development view of the 4+1 software model. It also describes the mapping of the software onto the file system.

The last model we will describe is from Clements et al. They use viewtypes to classify the various views on a software system. A viewtype describes a software system from a certain viewpoint and it defines types and relations that could be used in this viewtype. Each view falls into one of these three categories.

- **Module viewtype** – this view is the most used viewtype. It describes the static module organisation of a software system.
- **Component and Connector (C&C) viewtype** – this view describes all aspects of the system with a runtime character.
- **Allocation viewtype** – this view documents the relationship between the development environment and the software system.

Figure 3 describes the relations of viewtypes, styles and views. “A style is a definition of elements and relations that can be used in a view, together with a set of constraints and how they can be used. And a view is an instance of a style bound to a particular software system [CBB+02].”

We have chosen to use this last model, because it is problem driven: Styles and views are selected based on information needs by the different stakeholders. The other models we have described have a static set of structures and views. In the model of Clements et al. it is possible to apply a certain style within the set of viewtypes. We think that the possibility to choose styles could be important during the process. Furthermore Symphony is also problem driven, which means that we reconstruct only the views needed to solve the problems.

### 2.2. Component and Connector viewtype

In this section we will describe the Component and Connector viewtype. We will explain what a component and connector view is, and why we need such a viewtype.

Examples of components are processes, objects, servers, databases and clients. Besides components the component and connector models consists of connectors such as

---

**Figure 3 Viewtypes - Styles – Views [CBB+02]**
communication links, protocols, information flows and access to shared data storage. A characteristic of this viewtype is that an entity, which is instantiated several times at runtime, appears also more than one time in a view [CBB+02]. Table 2 summarizes the component and connector viewtype. After that we will look at each of the parts in more detail.

Table 2 Summary of C&C viewtype [CBB+02]

| Elements                        | Component types: principal processing units and data stores
|                                | Connector types: interaction mechanisms
| Relations                      | Attachments: component ports are associated with specific connector roles, a component port, p, attached to a connector role, r, if the component interacts over the connector, using the interface described by p and conforming to the expectations described by r.
| Properties of elements         | **Component**
|                                | • Name: should suggest its functionality.
|                                | • Type: defines general functionality, number and types of ports, and required properties.
|                                | • Other properties: depend on type of component, including things such as performance and reliability values.
|                                | **Connector**
|                                | • Name: should suggest the nature of its interactions
|                                | • Type: defines the nature of interaction, the number and types of roles, and required properties
|                                | • Other properties: depending on the type of connector may include protocol of interaction and performance values.
| Topology                       | The C&C viewtype had no inherent topological constraints.

2.2.1. Elements

In table 2 we see the component and connector viewtype elements. These entities are instances of component and connector types. Which types are available depends on the architectural style. The types within a style are chosen, because of the similarity among several components or connectors within an architecture [CBB+02]. But it also possible to use custom defined elements.

2.2.1.1. Components

Components are the elements that do the computation or data processing within a computer system. In a box and line diagram components correspond to the boxes. As can be seen in figure 4 a component consist of an interface and an implementation. The component type and player are part of the interface description. In this section we will describe all parts of a component.

The interface of a component specifies its capabilities. A component may have multiple interfaces. An interface is the part that communicates with the environment. The actual points of interaction are called ports. Ports should be well documented, because ports are the points that define how to communicate with the environment.

The component type is more an optional attribute of the interface specification. In the previous section we discussed architectural styles. Within a style types of components could be defined. For example a pipe and
filter viewtype could have a set of specialized filters that can be used within the view. These types are all examples of component types.

Within an interface multiple entities are used. These entities can be used in communication between elements or they could be used within the component itself. The entities that are used outside the component are called players. The players of an interface are important for connectors, because they interchange the defined players with the component.

Last part of the component is the implementation. The whole component is based on this implementation. An implementation could be composite or primitive. A primitive component is implemented directly in the code or just data stored in files. Composite means that the component can be decomposed in multiple objects.

A composite implementation description must provide three kinds of information:

- The parts: this is a description of the components and connectors from which the large component is built up.
- The configuration: this is a description how the parts are instantiated. And furthermore it is a specification how the components communicate.
- The abstraction: a description how the abstracted component interfaces map onto the interfaces defined in the real implementation. The semantics of the described interfaces should be derivable from the parts.

For the implementation it is important that it is consistent with the described interface. For a primitive component it is relatively easy to check the interface description with the implementation. But for a composite component it is difficult because the interface description should be consistent with the implemented interfaces of all parts that interact with the environment [GMW00, CBB+02, SDK+95].

2.2.1.2. Connectors

The communication and coordination between components is mediated by the connectors. The simplest kind of connection is the communication between two components, but there are also very complex connectors, for example a multi-cast, which is a n-ary connector. Garlan states that connectors form the “glue” for architectural designs [GMW00]. Examples of connectors are method calls, data accesses, SQL (structured query links) links or client-server protocols. In architectural box and line diagrams connectors form the lines between the components. But in the implementation this line is often more complex. For example an SQL link should handle buffering and it should guarantee the delivery of messages.

In figure 4 we can see that connector specifications consist of two parts: a protocol, and an implementation.

The protocol must have a description of the connector type and a specification of the role. The connector types are often defined in the view type. For example in the pipe and filter viewtype pipes represent the connectors. It is possible that specific pipes are used many times. In that case it is useful to define types. These types are called connector types. The connector role is a definition of the kind of interaction that could be established. For example a connector in a publish-subscribe style will have reader and writer roles, because the connector should be able to read and write information to the component ports. In a role specification we can find which components are supported and which players could be handled. It is possible that within one connector multiple roles are defined. Finally the connector protocol should explicitly document the assertions that constrain the entire connector. Examples of assertions are the ordering of messages, guarantees about delivery, or the maximum throughput.

Also for connectors the implementation could be composite or primitive. Again this means that there are complex connectors that consist of multiple components and connectors. Just like a composite component specification a composite connector specification should
also contain information about the parts, the configuration and the abstraction [GMW00, SDK+95, SHA94].

2.2.2. Relations and Properties
A relation between a component and connector is the connection between a component and a connector. It is a definition of the ports and roles that are connected and the players they use for their communication. In a formal definition a relation is:

“A component is attached to a connector if component port $p$, interacts over connector role $r$, using the interface described by $p$ and conforming the expectations described by $r$ [CBB+02].”

A component or connector may have multiple properties. Name and type are the most used properties. Other properties are also possible and should be used to support the meaning of the component and connector viewtype. For example, different properties will be chosen when the view is used for analysis, communication or construction. Some examples of well-known properties are [CBB+02]:

- **Reliability**: What is the chance of failure of a certain component? This property can be used to determine the overall system reliability.
- **Performance**: What response time can we expect? And how is the response time influenced by the system load? What is the expected latency and throughput of a certain connector? These properties can be used for example to determine the total throughput of a system.
- **Resource requirements**: What are the processing and storage needs of a component or a connector? They can be used to give answers to questions about resource requirements. For example what is the processor needed for a certain software system, can this hardware configuration be used for this system etc?
- **Security**: Which security features are implemented in the component and connectors? During security analysis software engineers can expose the vulnerabilities of a system.
- **Functionality**: This property tells something about the functionality of an element, and it is used for reasoning about a system [CBB+02].

2.2.3. Connector Abstractions
Most connectors perform multiple functions. This functionality is often realized with a composite implementation. For example an SQL link should be able to create connections, to close connections and it should guarantee that a query is executed.

It is possible to visualize this connector as one abstracted connector, but it is also possible to show the as implemented version with multiple components and connectors. In figure 5 we can see that abstracted connector 1 is replaced by two simple connectors and component 1.

In our opinion it is difficult to choose the right connector abstraction. Below, we will explain why we think this is the case. A component and connector view serves various purposes. Sometimes, it is meant for reasoning about a software system. The use of abstracted components and connectors makes it possible to get an overview of a system. Furthermore when we show all components and connectors it is difficult to know the difference between the essential components and the ones that only support the essential ones. For these purposes abstracted components and connectors are useful.

On the other hand a developer needs a detailed view of a component and connector view. And he wants to know all details even if a certain component has only a support function for an essential one.

Choosing how to document connector abstractions is one of the most difficult jobs. It may be difficult to map a component and connector view onto a more implementation-oriented view when the component and connector view is very abstract. On the other hand a view
becomes cluttered when it documents connectors and components that are logical part of
the connector mechanism.
Choosing the right abstraction level, is a matter of taste and it is influenced by the needs of
the stakeholders. Sometimes it is even better to create more than one component and
connector view [CBB+02].

![Diagram of Component and Connector Notation]

Figure 5 Replacement Connector 1[CBB+02]

2.3. **Component and Connector notation**

Before we can start reconstruction we need to know which ADL (Architecture Description
Language) will be used. We think that for AEGON the following characteristics are
important:

- *Completeness:* within a view it should be possible to document the characteristics
of components and connectors.
- *Usability for IT departments:* this means that the chosen language should fit in the
development process that is used at the IT department.
- *Usability for business departments:* the software is complex because many
business rules are implemented. The financial experts should be able to read
existing documentation of software systems.
- *Visual Clarity:* most of the existing documentation is used to reason about software
systems. The software is implemented by software houses, which means that the
documentation at AEGON is used for high level reasoning. We think that visual
clarity is important for high level reasoning.
- *In line with competences employees.*

Matevska-Meyer, Hasselbring and Reussner investigated different architecture description
languages. They presented their results in a paper Software Architecture Description
supporting Component Deployment and System Runtime Reconfiguration [MHR04].
In their opinion an ADL (Architecture Description Language) should support the following
kinds of views [MHR04]:

- *Structural view:* The structural view is the classic diagram of boxes and lines.
Components or objects are represented by boxes and connectors as lines. The
structural view is a decomposition of the system.
- *Dynamic view:* This is a view that represents the behaviour of a software system at
runtime. Dynamic means that it describes the flow in the system. For example an
object that is instantiated three times will appear three times in the view.
- *Resource mapping view:* The resource mapping is a mapping of the objects of the
available resources. Two mappings are needed. First there should be a mapping to
hardware units. And second there should be a mapping to organisational entities.
For documenting the component and connector view, only the dynamic view is important, because it is an example of the dynamic view. They evaluated a number of ADLs on several qualifiers. They concluded that Rapide and Wright have the important characteristics to document dynamic views. In the following subsections there are short descriptions of Rapide and Wright. Furthermore there is a description of ACME, because this ADL is a representative of an ADL family describing architectures as annotated graphs of components and connectors. Next, we describe UML, because it is so widely used in architecture descriptions that it is impossible to leave it out. Finally, we draw some conclusions regarding the best ADL to use in AEGON context.

2.3.1. Rapide

Rapide [LKA+95] is an ADL designed for prototyping system architectures. The designers of Rapide used four important design criteria. Firstly, it should be possible to define an executable from the designed architecture. So in an early stage of the design process it should be able to execute a prototype. Furthermore within the execution model it should be possible to represent concurrency, synchronization, dataflow and timing issues. Thirdly, the possibility to define constraints to provide support for reference architectures. A last design criterion was the link with industrial needs. For example scalability is an important issue in industry. Rapide is designed to support common industrial issues. The result of these design criteria is Rapide. It is a modular ADL, which means that there are five ‘languages’ that could be studied separately. Furthermore a change in one of the parts does not mean that the other parts should be changed.

The five languages in Rapide are [LKA+95]:

- **Types language**: meant for describing component interfaces.
- **Architecture language**: this language is meant to describe the flow of events between the components that are used. This language uses a set of components and a set of connections as input to describe the flow.
- **Specification language**: this language is meant to describe the constraints on behaviour of components.
- **Executable language**: in the introduction we already stated that it should be possible to create executables. This language is meant to write executable modules.
- **Pattern language**: Within architecture sometimes a pattern of events is performed. This language is used to specify these patterns.

Rapide is usable for the component and connector viewtype, because it represents the behaviour of the system as a history of events [LKA+95].

2.3.2. Wright

Wright [AG94] is designed with focus on the period after implementation. Many architecture description languages give an informal description of a software system. Programmers need detailed information about the parts they need to change. They should be able to get the ‘truth’ from the architecture description. In the Wright opinion this can only be derived from a formal architecture description. Wright is developed to support this concrete architectural information and to support architecture analysis on the description. It is based on the idea that relationships between components should be specifiable as a set of protocols, which characterize the interaction in a software system.

A Wright description is divided in two parts. First, the components and connectors are specified. Components are described as a set of ports or logical interaction points. These ports describe the component’s expectations and promises. Furthermore it is possible to specify how the interaction points are used in computations. A connector is characterized by a set of roles and a glue specification. A role gives us information about the
expectations of each of the interacting parties. The glue is a description of the coordination between the different roles within the connector [AG94]. Secondly, the configuration of a software system is described. The configuration is described by a list of component and connector instances. The topology between these instances is described in a list of attachments.

2.3.3. Acme

Acme [GMW00] is a second generation ADL that has been developed built on the experience of other languages. The resulting language has several advantages. Acme has a language supporting only the most essential arguments to describe architectural elements. In addition, it is also possible to use extensions to document more complex structures. The small standard set and the possibility to define structures results in a language, which can easily be learned and is powerful enough to describe complex architectures. Looking at many first generation ADLs, each language focuses on several aspects of architecture. Each ADL focuses on several properties like: resource consumption, interaction protocols, scheduling constraints etc. The designers of ACME decided to support an open-ended specification to allow for extra information. Each property that is document has a description, type and a value. Properties can be assigned to any of the core types used in ACME.

These seven core types are components, connectors, systems, ports, roles, representations and rep-maps. Components, connectors, ports and roles have the same components as described in section 2.2. Systems are the representations of Acme in graphs. Within these representations the nodes represent components and arcs represent the connectors. Composite representations are introduced to support hierarchical documentation. This means that components or connectors of a certain view can be represented by other lower level graphs. Even these low level representations could contain components or connectors that are represented by lower level graphs.

Representations help us to keep system overview by presenting a hierarchical description of an architecture.

As described earlier the interface of complex components is the set of interfaces of internal components that interact with the environment. A rep-map is a definition of the external interface and the correspondence with the internal ports. In figure 6 you can see an example of Acme [GMW00].

```plaintext
System simple_cs = {
    Component client = {
        Port sendRequest;
        Properties { requestRate : float = 17.0;
                     sourceCode : externalFile = "CODE-LIB/client.c" }
    }
    Component Server = {
        Port receiveRequest;
        Properties { idempotent : Boolean = true;
                    maxConcurrentClients : integer = 1;
                    multithreaded : Boolean = false;
                    sourceCode : externalFile = "CODE-LIB/server.c" }
    }
    Connector rpc = {
        Role caller;
        Role callee;
        Properties { synchronous : Boolean = true;
                     maxRoles : integer = 2;
                     protocol : WrightSpec = "..." }
    }
    Attachements {
        Client.send-request to rpc.caller;
        Server.receive-request to rpc.callee;
    }
}
```

Figure 6 Client-Server System with properties [GMW00]
2.3.4. Unified Modelling Language

Original UML is a modelling language to support detailed design. However, since it has possibilities to describe elements and relations, it can be used to describe more than just design. Consequently, the existing language is used and mapped onto a new purpose. The notation of the resulting component and connector viewtype could be incomplete, but had to capture most of the important aspects of the architecture. Because the language is not created to document the component and connector viewtype there are many possible strategies to document this view. We present four different strategies that all have their strengths and weaknesses. The third example uses the UML real time profile, this profile is used because a good mapping is possible [CBB+02]. Of course also other profiles could be used if a good mapping can be found.

The four documentation strategies are [CBB+02]:

- **Using Component Types as Classes and Component Instances as objects**
  The relation between a class and an object is similar to the relation of an architectural type and their instances. Furthermore classes describe the system vocabulary, just like the component and connector types in an architecture description.

- **Using subsystems**
  Packages have the advantage of having an underlying model. For example a complex component can be modelled in classes, objects and behavioural models. This complex component is visualized in the component and connector view as a package. In this strategy the relations between the components are drawn as package relations.

- **Using the UML Real-Time profile**
  Until now we used generic UML to describe the architecture concepts. Another approach uses the UML Real Time profile. In UML a profile is a bundle of stereotypes, constraints, and tagged values that form a domain specific language specialization.
  In UML Real-Time profile, the unit for describing computation is the capsule. These capsules have interfaces that can be hierarchically decomposed. The components of the component and connector types map to these capsules. Furthermore the connectors can be mapped to UML Real Time connectors. And ports and roles provide interfaces between components and connectors, which also exist in the UML Real Time profile.

- **The execution view of the “4+1” model view, described in section 2.1.2., has many similarities to the component and connector viewtype. So the proposed documentation strategy of the execution view can be used for documenting the component and connector viewtype.**
  They use:
  - UML class diagrams for showing the static configuration
  - UML sequence diagrams for showing the dynamic behaviour of a configuration, or the transition between configurations [HNS99 page 9].

To check the value of these strategies it is possible to use the four criteria of Clements et al. for a strategy [CBB+02]:

- **Semantic match:** The resulting UML model should be close to the interpretation of the component and connector viewtype description. Furthermore it should respect the semantics and intuitions of UML.
- **Visual clarity:** The advantage of a drawing technique is that it is easier to understand. So the result should bring conceptual clarity to a system design, it should avoid visual clutter and highlight key design details.
• **Completeness:** Finally it is important that the resulting model is complete. It should represent the component and connector types as well as the computational model.

• **Accessibility:** The notation should be readily accessible for all stakeholders.

All described strategies suffice to some extent to these criteria. In conclusion we can say that there are many ways to document component and connector views with UML. The different strategies use objects that were originally not meant for this purpose. This results in architecture descriptions that do not contain all aspects as described in formal specification languages. And furthermore it is difficult to create unambiguous views. A UML description could be complete and unambiguous, but it is not guaranteed by the language itself.

### 2.3.5. Conclusion

To choose a notation style we created a table with for AEGON important issues about documentation.

To check the competences of employees we asked several employees how familiar they were with our selected ADLs. Furthermore we asked if they preferred other languages than the four we have selected. The conclusion of this short inquiry was that the only known language was UML.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Rapide</th>
<th>Wright</th>
<th>Acme</th>
<th>UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completeness</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+/-</td>
</tr>
<tr>
<td>Visual Clarity</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Unambiguous</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Usability for IT department</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Usability for business departments</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>In line with competences employees</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

In table 3 we present six issues, which are to our opinion important for AEGON. The first three focuses on language qualities and the last three focus on usability for AEGON.

On the one hand our inquiry showed that UML is definitely the preferred language. At the other hand UML descriptions could be incomplete and it is ambiguous. The completeness is better guaranteed in a formal ADL, but formal ADLs have the disadvantage that they are not visual.

We think that UML is for several reasons the best option for AEGON:

• Knowledge of UML is already available

• Visual clarity

• Usable for all stakeholders

• Completeness and ambiguity are important, but most use cases in AEGON software systems are simple. Of course there are some exceptions, such as calculation modules.

Our choice is UML and from now we will use this language to describe OfferteNet.
3. Reconstructing with Symphony

Maintenance on existing software systems is a complex task. The availability of a software architecture description is crucial for maintenance tasks. AEGON would like to redesign OfferteNet, this is impossible without the architectural documentation of the current system. When certain information is not available software architecture reconstruction comes in. Software architecture reconstruction is the process of (re)creating the documentation.

For this reconstruction process we need information. This information can be obtained from many possible recourses. Recourses can differ from user experience to log files and from existing documentation till source code. As can been expected log files and source code are the most reliable information source.

Software architecture recovery is a complex process, because it consists of ad-hoc steps, in which we use simple tools and a lot of manual interpretation.

Symphony is developed to guide a reverse engineer through this reverse engineering process. Symphony makes it possible to report about reconstruction results and even to compare reconstruction approaches [DHK+04a].

In this section we will explain the Symphony method. “Symphony is a framework for describing reconstruction approaches [DHK+04].” It is not a technique itself, but more a guideline for a reconstruction process. The result of this process, executed with possibly different techniques, is a set of views that aid in solving the problem.

First we will describe what a view is and in section 3.2 we will describe the Symphony steps.

3.1. Views in Symphony

A software architecture description is a set of models and some knowledge that is shared over all views. In Symphony the models are called views, and the overall information is called rationale. Symphony’s goal is to reconstruct the models and their rationale.

One of Symphony’s key points is using views and viewpoints. “A view is a representation of a whole system from the perspective of a related set of concerns [IEEE00].” And a viewpoint describes the conventions to create, visualize and interpret a view. So the relation between viewpoint and view is that a viewpoint is a specification of the kind of information that is presented in a view [IEEE00].

A software architecture description consists of views, which are selected for the involved stakeholders. In forward engineering it is important to identify the stakeholders. When the stakeholders are known it is possible to select the viewpoints. Next step is creating the views from the viewpoints. This process of selecting stakeholders and viewpoints is important to reduce the complexity, because it is hardly impossible to document all possible views. Besides that it is irrelevant to document views in which nobody is interested.

In reverse engineering it is also possible to document too much information. Many techniques and methods are available to extract the information from all kinds of resources. Symphony supports in selecting the information that is useful for the reconstruction.

In a Symphony process three viewpoints could be used [DHK+04a]:

- source viewpoint: this view contains information that could be directly extracted from the system, system source or from creations of the system like log or build files
- target viewpoint: when the target view is extracted it is possible to (partially) solve the problem
• Hypothetical viewpoint: this is a view that describes the architecture, but it is not guaranteed that this view is accurate. A hypothetical view is for example an as-designed view or a reference view.

3.2. Symphony Steps

First of all Symphony is an iterative process. When a step is finished it could be revisited later in the process. Symphony consists of two parts. The first part is the reconstruction design and the second is reconstruction execution.

Symphony is problem driven, which means that it only reconstructs the architectural information that is needed to solve the problem.

First step in Symphony is the problem elicitation. The result of this step is a report of the available information (derived from interviews, existing documentation etc.) and a definition of the problem statement.

After defining the problem, one should be able to define the information needed to solve the problem. This phase is called concept determination. Within this step stakeholders are identified and one should summarize their information needs. This information makes it possible to create a list of candidate views. Last step is searching for a balanced set of views, by combining and removing views from the candidate list. The resulting set of views is the starting point for reconstruction execution.

The reconstruction design phase is not only useful for the current reconstruction, but it can also play a role in future reconstructions or for architecture conformance checking.

Reconstruction execution has an iterative character. All iterations are structured in the same steps, this clear structure makes it possible to compare results of iterations and even complete reconstructions can be compared. And above all the clear structure helps readers to understand the process of reconstruction.

The execution phase consists of three sub steps:

• Extracting the source view from the source code. This is called data gathering.
• Knowledge inference is the step in which the mapping rules are performed and the target view is created.
• In the last step the information is interpreted and if needed some steps will be revisited or the concept determination is refined.

Result of the design and execution phase is the information that is needed to solve the problem. It is possible that this result creates new opportunities for reconstruction. So even the whole process has an iterative character. The process is summarized in figure 7, which shows all steps and the roles (stakeholders, process designer, reconstructor) that are involved.
4. Reconstruction OfferteNet

4.1. Problem Elicitation

In the first stage of the Symphony reconstruction process we have to define the problem statement. Based on discussions with AEGON managers, project leaders and maintainers at Service Center Pensioen, we made an inventory of problems that call for software architecture reconstruction. In this section we will describe the problems.

Firstly, we want to say that AEGON has hardly any technical documentation, except some general overviews, about OfferteNet. Only the programmers of the external software house know how it exactly works. They have unwritten knowledge about the application. For AEGON it is difficult to do impact analysis when users come with new functional requirements.

The second problem is that there are currently three important web applications for AEGON Service Center Pensioen (SCP). A lot of their functionality is common for all three systems. The desirable situation is that this common functionality is shared among all web-applications. That is why the service based situation is developed (figure 2). At this moment each application is a separate system, there is no reuse of functionality over the three systems. This results in a system in which every front-office application has its own connection with the back-office systems (figure 1). In the target situation there is a mid-office system that serves all front-offices and furthermore there is no direct connection between front-office and back-office (figure 2). But before we can create such a situation we have to gather many details about the current application architectures. We should know what is the overlap between the applications and how the current architecture can be mapped onto the proposed architecture.

An example of multiple defined functionality is the calculation kernel, used in the proposal software. The OfferteNet application and the CAESAR mainframe both have such a kernel. This results in maintainability problems, because every change should be done twice. Furthermore there can be problems with the results. Currently there is an allowed margin between results of OfferteNet and CAESAR. The desired situation is a common calculation kernel resulting in equal outcomes in calculations.

The third problem is that the data sources differ for all applications. At this moment there is no standard for data transport between back-office and front-office. For each project other techniques and approaches are used for retrieving and sending the same mainframe data. The approach is sometimes even an employee that is typing data from system A to B by hand.

When we reach the situation of picture 2, overall costs will be lower because:

- Maintenance costs will decrease, because there is no data duplication, less code to maintain and only knowledge of standard techniques are required.
- Reuse is possible, a new application can use parts of the existing ones
- Portability will increase, because responsibilities are separated over the system parts. This means that for example the services can be used from a web interface, but also from a windows application. This is cost efficient because components have to be developed and maintained only once.

To handle all these challenges AEGON would like to migrate OfferteNet to a new architecture. The first issue is creating technical information about how OfferteNet is structured at this moment. So there is dearly a need for software architecture reconstruction!
4.2. Concept determination

Before the architecture can be reconstructed, we must choose which views might be important for the migration process. In chapter 2 we have described the model of Clement et al. [CBB+02]. We have to determine which views are must haves, and which level of detail is required for a successful reconstruction project.

We follow the ATAM method proposed by Clements et al. and create a list of stakeholders that will be involved in the process [CBB+02]:

- **Project manager:** his concerns are schedule, resource assignments. A manager is not interested in all technical details. He is mainly interested in the overall purpose, the constraints, and the interactions with other applications.
- **Developer:** the architecture for him is the view of the whole application. In practice a developer is interested in all details of the parts he is responsible for. Furthermore he wants to have a global idea of the whole system.
- **Architects:** this group is the most complicated, because they are interested in all documentation. Architects want to know all key design decisions and why they were made.

Regarding these three stakeholders we can create a list of candidate views (table 5). The candidate list (table 4) consists of a large number of views, but not all views are needed for this particular reconstruction.

For the module viewtype we will combine the decomposition with the uses view, because it is possible to draw relations between the decomposed elements. This reduction results in three styles that are interesting: decomposition, generalization and a layered style. We prioritise the decomposition/uses style, because we think that there will be enough information about layering in the component and connector view. Secondly, OfferteNet is a relatively small software system and for that reason generalization is less important.

For the component and connector viewtype various styles are possible. Two reasons contribute to our decision to use a shared-data / client server style. Firstly, one of the problems described in the previous section is data transport. Secondly, within the minimal set of existing documentation we read OfferteNet is a data driven application. We think that it is useful to combine this view with some client server characteristics, because a browser is the client for the web applications.

For this reconstruction process implementation and work assignment views are not important, because AEGON is not responsible for the real development and only interested in the technical documentation. Technical maintenance is outsourced to Getronics Pink Roccade Finance. Furthermore the deployment is on one single production machine. So we can also leave the deployment view, because we think that a description in the introduction is sufficient to understand the environment. Secondly, also hardware responsibilities are outsourced and thus we are only interested in the high level architecture of the hardware environment.

For this reconstruction we need two views (table 5):

- **Module viewtype:** decomposition/uses
- **C&C viewtype:** shared-data/client-server view

We will describe the process of recovering these views from the source code and artefacts in the remaining text of chapter 4.
**Table 4 Candidate view list [CBB+02]**

<table>
<thead>
<tr>
<th>Module Views</th>
<th>C&amp;C views</th>
<th>Allocation Views</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Decomposition</td>
<td>Uses</td>
</tr>
<tr>
<td>Project Manager</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>Developer</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>Architects</td>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

*Key: d = detailed information, s = some details*

**Table 5 Selected views**

<table>
<thead>
<tr>
<th>Module Views</th>
<th>C&amp;C views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition / Uses</td>
<td>Shared-data / client-server</td>
</tr>
<tr>
<td>Project Manager</td>
<td>s</td>
</tr>
<tr>
<td>Developer</td>
<td>d</td>
</tr>
<tr>
<td>Architects</td>
<td>d</td>
</tr>
</tbody>
</table>

### 4.3. Exploration of OfferteNet

The first step in recovering a software architecture is the initial exploration. The exploration is meant to get a feeling about the application within a couple of hours. In a short period we should get an idea about the complexity of the system. We will do two types of analysis: Firstly, we will calculate some metrics, which give an idea of the system in numbers. Secondly, we will create a view of the source code and their mapping onto the file system. This mapping will help us to navigate through the source directory.

#### 4.3.1. Metrics of OfferteNet

As preliminary exploration we calculated some metrics. We used JavaNCSS to compute these metrics [URL5]. With these metrics we can make some expectations about quality and identify potential risks.

The project consists of 210 packages, 1018 classes, 9,634 methods, 58,744 lines of code (LOC) and 4,746 Javadoc entries. LOC are the lines of code without comments. JavaDoc entry means here the existence of JavaDoc in advance of a method, the number of keywords is not measured. With these metrics we calculated average values for packages classes and methods.
In table 6 we show measurement averages. Of course an average value can be influenced by extreme values. To give insight in these extreme values we show all values into several plots. The calculated averages in table 6 give a good representation if and only if the individual points in the graphs have almost equal values. If not than there exist extreme values, which influence the averages.

In appendix F we show nine plots. On the X axes we find packages, classes or methods. In the package plots 210 points are displayed with the corresponding Y value. In the four classes plots we display 1018 points in the two method plots 9634 points. We ordered the X axe value by increasing Y value.

The average package consist of 4,8 classes and almost 46 methods. The JavaDoc is in many packages not complete. There are some packages that have a large number of classes and methods. Most packages have less then 10 classes.

After measuring for the packages we started to compute metrics for individual classes. We discovered that the average class has:

- 58 lines of code
- 9 methods
- in some case an inner class
- and 50% of the JavaDoc is available

Most classes have less then 200 lines of code, which is reasonable. There are some exceptions and these classes are potential risks for future maintenance, because they are too large. For the number of methods in a class the same holds. Most classes have a reasonable number of methods, but there are exceptions there is for example one class that has almost 450 methods. In a couple of classes inner classes are used.

Last measurements we made are method metrics. Again we measured the LOC for the methods and the cyclomatic complexity number.

The average method has:

- 6 lines of code
- Cyclomatic complexity of 1.8

In the method plots we find the same trend as in packages and classes. In most cases methods have a reasonable number of code lines. But there are exceptions that are potential risks in maintenance. The cyclomatic complexity is a measurement for the complexity of a method. A cyclomatic complexity bigger then 20 is a risk for future maintenance. Looking at the plot we can conclude that the majority of methods are not indicated as a risk.

From these computed software metrics we can conclude that there are some risks in OfferteNet, but the majority of packages, classes and methods do not indicate risks. Besides these risks we can conclude that the technical documentation in the source code is for about 50% complete.

4.3.2. Source mapping

When browsing the source code we can see the imports of the different classes. The *lib* directory shows the libraries used by OfferteNet. There are seventeen JAVA libraries in the *lib* directory. These libraries will be included in the class path when the source is compiled.
But within the source we will not see any dependency to a specific library, but a reference to the package and/or class in these libraries.

The possibility to find a class in the file system could be important for further investigation. For this reason we decided to create a view of the source code and its mapping onto the file system.

We mapped all source files and libraries to an XML structure. We created three mappings:

- a mapping of the original source directory to XML
- a mapping with libraries unpacked and decompiled in the source directory to XML
- a mapping of the unpacked libraries to XML

These three mappings are made to support all possible searches during our research. With the first mapping we can find if a class is placed in a library or in the source directory. With the second mapping we can follow the import links and find the desired classes. And with the unpacked libraries we can find out in which library a class is included.

The result of this step is a set of XML files.

### 4.3.2.1. Target, Source viewpoint and mapping rules

As told in the previous section our target is a mapping of the source to XML. In this XML mapping we should be able to see the allocation of the software modules to files or in the other mapping the module allocation to the libraries. The source viewpoint is the directory structure, package structure and the file names.

A directory is mapped to an XML-tag with the directory name. The content of that directory is mapped within the tag. This could be sub directories, which are handled as directories, or files. A file type is mapped to an XML-tag. The possible XML-tag types for files are listed in table 7. For example velocity.jar will be mapped to `<Library>Velocity.jar</Library>`.

### 4.3.2.2. Data Gathering

We wrote some scripts in order to gather all data. With help of some command line tools we created the mapping. Because JAVA source composition is close to the mapping onto the file system we use the file system for creating our view. The only sources that have no direct mapping with the file system are the libraries. To correct this for the second and

```
<aebot.components.batch>
  <aebot.components.batch.business>
    <class>BatchService.java</class>
  </aebot.components.batch.business>
  <aebot.components.batch.data>
    <class>BatchDAO.java</class>
    <class>BatchDAOException.java</class>
  </aebot.components.batch.data>
  <aebot.components.batch.model>
    <class>BatchModel.java</class>
    <class>Batch.java</class>
  </aebot.components.batch.model>
</aebot.components.batch>

<aebot.components.batchverwerking>
  <aebot.components.batchverwerking.business>
    <class>BatchverwerkingException.java</class>
    <class>BatchverwerkingHome.java</class>
    <class>BatchverwerkingRemote.java</class>
    <class>BatchverwerkingSLSB.java</class>
  </aebot.components.batchverwerking.business>
</aebot.components.batchverwerking>
```

*Figure 8 Part of XML file (sourceview.xml)*
third mapping we wrote a script that unpacks all JAR (Java Archive) files. After execution of this script all directory paths are equal to the package names. We created two versions of this unpack script:

- One that extracts all data in the root of the source. This script is used for the second mapping.
- One that creates a directory with the name of the library and extracts the data of that library in this directory. This script is used for the third mapping.

The first script version is shown in appendix C1.

When we want to browse the source code in further steps, it is still impossible because the JAR files contain just class files. To solve this we created another script that decompiles all Java code using JAD (see appendix C2) [URL2]. This script was not needed to create the mappings, but we think that the source code could have value in further reconstruction steps.

A third script was written to perform the mapping rules and create an XML file (appendix C3). We have chosen for an XML file because this is an often used plain text format and many software projects can import XML. Because of the clear hierarchal structure it is relatively easy to rewrite the XML to any format that is required.

A part of the resulting XML file is displayed in figure 8. Within this XML file it is possible to view all modules that could be referenced in the source. An example is figure 9, in the directory test/com/pinkroccade/efinance/rekenbox/data we can find nine descriptor files and in the directory test/com/pinkroccade/efinance/rekenbox/JSK we can find nine classes. We created the other XML-mappings by executing the described scripts.

4.3.2.3. Information Interpretation

When we analyse the directory structure there are several subdirectories: aebot, com, descriptors, documenten, lib, templates, test and XML. The directory aebot contains most of the source code. Com contains some Java code that is programmed by the suppliers of the calculation kernel. The descriptor directory contains a lot of XML descriptor files for the application server. In the directory documenten (documents) we find the rtf templates that are also stored in the database. The application uses the documents inside the database. The libraries of external parties are inside the lib directory. The velocity template, used by the FormulierBouwer is located within the template directory. Then there is the test directory, this one contains packages for test purposes. These packages are out of scope for our research, because we only recover the application architecture and not the test script architecture. Finally there is the XML directory. This directory is also important because it contains all the XML-files for the calculation framework. These are also in the database.

In short we can conclude that there are two important directories for our research i.e.: aebot and com.

4.3.3. Discussion

The exploration results were useful for us, because we get an idea of OfferteNet’s complexity. We think that metrics are a fast method to identify potential risks and to estimate the quality of the source code. We computed values for all packages, classes and methods. These values can be
represented in many views. We have chosen to display the individual points ordered by increasing Y value. There are also other options to visualize these metrics. Other views could give more information, but in our opinion this is not needed, because statistical interpretation of the measurements cost too much time for a quick scan.

4.4. **First Iteration – Execution view**

Our first iteration creates a view that shows all possible screen flows. It starts from the login screen and then visualizes all possible screens that could be accessed from the menu. Furthermore we visualise the connections with included java beans to show which JSPs use a certain bean. Finally, it is possible to see which pages redirect to each other.

4.4.1. **Target, Source viewpoint and mapping rules**

In the original documentation we can find a picture with information about the current structure of OfferteNet (figure 9). The architecture contains three main layers; web server, application server and data server. Furthermore we can read that there are also three layers in the software; presentation layer, business layer and persistence layer. Because OfferteNet is a J2EE application there is a presentation layer that consists of servlets and Java Server Pages (JSP). JSPs are a combination of Java code and HTML code. During our research we had no parser available capable of parsing JSP and JAVA files. Because of the small number of JSP files we decided to analyse the presentation layer by using grep and text editors, for that reason a parser is unnecessary. Our target viewpoint is a visualisation of modules that forward, redirect, or include other modules. The source viewpoint is the source itself. Within the source we searched for:

- Links: in the source code as `<a href ... /a>`.
- Form submits: in the source code as `<form action= ... >`.
- JSP Bean usage: in the source code as `<jsp usebean ... />`.
- JSP forwards: in the source code as `<jsp:forward page=.../>`.
- JSP includes: in the source code as `<jsp:include page=.../>`.

4.4.2. **Data gathering**

The data gathering phase included searching all JSP files by hand and with tools like grep. We searched for the described source lines and mapped these to our SDL diagram. An SDL diagram is originally meant for describing communication and telecommunication systems. We used SDL, because there were some license problems for TogetherJ. The drawing tool that was available supported SDL and it was possible to use the available objects for our view. For this reason we have chosen to use SDL. We used these objects but with another meaning. The meanings of our entities and relations can be found in figure 10. A part of the results can be seen in figure 11.
4.4.3. Information Interpretation
While analysing the created view of the system we concluded that the JSPs were just pages to:
- show menu options
- perform actions that do not depend on user input
- redirect the system to a servlet

These JSP-files use the jspBeans to trigger the business layer. This iteration does not deliver the information to solve the defined problems. We need an extra iteration. In the next iteration it is possible to leave out all the JSPs and start with the beans and servlets. This top-down approach leads to an overview of the system in which we can see all the modules that could be used by users. The advantage is that we can use classic java tooling to analyse the system. Of course there are also disadvantages, because available modules are always analysed but they may never be used by the presentation layer.

4.4.4. Discussion
In this iteration we created a view, with a drawing technique that is not standard in software engineering. The use of SDL is strange, but to our opinion the format is unambiguous and simple, because there are three entities and only one relation type. A more standard option was a flow/decision diagram. But we have chosen for an SDL diagram, because we had some relations that do not exist in a basic flow diagram, such as the include of beans.

Modelling unsupported relations, by adding special objects is to our opinion more confusing than changing the meaning of a ‘non’ software drawing technique with enough possibilities to model all the needed entities and relations.

Second point of discussion is the data gathering step. We read the code by hand and sometimes with a file search tool like grep.

There are other possibilities to parse the JSP files. There are many parses available that could parse HTML and also parsers for Java, so it would have been possible to create a new parser for JSP. However, we did not do this, because in our opinion the number of files (98 jsp files) was too small to create a new parser.
4.5. Second Iteration – Generate Decomposition View

In this step we will create a decomposition view of OfferteNet. As described in section 2.3.4 the execution view of the “4+1” model view can be described by a UML class diagram and UML sequence diagrams. The class diagram is a static view and the sequence diagrams give the dynamic character of the execution viewtype. This is our preferred documentation strategy, because with the available tooling we can create these diagrams. To avoid an unclear diagram, we have chosen for a decomposition view. This decomposition view has the possibility to create a hierarchical system view. The result of this iteration will be a deliverable for AEGON.

In the previous iterations we analysed the presentation layer. After this analysis we removed the JSP files from the source code and during the rest of this investigation we use the source code without the JSP files. The remaining code is just Java.

During this iteration we will create a decomposition view of the module viewtype.

4.5.1. Target, Source viewpoint and mapping rules

Our source view is again the source code itself, but without the presentation layer that was removed.

Triggers in the source are method calls for the relations between classes. Furthermore the contents of classes and packages are important during the mapping. The mapping rules are:

- package to a UML package object
- class to a UML class object
- interface to a UML interface object
- method to an entry in UML class object
- attributes to an entry in UML class object
- class relations to UML arrows that correspond to the relation

The target is a graphical view of the relations between the different modules. This view shows two types of relations the uses and contains relation on attributes, methods, classes, interfaces and packages.
4.5.2. Data Gathering
In almost every software architecture description a module view is included. Nowadays many vendors of UML tools created an automated generation of this view in their software suite. At AEGON we were able to use TogetherJ Control Centre. This tool is also able to generate our target, the decomposition view. We should select the source directory and import it in TogetherJ. It was possible to export the results to HTML files. The advantage of this HTML export is that the information can be accessed in a hierarchical way. We can start at package level and by simply clicking on the packages we can view the information on class level. An example of a class view can be seen in figure 12.

4.5.3. Information interpretation
Before we started this iteration we had some expectations based on our experiences with web applications. The results of this iteration are exactly what we expected: packages are independent beans and all these beans have almost the same structure. With XML-descriptor files these beans are bound to the application server, which makes it possible to use them. This step did not much clarify the structure of the system, because we could not observe any dependencies between packages. What we discovered was the similarity between the packages itself, because many packages contain classes with a similar structure and naming.

In the coming paragraphs we will discuss the different packages we found and the interpreted information about the different packages.

In aebot.beans we found the beans, used in the presentation layer. An important discovery was the aebot.components directory structure which contains a lot of packages having the same structure. There are some exceptions i.e. download, batchverwerking (batch processing), pdfconverter, uitvoer (output) and invoer (input). The other components have a business, model and data package. In the business package we can find a BeheerService, which is a session bean that is the interface to the environment. The BeheerService manages the entity. The business model is defined in the model package. These model packages consist of:

- A primary key: a unique identifier for the business model. This primary key is used to identify the entity. In some cases this primary key is not used.
- The model: this is the complete definition of the business data model. It is a specialisation of the primary key.
- the interface: this is the contract with the environment

An example of a model can be found in figure 13. Finally there is the data package; this package contains some classes for an entity bean (EB) and a data access object (DAO). The business objects that are stored in the database are represented by entity beans. In J2EE there are two types of entity beans. Firstly, there are beans with container managed persistence (CMP). These beans are not responsible for the mapping with the database, the container (application server) will do this. Secondly, there is bean managed persistence (BMP). These beans are responsible for all communication with the database. The advantage of bean managed persistence is that the developer has more control over how the database will be accessed. Thus in the data package we can find the methods to get a database connection and also the queries that are fired to the database [URL4].
OfferteNet only CMP is used for INSERT, DELETE and UPDATE queries. For SELECT queries also other structures are used.

In almost all component packages a DAO and a CMP implementation exist. These two data objects give us the expectation that they used the fast lane reader pattern. This pattern makes it possible to access data of multiple business entities simultaneously. It is possible to give a natural representation format of persistent data instead of objects. This means that it is possible to execute a query and only query the needed attributes instead of whole objects. And the performance of a fast lane reader is much higher than enterprise Java beans because of less overhead. A disadvantage is that a fast lane reader (DAO) is read-only, that is why there are also beans with CMP [URL3]. An overview of the fast lane reader pattern is given in figure 14. We can see that for managed access the EJB Container is used, which has many advantages. Transactional write is definitely the most important advantage. For read-only access the fast lane reader is used. Furthermore we discovered that the com.softwarehouse.efinance.components package contains components that have the same structure as the components in the aebot.components package.

In the aebot.ejb package we found also packages with the same responsibility as components in aebot.components. They are called packs, for example there is an
offertepack, a gebruikerpack and so on. Within these packs we find again an EB with CMP. It is not clear to us why there are two EBs for the same database object. Maybe this is caused by software evolution.

We summarized the three packages (com.softwarehouse.efinance.components, aebot.components, aebot.ejb) and their data access in figure 15.

In the package aebot.util we found some classes that could be used by all other packages. For example there is a class StringUtils to do some string operations and a class that reads information from the property files, stored on the server. The same holds for com.softwarehouse.efinance.util. This package contain utility classes with shared functionality used by many other classes, however some of these utility classes are mainly used by the calculation kernel.

There are two sub packages in the package com.softwarehouse. The first one is the calculation kernel. This is a commercial product and is out of the scope of this investigation. This does not mean that we did not spend some time on it, but we will describe the calculation kernel in section 4.2. The other package is the efinance package. Within this package again we can find a set of packages. Two of them are related to the calculation kernel. The third com.softwarehouse.efinance.aebot.web is a package with information that is published on the web. This is the batchServlet, which makes it possible to control the batch process with the admin pages of OfferteNet.

4.5.4. Discussion

A disadvantage of decomposition is that there is no overview. At the lowest level in the view only one package is decomposed into classes and the relations between these classes. Relations over multiple packages are not visible, although they do exist. The information we gathered in this iteration is about packages and information about the relations between classes and interfaces within a package.

In the next iteration we will try to get more information about the relations between packages and how they are related.

However there are disadvantages we have chosen this hierarchical view because we think that one extreme large picture would not help to get an overview of the system.

Finally we want to emphasize that it is difficult to visualise the relation with the entity beans, because of the JNDI look up. This JNDI lookup is a relation that is made with help of a descriptor file. It could be compared with the procedure for a DNS (Domain Name System) lookup. This lookup procedure causes that relations of components in the EJB container cannot be visualized without further analysis, because the beans are found by their JNDI name which are registered during initialisation of the application. To visualise these JNDI relations we could not use the tooling and we have to add these relations by hand.

4.6. Third Iteration – Sequence diagrams

During this iteration we will create our second deliverable for AEGON. In the previous iteration we created the decomposition view, in this step we will create some sequence diagrams. This iteration is the first that goes beyond the module viewpoint. We will analyse the calls in the source, instead of viewing the static class dependencies. The difference is that in case of two class instances also two class objects exist in the view.

4.6.1. Target, Source viewpoint and mapping rules

The target viewpoint in this iteration is a view that shows the different components and the calls between these components. This will be visualised in sequence diagrams. The source viewpoint is the source itself. We analyzed the following relations:

- Method calls method
- Class contains method
The mapping is that each used class will be mapped to an object lifeline and all method calls will be mapped to a call from the caller to the callee.

4.6.2. Data Gathering

In the beginning we thought that it could be extracted by using a UML generator. In fact this was working but there were some problems:

- When generating diagrams automatically there was an enormous overload because the system is too large to show into one diagram.
- The resulting diagrams show the links to the standard Java API which results in an exploding diagram.
- OfferteNet is a J2EE application and the application server accesses beans by remote method invocation. This means that there are stubs and the real implementation of a bean. In the generation of sequence diagrams the process ends when accessing a stub, because the tool was not able to detect the use of remote method invocation.

These three problems had to result in a partially automatic generation of the sequence diagrams.

The first problem was tackled by splitting the diagrams into multiple diagrams. Because we started our investigation by analysing the presentation layer, we started from the servlets and beans.

In our naming of the sequence diagram we started with Bean, Servlet or Business. Our top-down approach starts from the Bean and Servlet diagrams. In the case of Business it is a detail of a call in a bean or servlet diagram. The term business does not mean that all classes from the business layer are visualized as business, but only classes from the business layer that became too large to show in the bean and servlet diagrams. The second term in the names of the diagram is the class or JSP that is handled. The third term is added when only a part of the class is handled. Summarizing the previous text the name of a sequence diagram is:

\[<\text{bean OR servlet OR business}><\text{class OR jsp}><\text{part}]?>

An example of a splitted diagram is servlet InvoerServlet in the diagram the call printf(formBouwerRemote.bouwHtmlFormOp()) is executed. During the creation of this sequence diagram we discovered that the diagram would become too large. The inner call

\[['getAlgorithmen/valideer():Vector
[1.1: algoritmeValideer():get('algoritmeValideering')='algemeneValidaties'].Str...

Figure 16 FormulierBouwer getAlgemeneValidaties()
in the print line statement is worked out in a separate diagram *Business FormulierBouwer bouwHtmlFormOp*.

The second problem, the Java API in the diagram was easily to solve. It was possible to select packages that we would like to show in the diagram. In order to generate clear and understandable diagrams we discovered that in some cases it was indeed important to show some Java API. The *Servlet UploadServlet*, for example, is a servlet to import an excel file with employee data. As can be expected somewhere in the sequence the excel file should be read from the uploaded file, within the sequence it was important for us to show the *inputStreamReader*. This occasional behaviour makes it difficult to automate this generation. Again we would conclude that at this moment only partially automatic generation is possible. To allow further automation, it is worth investigating the features of a system to determine the chance that a java package is important. In a calculation kernel the *java.lang.Math* package has a good chance to have added value in sequence diagrams or *java.io* has important information for applications that do something with files. In future work this could be investigated.

The third problem we encountered during the data gathering phase was the remote method invocation used by the application server. In the source file there is only a link to the stubs. These stubs contain only empty bodies, so the sequence generation will stop. We had to change the class into the real implementation to restart the generation. Almost all data within this data-driven application is delivered through EBs which means that the generation of a sequence will stop several times. We replaced all stubs with its...
implementation to get complete sequence diagrams. This solution to analyse OfferteNet is time consuming, because we had to identify all implementations and replace the interfaces in the sequences. We have not found any other investigation with similar problems.

A completely different solution is instrumenting the source code. A. Mesbah and A. van Deursen used AspectJ to couple the interfaces [MD05]. Programming aspects could be used to gather information of the system. For us this instrumenting was not possible, because we were not able to use a running version of OfferteNet.

Finally all diagrams were created. An example is given in figure 17. This is the execution of the method Business FormulierBouwer getAlgemeneValidaties(). This diagram is splitted into multiple diagrams because when a FormulierBouwer bean is created many actions are performed. Figure 17 starts with the ejbCreate(...) call, which finds it origin in the InvoerServlet. In the FormulierBouwerSLSB life line several method calls are displayed: setFormulierId, setSecitieId etc. The details of these calls can be found in extra diagrams, which are named with the described naming conventions. An example of a detailed diagram is figure 16.

In the diagrams we left out the whole sequence of getting a variable. For example in InvoerServlet.java there is a call getSectieId() (figure 18). This call invokes the FormulierBouwerSLSB by remote method invocation this will invoke a Sectie, which finally will return its Id. We left these calls out because it saves a lot of space in the pictures and the sequence is performed several times and is equal for each variable. So the diagrams show that a get variable is performed, but the whole sequence in which a BeheerService till a model-object is requested is not shown.

Figure 18 getSectieId() example

4.6.3. Information Interpretation
The sequence diagrams give a lot of information about the system. What we see is that for almost all sequences, there is a sequence of a bean or servlet that calls a data manager and then some data will be requested. This data is formatted and sent back on response of the web server.

The data is delivered by components or EJBs. There is another difference between aebot.components, aebot.ejb and com.softwarehouse.efinance.components. When aebot.ejb is used there is a JNDI lookup to locate the bean. On the other hand for aebot.components we see a BeheerService creation, and then a JNDI lookup or a DAO creation. Most of the time in find or get methods a DAO is used in the set and add methods they create an EJB. In com.softwarehouse.efinance.components persistence is always performed by a DAO. This means that in the BeheerServices the responsibility for searching in de container is moved from the presentation layer to the business layer itself. The JNDI names differ for the aebot.ejb and aebot.components. For example the JNDI name for aebot.ejb.downloadpack is Download. For the aebot.components they use the prefix ejb/. The JNDI name for aebot.components.download is ejb/Download. This observation is useful because it is easier to find the related EJB when viewing the source
The possibility to use two JNDI names and EJB for the same data is wrong, because there should be a single point of definition. It is difficult to find clear separations between business and data layers. In section 4.4 we expected that we removed the presentation layer, except for the servlets because these are also called from the client. Starting from that point our expectation was that the beans, which would be included by the JSPs, would represent the business layer, however the sequence diagrams show that there is a sort of mid-office. This mid office is formed by classes that act as an interface for all related functionality. For example the `MidGebruikerPack` (MidUserPack) is an interface for all functionality related to OfferteNet users. After some research we discovered that sometimes a mid office is used and at other times the business layer is passed. In table 8 we summarized the used mid-office classes and the classes that represent tables from the database accessed from the bean.

<table>
<thead>
<tr>
<th>Beans</th>
<th>Mid-office classes</th>
<th>Data-layer classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>jspDownloadBean</td>
<td>aebot.ejb.downloadpack.*</td>
<td>aebot.components.download.*</td>
</tr>
<tr>
<td>jspGebruikerBean</td>
<td>aebot.ejb.midgebruikerpack.*</td>
<td>aebot.components.formulier.*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.ingevoerdedata.*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.invoerrelatie.*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.invoerveld.*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.keuze.*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.managementinfo.*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.offerte.*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.produkt.*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.sectie.*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.tussenpersoon.*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jspHelpTekstenBean</td>
<td>aebot.components.invoer.Midstore.*</td>
<td></td>
</tr>
<tr>
<td>jspPDFConverterBean</td>
<td>aebot.ejb.pdfconverterpack.*</td>
<td></td>
</tr>
<tr>
<td>jspRekenboxBean</td>
<td>aebot.ejb.rekenboxpack.*</td>
<td></td>
</tr>
<tr>
<td>jspSecurityBean</td>
<td>aebot.ejb.midgebruikerpack.*</td>
<td></td>
</tr>
<tr>
<td>jspUserSession</td>
<td>aebot.ejb.midgebruikerpack.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WerkgeverInvoerBean</td>
<td>aebot.util.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.invoer.Midstore</td>
</tr>
<tr>
<td>WerknemerInvoerBean</td>
<td>aebot.util.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.invoer.Midstore</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.ejb.midgebruikerpack.*</td>
</tr>
<tr>
<td>WerknemerUpload</td>
<td>aebot.beans.CSVParser</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>aebot.components.invoer.Midstore</td>
</tr>
<tr>
<td>Servlets</td>
<td>Mid-Office classes</td>
<td>Data-layer classes</td>
</tr>
<tr>
<td>BatchServlet</td>
<td>com.softwarehouse.efinance.</td>
<td>aebot.servlets.BatchService</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EmailServlet</td>
<td>aebot.beans.jspGebruikerBean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aebot.beans.jspSecurityBean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aebot.util.*</td>
<td></td>
</tr>
<tr>
<td>InvoerServlet</td>
<td>aebot.components.invoer.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aebot.util.*</td>
<td></td>
</tr>
<tr>
<td>MemoServlet</td>
<td>aebot.beans.jspGebruikerBean</td>
<td></td>
</tr>
<tr>
<td>Offreerbare</td>
<td>aebot.util.*</td>
<td></td>
</tr>
<tr>
<td>ProductenServlet</td>
<td>aebot.components.invoer.MidStore</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aebot.ejb.midgebruikerpack.*</td>
<td></td>
</tr>
<tr>
<td>StoreServlet</td>
<td>aebot.beans.WerknemerInvoerBean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aebot.beans.WerkgeverInvoerBean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aebot.components.invoer.Midstore</td>
<td></td>
</tr>
<tr>
<td>UploadServlet</td>
<td>aebot.frontoffice.servlets.Werknemer</td>
<td>aebot.components.tellergenerator.*</td>
</tr>
</tbody>
</table>
In the application and sequence diagrams we can see that most JSPs and servlets are static layouts, filled with dynamic data. In the other direction the filled forms will be stored in the database.

As described earlier this data is delivered and stored by components. Sometimes the mid-office mediates between the presentation layer and the components. There is one exception to this situation namely the InvoerServlet, because in this case data and layout are dynamic. There is only one template defined with the style of the screen. Furthermore all fields and labels are located in the database. Because of the complexity we will describe this servlet in section 5.1.

After creating these sequence diagrams we have a lot of detailed information about OfferteNet. But what we miss is a way to get a quick system overview. So we would like to leave out some details and compress the diagrams, after all we want to discover the high level architecture and not all source details. In the fourth iteration we will compress the sequence diagrams to get an overview.

4.6.4. Discussion

We have chosen sequence diagrams, because we preferred the theory we described in section 2.3.4. When we read papers about information systems the structure is mostly straightforward. The user enters information, then the information is transformed and maybe stored in the database. This route is vice versa travelled to the user again. Of course this transformation can be complicated, because there could be a dependency with data or other parts of the application.

Because of this expected behaviour we decided to use sequence diagrams. Sequence diagrams give the total flow for a certain user action. A disadvantage of sequences is that when many user actions use the same business classes, sequence diagrams give much redundant information.

Afterwards we can say that our expectations were right. The created diagrams give the information we need and the redundancy of information is limited. However many data classes have the same structure so the redundancy can be found in different components that perform the same steps.

For us this step was the most clarifying in the whole process. The sequence diagrams show which data classes are used by which method. However the iteration result is useful the data gathering step was difficult, because of the stubs and JNDI lookups in the EJB’s. The static references to the JNDI or to the stubs are interpreted at runtime and then forwarded to the EJB or the implementation. We analysed the static source code a better solution was probably instrumentation, or dynamic call analysis. We have chosen for static analysis, because:

- there was no test server available
- the database is only available in England and thus there were no possibilities to start OfferteNet in a debugger/analyser

For us this was extra work, but the result is the same in both cases.

4.7. Fourth Iteration – Compressing the Sequence Diagrams

In some cases the results of the previous iteration led to very large diagrams. Often we see sequences that become predictable after studying some detailed sequences. For example the calls from the business layer to the data layer, or the sequence in components where the data package fills a model. We think that diagrams hiding these implementation details would help us.

The diagrams we created in the previous section should be compressed to leave out the implementation details.

When we started this iteration we even did not know how to compress these diagrams. We searched for possibilities, but we did not find any technique to fulfil our wishes. Many
researchers investigated the same area for example Briand [BLM03], Systä [SKM01] and Riva [RR02], but all of them use dynamic techniques. Briand and Riva create the data by instrumenting the source code. Systä is using a customized debugger to gather dynamic data.

The sequence abstraction step is different for the three investigations. To manage the explosion of the diagrams Systä recognizes pattern repetitions and presents these patterns as sub scenarios or with a special repetition construct. These sub scenarios do not always improve the readability of the diagrams. The readability was improved by renaming the sub scenarios manually. Riga uses also abstraction by inserting hierarchical models. These models are recognized with help of prolog. A disadvantage is that the initial identification should be performed by architects and then they should be added to the abstraction model. Briand does not use any abstraction methods.

The models we found are not applicable, because in OfferteNet there are almost no equal patterns. After some brainstorming we decided to compress the sequences to package level, instead of class level. We expect this compression technique to have several advantages:

- In future it should be possible to automate this process
- Compression will reduce the number of objects in the sequence diagrams
- Getting a system overview is relatively easy.

4.7.1. Target, Source viewpoint and mapping rules

The target situation is of course an abstracted set of sequence diagrams. To create this abstraction we use the sequence diagrams created in section 4.6. These sequence diagrams represent our source viewpoint.

The mapping rules are a little bit more complex, because they are dynamic. When we decide to make an abstraction on package level we should also decide the level at which we want to abstract. For example grouping on aebot.components level is not a desired situation, because we loose too much information. In this case only object remains. In contradiction aebot.components.formulier.data could be too much information, because the abstraction level is too low. In this case aebot.components.formulier is the abstraction level required, but we still cannot conclude that a depth of three is desirable, because com.softwarehouse.efinance is not a good abstraction. In that case we prefer a depth of five (com.softwarehouse.efinance.components.beroep), because the number of objects is reasonable and the resulting view is still readable.

After a quick scan through the existing sequences we decided to choose a depth of three and adapt it when a sequence shows too much or too little information. So the mapping rule is that the objects within a sequence diagram will be joined at a package level of three.

4.7.2. Data Gathering

To create the sequence diagrams we use TogetherJ Control Center.

In the first place, we made a copy of the diagram. The second step was renaming all objects to the package name by hand. In the third step we merged all activations that addressed packages with the same name. We executed all these steps by hand, but we formalised the steps so it is possible to automate these steps in future.

We processed all sequence diagrams except the ones of the Business ...StepExecutor and the BatchServlet. In these cases we changed the depth, because a depth of three would not give us the desired results.
For *com.softwarehouse.efinance.aebot.web.servlets* in the *Servlet BatchServlet Overview* we used a depth of six, because we expected that the information represented by this class (a servlet) is crucial in understanding the system. A lower depth would result in a diagram with only one object, which hides all information about the servlet.

The same holds for the step executors. In that case the choice for a depth of three results in a diagram with just one object. We made the decision to use a depth of four.

A result of this data gathering phase can be seen in figure 19, the source for this sequence has already been given in figure 16. We can see that the following translations are executed:

- *aebot.util.ResourceManager* is translated to *aebot.util*
- *aebot.components.invoer.beans.Relatiemanager* is translated to *aebot.components.invoer* (depth of 3)
- *aebot.components.invoerveldvalidatieregel.model.InvoerVeldValidatieRegel* and *aebot.components.invoerveldvalidatieregel.business.InvoerVeldValidatieRegel* are translated to *aebot.components.invoerveldvalidatieregel* (depth of 3)

This example shows that two objects are merged. In larger diagrams many more objects are merged, which is to our opinion an advantage in getting an overview of the system.

### 4.7.3 Information Interpretation

After applying our abstraction method we were very pleased with the results. In many diagrams it delivered a good abstraction of the detailed view. For example all overhead of the business, data and model packages within the different component packages is filtered out and now it shows the name of the data that is collected during execution of this 'object'. Furthermore it hides all EJB overhead, the creation of a home and remote interface. And last but not least the diagrams are more compact so getting an overview is less complicated than before.
With these sequence diagrams we tried to create a system overview. In figure 20 we positioned the different packages in a sort of layered structure. After this positioning we tried to map the layers, promised in the documentation. We left out two parts of the system; the calculation kernel, which will be discussed later, and the stand alone batch client because this is not used anymore.

4.7.3.1. Presentation layer
Within the presentation layer we can see the `aebot.frontoffice` package, which contains all JSPs. Then on the borderline of presentation and business there are the servlets. We placed them on the borderline, because on the one hand they arrange some presentation tasks for example redirecting to the right JSP or even printing HTML-code. But on the other hand they also handle some business tasks, for example the `UploadServlet` will directly call the `TellerGenerator` for a new primary key.

4.7.3.2. Business layer
The relation of the JSPs with the beans could be deduced from their naming. But these are not the only classes they serve. They also assist the `Store`, `Memo` and `EmailServlet`. Table 8, in section 4.6 shows that there are some beans that have their own mid-office. For example the `jspRekenBoxBean`, which has the `aebot.ejb.rekenboxpack` as mid-office. Other examples are the `jspDownloadBean`, the `BatchServlet`, and the `UploadServlet`. The `aebot.ejb.midgebruikerpack` is mainly used by the `jspGebruikerBean`, but for example the `jspUserSession` and the `MemoServlet` use this package too. The `aebot.beans` package provides methods to the `aebot.frontoffice`. The front office gets the information needed within one call to the midoffice. The lower mid-office, found with a JDNI look-up, provides a similar interface to the beans and servlets, but this time within the context of the EJB-Tier.

![Figure 20 Hierarchical structure of OfferteNet packages](image-url)
This lower mid-office will perform a sequence of calls to the data layer and finally it returns the answer to the caller. Within this process there is again one exception the `aebot.components.pdfconverter` package. This package creates an ftp-session with the PDF-server to translate the rtf-documents to PDF-documents.

4.7.3.3. Data Layer
As explained in the previous section the data layer is called from the mid-office, the `jspGebruikerBean`, or the `UploadServlet`. Within this calling process there are two possibilities: there is a JNDI look-up from the mid-office (an `aebot.ejb` component) or a stateless session bean is created (a `BeheerService`). The data classes will query the database or in other cases the container will query the database. After querying a model is filled and finally returned to the `BeheerService`. The `BeheerService` will return the information requested to the mid-office.

4.7.4. Discussion
In this section we want to answer the following questions:
- Does this iteration deliver what we expected?
- Is this abstraction method possible and significant for each project?
- Can we automate this abstraction?

Let us start with the first question. This question is relatively easy to answer because in the first part of section 4.7 we stated requirements for our method: possibility to automate the process, compression of the diagrams and it should not be difficult to get a system overview. Automating the process is stated as a separate bullet and thus we will answer it later. Our second requirement, sequence diagram compression, was successful, because many diagrams consist of significant fewer objects. Furthermore we can conclude that in most cases the complexity of the diagrams has decreased. For example the home and remote interfaces for remote method invocation are not visible anymore, but it still gives us a good view of the system. The results of this iteration fulfil our requirements and thus this iteration helped us.

It is more difficult to conclude that this method is a good method for each project. Firstly, we use the Java characteristic that the mapping onto the file system is equal to the mapping of source onto the file system. Our method is thus applicable to all Java software systems. Secondly, we can say that this method meets our goals, but we think it depends on the hierarchical decomposition if compression succeeds. For example when we compare the `Business AssignmentStepExecutor Overview` (appendix D1) with the `Business ConditionStepExecutor Overview` (appendix D2) we could conclude that they may use the same classes, but when we look at the detailed sequence diagram we can see that there is almost no similarity between these two sequences. In this case compression with our technique will not give the desired results. Of course we tried to find the reason of this sundry in results. To answer this question we need a definition of a package: “A package is a collection of related types providing access protection and namespace management. Note that types refers to classes, interfaces, enums and annotations. [URL4]” Looking at the package structure we see that `aebot.packages.components` is divided into an extra level of packages. These package names point to the semantics of that package (`document`, `pdfconverter`) and the sub packages point to the functionality (business, data and model). In the `com.softwarehouse.calculationkernel` there is no sub packaging into semantics, but only into functionality. To get the desired results there should be a package structure that tells something about the semantics of that package.

Now we will answer the question if it is possible to automate this process. The software we used would not allow to export a sequence diagram into an editable format, so we could not automate this specific process. But it should be possible to create a sequence diagram in an editable format. We need this editable format because we have to group the
class objects to package level. The second step in creating the abstractions was defining the abstraction depth. Regarding to the text above we could advise to take a package-level that tells something about semantics. This step could not easily be automated, but it should be possible to assist a reverse engineer with a computer program. Thirdly, we cut off the last part of the package name. This step is just taking a substring and could be automated. Last but not least we should merge the sequences with the same package name. This is a string comparison followed by a merge operation. This could also be automated. We can conclude that the whole process cannot be automated, because the user should determine the depth. But after this determination the remaining process can be automated. It is possible to search for an optimum in number of objects, but we think that this is not always the best diagram, because we think that the semantics of names is also important. With relative less effort we can create an abstraction.

After only one application we cannot conclude that it works, but it is definitely worth to use and evaluate it for other systems.
5. Detailed Investigation

The documentation we created in the previous chapter is enough for almost all parts of OfferteNet. But there are two complex parts in the system, the input component and the calculation kernel. In this chapter we will give a more detailed description of these two parts.

5.1. InvoerServlet

When invoking a servlet the application server will first create an instance of it. During this creation the application server will call the init(...) method. This init(...) function will create two instances of the business layer, the MidStoreHome and the FormBouwerHome.

In the source code comments we can read the responsibilities of these two instances. The FormBouwer is responsible for building up screens and the MidStore is responsible for storing collected data in the database.

After the InvoerServlet is initialised the application server will call the service(...) method. This service method is the most important method for the input engine. In this section we will describe all steps that are executed. But before we start we will give a short introduction to the data model that is used. In the text we will use the names as stated in the OfferteNet source code, even if they are in the Dutch language. An English translation of the name is given between the parentheses.

Figure 21 Screenshot OfferteNet with data mapping
In figure 22 a part of the data model is visualized in an ER-diagram. This diagram is useful for understanding the relations between the elements. We deduced this diagram from the OfferteNet oracle data model. In this diagram the entities are visualized, which are needed to generate the screen during input of the proposal. Where the input data is stored is not visible in this diagram.

![Figure 22 Part of the OfferteNet data model](image)

A Sectie (section) is a screen that is shown when the InvoerServlet is accessed. It consists of InvoerRelaties (input relations). An InvoerRelatie has several attributes InvoerVeld (Input fields), Helpteksten (help texts), Keuzes (choices) and InvoerVeldValidaties (input field validation rules).

An InvoerVeld is a form field that is shown onto the web browser screen. An example of an invoerveld is given in figure 21. The line “Werkgever:”, above the boxed InvoerVeld, is also an InvoerVeld this time of type label. For InvoerVeld several types such as text, radio, input, dropdown and label are possible.

For certain invoerveld types something has to be chosen. In figure 21 we can see the possible choices for rechtspersoon. These choices are called keuzes (choices) within the application. Keuzes are used in the case of radio-buttons, drop down boxes etc. Sometimes it is possible to view extra information about an invoerveld, this is called a helptekst. The helptekst appears by clicking on the boxed ‘i’-picture. After clicking a pop-up will appear providing some extra information about the field.

An advantage of OfferteNet is that Produkten (products) can use the same sectie. This is done by adding a FormulierId (Form Id) into the table Sectie. Formulier has an n-to-1 relation with Produkt. Formulier makes it possible to have more than one flow for the same product. When two products use the same sectie they have almost the same entry in the Sectie table. What differs are the SectieValidatieRegels (section validation rules), these rules determine the next section. The decision on what is the next section depends on the true or false of the validation rule.

The table SectieGeschiedenis (section history) is used to fill the “spring naar”-drop down box with secties that have already been passed.
In the coming subsections we will explain the InvoerServlet steps.

5.1.1. Step 1: Deriving parameters
Firstly, all request parameters will be derived and possibly completed with session parameters. These parameters are important for other steps.
In this parameter gathering we found a security leak. When there is no session a new session is automatically created without asking credentials and filled with URL/request parameters. A hacker should have knowledge of the obligatory parameters and give them a correct value in the URL to exploit this vulnerability. This results in an OfferteNet screen with all the proposal information. For security reasons we will not publish the possible parameters. This security leak could be fixed by creating a dependency on an existing session or other server side data.

5.1.2. Step 2: Determining section information
In this step the current section is determined. This is done by getting the variable sectieid from the request. There are several options:

- No sectie id in request: it tries to find the sectie id in the session-data otherwise sectie-id is set to -1 and during the process it will be translated into the first section of the product.
- sectie id > -1: this is the sectie id we use
- sectie id < -1: in this case it is a special request

The ordinary sequence of actions is validation of the fields, storing data, validating the section, storing history, change the status and finally redirect to another servlet or building up the next section.
Of course there are some exceptions to this process. The buttons “Spring naar...” (jump to) and “Vorige” (back) implement functions to move around. When we try to edit an existing proposal, the system should also pass validation, storing and section validation, because there is no new data on the request. After this first screen OfferteNet will take the normal sequence again. In these three cases the steps described in the sections 5.1.3 till 5.1.7 will be passed.
The last sub step is the determination of the next section. The next section consists of two possibilities depending on the validation result.

5.1.3. Parameter field validation
As described earlier an InvoerRelatie has some attributes being foreign keys. One of these attributes is the InvoerVeldValidatie. An InvoerVeldValidatie is a rule to validate the user’s input. An example a telephone number check. A (Dutch) telephone number may consist of ten digits and an optional dash. This check is written in Java script and server side executed. When the check is finished the result is “goed” (Ok) otherwise an error message will be returned.
This whole check is initiated by the FormulierBouwer.validate(...) method: It reads the InvoerRelatie from the database by using the RelatieManager. With the InvoerRelatie it is possible to get all the attributes needed. These attributes are validated by applying the Java scripts from the database belonging to that field. Furthermore there is a check of some global validation rules. All fields that pass all checks will be returned to the InvoerServlet in a hash table.

5.1.4. Store valid values
The next step is storing all valid fields (from the first step) with the MidStore in the database. The MidStore will invoke the StoreManager.storeSectie(...). When it is the first time the data is stored then everything will be inserted into the database, otherwise the old data will be updated. Within the OfferteNet database the data will be inserted in the ingevoerdedata table. This table is attached to the offerte (proposal) table. In figure 23 the
whole ER-diagram of the proposal data can be viewed. This diagram visualizes all entities that are important for the data input. In figure 22 we can see all entities to generate the screens. This is also an ER diagram derived from the oracle data model. Besides the entities for input we show also the entities with info about intermediaries and users.

Figure 23 Part of the OfferteNet data model

5.1.5. Section validation

This step will only be executed when the parameter validation was successful. A section validation is not a validation in the sense that it checks parameters or the constraints between parameters. But it makes a decision depending on the choices made and the input parameters given. For example, when a customer wants an “Excedent-regeling” it checks if field v141 has value 1. Depending on this result (true or false) the system will decide what the next section is. In section 5.1.2 we determined already the sections to jump to depending on

Figure 24 Screenshot OfferteNet
the result. If the section validation is true we jump to X and otherwise the next section will be Y.
These section validation rules are also a weakness of OfferteNet, because this structure means that only one decision could be made in a section. The result of this approach is that sometimes only one question is shown on a screen (figure 24).

5.1.6. **Store history**
In figure 24 we boxed the stored history, this drop down box makes it possible to jump to a previous section. In the example of figure 24 there are 4 sections. This is done by the MidStore which gives the control flow to the StoreManger. This history makes it possible to use the “Spring naar..”-button in the screens. After a jump to a previous section the user may change a variable influencing the future flow. This possible change of the flow means that a jump forward is not allowed after a change.
To implement this behavior there is a method in the StoreManager that will insert or update the offerteid/sectieid combination and the offerteid/sectieid combinations having higher numbers will be deleted.

5.1.7. **Change proposal status**
New proposals will get the initial status A which means “to be calculated”. The proposals that have already a status higher then A are changed to the initial status A before accessing the **InvoerServlet**.
The possible status conditions in OfferteNet are:

- A: to be calculated
- B: published
- C: accepted
- D: juridical documents send
- E: done
- Y: elapsed (not accepted)
- Z: elapsed (discontinued)

5.1.8. **Build next screen**
Generally we can say that the next screen is also a section, but in some cases a redirect to another servlet should be performed. OfferteNet uses special values for these redirects:

- -10 for input of employees
- -20 for the list of employees
- -30 for the redirect to the calculations kernel

These special values are formulated in a routine. In table 9 we present the possible options of redirection. The first six columns present values of attributes OfferteNet calculated in the previous sections. The last column is the redirection URL.
When for example the parameter validation and section validation result is true. Furthermore, the **next section by true** equals -10 and the **terug (back)** attribute is false. The system will redirect the browser to StoreServlet?page=start.
<table>
<thead>
<tr>
<th>Parameter validation</th>
<th>Sectie Validatie</th>
<th>Next sectie when true</th>
<th>Next sectie when false</th>
<th>Terug (Back)</th>
<th>Sectie ID Start</th>
<th>Redirect to</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>-10</td>
<td>False</td>
<td>*</td>
<td>*</td>
<td>StoreServlet?page=start</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>*</td>
<td>-10</td>
<td>False</td>
<td>*</td>
<td>StoreServlet?page=start</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>-20</td>
<td>False</td>
<td>*</td>
<td>*</td>
<td>StoreServlet?page=werkgeverlijst</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>*</td>
<td>-20</td>
<td>False</td>
<td>*</td>
<td>StoreServlet?page=werkgeverlijst</td>
</tr>
</tbody>
</table>
*                      | *               | *                    | True                  | -10         | *              | StoreServlet?page=start |
*                      | *               | *                    | True                  | -20         | *              | StoreServlet?page=werkgeverlijst |
| True                 | True            | -30                  | False                 | *           | *              | Calculation Kernel |
| True                 | False           | *                    | -30                   | False       | *              | Calculation Kernel |

* = Wildcard

If none of the redirection rules could be applied then the system should generate a new page. Because the variable sectie id is set to the next section, there is only a call to the FormBouwerBean to build a new HTML page. The results are written to the client’s browser by using a print writer.

Because the written HTML to the print writer contains a form posting its variables to the InvoerServlet, this cycle is executed several times until the input is complete and the system ready to generate a proposal.

### 5.2. Calculation Kernel

In fact it is not only one calculation kernel, but in OfferteNet there are two calculation kernels. Firstly, we will tell something about the history of these kernels. Once there was a programmer who programmed a calculation kernel in C++. Several years ago OfferteNet was created and it needed a calculation kernel. The external software house translated the old C++ kernel into Java and from that time on it calculated all collective pension proposals for OfferteNet.

After a couple of years AEGON made a request for change to add more products, the individual pension proposals. During impact analysis of that request it was discovered that the old kernel was too complex to add another product. A new kernel had to be developed. The new one is based on the Java Calculation Framework. The JCF-Calculation Kernel is a product from an external software house and is developed for insurance companies.

During this investigation the old calculation kernel is ignored, because there is already an impact analysis to migrate all products to the new kernel. To make this documentation complete we created two pictures of the old kernel which can be found in the appendix E, but we will not explain them.

A calculation kernel is very domain specific software, and many colleagues told us that it is the most complex software an insurance company uses. When we started our research we did not have any knowledge of the financial world at all. The first step for us was extending our knowledge about calculation kernels. To get this knowledge we visited TKP Pensioen in Groningen. They explained how their calculation kernel works and the described the concepts and ideas behind the kernel.

In this section we will not explain all ins and outs of the calculation process in terms of pension domain knowledge, but more in general, how the framework works.
5.2.1. Connection with OfferteNet

What we could see in figure 25 is that the calculation kernel itself is a separated entity in OfferteNet. It is bound to OfferteNet because it uses the same data-layer. The only arguments that are given to the calculation kernel are the model id and the proposal id. The calculation kernel will get its data from the database. It uses the aebot.util.Resourcemanager to get a database connection from the JDBC-pool. In the sequence diagram we have seen that first a CalculationKernelFactory is created. This factory will get the XML-model from the database and set it to the new CalculationKernel object. Furthermore it will create a StepExecutor with a new worksheet and finally it returns the CalculationKernel. The RekenBoxBean will call the CalculationKernel calculate method and the calculation will start. When finished the calculation kernel will return a boolean to the RekenBoxBean. This result is propagated to the JSP file. Depending on this result the JSP will redirect to the Uitvoer (output) page or the rekenbox_fout.jsp (calculation box failure) page.

5.2.2. How JCF works

The framework is in essence a classic calculator. In theory everything can be computed with this framework, but of course this application fits very well in the world of financial products. The vendor of the JCF published a figure of the situation for AEGON (figure 26). We think that this picture should be future but it is definitely not the current situation.

![Figure 25 Positioning Calculation kernel](image1)

![Figure 26 JCF architecture overview](image2)
In picture 26 we see calls from other systems to the calculation kernel. Within the concept this should be possible, but at this moment there is no data push model implemented. In the current situation there is a formula plug-in that imports all data onto a worksheet. This plug-in connects to the OfferteNet database, which means that only OfferteNet data can be used.

Moreover in the left part of the picture there is a release management system has been drawn. At this moment the XML-model is stored as a CLOB-object in the OfferteNet database and at AEGON there is nobody that has a client application to do model maintenance. Again we think that the proposed figure could be realized with the JCF concept but at this moment that situation is not relevant.

The reason why we inserted figure 26 is the right part of the system. Here we can see the structure of the kernel and the objects. In the remaining text we will discuss the calculation kernel, formula plug-in, worksheet and model.

As mentioned before the calculation kernel is a calculator for the financial world. When creating an instance of the kernel, a model is loaded into the executor. This model is also XML, it consists of model steps and the parameters needed for the step. Several model step-types are possible. Below we present a list of the possible steps. The \textit{input} and \textit{parameter} reference steps are in italics because they are not in use at the moment.

- Assignment-step
- Condition-step
- Data Reference-step
- Formula-step
- Include-step
- Input-step
- \textit{Parameter Reference-step}

The worksheet is the data supplier and data storage for the system. Data is stored in XML-format.

The calculation kernel uses JEP, which stands for Java math Expression Parser. JEP is included as a library and it is used for parsing and evaluating mathematical expressions. It is possible to insert a formula as a string and instantly evaluate it. The power of JEP is the possibility to define variables, constants and functions. Of course a number of standard functions are already included in JEP.

These functions are written in Java and should be announced to JEP. This is done by the \texttt{com.softwarehouse.calculationkernel.business.engine.formula.JEPPFactory} which has the method \texttt{addCustomFunctions}. Here functions will be added to the JEP parser, by using static strings from the \texttt{JEPPFormulaNames} class.

Functions are implemented in Java and they are called plug-in in the calculation kernel. All plug-ins contain a \texttt{run} method with the following signature \texttt{run(Stack inStack)}, which throws ParseExceptions. The argument \texttt{inStack} is a stack where arguments can be popped, and the results can be pushed on the stack after calculating.

These plug-in help us to define specific pension functions, for example to calculate the pension when a person reaches the age of 65.

There is also a second type of plug-in which is used for the \textit{datareferencesteps}. These plug-in contain an \texttt{execute} method. They insert data into the worksheet in figure 26. This data is called bulk data.

In the subsections below we will discuss the different modelsteps. In the sequence diagrams we created there are also sequences about these steps. But many links being made by parsing XML and reflection, it is not possible to create complete sequences. Because of these XML-dependencies we think that some extra information about these steps will help the reader to understand system working. We explain the modelsteps by using an example.
5.2.3 Assignment-step

This model step is a step to define some variables that can be used later. These data will be added to the worksheet.

To search in the worksheet XPath is used. XPath uses path expressions to select nodes or node-sets in an XML document. These path expressions are very similar to the expressions you encounter working with a traditional file system.

The variable giltigheids_maanden from the example can be used in other samples by referencing ‘/worksheet/stepresult/temporaryData/geldigheids_maanden’.

Figure 27 Example of Assignmentstep

Figure 28 Example of Conditionstep
5.2.4. **Conditions-step**

Within the concept it is also possible to create conditions. Between the tags `<conditionContent>` we can formulate the condition. The condition is also written in the XPATH language. Depending on the result a part of the ConditionResult will be executed. There are two possibilities the `<IfTrue>` tag or the `<IfFalse>` between these tags it is possible to define other modelsteps. The `<IfFalse>` tag is optional, it does not need to exist.

5.2.5. **Data Reference-step**

The data reference step is used to import the bulk data. What is important is the paramReference, given in an XPATH query. In the given example this is the proposal id. The second important tag is the `<dataClass>` tag, because this tag will point to the class that should be executed. The method execute will be called with the parameter as reference, after performing this method the result will be added to the worksheet in ‘/worksheet/stepresult/DataReferenceIngevoerdedata’.

**Figure 29 Example of Datareferencestep**

5.2.6. **Formula-step**

There are two types of formula steps, JEP or XPATH. In the case of JEP a JEPFormulaParser will be created, when the XPATH type is selected an XPathFormulaParser is created. For almost any modelstep the JEP type is used. The formula will be parsed and the result is again written to the worksheet.

In the case of a JEP parser the functions will be used as described in section 5.2.2.

**Figure 30 Example of Formulastep**

5.2.7. **Include-step**

An include step is a step that makes it possible to insert modelsteps into the modelstep array already existing.
A model-id is given between the tags <ModelID>. The modelsteps of that model id will be taken from the database and inserted into the existing modelstep array.

<ModelStep>
    <IncludeStep>
        <ModelID>6</ModelID>
        <description>Doelvermogens voor NP</description>
    </IncludeStep>
</ModelStep>

Figure 31 Example of includestep

5.2.8. Discussion
In this section we give an overview of advantages and disadvantages of the chosen architecture.
The architecture of a calculation kernel is worth to analyse with a technique such as ATAM (Architecture Tradeoff Analysis Method), but we left this part out during our investigation. Nevertheless AEGON asked us to make some statements about the calculation kernel. To meet this wish we make some statements about the chosen architecture with help of some quality attributes of software architecture.

Especially pension products will be administrated for about forty years. During lifetime of a product laws will change and a product will be adapted to these new laws. Because of these reasons modifiability and maintainability will be important. However products will be adapted the essence of assurance has not been changed in years. This means that parts of pension products can be reused for new products. Re-usability and documentation are quality factors that help us to realize a fast implementation of new products. Performance is needed, because pension calculations consist of many sequential calculations. Last quality factor is scalability we selected this quality factor, because we think it could be important for the future. Summarizing the previous text we think that modifiability, maintainability, performance, scalability, re-usability and documentation are the most important quality factors for a calculation kernel.

Modifiability is one of the strongest points of this calculation kernel. Since the elementary framework is just a calculator, we think that this kernel can be used for every product that needs calculation.

A second strong point is that complex calculations can be formulated in a plug-in, which is re-usable for other products. The question for this system is how specific do we make plug-in? We can define two extremes, one in which we define many pretty small functions or the situation in which we define functions that calculate large parts of the total calculation. The optimum should be somewhere between these extremes. The configuration manager of the calculation kernel has to guard the optimum. He should think in general calculations, but they should be specific enough to keep the formulas obvious.

So modifiability depends not only on technical aspects, but also on the product definition in the kernel itself.

About maintainability we are more negative. The scripting language we can use for defining calculations is a sort of meta language. With use of an XML-language we can address classic implementations of functionality (plug-in) with statements in XML. This language makes the kernel extremely flexible, but at the other hand we do not have the advantages of a compiler. In this situation the compiler cannot do static type checking, because the method call is made by using an Object stack. All errors that would be compile errors in classic implementations will become runtime errors. For example calling a method with two Strings as arguments would become in a classic implementation exampleMethod(String stringOne, String stringTwo), if we make a call to this method with exampleMethod(int, int) compilation will fail. In the case of an Object stack the compiler
would not complain, because the type of the stack is identical. But if there are two integers on the stack while the method tries to pop two Strings an error will occur at run time. Furthermore it is not possible to debug the XML-script, because there is no debugger for this kind of scripts. This maintainability problem has become worse, because all variables are numbered in OfferteNet and do not have a descriptive name. So field 95 could be numeric while 96 could be a string, these numbers make writing scripts very complicated. Performance is a less important issue, because it required a proposal to be made within ten minutes. The calculation kernel calculates a proposal within a minute so this requirement is easily fulfilled. When we compare the performance of this framework to a classic implementation the framework performs worse. First reason for this difference in performance is the XML-parsing and unparsing, which are all string operations that perform badly. A second reason is the use of reflection, which is also relatively slow. Scalability is not a problem, because the kernel is written in Java and is accessed from a session bean. Scalability is one of the issues that is covered in the J2EE technique. Re-usability is also excellent in this framework, because it is possible to use a plug-in in another project. Furthermore we can use this framework for more then only pensions. So we think that the components of the kernel are qualified to use in other projects. Documentation is not so accurate, there are almost no documented functions in the code. And in case of comments in the code it is just a line indicating what happens there. It is recommended to use Javadoc in the code, because this will help the developers to understand the system. To improve the calculation kernel documentation should be added to the methods. A second improvement for the near future is the use of naming conventions within the calculation kernel. Of course it is normal to use IDs in the database, but when we pass the data layer in the Java application it is the strength of Java that objects have descriptive names. Furthermore we recommend that the structure is guarded by a configuration manager. The flexibility in creating plug-in allows developers to create workarounds. These workarounds will increase the entropy of the system. Finally there is a model maintenance application drawn in figure 26, the existence of such a model maintenance application will help us to create and debug modelsteps.
6. Conclusions

6.1. Current Situation of OfferteNet

Interviews with users and functional administrators taught us that they were very content with the application. Use of the application is increasing every day. Of course there are some wishes for the application but they could be fixed in regular releases. Most of these changes are changes related to amendments in law or modification of the products. But there is also a seamy side and that is the technical aspects of OfferteNet. We will describe these technical aspects in this section. In the last section we will present some motivations for the implementation problems.

6.1.1. Dead code

We started with the presentation layer to create the sequence diagrams. We discovered the fact that several methods in the included beans were not in use. Examples of such methods are:

- jspGebruikerBean.getAlleWerkgeverNamen(...)
- jspGebruikerBean.getWerkgeverOfferteid(...)
- jspSecurityBean.removeAlleGroepBevoegdheden(...)
- jspSecurityBean.getRandomImage(...)
- jspWerknemerInvoerBean.checkData(...)

Even within the system there are many components that are not in use. For example the XlsServlet is still running, but not in use by the application. The XlsServlet sends the template for uploading employees to the client, but in the code we find a direct link to the file. These direct links give information about the file system on the web server, which is a possible security risk. The most important problem is maintenance. When the template name changes, both objects should be changed and thus the cost of changes will increase. There is also a standalone batch application within the source code. We believe that this client is not used anymore, because there is an online batch service running in a separate thread.

Last example of dead code is the jspPDFConverterBean, which is the only client of the aebot.ejb.pdfconverterpack.*. This PDF converter pack is again the only user of the aebot.ejb.tellergeneratorpack. In the current source the jspPDFConverterBean is not in use, because PDF documents are generated by a special thread. So there are two packages and a bean that are not in use anymore.

We expect that this dead code is also adapted during maintenance, and thus it causes extra costs for AEGON. Besides, it is confusing for developers to have two packages that convert PDF-documents. (aebot.ejb.pdfconverterpack and aebot.components.pdfconverter (used by the generator thread). This confusion causes extra time to do impact analysis and again extra costs.

6.1.2. Architectural comments

Good software engineering starts with a clear architecture. The OfferteNet architecture overview document describes a controller structure. These controllers (servlets) should provide the coordination between user-interface and business logic. During our investigation we have seen that beans are included in the JSPs and these beans contact the business layer. Only the InvoerServlet and StoreServlet act as a controller. For the calculation kernel and output we could say that this controller function also exists, but in this case there is almost no interaction with the presentation layer.
In our opinion the separation between presentation, business and data is a little bit vague. We know that it is difficult to create strict layers. But when a developer has to implement some new functionality, he should know where to place it in the architecture. In OfferteNet this structure is not clear. For example what is the business layer? Is it the jspGebruikerBean that accesses data classes? Or should we place this functionality in the aebot.ejb.midgebruikerpack, which also has methods that access data providers? We think that this vague situation must not exist. Or when there are indeed two business classes needed their naming and documentation should describe the difference between them. This complexity is worse in the data layer. For some tables in the database there are three components that provide data from that table. Probably in the first period only EJB technology was used, because of performance problems EJB/DAO structure has been introduced. It is good practice in software engineering to remove the old code and refactor the system. In the current situation a change in the database results in three changes in the code. This redundancy causes extra maintenance costs and confusion for the developers.

6.1.3. Performance DAO/CMP/BMP

A J2EE environment includes two different containers; the web container and the EJB container. For performance the EJB container is often the bottleneck [CMZ02]. In the reconstruction process we found several EJB implementations:

- **Bean/Servlet – Session bean:** Bean/Servlet in web container uses a session bean in the EJB container that accesses the database.
- **Bean/Servlet – EB:** Bean/Servlet in web container uses an entity bean in the EJB container that accesses the database.
- **Bean/Servlet – Session bean – EB:** Bean/Servlet in web container uses a session bean in the EJB container. The session bean uses an entity bean in the EJB container that accesses the database.

Use of EBs has several advantages, because the container can be responsible for providing component pooling, lifecycle management, client session management, database connection pooling, persistence, transaction management, authentication and access control. There are enormous performance differences, between implementations used, because not all implementation make use of all container possibilities.

Cecchet, Marguerite and Zwaenepoel investigated the performance and scalability of EJB applications [CMZ02]. Figure 32 is one of their results. In this figure the throughput in request/minute is measured for several implementation types.

The session bean implementation has the highest throughput, this implementation correspond to the bean/servlet – session bean implementation in OfferteNet. This implementation uses the fewest services from the EJB container. In OfferteNet this implementation is only used for read-only access.

When writing is needed OfferteNet uses an implementation with an EB. The Bean/Servlet – EB implementation correspond to the EB-CMP implementation. There is one example of a BMP bean. But this does not matter for performance as can be seen in the figure. The
Bean/Servlet – Session Bean – EB implementation corresponds to the session façade. The throughput in these implementations is almost half of the session bean implementation [CMZ02].

Within OfferteNet there are already fast lane reader implementations, but not for every component this pattern is implemented. To improve the performance it is useful to use session beans for read access. If also write transactions are needed it is useful to use the advantages of an entity bean.

6.1.4. Security

The developers of OfferteNet did not pay much attention to system security. The security model is quite simple. A user of OfferteNet belongs to a group. This group has rights in OfferteNet. Depending on these group rights menu options and pages will appear.

To check these rights the `aebot_security.inc` should be included in the JSPs. And in servlets the rights should also be checked. These steps are not consequently performed and thus there are many security leaks within OfferteNet.

A user is authenticated by a user id and a password this combination gives a session. A password is valid for three months and then it should be changed. When a user gives a valid combination a session will be created. This session should be checked before an action is performed. Also in this session management we found some vulnerabilities. At this moment it is possible to create a session without a valid combination of user ID and password.

Because of security reasons we will not present the vulnerable parts of OfferteNet in this report.

6.1.5. Code

During our investigation we scanned a lot of Java code. Our investigation was not meant as a code review but rather as an architectural study.

However during the reconstruction we read a lot of code and sometimes we identified bad coding. So this section is not the conclusion after an extensive code review, but rather facts we found while browsing the code.

We think that every developer should use the Java code conventions as published by Sun. These conventions should be used because [KND+97]:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve software readability, allowing engineers to understand new code more quickly and thoroughly.

Within the application these code conventions are not always used. For example the variables that are collected during proposal input are numbered.

Variables and parameters should follow the naming conventions. A variable name should be [KND+97]:

- written in mixed case starting with a lowercase first letter. And internal words start with capital letters
- short yet meaningful
- mnemonic, which means that other engineers comprehend the intended use

<table>
<thead>
<tr>
<th>Jaar</th>
<th>Premie in euro (*)</th>
<th>Premievrije waarde in euro</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;= rr260 format=&quot;num0&quot;&gt;</td>
<td>&lt;= rr270 format=&quot;num0&quot;&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;= rr262 format=&quot;num0&quot;&gt;</td>
<td>&lt;= rr272 format=&quot;num0&quot;&gt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;= rr264 format=&quot;num0&quot;&gt;</td>
<td>&lt;= rr274 format=&quot;num0&quot;&gt;</td>
</tr>
<tr>
<td>10</td>
<td>&lt;= rr265 format=&quot;num0&quot;&gt;</td>
<td>&lt;= rr275 format=&quot;num0&quot;&gt;</td>
</tr>
<tr>
<td>15</td>
<td>&lt;= rr266 format=&quot;num0&quot;&gt;</td>
<td>&lt;= rr276 format=&quot;num0&quot;&gt;</td>
</tr>
<tr>
<td>20</td>
<td>&lt;= rr267 format=&quot;num0&quot;&gt;</td>
<td>&lt;= rr277 format=&quot;num0&quot;&gt;</td>
</tr>
<tr>
<td>25</td>
<td>&lt;= rr268 format=&quot;num0&quot;&gt;</td>
<td>&lt;= rr278 format=&quot;num0&quot;&gt;</td>
</tr>
<tr>
<td>Einddatum</td>
<td>&lt;= rr269 format=&quot;num0&quot;&gt;</td>
<td>&lt;= rr279 format=&quot;num0&quot;&gt;</td>
</tr>
</tbody>
</table>

Figure 33 Part of unparsed document
As an example of bad naming we would like to show an unparsed table from a proposal, see figure 33. Keep in mind while reading this example that a proposal consists of 18 pages, and that there are 29 different proposal types that could be parsed. If all calculation results are numbered it is hardly impossible to maintain the documents.

Another naming problem we discovered many times is the method name. According to code conventions method names must start with a lowercase character and internal words must start with a capital letter. In figure 34 we can see two examples of wrong method names. The same example of the class Persoon (Person) shows that a person has two sexes. Two sexes is a contradiction with the real world. In this example the naming of the class failed and/or the attributes names are not well considered.

Error handling is a powerful Java mechanism. When something goes wrong the developer is able to correct the situation or to write an error message to the log-file. In case of classes throwing exceptions, the developer is forced to handle it or re-throw it. In some cases the developers of OfferteNet catch errors without doing anything (see figure 35).

A possible scenario could be that a customer calls our helpdesk with a problem. The system failed while it was creating a proposal. Because of the empty catch blocks it is not possible to trace what went wrong. In short we cannot help our customers.

A second phenomenon, not recommended in Java, is the use of catch blocks in the normal program flow. An example can be found in figure 36. This method always returns true, except when an exception is thrown [GJS+05]. However, these exceptions could be used to correct the flow it is better to use catch blocks to correct or log the error and after that a finally block to continue with the normal flow.

6.1.6. Documentation

In chapter 2 we already discussed the importance of software architecture. To maintain an architecture we need documentation. Besides high level documents we need also documentation on the lowest level. This documentation tells us:

- What the functionality of a method is.
• What the possible values of the parameters are?
• What can be expected from the result?
• What happens when something goes wrong?

Many times researchers and software engineers have emphasised the importance of documentation [BAI99]. Within OfferteNet we found comments that have nothing to do with professional software engineering. In table 10 we will give some examples of unprofessional documentation. These citations prove that engineers do not understand what the software is doing or that work arounds, spoiling the architecture, have been programmed.

Table 10 Comments in OfferteNet

<table>
<thead>
<tr>
<th>Comments</th>
<th>Free Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peril sensitive sunglasses recommended, nja wat de code zou moeten doen</td>
<td>Not clear programmed code. What the code should do</td>
</tr>
<tr>
<td>(denk ik).</td>
<td>(in my opinion).</td>
</tr>
<tr>
<td>ARGH!! “WgIdStore” “WnldStore” “KindStore” Wat zijn dat voor dingen?</td>
<td>“WgIdStore” “WnldStore” “KindStore” What do these variables</td>
</tr>
<tr>
<td></td>
<td>mean?</td>
</tr>
<tr>
<td>Gimme a break! Het is 20:00u en ik ben moe...</td>
<td>Give me a break! It is 20:00h and I am tired...</td>
</tr>
<tr>
<td>Ja, ik weet dat het niet netjes is, maar ik ga niet de hele Invoer</td>
<td>Yes, I know it is a workaround. But I am not</td>
</tr>
<tr>
<td>ombouwen.</td>
<td>willing to refactor the whole input component.</td>
</tr>
<tr>
<td>... waardoor het een redelijk zootje is geworden.</td>
<td>... why it has become a real mess.</td>
</tr>
</tbody>
</table>

These comments are just examples, many more examples can be found in the code. We think that these remarks prove that documentation is needed to maintain a system for a long time.

6.1.7. Discussion

In this section we discussed several issues that influence the quality of OfferteNet. These quality issues are caused by:
• short development time, due to marketing organisation pressure.
• modification, without architecture observance.
• improvements, without clearing past solutions.
• lack of documentation.
• discontinuity in developers.
• AEGON’s first J2EE project.
• no audits on the releases delivered by the suppliers

Many of these are classical examples of software evolution problems. To our opinion it is possible to correct these issues, but it is important to do this in near future. In the next section we will discuss more structural changes for OfferteNet.

6.2. Future of OfferteNet

OfferteNet is the first application that should be adapted to new product, is because quotations are being made with this application. In many cases fast implementation is expected. When software is used, there is the law of continuing change: “A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it [BL76].”

Furthermore during maintenance the law of increasing entropy holds: “The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it [BL76].” These laws of Belady and Lehman, are generally accepted in the software engineering world.
The large investment in OfferteNet leads to a rather long period of cost recovery. After this period there should be a period of benefit. This period of benefit is the most important for an organisation. Of course it is important that this period lasts as long as possible.

During the whole application life time maintenance is needed. The law of increasing entropy tells us that specific work is needed to keep the application well structured. Specific maintenance is needed, in order to postpone the moment of application freeze.

In the subsections of this section we will make some recommendations to improve OfferteNet.

6.2.1. (Re)applying the MVC pattern

Modularity has enormous benefits in designing applications. The programmer need not know all details of each component to change things successfully. There is in the model view controller pattern a separation of [KP88]:

- the parts that represent the underlying application domain.
- the way in which the model is presented to the user.
- the way in which the user interacts with it.

This division in three parts offers the possibility to update the user interface without adapting the model and controller. For example the introduction of a new look-and-feel will be easier.

This model-view-controller pattern also fits in the proposed architecture. The view should be placed in the portlet and the controller can be placed in the UI flow components. A framework such as struts can be used to introduce the model-view-controller model.

With this recommendation we will reach a situation in which:

- We can adapt the look-and-feel with less effort. This is important because the look of an application becomes more important in web design. AEGON needs possibilities to change very fast if the marketing unit asks for it.
- A better structured application. At this moment there are servlets that generate screens and interact with the user. This should be divided in a component that delivers the presentation and a component that interacts with the user.
- Many methods can be reused in the front-office.
- The front-office security can be improved.

6.2.2. One stand-alone calculation kernel

At this moment the calculation kernel is a separated component. But with a formula plug-in it reads data from the OfferteNet database and after calculation it writes data to the database.
We think that AEGON should isolate the calculation kernel and make it possible to deploy it at another physical machine that could be reached by a web-service. This situation will have several advantages:

- reuse by other quotation tools.
- no difference in calculation results over multiple systems (back office and front office).
- single point of definition.
- Independency of calculation kernel. Only the interface definition is important.

As early as June 2004 the software house created a budget planning to isolate the kernel. The proposed solution should be implemented. An overview of this solution is presented in figure 37. After proposal input an interface component of OfferteNet creates an XML with the input variables. Second the calculation model is determined. The calculation kernel receives the XML and model and starts calculating. The results are again an XML and the same interface component can write it to the database.

6.2.3. **Mapping onto the proposed architecture**

The proposed architecture stated in figure 2 has many layers. In essence it is the default three tier application: a presentation layer, a business layer and a data layer. But in this architecture overview more layers are drawn. We will discuss the architecture overview bottom-up.

The communication components provide us communication with the different databases. This could be the data warehouse, the mainframe (CAESAR), the calculation kernel or the output component. The crux of this layer is that AEGON would like to be application server independent. Currently OfferteNet is using a WebSphere data source to communicate with the database. It is possible to switch to another communication layer, but we would not advise this change. We think that AEGON has not the expertise to build and maintain such difficult software components. Furthermore vendors of application servers will have this expertise. The supplied components are used by thousands customers, so it is proven technology. Last but not least AEGON is prescribing WebSphere as preferred application software, which means that license costs are no issue.

Secondly, there is a financial risk every time a database driver is changing AEGON should maintain its communication layer. This change is very complex, which results in high costs. If we use a commercial product for the data communication the vendor will take these costs, and we only have to update our license to a newer version.

The second layer is the model, divided in two layers: data-model and business-model. The difference between these two is that in many cases data from the databases have many attributes that are not needed in front and mid-office applications. So the business model should be a data model optimized for the application. Of course there is a one-to-one mapping possible between the business model and the data model.

OfferteNet is a quotation system that is independent of mainframe and back office systems. The designed business model is almost the same as the underlying data model. If we implement the layers as proposed by the architecture we should add a layer data model which is almost the same as the business model. To our opinion we can leave out the business model.

Business model and data model will become important, when a connection with the mainframe is established. Until now there is no difference between these two layers. Our advice is to leave the current situation, because OfferteNet is a data driven application and many data accesses are executed.

Then there is the business service layer. AEGON wishes a service oriented application, so every presentation layer can use all application services. When we see figure 20 in section 4.7 of OfferteNet we have already a kind of services in the EJB tier. These services are not suitable to publish as service because it would become micro services. What we should do
is integrate multiple services. For example when asking a proposal we should not only return the proposal with an employer id, but the whole employer object. So within one question we return more then one attribute. This large object should be serializable to XML and with a service request it should be possible to request the object. OfferteNet is a data driven application, which results in many database transactions. We described the generation of the screens for the quotation input. In this input component sections are fetched from the database and input is stored again. The tables used by this component are designed for OfferteNet and will never be used by other applications. For these database calls performance is important. If we create a service layer this performance will decrease, because serializing and deserializing of XML is a heavy operation. So introduction of a web service on these parts will cost performance and there are no advantages.

Of course there are parts of OfferteNet that are useful for other applications, for example complete quotations or the management information. AEGON should determine which part of the data model is useful for other applications. The interesting parts should be published by introducing web services.

The business process layer is the layer, which should apply the business rules. In OfferteNet this layer is a very thin layer, because business rules are stored in the database. For example all validation rules in the input component are stored in the database. These are applied to the user input and depending on the result a decision is made. The largest part of the business layer is formed by the calculation kernel. In the previous section we already advise to create a stand alone calculation kernel.

Last but not least there is the presentation layer. AEGON has chosen for a portal strategy. We think that use of portals and portlets is a solution for a business problem. A strategy could not be built on a technology. The business reasons for portal technology are [SUN03]:

- possibilities for personalisation, presentation and security
- accommodating the unique business need for many audiences and departments
- reusability of code
- many portlets into one portal

In our opinion there is only one group of users and that are intermediaries. So there is no real need for personalisation. Furthermore at this moment there is only one business need for one audience.

OfferteNet is a full screen application; many screens present too much information to use only a part of the screen. This means that OfferteNet will become a large portlet and the advantage of many portlets into one portal does not hold anymore.

The only relevant advantage for AEGON is reusability. We think that portal technology is too complex and at this moment it is not the standard in industry. Our advice is to use a MVC pattern, because this is more cost efficient and realizes all advantages that are relevant for AEGON.

With this mapping onto the new architecture we think that maintenance will become easier, because responsibilities are better separated. One architecture means that AEGON-developers need knowledge about one architecture to maintain all applications. Of course agreements should be made in package partition and naming conventions, to be sure that developers can find the code that should be adapted for certain maintenance tasks.

6.2.4. More decisions on a screen

During the investigation we have spoken with users of the system. We discovered that for creating a proposal too many screens have to be visited. And often only one question is presented on a screen.
We think that this point could also be improved in OfferteNet, without changing the whole structure. As an example, we put a possible flow into a flow-diagram, figure 38. The decisions are of course section validations and the screens are sections.

When we look at this flow we can see three decisions. The second decision has a dependency with the first decision, though we will not decide if the first decision is ‘no’. The third decision is independent of the first decision, though we always have to make it. So it should be possible to integrate decision 1 and 3 and their screens 1 and 5.

To make this work we have to introduce a section block and a stack. An example of a block is screen 7 and 8, because they form a sub-tree in the flow. On screen 1 we can present the question of screen 1 and 5. We make decision 3 and if the answer is no, we put block B on the stack. Then we make decision 1 and depending on the answer we put block A on the stack. The next screen is determined by popping the blocks from the stack and evaluating the section validation rules.

This improvement will result in a better user satisfaction. Proposals could be made in less time and fewer screens should be filled.

6.2.5. Security

The last recommendation we would like to make is security improvement. At this moment the security component guards only the front-doors. Within the presentation layer functionality will be hidden for users depending on their group.

When we implement a service architecture authentication and authorisation become more important also in the business and data layer.

We suppose to connect with the single-sign-on services that are already in use by the two other web-applications of Service Center Pensioen. When a user logs in it creates a session. With this session id it is possible to get information about the user. This session id should become a user session variable in OfferteNet. The advantage of this session variable is that authorization can occur everywhere in the application and not only in the front office.

This improvement means that the whole user administration should be removed within OfferteNet.
During the investigation AEGON already created a connection with SSO, but decided to keep a redundant user administration. So each user should be added twice in OfferteNet and in SSO. We think that AEGON should move the whole user administration to SSO, although it is a complex operation. To avoid structural changes, a security interface component should be maintained. The backend of this component should not point to the OfferteNet database, but it should use the SSO-client to check rights and permissions. This recommendation is a win-win situation. Users will have only one username and password for all applications and AEGON should maintain one user administration.

6.3. Symphony and the reconstruction process

During our reconstruction we used Symphony to recover OfferteNet. In this section we will present our experience with Symphony and we will present its strengths and weaknesses. Symphony is not a technique to reconstruct a software system, rather a framework to select views and to guide the reconstruction process.

Symphony consist of two main steps, reconstruction design and reconstruction execution. Both steps have an iterative character, at any time it is possible to return to an earlier step. The first step, problem determination, is defining the problem statement. Several techniques can be used to gather one overall picture of the problem. The second step in reconstruction design is concept determination. Because Symphony is problem driven, we were forced to think about views that could solve our problem. Reconstruction design was a difficult step for us, because our problem statement was broadly defined. There were no problems with OfferteNet, except the need for more information about the system and how we could improve the system in the future. Anyhow after performing these steps we were able to reduce the number of important views to two necessary ones.

Second advantage of the problem elicitation step is that we have a guide for reconstructing. By selecting the views we defined our direction for the rest of the process. Of course Symphony is an iterative process and thus it is possible to modify the problem elicitation during the process. Because of this structured process we have not executed steps that were not needed.

The second part of Symphony is reconstruction execution. In this step the views, which were defined during problem elicitation, will be reconstructed. These views are reconstructed in the iterations, which are also well structured.

An iteration consists of five steps. The first step is defining the target viewpoint. The target viewpoint is a definition of the desired result. Then we have to define the source viewpoint, in short from which information do we extract the target. After performing these two steps we have to define how to reach the target with the defined source.

Next step is data gathering. During this step we selected a technique to create the target viewpoint with the defined mapping rules and source viewpoint.

Last step is discussing the information we extracted from the system. We could answer questions like: Is this the information we need? Do we need more information? Should we improve the information by applying an extra iteration over the steps?

Our experience with Symphony is that it introduces a very structured process. It structures the whole process but also the sub steps. Because of the outlined structure it is possible to compare step results, but also a whole reconstruction process.

We think that Symphony is a powerful method to reduce the amount of work, when solving a problem. Symphony forces the reverse engineer to think before starting the reconstruction and after a step it forces the engineer to evaluate the results.

The problem-driven approach is hard to apply, when there is no clear defined problem. For example, when a principal asks for system redocumentation, we are still forced to select views and stakeholders. In that case it is difficult to argue why certain views are selected and why others are not. Possible stakeholders may still be selected and views maybe reduced by selecting views that are interesting for almost all stakeholders.
6.4. **Further Research**

During our research we encountered some problems that need further research. While creating the sequence diagrams we discovered that showing all objects in the diagrams was not always desired. When for example a `toString()` method is called, it is not necessary to show all further calls that are made. It is not interesting to know how all attributes are translated to strings within this `toString()` method. But in other cases, for example the input component, it is indeed important to show all calls that are made inside the `service()` method. Within this service method a call is made to the `PrintWriter`, which gives us the information that a new screen is printed to the client. We think that it is possible to make a rational guess, for which components it is important to show all calls that are made and for which components it is not important.

For us the sequence diagram compression was a method that we applied by hand. But in the future it could be important to automate this process. That is why we applied the method in very stringent steps. Each time we applied the same actions and after applying all steps we start to interpret the results. So our process can be automated when it is possible to read and write sequence diagrams to a suitable data format. More investigation is needed to define this format. We should be able to export sequence diagrams. Then edit the objects and change the object names into the package names. And after applying this renaming step it should be possible to translate the result into a diagram again.

During the sequence diagram compression we needed a package level. We made a choice by analyzing the result and do it again if the information was not what we expected. But it is worth to investigate if it is possible to reduce the iterations, by creating expectations about the results before the compression is executed.

Finally we think that more investigation is needed in the architecture of the ideal calculation kernel. During our research we found two different styles. One in which the calculations were programmed in a classical implementation and two kernels which created a framework and a meta-language to define the calculations. Should we create a framework with a meta language or should we take the advantages of a compiler? Which structure can be maintained in the future, when law is changed and calculation should be adapted?
References


[HIL01] R. Hilliard, IEEE Std 1471 and Beyond, 2001


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title and Details</th>
</tr>
</thead>
</table>
## Appendix A: Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>BMP</td>
<td>Bean Managed Persistence</td>
</tr>
<tr>
<td>CMP</td>
<td>Container Managed Persistence</td>
</tr>
<tr>
<td>DAO</td>
<td>Data Access Object</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>EB</td>
<td>Entity Bean</td>
</tr>
<tr>
<td>EJB</td>
<td>Entity Java Bean</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper Text Markup Language</td>
</tr>
<tr>
<td>JAR</td>
<td>Java ARchive</td>
</tr>
<tr>
<td>Javadoc</td>
<td>Java Documentation</td>
</tr>
<tr>
<td>JEP</td>
<td>Java math Expression Parser</td>
</tr>
<tr>
<td>JSP</td>
<td>Java Server Page</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Links</td>
</tr>
<tr>
<td>XML</td>
<td>eXtended Markup Language</td>
</tr>
</tbody>
</table>
# Appendix B: Symphony Iteration Summary

<table>
<thead>
<tr>
<th></th>
<th>Starting Viewpoint</th>
<th>Target Viewpoint</th>
<th>Source Viewpoint</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploration</td>
<td>XML mapping</td>
<td>Module allocated to file</td>
<td>Directory structure</td>
<td>directory to package-tag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Module allocated to library</td>
<td>File structure</td>
<td>extension to type tag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>File extension</td>
<td></td>
<td>filenames to tag content</td>
</tr>
<tr>
<td>First Iteration</td>
<td>Execution view</td>
<td>page links to page</td>
<td>Source code jsp files</td>
<td><code>&lt;a href&gt;</code> to link-relation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>&lt;form action..&gt;</code> to link relation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>&lt;jsp:forward..&gt;</code> to link relation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>&lt;jsp: include page..&gt;</code> to include entity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>&lt;jsp usebean&gt;</code> to use bean relation</td>
</tr>
<tr>
<td>Second Iteration</td>
<td>Decomposition view</td>
<td>Module acts as interface</td>
<td>Method calls method</td>
<td>Package to UML-package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Package to UML-package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Module uses interface</td>
<td>Class fwd/incl/etc. class</td>
<td>Class to UML-class</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Module uses module</td>
<td>Class contains method</td>
<td>Interface to UML interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Module contains module</td>
<td>Package contains class</td>
<td>Method to an entry in UML object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Module fwd/incl/etc. module</td>
<td></td>
<td>Uses relations to uses UML relations</td>
</tr>
<tr>
<td>Third Iteration</td>
<td>Sequence diagram</td>
<td>Module calls module</td>
<td>Method calls method</td>
<td>Method call to call relation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Method call to call relation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Module contains module</td>
<td>Class contains method</td>
<td>Class to class object</td>
</tr>
<tr>
<td>Fourth Iteration</td>
<td>Abstracted</td>
<td>Module calls module</td>
<td>Method calls method</td>
<td>Method call to call relation</td>
</tr>
<tr>
<td></td>
<td>Sequence Diagrams</td>
<td></td>
<td></td>
<td>Package to package object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Module contains module</td>
<td>Class contains method</td>
<td>Package to package object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Package contains method</td>
<td></td>
<td>Package to package object</td>
</tr>
</tbody>
</table>
Appendix C: Source Code

C1 Jar unpack script

```bash
#!/bin/sh
#Script to unpack all jar files from lib/ into the root
# of the sourcedir
# cp lib/*.jar .
FILES=`dir -1 *.jar`
# This for-loop will unpack all jar files. When finished the jar will be deleted
# for file in $FILES
# do
#   echo "jar -xvf $file"
#   'jar -xvf $file'
#   'rm $file'
# done
'rm -rf lib'
'./jad_script'
```

C2 Script to decompile class files and delete unnecessary files

```bash
#!/bin/sh
#Script to jad all class files recursively
# find . -name *.class > class_files
aantal=`cat class_files | wc -l`
i=1
# This while loop will visit all class files and decompile it in the same directory
while [ "$i" -le "$aantal" ]
do
  regels=`cat class_files|head -n$i|tail -n1`
directory=`echo $regels | sed "s/\/[a-zA-Z0-9.]*$/\//\"`
  `jad -o -s java -d $directory $regels`
i=`expr $i + 1`
done
rm class_files
CLASSES=`find *|grep .class`
# Script is ready to delete all class files because they are all unpacked
for class in $CLASSES
do
  `rm $class`
done
VERSIONCONTROL=`find * | grep vssver.scc`
# In every directory there is an unnecessary files called vsscer.scc this will be deleted
for versionc in $VERSIONCONTROL
do
  `rm $versionc`
done
```

C3 Script to create XML file

```bash
#!/bin/sh
#Script to make a XML structure of a directory ./ will not be shown)
#This is a recursive function that generates the XML
#$1 is the directory where the function should start
#$2 is the variable to outline the XML file
recursive_generator()
{
  local packagenaam=`echo $1 | sed "s/\//./g"`
  echo $packagenaam
  current_diepte=$2
  local htabs=""
  while [ "$current_diepte" -gt 0 ]
  do
    local htabs="$htabs\t"
    current_diepte=`expr $current_diepte - 1`
  done
  # When we are still in the root do not show files otherwise scripts will appear
  # in the XML file
```
if [ "$2" -gt 0 ]
then
    FILES=`find ./$1 -mindepth 1 -maxdepth 1 -type f -print | sed "s/([a-zA-Z0-9.]*\[/]*\[/]*)//g"
    echo -e "$htabs<Packagenaam>">>sourceview.xml
fi
# In this for-loop files will be mapped into the XML. In the case we can add file types.
for file in $FILES
do
    extensie=`echo $file | sed "s/[A-Za-z0-9_]*\[[A-Za-z0-9_]*\]/g"
    case $extensie in
        "jar") echo -e "$htabs<library>$file</library>">>sourceview.xml;;
        "java") echo -e "$htabs<class>$file</class>">>sourceview.xml;;
        "jsp") echo -e "$htabs<jsp>$file</jsp>">>sourceview.xml;;
        "gif") echo -e "$htabs<fo-image>$file</fo-image>">>sourceview.xml;;
        "jpg") echo -e "$htabs<fo-image>$file</fo-image>">>sourceview.xml;;
        "js") echo -e "$htabs<fo-script>$file</fo-script>">>sourceview.xml;;
        "mid") echo -e "$htabs<fo-easterregg>$file</fo-easterregg>">>sourceview.xml;;
        "htm") echo -e "$htabs<fo-htm>$file</fo-htm>">>sourceview.xml;;
        "inc") echo -e "$htabs<fo-inc>$file</fo-inc>">>sourceview.xml;;
        "xls") echo -e "$htabs<fo-xls>$file</fo-xls>">>sourceview.xml;;
        "css") echo -e "$htabs<fo-css>$file</fo-css>">>sourceview.xml;;
        "rtf") echo -e "$htabs<templates>$file</templates>">>sourceview.xml;;
        "vm") echo -e "$htabs<velocity>$file</velocity>">>sourceview.xml;;
        "jbx") echo -e "$htabs<jbuilder>$file</jbuilder>">>sourceview.xml;;
        *) echo -e "$htabs<descriptor>$file</descriptor>">>sourceview.xml
    esac
    done
AANTAL=`find ./$1 -mindepth 1 -maxdepth 1 -type d -print | wc -l`
# This is the stop in the recursion. When there are no directories the script will stop and give control back to the previous 'process' on the stack.
if [ "$AANTAL" -eq 0 ]
then
    echo -e "$htabs</Packagenaam>">>sourceview.xml
    return
else
    # Enter the recursive part. When there are directories untouched put them on the stack.
    for dir in $DIRS
do
        echo "/$dir"
        diepte=`expr $2 + 1`
        recursive_generator "$dir" $diepte
    done
fi
# All directories and files are processed we can include the end-tag
if [ "$2" -gt 0 ]
then
    echo -e "$htabs</Packagenaam>">>sourceview.xml
fi
return 0
}
Appendix D: Sequence diagram examples

D1 Assignment step executor overview
Appendix E: Pictures of calculation kernel
E1 View of old calculation kernel with Bauhaus
E2 Module view of old calculation kernel