Diploma Thesis

June 12, 2006

Reengineering of a Ticket Sales System

Alex Müller

of Schaffhausen, Switzerland (99-808-214)

supervised by

Prof. Dr. Harald Gall

Beat Fluri

University of Zurich

Department of Informatics

s.e.a.l.

software evolution & architecture lab

Diploma Thesis

Reengineering of a Ticket Sales System

Alex Müller

University of Zurich
Department of Informatics

Diploma Thesis

Author: Alex Müller, alex.mueller@access.unizh.ch

Project period: December 12, 2005 - June 12, 2006

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank all the people involved in the success of this thesis—they were always helpful and devoted their time to my issues. This includes the people at the Institute for Informatics at the University of Zurich, the company I did this thesis in conjunction with, and my family and friends. Special thanks go to Professor Harald Gall for mounting this practically oriented thesis, Beat Fluri for supporting me in the reengineering tool jungle and helping me out with miscellaneous problems during the thesis every day life, Helmut Rainel and Andreas Schabmann from the company for taking the risk of giving a valuable software asset into the hands of a student, and the senior developer Oliver Hönigsberger for taking the time to answer a large number of questions related to the software component under investigation. I am also grateful to my parents and my girlfriend which looked after me in terms of motivation and healthy food every once in a while. Last but not least, I am very pleased about the philosophy of open source software, which I used intensively during the thesis.

Abstract

The goal of this diploma thesis is to develop a reengineering plan for the front-end of a ticket sales system written in Java using reengineering techniques and tools. The front-end component is operational, but defect from a maintenance point of view. The analysis of the component and the change propositions are based on theoretical cognitions, practically established reengineering patterns, and supporting software tools. Elements of these domains are applied in the reengineering process to find a best practice solution for the given task. The thesis leads through the reverse and forward engineering process. It is organized in three parts. The first part gives an overview to the software reengineering domain and introduces the instruments used in the thesis. The second part covers the reverse engineering of the component. A stepwise organized and goal-oriented analysis process supported by object-oriented reverse engineering patterns and tools is applied. The result of the second part is the set of classes that need to be reengineered. The third part discusses the options based on the reverse engineering results and delivers a goal architecture and a plan describing how to change the ticket sales component to make it easier to maintain and to enable its further evolution.

Zusammenfassung

Das Ziel dieser Diplomarbeit ist das Entwerfen eines Reengineeringplans für ein in Java geschriebenes Front-End eines Ticketverkaufsystems. Das Front End ist in Gebrauch, aber defekt aus der Wartungsperspektive. Die Analyse der Komponente und die Änderungsvorschläge basieren auf theoretischen Erkenntnissen, praktisch bewährten Reengineering Mustern und hilfreichen Software Tools. Die Elemente werden während dem Reengineeringprozess angewendet um eine bestmögliche Lösung des gestellten Problems zu erarbeiten. Die Diplomarbeit führt durch den Reverse- und den Forwardengineeringprozess. Die Arbeit ist in drei Teile gegliedert. Der erste Teil gibt eine Übersicht zum Software Reengineering und stellt die im Verlaufe der Arbeit angewendeten Instrumente vor. Der zweite Teil dokumentiert den Analysevorgang der Komponente. Ein schrittweise organisierter und zielgerichteter Analyseprozess begleitet von objektorientierten Reverse Engineering Mustern und Software Tools wird durchlaufen. Das Resultat des zweiten Teils sind die identifizierten Klassen, die einem Reengineering unterzogen werden müssen. Der dritte Teil diskutiert Optionen aufgrund der Analyseresultate und gibt eine Zielarchitektur vor. Ein Plan beschreibt welche Veränderungen an der Komponente vorgenommen werden müssen um die Wartbarkeit zu vereinfachen und um eine weitere Evolution zu ermöglichen.

Contents

1 Introduction **1**
1.1 The Patient . . . 1
1.2 Goals . . . 1
1.3 A Brief History of Software Reengineering . . . 2
1.4 Object-Oriented Legacy Systems . . . 2
1.5 Structure of the Thesis . . . 3

2 Reengineering Methodology **5**
2.1 Reengineering Terminology . . . 5
2.2 Reengineering Patterns . . . 6
2.3 Patterns and the Reengineering Life Cycle . . . 8
2.4 Refactorings . . . 10
2.5 Software Reengineering Tools . . . 11
2.5.1 Creole for Code Visualization . . . 12
2.5.2 CodeCrawler for Metrics Visualizations . . . 12
2.5.3 Omondo UML for an Abstract View of the Classes . . . 14
2.5.4 Java Metrics to Collect Relevant Metrics of the System . . . 14
2.5.5 Eclipse for Automated Refactorings . . . 14

3 Reverse Engineering **17**
3.1 Setting Direction . . . 17
3.2 First Contact . . . 18
3.2.1 Interview during Demo . . . 19
3.2.2 Chat with the Maintainers . . . 21
3.2.3 Skim the Documentation . . . 23
3.2.4 Read All the Code in One Hour . . . 24
3.2.5 Do a Mock Installation . . . 25
3.2.6 Summary of the First Contact . . . 25
3.3 Initial Understanding . . . 25
3.3.1 Speculate about Design . . . 26
3.3.2 Study the Exceptional Entities . . . 32
3.3.3 Summary of Initial Understanding . . . 38
3.4 Detailed Reengineering Definition . . . 39
3.5 Detailed Model Capture . . . 41
3.5.1 Step through the Execution . . . 41
3.5.2 Look for the Contracts . . . 47
3.5.3 Detailed Model Capture Summary . . . 49
3.6 Reverse Engineering Results . . . 50

4 Reengineering Options **51**
4.1 Rebuild from Scratch? . . . 51
4.2 Alternatives to Java . . . 51
4.3 Graphical User Interfaces . . . 52
4.3.1 GUI Principles . . . 52
4.3.2 GUI Creation Frameworks . . . 53
4.4 Unit Tests . . . 54
4.5 Evaluation of the Proceeding . . . 55
4.5.1 SWT and JWS together? . . . 55
4.5.2 View or Controller first? . . . 55
4.5.3 Mapper or Direct Access to the View? . . . 56
4.6 Prototype the Target Solution . . . 57
4.6.1 Feasibility of JWS Type Substitutions . . . 57
4.6.2 Feasibility of the Mapper . . . 57
4.6.3 Feasibility of the Refactorings . . . 58

5 Reengineering Plan **59**
5.1 Overview . . . 59
5.2 Substitute Local JWS Elements in Controller . . . 59
5.3 Create JWS Mapper . . . 59
5.4 Build SWT GUI . . . 60
5.5 Controller Refactoring . . . 62
5.6 Documentation . . . 66

6 Conclusions and Outlook **67**
6.1 Contribution . . . 67
6.2 Experiences Made . . . 67
6.3 Outlook . . . 67

A Source Code Listings **69**
A.1 JWS Substitute Data Types . . . 69
A.2 Mapper Class . . . 69
A.3 Move Method Refactoring Example . . . 69

B List of Used Software Tools **75**

List of Figures

2.1 Graphical overview of the reverse engineering taxonomy. 7
2.2 The software reengineering life cycle. 9
2.3 Pattern clusters assigned to the reengineering life cycle. 10
2.4 Creole view of class interactions and intra class variable accesses. 13
2.5 The principle of a polymetric view (a) and a class hierarchy generated by CodeCrawler using polymetric views (b). 13
2.6 Example of metric results calculated by the Java Metrics plug-in. 15

3.1 GUI of the main working surface of the ticket sales component. 20
3.2 Component view of the event managing system. 22
3.3 Communication path of the ticket sales component with the database. 24
3.4 An UML class diagram showing inheritance relations. 26
3.5 Typical method call path for handling GUI user input events that trigger database access. 29
3.6 Hierarchy of classes in the `odp` package. 30
3.7 The class hierarchies of the group package. 31
3.8 Assignment of classes to model, view, or control layer. 32
3.9 Visualization of class hierarchies, interface hierarchies, and interface implementations in the ticket sales component. 48

5.1 The architecture of the reengineered ticket sales component. 60
5.2 The class hierarchy of the JWS mapper classes. 62
5.3 UML class diagram of classes that contain methods moved from the `form_clCache` class. 65

List of Tables

2.1 The format of a reengineering pattern. 8
2.2 A condensed example of a reengineering pattern. 8
2.3 Overview of refactoring categories. 11

3.1 Overview of the patterns used from the *setting directions* cluster. 18
3.2 *Setting direction* patterns stating generic rules to keep in mind during the reengineering project. 19
3.3 *First Contact* patterns used to get to know the system. 19
3.4 *Initial Understanding* patterns used to discover the details of the ticket sales component. 26
3.5 Overview of loose classes. 27
3.6 Abbreviation and full description of the metrics used on the package level. 33
3.7 List of metric values on package level. 33
3.8 Abbreviation, full description, and threshold values of the used class size metrics. 34
3.9 Classes with exceptional metric values. 35
3.10 Manual evaluation summary of classes identified to have exceptional class size metrics. 37
3.11 Abbreviation and full description used class inheritance metrics. 37
3.12 Significant metric values of class inheritance metrics. 38
3.13 Abbreviation, full description, and thresholds of the used method metrics. 38

3.14 List of methods having exceptional metrics. 39
3.15 The *load the hall map of a chosen event* feature split up into steps with the corresponding logic methods. 47
3.16 Interfaces, their implementing classes, their referencing classes, and their purpose. 49
3.17 Reengineering dealing for the loose classes. 50
3.18 Reengineering dealing for the ODP classes. 50

4.1 Line up of SWT and Swing properties. 54

5.1 Overview of the reengineering plan including milestones. 61
5.2 List of JWS types in the controller that need to be substituted. 61
5.3 Subjects representing new classes to split up the `form_clCache` class. 64
5.4 Methods that need to moved to an existing class or a substitute thereof. 65

List of Listings

3.1 A snippet of debug information obtained when loading the hall map for an event. 44
3.2 A snippet of the debug information with added print statements obtained for the `form_clCache` class when loading the hall map for an event. 45
3.3 Code of the method `select_pictureChoice` in the `form_clCache` class. 46
4.1 An extract of the `method(String sSubMethodTyp,Event evt)` code calling a method candidate for refactoring. 58
5.1 Use case of the ODP communication protocol. 64
A.1 Substitute type for the JWS type `Attribute`. 70
A.2 Substitute type for the JWS type `AttributeList`. 71
A.3 A mapper class containing methods to delegate the ODP method accesses to the view. 72
A.4 The newly created `StatistikController` class which holds the moved method `besuchsstatistik`. 73

Chapter 1

Introduction

> Programs, like people get old.
>
> David Lorge Parnas, 1994

Software reengineering is a relatively young research area which is not perceived as much as software engineering. The reality is that often an already existing program has to be quickly extended and the code added to it by different people leads to the decay of the system on all levels. The loan taken with this practice and the cumulated interests are paid back when the reengineering of the system has become inevitable. The reengineering work is painful and unpopular. In this thesis, we examine how the reengineering of an object-oriented legacy system can be tackled with existing tools and techniques. The work is done in collaboration with a company, which will use the results of this work to enable the successful future evolution of the ticket sales component. The contribution of this thesis is a reengineering experience report. This work is of use for the scientific software reengineering community looking for a case study as well as for software project managers and developers faced with object-oriented reengineering projects.

1.1 The Patient

The legacy ticket sales component to be reengineered is written in Java and operational, but broken from a maintenance point of view. Instances of the component connect to a server which handles the traffic with the database. Its maintenance effort to keep the current level of functionality offered and to make minor enhancements is too high. It is not possible to make larger extensions demanded by the customers.

1.2 Goals

This thesis in the form of a field report aims at applying reengineering software tools, an organizational framework, and best practice refactoring know-how to an object-oriented legacy system component to enhance the understanding of how those different aspects of software reengineering play together in practice. We apply reverse engineering techniques to understand the component and to identify the reasons that make the component hard to maintain and extend. The set of classes that need to be changed is presented and a reengineering plan including instructions, templates, and examples to renew the ticket sales component is delivered.

1.3 A Brief History of Software Reengineering

Early work on software reengineering—though not yet called like that—date back to 1966 when academic discussions about the GOTO statement in procedural programs where lead. The GOTO statement makes programs more difficult to understand. Parsing and mapping of programs onto a flow graph have been introduced. The graph was then restructured and converted back into a program. In the 1970s, influences of the GOTO discussion were considered in structural programming methodologies. The 1970s were also a germination period for reengineering ideas and technologies. The work done went under the banner of software maintenance and would only gain widespread attention in the 1980s. In that decade, commercial software restructuring tools had appeared and larger-scale improvement of legacy code had started, the calculation of various software metrics has also been added to the tools. The term reengineering has been introduced as a generic term to describe the different activities involved in program renovation. In the early 1990s, forward- and reengineering technology came closer together as forward engineering tool vendors integrated maintenance technology into their environments due to the risen interest in rather maintain code than write new one. Nowadays, with the still ongoing trend of getting more out of existing systems, the demand for reengineering technologies and services remains strong. There are different IDEs (Integrated Development Environment) available today that offer reengineering features such as refactoring. Still, software reengineering has not become easier, it requires a lot of knowledge and experience and there remains a lot of handwork and thinking to do [Arn94]. Today, software reengineering is a many faceted research domain. Current research activities in the software reengineering domain include:

- **Visualization techniques** of program artifacts and collected data about them. The goal is to develop tools that enable intuitive, information rich perspectives on a code base. In [PGFL05], a visualization approach is proposed considering the time dimension (*i.e.,* the history data of several releases of a program).
- **Architecture recovery** focuses on how to retrieve the architectural state of a software system on different levels of abstraction. For example in [RR02], a technique for combining static and dynamic architectural information is combined to support the task of architecture reconstruction.

Another interesting approach is to consider the features of a program as a unit to describe a software system. Tracking features aims at finding scattered code belonging together with the help of other artifacts [FPG03a]. Most recently, endeavors have been made to better understand a program by considering and combining historical information stored in bug tracking and version management systems [FPG03b]. Visualizations thereof give new perspectives on a program and enables to see where the examined system has weaknesses.

1.4 Object-Oriented Legacy Systems

The concept of object-oriented programming is a paradigm that has radically changed the way software is developed. It has entered the mainstream in the eighties when established programming languages such as C have been enhanced to support object-oriented programming in terms of statements (*e.g.,* `class`) and performance. Object-oriented programming is still static. A change in requirements implies that additional code to meet them has to be added at different locations in the code. In the late nineties, the paradigm of aspect-oriented programming[1] (AOP) has

[1] Aspect-oriented programming complements object-oriented programming by allowing to dynamically modify the static object-oriented model to create a system that can grow to meet new requirements. http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html (27.5.2006).

been formulated to mitigate the problem of scattering code due to the static nature of the object-oriented programs and their evolution. Scattered added code increases the complexity of the system which results in symptoms object-oriented legacy systems suffer from, such as a decayed system architecture or objects. Object-oriented legacies start to pop up in the various domains they have been applied in. Their reengineering demands special techniques due to the program structure imposed by the object-oriented paradigm. External drivers that fostered object-oriented legacy systems are:

- Object-oriented programming is not that young anymore: It has become popular in the 1980s with the introduction of window based GUIs.[2]
- In the beginnings of mainstream object-oriented programming, object-oriented languages were often abused to write programs in a procedural fashion.
- With the beginning of the public Internet era, distributed applications unexpectedly gained popularity, but the technologies to handle data across a network were not yet sophisticated enough for convenient use. Developers and organizations came up with their own solutions.

The consequence of using object-oriented programming techniques is that the architectural structure of a software system is radically different from traditional procedurally written software. Designing object-oriented programs from scratch is powerful and efficient due to the inherent reuse aspect of code in the object-oriented programming paradigm. The reengineering of object-oriented code is more complicated than in the procedural programming case. This is because multiple instances of a class can exist at runtime and the class hierarchy does not necessarily represent semantic groupings which makes it difficult to understand the program. Where the latter point is also a problem when a procedurally approached software system is reengineered, the former is specific to object-oriented programming. The powerful concept of polymorphism additionally hardens the understanding of object-oriented programs in order to reengineer them.

1.5 Structure of the Thesis

Chapter 2 introduces the methodology and tools chosen for the reengineering of the ticket sales component. Chapter 3 introduces the whole software system, the component to be reengineered, and documents the reverse engineering steps. The given code base is analyzed to gain an understanding of the component as a preparation to make the appropriate reengineering decisions. The result is an architectural view and the classes that are subject to reengineering. Chapter 4 evaluates and discusses the reengineering options based on the reverse engineering results, the available technologies, and the constraints imposed by the company. The feasibility of the target system is verified by exemplary prototypes. Chapter 5 gives a goal architecture and a reengineering plan describing the steps to achieve it. Instructions and templates are delivered. Chapter 6 summarizes the experiences made, gives an outlook of the future evolution of the component and discusses the lessons learned during the work.

[2]Smalltalk was the first object-oriented programming language with an integrated user interface using windows to display information. http://www.parc.xerox.com/about/history/default.html (5.4.2006).

Chapter 2

Reengineering Methodology

This chapter introduces reengineering terms, develops the organizational framework for the project, and introduces the tools chosen for the support of the reengineering activities.

2.1 Reengineering Terminology

In literature, the interpretation of the reengineering terminology does vary and sometimes the concepts behind the terms are not clearly distinguished from each other. The following set of definitions enables a common base of understanding [CC90].

> **Definition 1 (Forward Engineering).** *Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.*

This definition has been given to better understand the definition of reverse engineering:

> **Definition 2 (Reverse Engineering).** *Reverse engineering is the process of analyzing a subject system to:*
>
> - *Identify the components of the system and their interrelationships*
> - *Create representations of the system in another form or at a higher level of abstraction*

Two typical reverse engineering subtasks are redocumentation and design recovery:

> **Definition 3 (Redocumentation).** *Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level.*

> **Definition 4 (Design Recovery).** *In the design recovery process, domain knowledge, external information and reasoning are combined with the observations of the system to identify useful higher level abstractions.*

The result of redocumentation and design recovery together with other reverse engineering activities are useful architectural views of the system. Architectural views aim at enabling an understanding of the system and its problems in preparation for the changes to be made during the forward engineering.

> **Definition 5 (Reengineering).** *Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.*

While reverse engineering is essentially a process of examination, reengineering also refers to the process of changing the system. Reengineering is not the same as software maintenance, which is concerned with the modification of software after delivery to improve performance or other attributes [otICS98].

> **Definition 6 (Restructuring).** *Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the external behavior of the system in terms of functionality and semantics.*

Restructuring generally refers to source code translation (such as the automatic conversion from unstructured "spaghetti" code to structured, or "goto-less" code), but it may also entail transformations at the design level[DDN03]. Restructuring in an object-oriented context is called refactoring and has been defined by [FBB+99] similar to restructuring:

> **Definition 7 (Refactoring).** *Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.*

> **Definition 8 (Remodularization).** *Remodulariszation is the process of dividing a system or program into several smaller, more manageable modules and/or recombining those programs along functionally cohesive lines. This may include reconciliation of redundant logic and functional realignment of physical systems.*[1]

Remodularization consists of a set of refactorings done based on an analysis to identify code clusters depending on application contexts. The definitions given in this paragraph cover the most important aspects of software reengineering. There are a lot more definitions aiming at formally describe reengineering activities, but they are often not standardized and do not have a strong generic character and therefore cause redundancy between each other (*e.g.,* definitions for the same process in a procedural and an object-oriented context).

Figure 2.1 [CC90] shows to which artifacts the defined terms can be applied. The figure suggests, that the whole reengineering process is not strictly sequential, but rather a mixture of different kinds of advancements depending on the task given and the problems discovered in the code.

In this thesis, the term component is used to describe the set of Java classes to be investigated that constitute the ticket sales client (*i.e.,* the ticket sales component). Our use of the term component does not exactly follow the definition of the term software component [Szy98].

> **Definition 9 (Software Component).** *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

The defined and discussed reengineering related terms enable a concise usage of them in this thesis.

Note: Throughout the thesis, Java code, packages, and files are marked with the `typewriter` font. Signatures of methods are only written out when necessary for the understanding, otherwise they are left away for a better readability of the text.

2.2 Reengineering Patterns

The terms introduced in the previous section characterize the kinds of potentially applicable procedures to the artifacts of a software system in a general sense. Those procedures have to be divided into smaller parts and executed at the right point in time to achieve the desired sub-goals.

[1] http://www.comsysprojects.com/SystemTransformation/tmglossary.htm (4.4.2006).

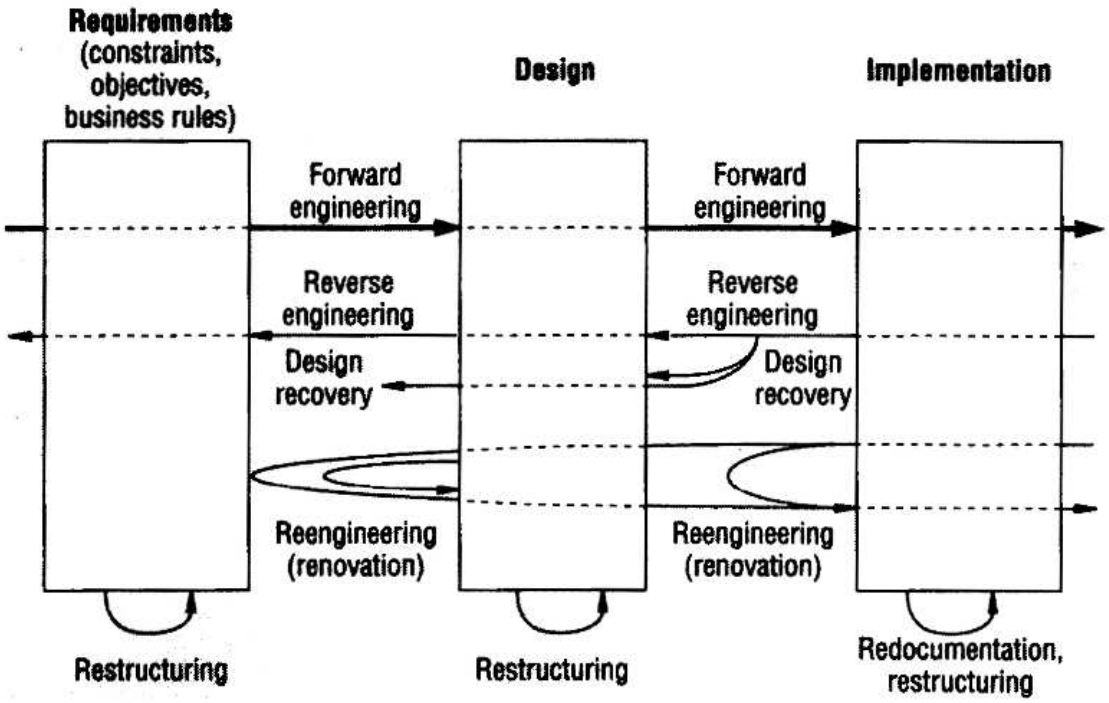

Figure 2.1: Graphical overview of the reverse engineering taxonomy.

This is difficult because each software system is different. As a consequence, no easily applicable and detailed standard procedure for the reengineering of software can be defined. Nevertheless, certain best practices have been established to handle a lot of the set of all possible situations in the reengineering process. This know-how is stored in reengineering patterns. [DDN03] defines reengineering patterns as follows:

> **Definition 10 (Reengineering Pattern).** *Reengineering patterns codify and record knowledge about modifying legacy software: They help in diagnosing problems and identifying weaknesses that may hinder further development of the system, and they aid in finding solutions that are more appropriate to the new requirements.*

Patterns are stable units of expertise that can be consulted in any reengineering effort. They describe a process without proposing a complete methodology and they suggest appropriate tools without "selling" a specific one. In the context of a whole reengineering project, a pattern can be seen as a step on the way to the reengineering goal. Some of the patterns state rules to follow that promise good results. Others imply questions that have to be answered relating to a reengineering project. A pattern typically entails a number of conflicting forces, and the solution described entails a number of trade-offs. Table 2.1 [DDN03] shows the basic structure of a pattern. The first column contains an aspect of the pattern that has to be specified in the second column. Table 2.2 gives an example of a pattern. Not necessarily all aspects of a pattern have to be considered equally or at all to successfully apply it. Demeyer et al. suggest patterns for all phases of a reengineering project. They describe aspects of patterns in detail, relate the patterns to others, and suggest which pattern(s) can be applied next. Patterns are a convenient knowledge repository, but do not provide a "silver bullet" for a reengineering project. They need to be applied depending on the situation during the reengineering project.

The patterns selected to apply in this reengineering project will be briefly introduced when they are used. The reengineering patterns will only be explained as detailed as necessary since the center of work in this thesis is the practical reengineering.

Name of the pattern	
Intent	
Problem	Mostly formulated as question, discussion of forces, key to the solution.
Solution	State the solution.
Trade-offs	Description of trade-offs.
Rationale	Why the solution makes sense
Known Uses	List of examples.
Related Patterns	Alternative actions.
What Next	Logical follow up action.

Table 2.1: The format of a reengineering pattern.

If it Aint't Broke, don't fix it	
Save the reengineering effort for parts of the system that will make a difference.	
Problem	Which part of the system should be reengineered?
Solution	Only fix the parts that are broken, those that can no longer be adapted to planned changes.
Trade-offs	Pros: No time is wasted for uncritical issues. Cons: Delaying non critical repairs may cost more in the long run.
Rationale	Ugly parts of the system that work can be wrapped and it may never be necessary to replace them.
What Next	Start with the most valuable asset first.

Table 2.2: A condensed example of a reengineering pattern.

Patterns are implementable because they are compact and embed proceedings in real world constraints (*i.e.,* forces and trade-offs). Still, time constraints, such as deadlines or the busy schedules of people involved detain the application of patterns. Time constraints tend to result in situations where several patterns are in use concurrently—the danger of loosing the focus arises. This is unavoidable and the reengineer needs to be aware of this problematic. Patterns are flexible, they can be adapted to the given problem. Each pattern is a mini framework for a situation during the reengineering project. For the patterns selected to use during this reengineering project, adapted forms are often applied.

2.3 Patterns and the Reengineering Life Cycle

Demeyer et al. use the software reengineering life cycle in [DDN03] to align the use of the patterns. Figure 2.2 shows the software reengineering life cycle. It is a generic view of what happens during the reengineering. The bended arrow symbolizes that reengineering is not a direct code conversion, but a process that involves different artifacts on different levels of abstraction. The images along the arrow symbolize those involved artifacts. Depending on the reengineering goal and the current state of the given artifacts, the reverse engineering phase can be interrupted by forward engineering phases, for example to get the software running again if it is functionally broken. Figure 2.3 [DDN03] shows sets of patterns along the previously discussed reengineering

life cycle arrow. The time axis indicates when during the reengineering a set of patterns is typically applied. It is important to adapt the use of patterns depending on the reengineering goals and the personal experience. A custom adaption can imply that certain reengineering phases have to be traversed iteratively, for example to augment the understanding for an important part of the system. The first four sets of patterns suggest what to do during the reverse engineering phase:

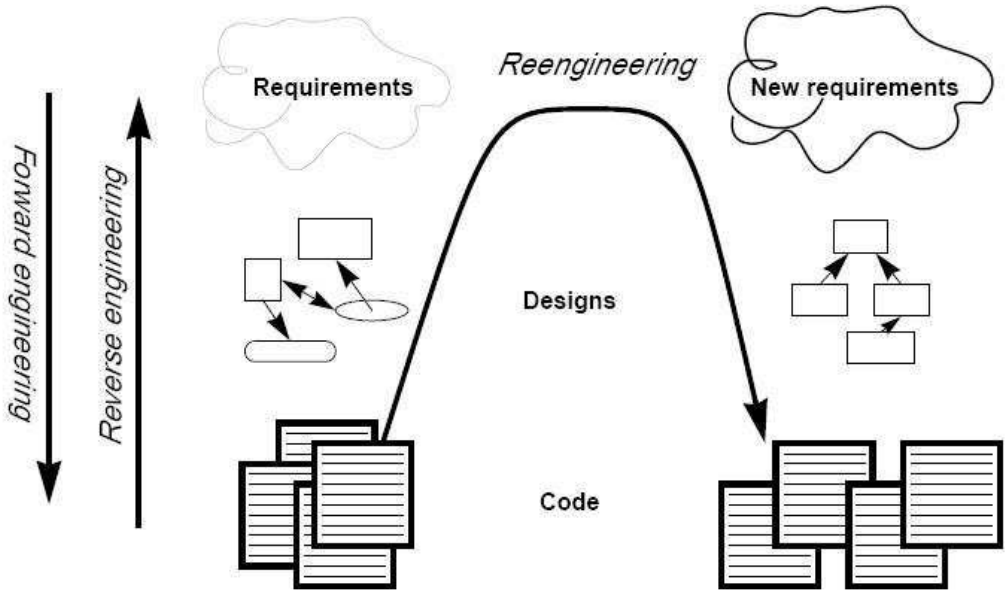

Figure 2.2: The software reengineering life cycle.

- **Setting Direction.** This set of patterns can be applied to any project. The patterns aim at setting the right path, suggest how to coordinate, and maintain the chosen path.
- **First Contact.** Wasting time is the largest risk when having the first contact with the system. The first contact patterns suggest how to gather basic information efficiently.
- **Initial Understanding.** When it comes to develop an initial understanding of the system, incorrect information is a big concern. The patterns in this set rely on source code, since this is the most trustworthy source of information.
- **Detailed Model Capture.** These patterns help to build up a detailed model of those parts of the system that will be relevant for the reengineering. The activities proposed help to expose design artifacts hidden in the code.

The remaining pattern sets group the forward engineering activities:

- **Tests: Your Life insurance!** Provides effective ways of using tests in a reengineering context to reduce the risks posed by reengineering changes.
- **Migration Strategies.** Deals with concerns such as user acceptance and migration strategies to incrementally move to the target system.

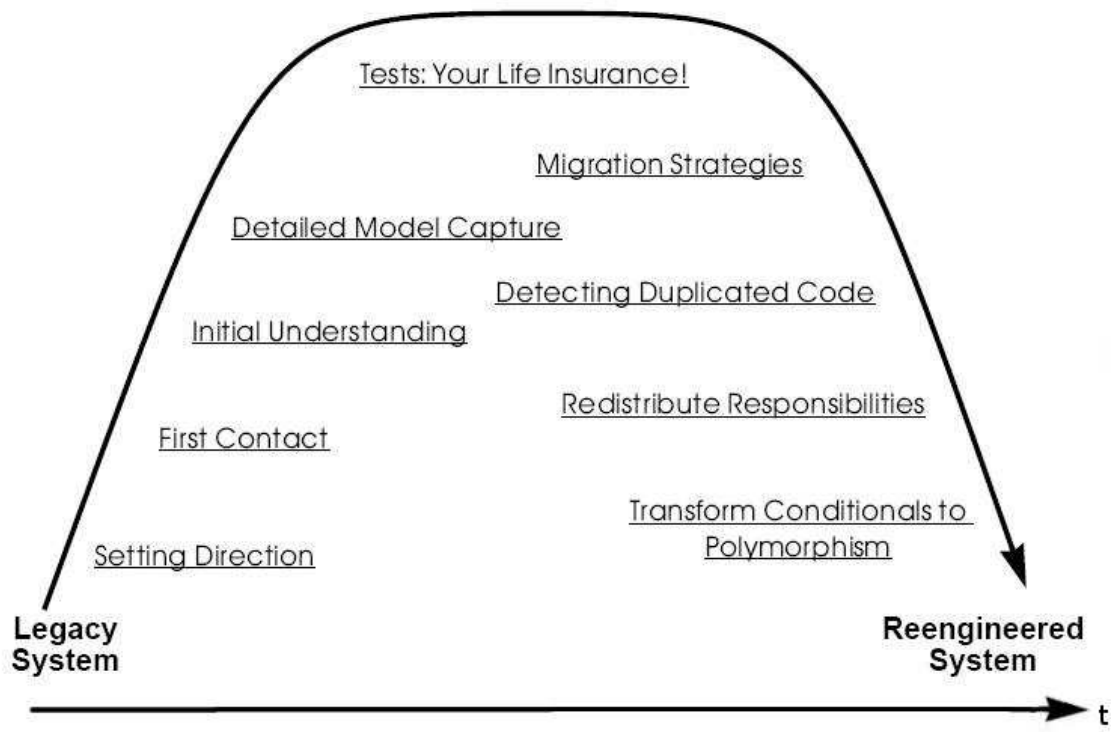

Figure 2.3: Pattern clusters assigned to the reengineering life cycle.

- **Detecting Duplicated Code.** The patterns aim at detecting duplicated code to reduce maintenance costs.
- **Redistribute Responsibilities.** Pure data containers imply navigation code and god classes can cause higher maintenance costs. The patterns suggested can help to mitigate this sort of problems.
- **Transform Conditionals to Polymorphism.** Large conditionals imply couplings between classes. The patterns describe how to eliminate them and thereby improving the flexibility in future changes.

The ordering of groups of patterns along the reengineering process visualized in the software reengineering life cycle form the organizational reengineering framework we are going to apply for the reengineering of the ticket sales component. It suggests what to do when and to a certain degree how.

2.4 Refactorings

Patterns that suggest to change the code describe the steps that need to be undertaken general manner. The refactorings deliver the details of how changes in the code have to made. Refactoring as defined in Section 2.1 has to be accomplished in a structured way to be successful. Fowler et

al. propose a disciplined way to change code that minimizes the chance of introducing bugs in [FBB+99]. It consists of small and simplistic changes to the code to eliminate a bad code smell or to stepwise add code to slowly converge to a target form of the program. The cumulative effect of these small changes improve the structure of the program. They propose a collection of refactorings for bad code smells and design problems a program can have. For each refactoring, a generic template is given. The 72 described refactorings can be grouped into the categories given in Table 2.3.

Category	Description	Example
Composing Methods	Pack long sections of code into a method or put code contained in a short method at its place.	Extract method
Moving features between objects	Refactorings to move responsibilities at different levels of granularity between objects	Move field
Organizing Data	The refactorings make working with data easier.	Replace record with data class
Simplifying Conditional Expressions	-	Replace conditional with polymorphism
Making Method Calls Simpler	Make methods that represent the interface to an object easier to understand	Rename method
Dealing with Generalization	Make code more generically usable	Pull up method
Big Refactorings	Often occurring cases that need several of the "small" refactorings.	Separate domain from presentation

Table 2.3: Overview of refactoring categories.

The analysis of the ticket sales component during reverse engineering will identify the refactorings we have to make to achieve the reengineering goals. It is essential that testing whether the code has been broken goes hand in hand with the refactorings. Automated refactorings (Section 2.5.5) change the testing policy, but do not substitute it.

2.5 Software Reengineering Tools

The organizational reengineering framework specifies a structure for what to do and when, the question how to do it is not considered in detail. The answer is the knowledge of techniques on how to investigate and change the system. Reengineering tools represent techniques to support the investigation and the change of the system.

Tackling with the source code, the hidden architecture of the system, and its problems remains difficult and requires a lot of expertise. Reverse engineering tools help to understand the code, derive architectural views and identify design problems by providing software metrics and visualizations. The tools offer useful features and ease the reverse engineering work, but they do not make decisions—neither strategic nor common sense ones.

The output of tool supported software analysis in the reverse engineering phase has two facets: Metric values and visualizations. Visualizations can be high level UML abstractions or two-dimensional drawings illustrating metric information with polymetric views (Section 2.5.2).

The optimal amount of tool generated information to support the understanding of the program depends on its complexity and the experience of the programmer. Intensive tool use is dangerous because the understanding of the system can become biased by the perspectives the tools offer. A good approach to minimize this risk is to wait as long as possible with the use of tools.

A considerable amount of reengineering tools with useful features are available to speed up the reverse engineering process. Some of them come as a set of features integrable in IDEs such as Eclipse[2], others are standalone programs. A lot of them are open source, of academic nature, and the features they offer implement yet young research results from the software reengineering domain. As a consequence, the quality of the tools varies a lot. Considering this, and the fact that tools do not solve the actual reengineering problems, they have to be chosen and used selectively. To work efficiently with the once selected tools, it is advisable to first think about what kind of information needs to be gained before starting to work with them. Some of the tools offer a lot of features, but not all of them provide additional relevant information. The IDE used for this project is Eclipse 3.1 which comes with built-in refactoring features. The tools considered suitable and chosen to support the given reengineering task are Creole[3], Omondo UML[4], Java Metrics[5], and CodeCrawler.[6] All except CodeCrawler are available as Eclipse plug-ins.

2.5.1 Creole for Code Visualization

Creole is an open source Eclipse plug-in that generates visualizations of a code base. It is able to display class hierarchies, views of source code sets, and their interactions at different granularities. For example, Figure 2.4 shows a set of classes. Their interrelation in terms of method calls and attribute accesses is expressed with the red arrows. Intra class relations such as methods accessing variables within the classes can also be seen. Creole can be used during the whole reverse engineering phase: To get a quick overview of a code set for the initial understanding of a software system, in the detailed model capture phase to verify hypotheses about architecture, and to detect bad smells such as too large classes or gravitation classes. Creole runs stable, but uses a lot of memory to generate visualizations.

2.5.2 CodeCrawler for Metrics Visualizations

CodeCrawler is a language independent open source software visualization tool [Lan03]. It is integrated into the Moose[7] reengineering environment which is a language independent environment for reverse engineering and reengineering software systems. Moose provides a set of services including a common meta model, metrics evaluation and visualization, a model repository, generic GUI support for querying, browsing and grouping [NDG05]. The visualizations CodeCrawler offers are less fancy compared to Creole, but contain more information. The visualizations of CodeCrawler are based on polymetric views. A polymetric view can display up to five metrics (Figure 2.5 a) [Lan04]). We used CodeCrawler in addition to Creole to detect exceptional entities Creole was not able display, such as the number of methods overridden. A lot of overridden methods in extended classes can indicate that the implemented class hierarchy does not correspond to a good architecture. CodeCrawler offers a lot of predefined visualizations. Figure 2.5 b) shows one of the class hierarchies of the classes to reengineer. The classes are represented as rectangles, their width is proportional to the number of attributes and their high to

[2]http://www.eclipse.org
[3]http://www.thechiselgroup.org
[4]http://www.omondo.com
[5]http://metrics.sourceforge.net
[6]http://www.iam.unibe.ch/∼scg/Research/CodeCrawler
[7]http://www.iam.unibe.ch/ scg/Research/Moose/

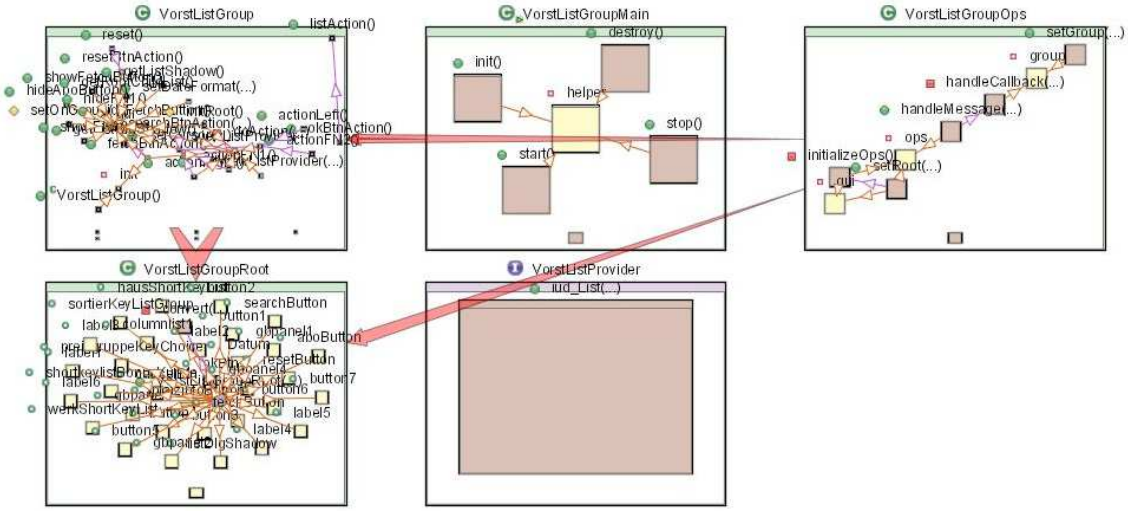

Figure 2.4: Creole view of class interactions and intra class variable accesses.

the number of methods they contain. The color represents the sum of lines of code of all methods the class contains. The darker the filling, the more lines of code. Such system complexity views allow to quickly identify potential design problems (*e.g.,* an unbalanced class hierarchy). The names of the classes are not contained in the rectangles. Class names are shown in a text bar in the Moose environment when the mouse pointer is positioned on an entity. CodeCrawler crashes when trying to execute certain operations and the overall program handling is less convenient than in Creole since the data for the common meta model first needs to be generated. The jFamix[8] Eclipse plug-in allows to export the data needed for visualizations in CodeCrawler. The data is exported into a XMI (XML Metadata Interchange) file that can be loaded into CodeCrawler.

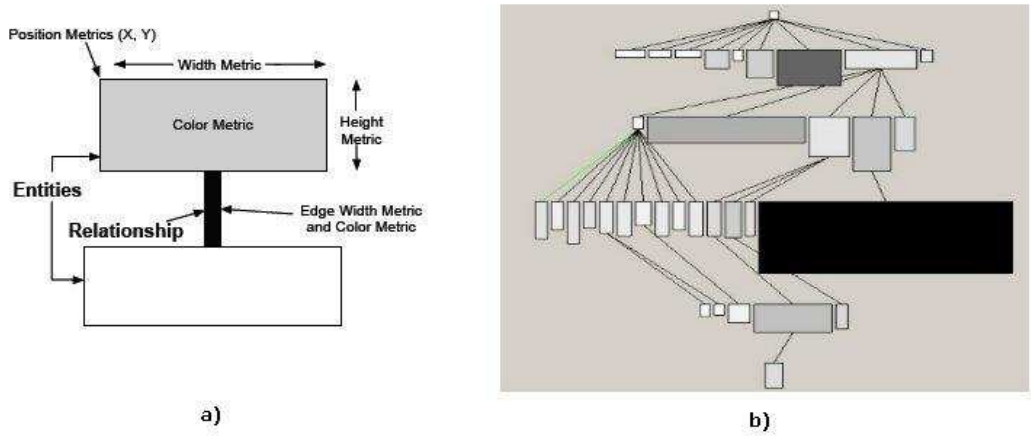

Figure 2.5: The principle of a polymetric view (a) and a class hierarchy generated by CodeCrawler using polymetric views (b).

[8]http://jfamix.sourceforge.net/eclipse/index.html

2.5.3 Omondo UML for an Abstract View of the Classes

Omondo UML is an Eclipse plug-in for UML modeling. The software is commercial, but the free edition comes with enough features for this reengineering project. Reverse engineering of class diagrams from source code is possible. Synchronization between the class diagram and the Java code is supported. UML class diagrams are used for redocumentation and to provide templates in the forward engineering phase. After the changes to the ticket sales component have been made, different types of diagrams will be generated for the technical documentation.

2.5.4 Java Metrics to Collect Relevant Metrics of the System

Java Metrics is an open source Eclipse plug-in that calculates various metrics of Java source code and includes a dependency analyzer. There are commercial tools[9] which can calculate a lot more metrics than Java Metrics. The metrics computed by Java Metrics are sufficient for this project since the goal is to understand the code base and to find design problems rather than to make a quality assurance of the code. The problem with metrics is that it is hard to say starting from which threshold on, or out of which range a metric value is bad. Defining an exceptional value for a metric depends on the type of program. For example, classes that build up a GUI inherently contain a lot of code to describe all the widget details. The rule of thumb when using metrics for software reengineering is to stick with the simple metrics because complex metrics and ratios of metrics tend to give few additional information [DDN03].

Figure 2.6 shows an example of metric results calculated by Java Metrics of all classes subject to reengineering. A line in the table shows the name of the metric, its aggregated value (total), mean, standard deviation, and maximum values as well as the resource causing the maximum and the method of that resource. A metric can be expanded to see how it is composed on the package level (this has been done in Figure 2.6 with the Total Lines of Code metric). The package level can further be expanded to the class level and the class level to the method level. The results displayed depend on the entity, the fields are left blank if a metric value is not applicable to an entity (in Figure 2.6 for example, there is no total value for the depth of the inheritance tree measured over all classes (*i.e.*, on the project level)). The plug-in does not support the creation of customized metric sets, that is the user can not choose which metrics to apply to which set of classes. The tool offers an Extensible Markup Language (XML) export feature. As a workaround to create customized metric sets, the exported raw XML data can be transformed with an Extensible Stylesheet Language (XSL) sheet to show customized results in HTML. The Java Metric tool is used to collect the overall metric values of the project and to detect exceptional entities.

2.5.5 Eclipse for Automated Refactorings

Programming languages in which refactoring is not part of the culture often have insufficient refactoring tool support. This implies that most of the refactoring needs to be done manually. It prevents programmers from making refactorings because it is seen as a separate task that costs too much [FBB+99]. The refactoring features in Eclipse integrate programming and refactoring and let refactoring appear as a less separate activity. Eclipse offers a set of automated refactorings for Java. The possible refactorings for a marked Java element can be accessed via the context menu. Automated refactorings eliminate the possibility of human errors, such as spelling mistakes. Yet, testing is advisable for technically complex automated refactorings because personal testing showed that they do not work seamless. Typically, not all references to a modified element are updated. For example, the automated *Push Down Method* refactoring does not always update

[9] For instance the Borland Together product series is able to compute a vast set of metrics for quality assurance (http://www.borland.com/us/products/together).

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Total Lines of Code	37832					
(default package)	37832					
(default package)	13866					
group	10445					
odp	4154					
com.culturall.printspooler	1715					
com.culturall.bankomatkasse.rmi	1618					
com.culturall.bankomatkasse.message.response	1175					
com.culturall.javax.comm.modem	991					
com.culturall.administration	973					
com.culturall.javax.pay	592					
com.culturall.magnetkartenleser	438					
com.culturall.bankomatkasse.message.request	379					
com.culturall.bankomatkasse.protocol	362					
com.culturall.terminal	262					
com.culturall.magnetkartenleser.modelle	205					
data	192					
com.culturall.util	171					
com.culturall.bankomatkasse.message	168					
com.culturall.bankomatkasse.debug	126					
Number of Classes (avg/max per packageFragment)	215	11.944	13.954	60	/Cult_Source/group	
Number of Methods (avg/max per type)	1834	8.53	13.135	157	/Cult_Source/form_clCache.java	
Number of Overridden Methods (avg/max per type)	156	0.726	1.489	9	/Cult_Source/form_cl.java	
Number of Children (avg/max per type)	77	0.358	1.268	11	/Cult_Source/com/culturall/bankomatkasse/message/Resp...	
Depth of Inheritance Tree (avg/max per type)		2.847	1.478	7	/Cult_Source/group/MapEditVJCanvasShadow.java	
Number of Parameters (avg/max per method)		0.958	1.536	22	/Cult_Source/KartendruckThread.java	KartendruckThread
Number of Static Methods (avg/max per type)	106	0.493	1.494	12	/Cult_Source/com/culturall/bankomatkasse/protocol/EPS48...	
Number of Attributes (avg/max per type)	1642	7.637	33.296	470	/Cult_Source/form_clRoot.java	
Number of Static Attributes (avg/max per type)	538	2.502	7.784	73	/Cult_Source/odp/OdpColor.java	
Nested Block Depth (avg/max per method)		1.549	1.206	10	/Cult_Source/group/MapEditVJCanvas.java	update
Afferent Coupling (avg/max per packageFragment)		7.222	10.512	38	/Cult_Source/odp	
Normalized Distance (avg/max per packageFragment)		0.312	0.272	1	/Cult_Source/com/culturall/util	
Specialization Index (avg/max per type)		0.513	1.266	5	/Cult_Source/form_clMain.java	
Instability (avg/max per packageFragment)		0.62	0.339	1	/Cult_Source	
Number of Packages	18					
Method Lines of Code (avg/max per method)	29368	15.138	102.112	4086	/Cult_Source/form_clRoot.java	form_clRoot
Weighted methods per Class (avg/max per type)	7173	33.363	111.234	1504	/Cult_Source/form_clCache.java	
Lack of Cohesion of Methods (avg/max per type)		0.315	0.364	1.125	/Cult_Source/com/culturall/bankomatkasse/rmi/BK_Status.j...	
McCabe Cyclomatic Complexity (avg/max per method)		3.697	10.286	223	/Cult_Source/com/culturall/printspooler/CardPrintConverte...	Befehl
Abstractness (avg/max per packageFragment)		0.183	0.189	0.714	/Cult_Source/com/culturall/bankomatkasse/message	
Number of Interfaces (avg/max per packageFragment)	42	2.333	4.243	19	/Cult_Source/group	
Efferent Coupling (avg/max per packageFragment)		8.056	16.437	70	/Cult_Source/group	

Figure 2.6: Example of metric results calculated by the Java Metrics plug-in.

all the references to the moved method. We will use the Eclipse refactoring features whenever possible in conjunction with a testing framework. Automated refactorings improve the productivity. The empirically observed effect on the testing policy is that the frequency of tests run remains the same, but more program changes are verified [FBB+99].

Chapter 3

Reverse Engineering

This chapter documents the core reverse engineering work that has been carried out to gain an understanding of the component to the necessary level of detail for the forward engineering phase. After basic analysis has been terminated, the reengineering priorities will be stated and further goal specific analysis work will be carried out.

3.1 Setting Direction

The goal of the reengineering process is a plan including a set of propositions, a number of templates and examples that suggest what changes have to be made in the ticket sales component to enable its further successful evolution. Before beginning with the analysis, the reengineering goals that caused the company to start a reengineering project can already be stated:

- The classes of the component need to be refactored to reduce future maintenance and expansion efforts of the component.
- Adapting the component to be compliant with the current 1.5 JDK.
- Introduction of a new GUI, since the current Java WorkShop[1] (JWS) based GUI is broken from a maintenance point of view and does not support the widgets expected from a modern GUI.

With this general goal description at hand, a goal directed reverse engineering process from the beginning is practicable (*i.e.,* issues to look for when analyzing the artifacts). The hardest part in a software reengineering project is the reverse engineering. Program understanding done the classical way, that is stepping through the program execution with typical use cases at hand and following the execution path consumes a lot of time. The object-oriented programming paradigm with the conceptual gap between the static program code and the dynamic runtime behavior hardens this work additionally. Given a large code base and limited amount of time, the focus at the beginning has to be to quickly verify the feasibility of the reengineering project and then find out enough about the software to be able to reengineer it according to the already given reengineering goals and the ones that will be stated during the reverse engineering process as more deficiencies of the system might be discovered. The early stage allocation of time slots to reengineering subtasks is difficult because the information about the software and its context is incomplete.

[1] Java WorkShop was an IDE offered by Sun Microsystems. It was shipped with Abstract Windowing Toolkit (AWT) based classes aiming at easily creating GUIs.

The starting line of this reengineering project is a folder of about 24 Mb of data including Java source code, class files, batch files, Oracle Forms, a few C files, graphic files, and JWS generated project files. What to do first? In which direction has the reengineering effort go at the beginning? There are several roads one can start to walk down, but quite a few of them will turn out to be painful deviations or even lead to a point of failure. For example, people involved in the project can be seduced by technical problems that they are most interested in, instead of focusing on what is the most useful for the reengineering project [DDN03]. The *Setting Directions* patterns proposed below allow to get a smooth launch and keep important issues in mind from the beginning. The problems they refer to are largely of non technical nature and seem simple. Still, not to consider them is negligent. The patterns aim at setting the right path and suggest how to coordinate and maintain the chosen path. The patterns listed in Table 3.1 demand a project specific answer while those stated in Table 3.2 give facts to keep in mind for the whole project. The patterns are condensed to a problem-solution scheme.

Pattern Name	Problem	Solution
Agree on maxims	Establish a common sense of purpose with the people involved.	Establish key priorities of the project.
Speak to the round table	How to keep the involved people synchronized?	Regular communication with the people involved.
Most valuable fist	Which problems to focus on first?	Start with the aspects most valuable to the customer.
If it ain't broke, don't fix it	Which parts should be reengineered ?	Only fix the broken parts, those that can no longer be adapted to planned changes.

Table 3.1: Overview of the patterns used from the *setting directions* cluster.

The answers specific to the ticket sales component found trough discussions with the responsible developer considering the problems stated in Table 3.1 are listed below:

- **Agree on maxims.** The key priorities are to refactor existing logic classes, to make the component JDK 1.5 compliant, and to propose the implementation of a new GUI.
- **Speak to the round table.** Emails to the people involved will regularly be sent to report progress and get feedback.
- **Most valuable first.** Work focus is on the GUI component and the classes handling the precious and complex business logic.
- **If it ain't broke, don't fix it.** The ODP communication fulfills its purpose and is therefore not a reengineering priority.

3.2 First Contact

After having clarified the direction in which the project has to go, time has come for a closer look at the ticket sales component. The *First Contact* patterns suggest to chat with the maintainers and an interview during a demonstration of the software by a user. To validate and expand what has been heard and seen the *Read All the Code in One Hour*, *Skim the Documentation*, and *Do a Mock Installation* patterns from Table 3.3 can be applied.

Pattern Name	Problem	Solution
Fix problems, not symptoms	How to tackle all the reported problems ?	The source of the problem needs to be addressed, rather than stakeholder specific interests.
Keep it simple	How much flexibility should be built into the new system ?	An adequate, but simple solution should be preferred to a more general, complex one.

Table 3.2: *Setting direction* patterns stating generic rules to keep in mind during the reengineering project.

Pattern Name	Problem	Solution
Chat with the maintainers	How to get a good perspective on the historical and political context of the legacy system?	Discuss the problems with the maintainers.
Interview during demo	How to get an idea of typical use cases and the main features of the system?	Observe the system in operation by seeing a demo and interviewing the person who is demonstrating.
Read all the code in one hour	How to get a first impression of the quality of the source code?	A short, but intense time to study the code.
Skim the documentation	How to identify the parts of the documentation that might be of help?	Preparation of a list summarizing the important aspects for the reengineering project. Then, matching of that list against the documentation and meanwhile make an assessment of how up to date the documentation is.
Do a mock installation	How to be sure to be able to rebuild the system?	Build and install the system in a clean environment during a limited amount of time.

Table 3.3: *First Contact* patterns used to get to know the system.

The difficulty in this phase of the project is being faced with a software that is not understood at all and being forced to make substantial decisions on how to proceed in detail [DDN03]. One also has to be aware that the first impression can mislead. To minimize that risk, the hypotheses about the source code always have to be verified. The following sections summarize what we learned about the component applying the patterns in Table 3.3 while interviewing users working with the ticket sales client, chatting with the developer who is at the same time maintainer of the component and our own first look at the system and its components.

3.2.1 Interview during Demo

The goal of interviewing is to get an idea of typical use cases, the main features of the software system, and to learn about problems the users may experience when using the software. An interview was conducted with a person responsible for the deployment of the whole event management software at client sites. The ticket sales component demonstration aimed at getting familiar

with the main features. Figure 3.1 shows the main working surface of the ticket sales component. A hall map of an event is loaded where the user can select seats for reservation or sale. The following technical facts were be extracted from the interview:

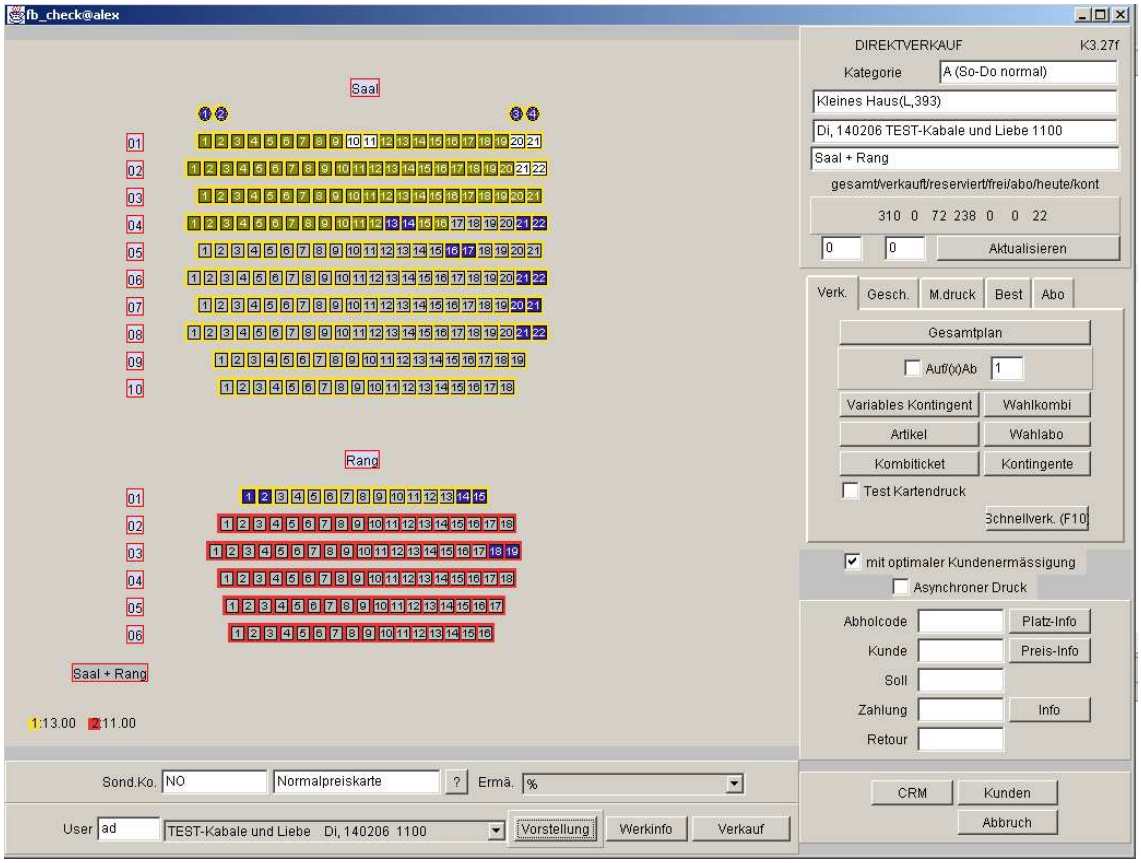

Figure 3.1: GUI of the main working surface of the ticket sales component.

- The ticket sales component uses sockets for communication with the an application server.
- The version management software used is Source Safe
- The tickets are printed out with a bar code for identification and cancellation.

The identified features are:

- Lockup events and show the state of the available seats
- Selling or reservation of seats for events
- Change or add new customer information to the customer base
- Choice between cash, billing or electronic payment is possible
- When the hall map of an event is very large, it is possible to only load segments of it. This is useful to relieve the database and to have a better overview

- Batch printing of tickets at the end of the day in the call center
- Management of different types of subscriptions
- Selling of merchandise articles
- Ticket selling of events without seats or unnumbered seats
- Automatic calculation of discounts

The features listed represent the main functionality. More functionality is available due to the complicated sales policies common in the cultural event sector. Additionally, short interviews were conducted with users of the ticket sales component. The users seemed comfortable with the system. They used it efficiently and knew workarounds for minor problems they faced during operation. The most common use cases are over the counter ticket sales at the point of sale and ticket reservations and selling in a call center which is implemented considering a customer base to identify the customer. The standard procedure to buy a ticket at a sales point is as follows:

1. Search an event with constraints given by the customer, for example searching by date or title of the event
2. Click on free seat(s) in the displayed hall map or enter the number of tickets to be sold in case of an event having non numerated seats or no seats at all (*e.g.*, a museum visit).
3. Click on the sales button to proceed to the payment method. The user has the possibility to enter a new customer or to search an already existing one in the database to associate the transaction with a customer.
4. After successful payment, the amount of tickets bought will be printed out on a ticket printer.

The information collected in the interviews while the ticket sales component was running gave a rough functional view. Next, the *Chat with the Maintainers* pattern can be applied to learn more about the underlying technical mechanisms of the described functionality.

3.2.2 Chat with the Maintainers

The discussions we had with the maintainer about the whole cultural event management system and the ticket sales component as a part of it are documented in the next two sections.

A Software System to Manage Cultural Events

The software system for managing cultural events consists of six components. The system offers functionalities to register an event such as a theater piece in a database, ticket selling thereof, and reporting about the stored data, for example how often an event was frequented during a certain time period. The database stores all the information necessary relating to an event and a customer base. Figure 3.2 gives an architectural overview of the system. The components of the event management system are centered around a database. They use different forms of communication for historical reasons. The database is also accessed by other IT systems, for example accounting applications.

- The **hall map editor client** is an Java client to add new plans of event locations to the database via the application server.

- The **administration tool** is an Oracle Forms[2] application that offers report generating functionalities for the back office.
- The **application server** is a Java application that interconnects the ticket sales and hall map editor clients to the database.
- The **ticket sales client** is a Java application to manage the over the counter and call center ticket sales processes. The offered functionalities are ticket reservations, ticket exchange, ticket selling and customer base management.
- The **Web ticket sales component** is a Java applet to integrate ticket reservation and selling into a Web page.
- The **Web server** is a Java application that interconnects the Web ticket sales component to the database.

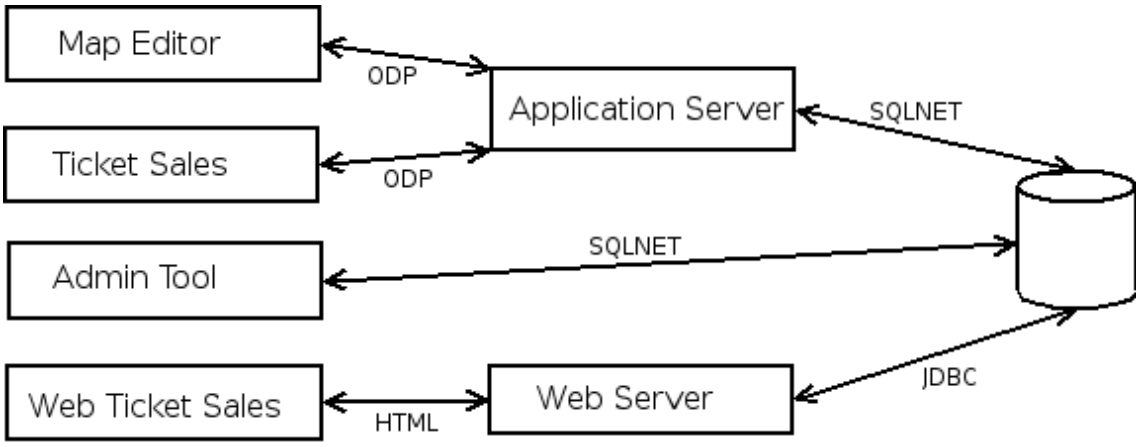

Figure 3.2: Component view of the event managing system.

The hall map editor and the ticket sales client use an ODP[3] based protocol for communication with the application server. Communication between the application server and the database as well as between the administration tool and the database is done with SQLNet.[4] The Web ticket sales component communicates via HTML with the Web server and the Web server uses JDBC[5] to interact with the database.

The four client components can be run independently from each other. Several instantiations of one application can run concurrently. The software has been gradually developed and introduced starting from 1992 on at the Bundestheater in Vienna. Today the whole system is operational at different customers in the cultural domain. Customers are typically ticket sales orga-

[2]Oracle Forms is a rapid application development environment for developing database applications. The language used inside is PL/SQL. PL/SQL (Procedural Language/Structured Query Language) is a proprietary server-based procedural extension to the SQL database language from Oracle. http://en.wikipedia.org/wiki/Oracle_Forms (3.2.2006).

[3]Open Distributed Processing (ODP) is a standard proposed in 1992 by ISO with the aim to establish a common reference model to develop distributed applications in open, heterogeneous system environments. http://www.rm-odp.net/index.html#standards (6.4.2006).

[4]SQLNet is the client/server middleware product from Oracle that offers transparent connection from client tools to the database, or from one database to another. http://www.orafaq.com/faqnet.htm#WHAT (6.4.2006).

[5]Java Database Connectivity (JDBC) is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. JDBC is oriented toward relational databases. http://en.wikipedia.org/wiki/JDBC (5.4.2006).

nizations acting on behalf of theaters, opera houses, and museums. The ticket sales client is the component to be reengineered.

The Ticket Sales Component

The ticket sales program was originally implemented with Oracle Forms in 1993. When the requirement to implement dynamic hall maps arose, the Oracle Forms environment came to its limits. In a dynamic hall map, attributes of seats such as price category or marking for disabled persons can change for each event the hall map is assigned to. Whereas in a static hall map, the once entered seat attributes configuration was not changeable anymore after it had been defined. Dynamic hall maps were the driver for a new implementation of the ticket sales client. In 1996, the company decided to develop a new GUI for the business processes needed at sales points. The GUI was implemented in Java using the JWS WYSIWYG editor. To handle server communication, an ODP based protocol using sockets was developed in Java by the company. This has been done because Java RMI (Remote Method Invocation) was just about to come out, not yet well known, and the company already had considerable knowledge about ODP. The purpose of the ticket sales component classes is either GUI related or ODP communication related. The work invested for the first productive release is about two man-years.

Since its introduction, the component has undergone a lot of changes and the JWS classes are no longer supported by Sun Microsystems. Changes to the GUI classes cannot be made via WYSIWYG editor anymore due to memory problems. Today, the component still runs with the originally used JDK 1.1.8 because changes made by Sun Microsystems in the AWT classes of later JDK versions where incompatible with the JWS classes. The business logic has grown large in an uncontrolled manner with the numerous changes that have been made over the years and is hard to maintain.

The ticket sales component class files are delivered to customers together with the hall map editor class files and the application server class files in a ZIP-file called `inhouse.zip`. Figure 3.3 shows that the client and the server side use different files from the same ZIP-file. All files having `''form_cl''` and `''map_cl''` in the beginning of their names are deployed on the client side and all files starting with `''data_sv''` are deployed on the server side. Files implementing the ODP protocol are used on client and server side.The server then redirects the traffic to a database stub using JNI.[6] The database stub contains configurations files used by the SQLNet middleware to connect to the database.

A user manual for the ticket sales component exists, but is outdated. Component testing is done with statements embedded in methods generating runtime data and writing it on the console and in log files. The statements can be activated via a parameters given at startup. The ticket sales component maintainer was also the main developer of the component. When the requirement for ticket sales via the Internet came up, a separate component was created, which also shows dynamic hall maps in an Java applet the customer can choose a seat from, buy it electronically, and print it out at home. Authenticity of Web-tickets is assured with a bar code.

3.2.3 Skim the Documentation

The purpose of this pattern is to assess the relevance of the available ticket sales component documentation. The technical documentation is limited to a couple of descriptions of how to set up the ticket sales component at client sites. A detailed 84 pages user manual describing the features of the ticket sales component exists, but is outdated. Recently added features are not mentioned

[6]The Java Native Interface (JNI) is a programming framework that allows Java code running in the Java virtual machine to call and be called by native applications and libraries written in other languages, such as C, C++, and assembly. http://en.wikipedia.org/wiki/JNI (5.4.2006).

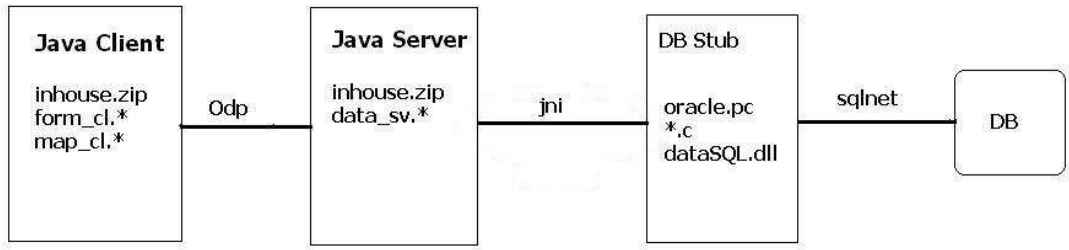

Figure 3.3: Communication path of the ticket sales component with the database.

and the GUI layout looks different today as shown in the manual. Although, browsing the manual was useful for a better understanding of a number of the features available in the ticket sales system. The existing documentation is not a key to technically understand the ticket sales component.

3.2.4 Read All the Code in One Hour

The intent of this pattern is to get a first impression of the quality of the source code. The pattern name suggests to only browse the code without studying details. In preparation for that, the source code and the necessary JAR (Java Archive) files had been loaded into Eclipse. Our goals were to find out which classes are used by the ticket sales component (since classes for the server, the hall map editor and the ticket sales component came together in one zipped package), to understand the purpose of the packages, try to identify potentially problematic classes, and to look for comments in the classes describing their functionality.

The set of classes having their name beginning with `form_cl` is responsible for the instantiation of the whole GUI panel. The three packages identified to be part of the ticket sales component are used by most of the classes of the `form_cl` set.

- The `group` package contains about 70 classes responsible for the creation of sub-panels in the main GUI for the ticket sales component. A part of the group-package classes were automatically generated by the JWS environment.
- The `odp` package contains about 30 classes that handle the communication between the ticket sales component and the application server. Most of the classes in this package are members of a hierarchy. The root class of this hierarchy is `odp`. It contains the abstract methods `send(DataOutputStream stream)` and `recv(DataInputStream stream)` which both return an integer value and are responsible for the transfer of data between the client(s) and the server.
- The three Java classes in the `data` are either needed for ticket printing or only used on the server side.

The class `form_clCache` is about 9000 lines long. It handles the GUI controll logic and calls the database with ODP methods. A part of the classes are not commented at all, others are commented in detail. Additionally, we created a Javadoc to get a feeling of the overall commenting used in all the classes. The `jws.zip` archive contains the JWS GUI class files. The `classes.zip` file contains the JRE 1.1.8 delivered with the ticket sales component ZIP-file. we also took a brief

look at the Bugzilla[7] software management system used by the company. The Bug activity related to the ticket sales component was about equally weighted between feature and bug requests. `form_clCache)` is the hot spot in the ticket sales component.

3.2.5 Do a Mock Installation

Doing a successful mock installation ensures that all the code necessary for the reengineering is available and gives the possibility to test the features with a non productive instance of the software. In preparation for that, the person responsible of the database resort in the company already set up a test database on a mobile hard drive and the server part on our workstation. Classes drawn from SourceSafe[8] have been added to the Eclipse Project until all class dependencies were resolved. as a last step, the correct program startup parameters and compile options had to be passed over to the compiler. we took the outdated user manual and tested the described features to further get acquainted to the ticket sales component.

3.2.6 Summary of the First Contact

The ticket sales component has an old fashioned GUI offering a lot of functionality to the user. Communication with the server is done with an inhouse-made class hierarchy implementing the ODP protocol. Large sections of code are not documented at all. The user manual is outdated. The system is operational, but hard to maintain and enhance. A mock install was successful indicating that the necessary code base is complete. Bugzilla shows an equal weight between bug reporting and feature requests. The GUI widget classes are automatically generated by the JWS WYSIWYG editor and a number of the GUI controller classes have grown to god classes. The focus when applying as a next step the *Initial understanding* pattern set will be on the understanding of the classes containing a large number of code. The so far acquired knowledge about the ticket sales component is not sufficient enough yet to refine the reengineering goals already stated at the beginning of Section 3.1. Nevertheless, the reengineering project is feasible because no significant artifacts are missing and the people who where involved in the development of the ticket sales module are still with the company.

3.3 Initial Understanding

The *First Contact* patterns allowed to get familiar with the general set-up of the system as well as a rough identification of its structure and of problem zones in the code. The more detailed design of the ticket sales component is not yet understood and it is unknown how big the effort is going to be acquire an understanding as base for reengineering decisions. The *Initial understanding* pattern set allows to refine the discoveries made in the *First Contact* pattern set. Table 3.4 summarizes the two patterns applied to the component. The *Speculate about Design* pattern represents a top-down analysis approach whereas the *Study the Exceptional Entities* pattern incorporates a bottom-up approach.

[7] http://www.bugzilla.org

[8] Microsoft Visual SourceSafe is a source control software package oriented toward smaller software development projects. http://en.wikipedia.org/wiki/Sourcesafe (7.4.2006).

Pattern Name	Problem	Solution
Speculate about Design	How to recover the way design concepts are represented in the source code?	Appliance of development expertise to conceive a hypothetical class diagram representing the design.
Study the Exceptional Entities	How to quickly identify potential design problems in large software systems?	Measurements of structural entities forming the system and identification of exceptions in the collected quantitative data.

Table 3.4: *Initial Understanding* patterns used to discover the details of the ticket sales component.

3.3.1 Speculate about Design

Manually drawing an initial hypothetical class diagram and to refine it through browsing the code of a component consisting of approximately 200 classes with an insufficient understanding of the system even to formulate hypotheses is useless. Instead, we decided to generate UML diagrams of potentially interesting sets of classes of the component. A small example of a class diagram is given in Figure 3.4. This allowed to quickly see the number of methods and attributes classes have as well as how classes are related to each other. We then formulated the questions listed beneath. The goal of this modified pattern is therefore to find answers to those design related questions. The knowledge gained in answering those questions will allow to create a useful initial architectural view in which classes are grouped depending on their purpose.

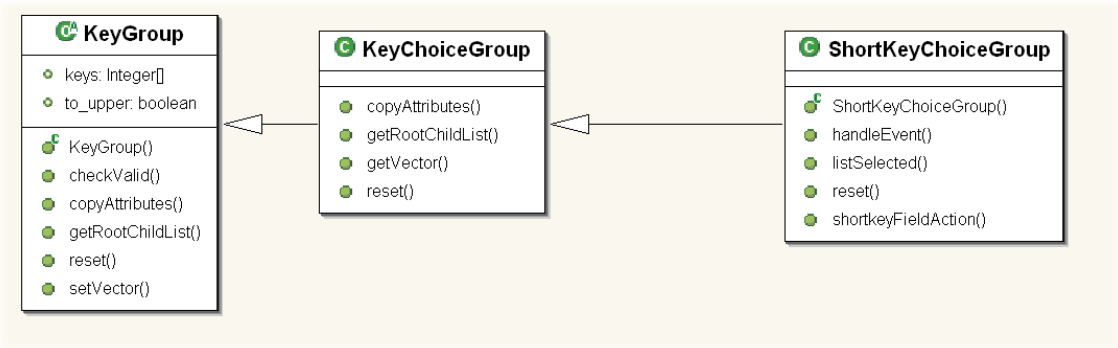

Figure 3.4: An UML class diagram showing inheritance relations.

- Which classes are used by the ticket sales component?
- How is the JWS event handling mechanism embedded in the code?
- How is the hierarchical structure of the classes?
- Can all the classes be grouped in sets depending on their purpose?

Ticket Sales Component Classes

The editor, server, and ticket sales classes are not kept separate from each other. There is little JWS class documentation available. As a first step, to create a better overview of the project, classes

have been iteratively removed from the Eclipse project because their name suggested that they do not belong to the ticket sales component. We recompiled the ticket sales component to verify that the removed classes are not needed. If there are more classes that are not part of the ticket sales component, they will definitively be identified later when stepping through the execution to relate features to code. The classes listed in Table 3.5 are the only ones left that are not in a package. We call them loose classes from now on. A closer look allowed to identify their purpose. The "Type" column in Table 3.5 refers to whether the class has been automatically generated by the JWS WYSIWYG editor, partially automatically generated as a template (to wire user input actions with handwritten classes or to overwrite them to have specific actions implemented) to be filled up by the programmer, or are handwritten.

Name	Type	Purpose
`form_clMain`	automatically generated	Contains the main method. Instantiates `form_cl`.
`form_cl`	template	Instantiates `form_clRoot` and `form_clCache`. "Relay" class taking events from `form_clOps` and delegating them to `form_clCache`.
`form_clCache`	handwritten	Communication god class between the GUI and the application server.
`form_clRoot`	automatically generated	All the GUI widgets are defined in this class.
`form_clOps`	automatically generated	The GUI elements that need to react to user input are registered in a vector. A case statement delegates the GUI events.
`KartendruckCache`	handwritten	Needed by `KartendruckThread`.
`KartendruckThread`	handwritten	A thread class to print tickets, instantiated in `form_clCache`.

Table 3.5: Overview of loose classes.

The classes named with `''form_cl''` are the center of the component: They start the component, are in charge of assembling the GUI, handle client side logic, and communicate with the server by instantiating ODP classes. The three packages (`data`, `group` and `odp`) are corded to be deployed by the `''form_cl''` classes.

JWS Knowledge

We decided to learn more about the JWS API[9]. A series of questions concerned with the JWS terminology and the JWS class model showed up when we looked at classes generated with Java Workshop. The JWS Knowledge stated here helps to better understand the code, the design decisions that were forced by using JWS, and to verify the stated propositions of the purpose of classes investigated in the last section. The JWS GUI builder introduces two new terms: *group* and *shadow*. Shadows are wrapper classes for an AWT components. They hide the event model and AWT classes behind an extra layer of code. They propagate AWT events to groups. A group is a container to bundle a set of GUI elements. It can contain shadows and other groups. Groups

[9]Useful JWS documentation was not findable on the website of Sun Microsystems anymore. http://www.dimi.uniud.it/labs/ documentazione/jws2.0-help/jws/visual/generate.html (15.3.2006) gives an overview to JWS.

that are included inside other groups are called subgroups. The JWS GUI builder is used to create new groups out of existing shadows and groups. New groups must be subclasses of the JWS `Group` class. The top-level group is called the *base group*. A tree of groups exists while the application is running. Messages sent by subgroups propagate up toward the top of the tree until they are handled or they reach the base group.

A group is accessed via a set of attributes and methods that the Group class defines. Subclasses of the Group class add their own customized attributes and methods. Applications created using the JWS GUI builder are groups. A large application typically consists of a base group that contains a lot of other shadows and subgroups. Each group represents a project in JWS. The JWS project manager allows to integrate projects, that is GUI components are added to a main GUI. A Java WorkShop project can contain only one group, but projects can contain other projects.

When the code of a group is generated with the Java WorkShop application based on the designed WYSIWYG layout, the following files result in the source directory of the project:

- `example.gui` contains the code that describes the GUI for Java Workshop based error checking. This file is not important since Java Workshop will no longer be used.
- `exampleRoot.java` contains the code necessary to initialize the components for the group. This file is recreated every time a GUI is built.
- `exampleOps.java` contains the operations for the group. This file is generated only if operations have been defined on one or more of the components in the corresponding group. This file is recreated every time a GUI is built.
- `exampleMain.java` contains the main method that is used to run the group as an applet or a standalone application. This file is recreated every time a GUI is built.
- `example.java` contains a template for the group. This file is generated only once. This file can be modified to implement event handlers, action handlers, and other functionality.

The JWS imposed class structure clarifies the purpose of most of the `form_cl`-named classes discussed in Table 3.5 and of the GUI classes in the group package. The group package further contains interface classes which will be examined with the *Look for the Contracts* pattern in Section 3.5.2. The identification of sets of classes that represent GUI components is usable as an architectural view of the GUI part of the ticket sales component. This structure can help to compose the GUI renewing task into steps in the forward engineering phase.

JWS Event Handling

The used JWS (version 2.0) API is based on Java SDK 1.0 AWT classes. "The model for event processing in version 1.0 of the AWT is based on inheritance. In order for a program to catch and process GUI events, it must subclass GUI components and override either `action()` or `handleEvent()` methods. Returning `true` from one of these methods consumes the event so it is not processed further; otherwise the event is propagated sequentially up the GUI hierarchy until either it is consumed or the root of the hierarchy is reached."[10] Since SDK 1.1, the event handling model has become delegation-based (*i.e.,* an event is propagated from an originating object to a "Listener" object by invoking a method on the listener and passing an instance of the event.).

The `form_clOps` class contains a registration vector for GUI elements. When an event from the GUI arrives, the overridden method `handleMessage(Message msg)` is automatically notified. This class has been taken as a starting point to validate how user input is handled in the

[10] Java AWT: Delegation Event Model, http://java.sun.com/j2se/1.3/docs/guide/awt/designspec/events.html (19.4.2006).

ticket sales component (*i.e.,* which methods are called and how the information is prepared for the ODP communication). This allows to verify the assumed SDK 1.0 AWT based JWS event model and to see which classes are involved in often occurring user operations. Following the method calls has been done statically since the JDK 1.1.8 needed to run JWS is not compliant with the Eclipse debugger front-end. As a remedy to this, the feature Eclipse offers to show all the references of a highlighted method name in the workspace has been used. The result of statically following the execution string is shown in Figure 3.5.

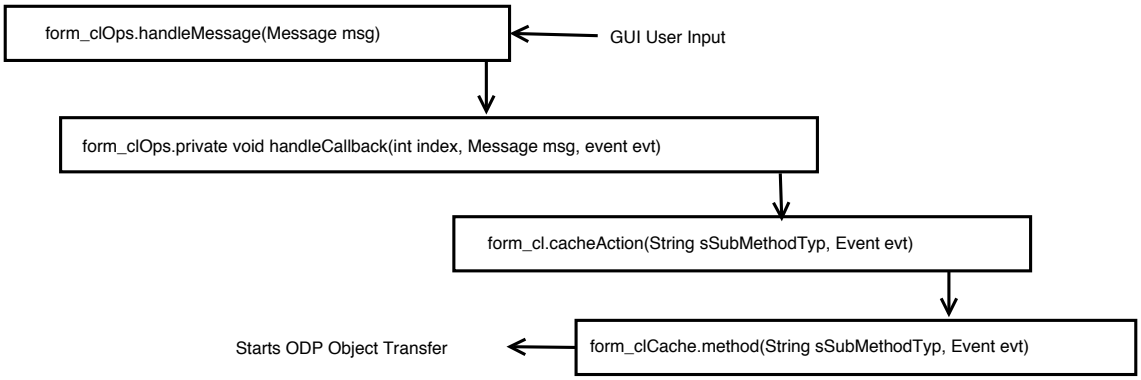

Figure 3.5: Typical method call path for handling GUI user input events that trigger database access.

In the `form_clOps` class, `handleMessage(Message msg)` calls `handleCallback (int index, Message msg, Event evt)` where `Message` is a public JWS class derived from Object that represents an event structure that can hold either an AWT event or a specialized event passed between instances of type `Group`. The method `handleCallback(int index,Message msg, Event evt)` contains a case statement that offers a choice of between 108 options based on the vector `index` integer determined in `handleMessage( Message msg)`. Each case calls the method `cacheAction( String sSubMethodTyp,Event evt)` in the `form_cl` class with a different string. `cacheAction( String sSubMethodTyp,Event evt)` then calls `method( sSubMethodTyp, evt)` in the `form_clCache` class. The `form_cl` class is used for navigation toward the `form_clCache` class that controls the GUI and communicates with the server. The `method( String sSubMethodTyp,Event evt)` method in `form_clCache` contains about 1000 lines of conditional code to decide which methods—mostly in the same class—have to be called based on the event. The `form_cl` class actually foreseen as a controller class, but the events from the GUI are delegated to be handled in the `form_clCache` which is an ODP class and has capabilities for communication with the database.

The Hierarchical Structure of the Classes

Visualizations of class hierarchies are useful for the understanding of a program and to spot design problems. Caution is advised when class hierarchies are used for design recovery because a design can be implemented in various ways. The hierarchical structure of the `odp` package is shown in Figure 3.6. The classes in the subtree on the left represent class types that can be used for communication with the server. The subtree having as the root the `OdpUnit` class represent communication types used by the classes responsible for loading and drawing hall maps. `MethodOdp` is a superclass of the `form_clCache` logic god class and implements client-server communication methods. `OdpArray` holds basic ODP types for data transfer. `OdpColor` holds class constants defining the colors used in hall maps. 26 of total 31 classes in the `odp` package are contained in the hierarchy. One of the remaining five classes is an interface to the class `SQL` in the `data` package.

The other four are concerned with threading or ODP object management. The root class `odp` contains two abstract methods `public abstract int send( DataOutputStream stream )` and *public abstract int recv(DataInputStream stream)* that each non-abstract subclass needs to implement.

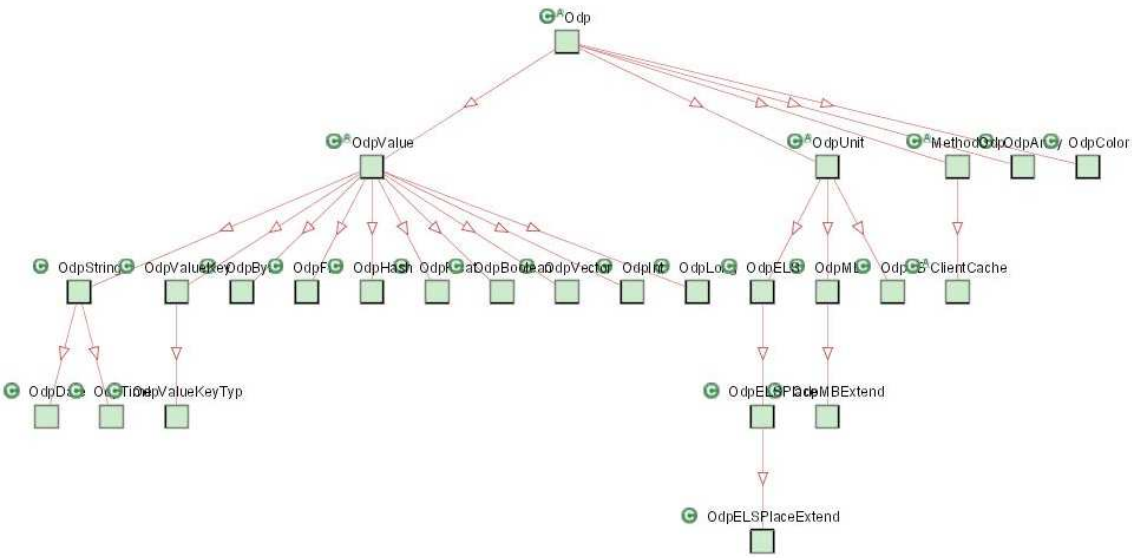

Figure 3.6: Hierarchy of classes in the `odp` package.

Figure 3.7 shows the classes that are members of hierarchical orderings in the `group` package. A majority of the classes in the `group` package obey to the JWS driven design pattern introduced in Section 3.3.1. The classes (of type `example.java`, which is a subclass of `Group`) in the hierarchy are ordered to represent specializations of groups, the classes representing leave nodes of the tree imply the generation of an `exampleRoot` and an `exampleMain` class. A class of type `exampleOps` is not always generated. Instead, events are directly caught in the `Group` subclasses. The classes of type `Group` in the `group` package not shown in Figure 3.7 are direct subclasses of JWS classes. Creole does not show them since it is only able create visualizations of source code classes and not class files.

MVC Perspective

The model-view-controller (MVC) design pattern is a must contract in software systems including complex GUIs. Its interpretation depends on the level of granularity (*i.e.,* at the widget level or component level). We consider the component level. We want to verify whether all the classes can be assigned to either the model, the view, or the controller. If this is possible, the work in the forward engineering can be done per layer. If not, the separation of concerns also has to be considered.

View classes are automatically generated subclasses of type `Root`. The controller classes are automatically generated subclasses of the class `Operations`. The controller listens to events in the view classes and handles them by calling corresponding methods in the subclasses of the `Group` class or in the `form_clCache` class. The custom classes derived from the `Group` class are as well controller classes. They access `Root` classes of their group and call the `form_clCache` class to invoke methods that globally affect the GUI (*e.g.,* to access the database). The custom

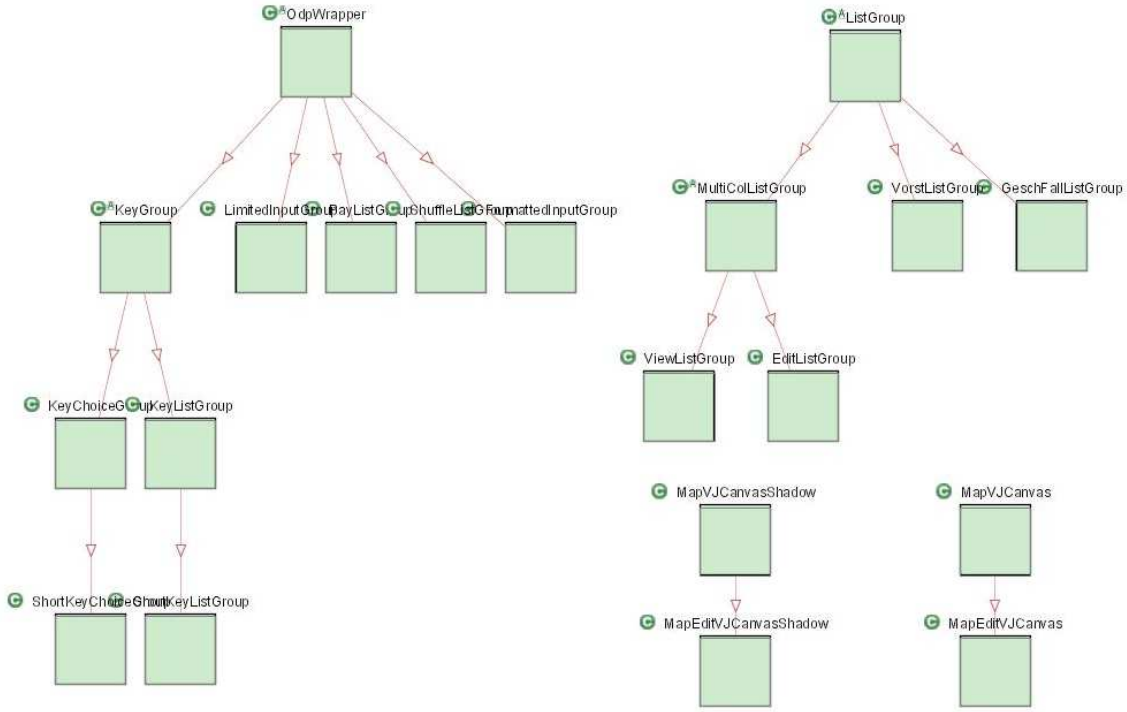

Figure 3.7: The class hierarchies of the group package.

`Group` classes also contain business logic related to the GUI component they represent, for example the validation if GUI input data corresponds to a given format). The `ClientCache` and the `form_clCache` classes contain global variables of the application model, communication logic, and access the view. They contain controller and model characteristics. The custom groups in `form_clRoot` use their own `Operations` classes to handle events of the custom group. If the case of local events, they call methods of their group and in case of global events, they access `form_clCache` via interfaces. The classes the JWS editor creates follow the MVC pattern, except for the `ClientCache` and `form_clCache` classes. Figure 3.8 gives a top level visualization of the considerations made.

Note: `form_clCache` contains communication logic, business logic, and GUI controll logic. If only *logic* is written, we refer to all three types of logic in the controller (which includes `form_clCache`. Each JWS group has a controller class which only contains business and GUI controll logic and no communication logic because the JWS group controller class do not have communication abilities. If they have to access the database due to an event handled locally in the group, they use interface types which reference `form_clCache`.

Combining the identified JWS driven architectural structure, the understanding of the event handling mechanism, and the MVC concept, it is now possible to clearly assign the majority of the classes to one of the three layers. This rough grouping is a base for a reengineering plan. View classes can be amputated and substituted with new GUI classes, the global controller classes can be refactored, code of the local controller classes (*i.e.*, of subtype `Group` can be partially reused, and the ODP classes that represent communication types can be left aside.

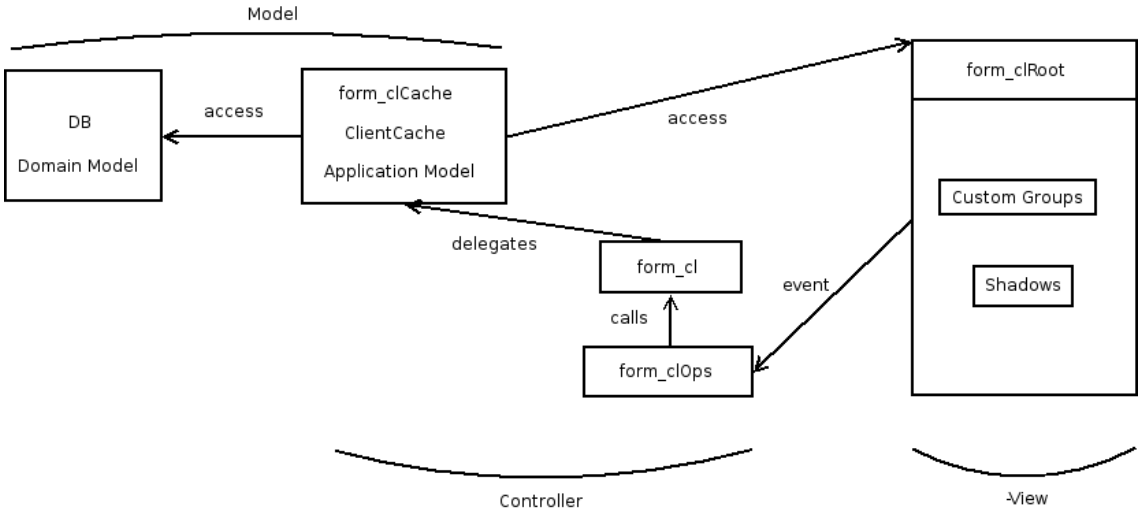

Figure 3.8: Assignment of classes to model, view, or control layer.

3.3.2 Study the Exceptional Entities

The *Speculate about Design* pattern approached the code top-down with questions that had to be answered to gain insight to the architecture of the component. Studying the exceptional entities approaches the code bottom-up examining it from a metric perspective. This allows to refine the architectural view and to spot potential design problems indicated by abnormal metrics. There is always a human assessment necessary to verify whether the metric value truly indicates a design problem. The difficulty with using metrics is to define when a metric value is exceptional. This depends on the type of program and from the notion the examiner has of program design practices. The available metric tools offer a great variety of metrics. Many of the "high-end metrics" do not bring much additional information gain for our purposes. We will therefore perform the analysis with a set of simple metrics. Metrics can also deliver useful information for the quality assurance of a software system. They can be used to rate the code changes during the reengineering and afterward during regular maintenance cycles. The Java Metrics tool (Section 2.5.4) is used to generate metric values from the code and CodeCrawler (Section 2.5.2) to visualize them. The metrics used to analyze the beneath listed structural entities are explained as they are used. The metric acronyms vary in the literature. The acronyms in this thesis have been chosen to best represent their meaning in the context of the entity considered. The threshold values of the metrics have been chosen that they are exceptional in the context of the whole project. This produces good results because the component has a lot of classes with normal metric values and a few with extremely exceptional values.

- Metrics on **package** level give an overall impression of the program part under investigation.
- **Class** size metrics deliver information of how it looks inside classes.
- Metrics relating to the **class inheritance** highlight anomalies in the class hierarchy which are often coupled with architectural deficiencies.
- **Method** metrics allow to spot bad method smells such as too much control.

Package Level Metrics

The whole project has 17 packages and a total of 36,841 lines of code. Table 3.6 shows the used package level metrics. Table 3.7 lists the lines of code and the number of classes a package contains.

Metric Acronym	Full Name	Comment
LOCP	Lines of code per package	Without blank lines and comments
NOC	Number of classes	-

Table 3.6: Abbreviation and full description of the metrics used on the package level.

Package	LOCP	NOC
loose classes	13866	9
`group`	10445	60
`data`	192	3
`odp`	4154	31
`com.culturall.bankomatkasse.rmi`	1618	25
`com.culturall.bankomatkasse.message.response`	1175	12
`com.culturall.bankomatkasse.message.request`	379	9
`com.culturall.bankomatkasse.protocol`	362	7
`com.culturall.bankomatkasse.message`	168	5
`com.culturall.bankomatkasse.debug`	126	3
`com.culturall.javax.pay`	592	5
`com.culturall.terminal`	262	4
`com.culturall.magnetkartenleser.modelle`	205	3
`com.culturall.magnetkartenleser`	438	2
`com.culturall.util`	171	3
`com.culturall.printspooler`	1715	16
`com.culturall.administration`	973	7

Table 3.7: List of metric values on package level.

The packages mentioned alongside the already known ones (`group`, `data`, `odp`, and the loose classes) in Table 3.7 are needed to resolve import dependencies imposed by the classes in the packages given to reengineer. Most of them have been added to the ticket sales component to keep up with the time enabling electronic payment functionality. The metric results for those classes are all normal and browsing through the code convinced us that they are well structured and documented sufficiently. They are therefore not considered further for the reengineering and it is supposed that they will be imported if necessary into the code to be restructured. They will only be reconsidered if a major design deficiency relating to them would show up when applying the *Detailed Model Capture* patterns. Table 3.7 shows that there are nine loose classes with 13'866 lines—few classes with a great number of lines of code. The metrics for the `data` and `odp` packages are normal. A look inside the `group`, `data`, and `odp` packages shows that no class has more than 1000 lines of code (LOC) which we consider as an alert value.

Class Size Metrics

The whole project consists of 204 classes. Only the ones violating one of the constraints defined for the chosen metrics in Table 3.8 are be shown. The bold values in Table 3.9 indicate a metric constraint violation. The classes in Table 3.9 have to be manually evaluated to assess whether the exceptional metric value(s) truly indicate a design problem. The listing in Table 3.10 shows the purpose and an assessing commentary of the manually investigated classes.

Acronym	Full Name	Threshold	Comment
LOCC	Lines of code per class	500	Lines of code without blank lines and comments. Automatically generated GUI classes can have a lot more LOC and do not pose a design problem (if managed with an editor).
NOM	Number of methods	30	Big values indicate too much functionality. Static and abstract methods are included.
NORM	Number of overridden methods	5	Methods in GUI API frameworks are often overridden to implement event handling or customizing GUI elements. Abstract inherited methods are not counted.[11]
NOF	Number of fields[12]	50	Does not include static fields. Indicates a data container smell.
NSF	Number of static fields	50	indicates where commonly used information is located.

Table 3.8: Abbreviation, full description, and threshold values of the used class size metrics.

[11] The ticket sales component has 213 abstract inherited methods. A lot of them are due to the abstractness of JWS classes.

[12] A field is also called attribute.

[13] Abstract inherited methods and methods invoking super are not counted. Overwriting of the methods `toString`, `hashCode`, and `equals` are not counted.

Class	LOCC	NOM	NORM[13]	NOF	NSF
loose classes					
`form_clCache`	**7067**	**160**	2	**116**	8
`form_clRoot`	**4594**	2	0	**470**	0
`form_clOps`	**1479**	6	1	3	0
`form_cl`	321	19	**9**	7	1
`group` classes					
`MapEditVJCanvas`	**700**	21	4	31	5
`MultiColListGroup`	**694**	**34**	2	12	0
`EditListGroup`	**642**	**34**	4	7	0
`KeyListGroup`	480	**35**	4	11	3
`VorstListGroup`	248	**30**	1	33	0
`ShortKeyListGroup`	240	22	**9**	3	0
odp classes					
`ClientCache`	**898**	**41**	0	**66**	**60**
`ObjectOdp`	**530**	**30**	1	16	12
`MethodOdp`	191	**30**	0	12	2
`OdpELSPlace`	224	13	**6**	12	22
`OdpColor`	282	8	0	1	**73**

Table 3.9: Classes with exceptional metric values.

Class	Purpose	Assessment
loose classes		
`form_clCache`	Communication class between GUI and application server	Contains a lot of logic and has to be split up. The center of the refactoring work!
`form_clRoot`	GUI creation	Contains widgets that need to be considered in the reengineering to conserve familiarity. Will be replaced by the new GUI.
`form_clOps`	Stores event sensitive GUI widgets in a vector and delegates the events to the `form_cl` class.	It will be important for the development of a refactoring strategy since it indicates the relations between operations and GUI elements.
`form_cl`	The top-level group. Delegates events to `form_clCache`.	Contains a set of default parameters that need to be considered in the reengineering. Will be replaced by the new GUI.

continued on next page

continued from previous page

Class	Purpose	Assessment
`group` classes		
`MapEditVJCanvas`	Responsible for the display of elements and caption of user inputs in a hall map.	Has to be understood in detail for the reengineering.
`MultiColListGroup`	Abstract base class for `EditListGroup` and `ViewListGroup`. The two classes enable the selection of database entries via a search field.	Well structured and documented. A part of the code can be reused in the corresponding element of the new GUI.
`EditListGroup`	Enables the selection and modification of database entries via a search field.	Derived from `MultiColListGroup`, they need to be considered together in the reengineering.
`KeyListGroup`	To select a database entry. Consists of a text field, a fetch button, and list that stores the returned database entries	The code is well structured and documented. The class is used by many other classes and needs to be considered in the reengineering.
`VorstListGroup`	This class represents and manages a list that holds information for cultural events	Derived from `ListGroup`. Needs to considered for the reengineering together with the other list related classes.
`ShortKeyListGroup`	Extends `KeyListGroup` with a text field that allows to enter shortcuts.	see `KeyListGroup`
`odp` classes		
`ClientCache`	Communication class between the GUI ant the application server. Fills the GUI with information retrieved from the database.	The contained communication methods are not considered for the reengineering. Contains global variables that represent the application model. They need to be considered in the forward engineering.
`ObjectOdp`	Administration class for the ODP-methods.	Not affected by the forward engineering.
`MethodOdp`	Transfer of ODP objects, allows to execute operations on distant computers.	Not affected by the forward engineering.
`OdpELSPlace`	Element buttons for seats.	Has to be considered in the forward engineering, GUI and communication and logic have to be separated.

continued on next page

continued from previous page

Class	Purpose	Assessment
`OdpColor`	Defines colors for custom painting of the canvas	Contains 73 static attributes that define colors. They are accessed by methods that paint elements of the GUI. This class is a data container.

Table 3.10: Manual evaluation summary of classes identified to have exceptional class size metrics.

The metric values for `form_clCache` further underline the god class smell. The `form_clRoot` class is automatically generated and contains the GUI widgets. The constraint violations are therefore reasonable. The `group` classes create and controll elements of the main GUI, handle events, and invoke ODP communication with the application server. The `ClientCache` class in the `odp` package is responsible for handling the communication mainly coming from `form_clCache` with the application server. It has similar functionality for the main GUI component than the classes containing `''group''` in their name in the `group` package. It is not surprising to see that the `odp` classes tend to have a lot of methods since they are responsible for communication (*e.g.,* load in a vector for communication).

Class Inheritance Metrics

The class size metrics allowed to spot several classes with unusual metrics in the ticket sales class set. Keeping those selected classes in mind, it is interesting to see where they are located in the class hierarchy. The class inheritance metrics in combination with visualizations generated by CodeCrawler of the inheritance hierarchies allow to learn about the hierarchical context they are situated in. Table 3.11 shows the metrics chosen to use. Table 3.12 shows the collected results. The `OdpValue` is the base class for all ODP data types. The group classes with high DIT values are specialized custom classes. Class inheritance metrics are useful to identify exceptional parts in a class hierarchy, but they are not informative to obtain an understanding of the system because a system architecture can be implemented in various ways via a class hierarchy.

Metric Acronym	Full Name	Threshold	Comment
NSC	Number of subclasses	5	direct subclasses are counted.
DIT	Depth of inheritance tree	6	path length from root to referred class.

Table 3.11: Abbreviation and full description used class inheritance metrics.

Method Metrics

The metrics chosen to investigate the code on the method level are explained in Table 3.13.

The values in the "Method" column in Table 3.14 are given in the `''class.method''` format.

The `form_clRoot.form_clRoot(Group group)` method initializes the main GUI component with one method. The `form_clOps.initializeOps()` method registers the GUI widgets in a vector to manage the event handling. The `handleCallback( int index, Message`

[14]http://www.teaminabox.co.uk/downloads/metrics/descriptions/NumberOfParameters.html (24.3.2006).

Class	NSC	DIT
odp classes		
`OdpValue`	**10**	2
`Odp`	**5**	1
group classes		
`OdpWrapper`	**5**	3
`MapEditVJCanvasShadow`	0	**7**
`ShortKeyListGroup`	0	**6**
`ShortKeyChoiceGroup`	0	**6**
`PictureChoiceGroup`	0	**6**
`MapVJCanvasShadow`	1	**6**

Table 3.12: Significant metric values of class inheritance metrics.

Metric Acronym	Full Name	Threshold	Comment
LOCM	Lines of code per method	500	Lines of code without blank lines and comments.
PAR	Number of parameters	7	Methods with a large number of parameters can indicate that classes are missing.[14]

Table 3.13: Abbreviation, full description, and thresholds of the used method metrics.

`msg, Event evt)` method in the same class contains a switch statement with 108 cases to add a string before calling the `cacheAction( String sSubMethodTyp, Event evt)` method with the corresponding parameters in the `form_cl` class. The `form_clCache.method( String sSubMethodTyp, Event evt)` method consists of a conditional statement to decide which business logic method has to be called with the given event and string identifier. The method is called by `form_cl.cacheAction( String sSubMethodTyp, Event evt)`. The `init Application()` method in `form_clCache` makes server calls to get data to be filled in the GUI elements. The methods receiving too many parameters are either constructors or handle tasks which involve inherently a lot of parameters, such as to insert a new customer into the database.

Summary of the Metric Analysis

The analysis of the structural entities with simple metrics and polymetric views confirmed the already suspected reengineering candidates and allowed to identify new suspects. `form_clCache` is the god class containing a lot of LOCs, methods, and attributes. The classes having "group" in their class name contain controller and business logic. They have a lot of methods and are also crucial for the reengineering of the system. The exact relations between the classes yet has to be determined with the dynamic testing of use cases to identify functional couplings.

3.3.3 Summary of Initial Understanding

The result of the initial understanding phase is an overall understanding of the responsibilities of the classes of the ticket sales component and how they interact with each other. The revealed architecture is JWS and historically driven, meaning that the structure of the ticket sales component is to a certain extent given by the automatically generated JWS classes and the preceding Oracle Forms application of the ticket sales component. `form_clCache` has procedural character (*e.g.*,

Method	LOCM	PAR
loose classes		
`form_clRoot.form_clRoot`	**4086**	1
`form_clOps.initializeOps`	**943**	0
`form_clOps.handleCallback`	**500**	3
`form_clCache.method`	**821**	2
`form_clCache.initApplication`	**546**	0
`form_clCache.getAndclickNextFreePlaces`	32	**14**
`form_clCache.insert_kundenrelation`	44	**12**
`form_clCache.kartendruck`	17	**10**
`form_clCache.postClick`	154	**10**
`form_clCache.getNextFreePlaces`	18	**9**
`KartendruckThread.KartendruckThread`	22	**22**
`form_clCache.StatusELS.StatusELS`	12	**12**
`odp` classes		
`OdpELSPlaceExtend.OdpELSPlaceExtend`	20	**18**
`OdpELSPlace.OdpELSPlace`	16	**14**
`ClientCache.click`	57	**9**
`data` class		
`SQL.handlePayment`	80	**12**
`group` classes		
`MapVJCanvasShadow.setCanvas`	64	**9**
`VJCanvasShadow.newCanvas`	41	**7**
`MapEditVJCanvas.createUnits`	65	**7**
`MapEditVJCanvasShadow.createUnits`	1	**7**

Table 3.14: List of methods having exceptional metrics.

communication logic is not called, but each time locally executed). Logic from the predecessor application has been packed into this class. In a pure JWS architectural context, `form_cl` would have played the role of `form_clCache`, instead its purpose is to serve as a navigation class. The packages are organized in: GUI subgroups, network communication, and database access. The loose classes intensively use the packages. A good approach to identify the purpose of the classes was to follow the methods involved in the handling of GUI events. The loose classes are inherently big because they are related to the main GUI component. A risk is the absent automated testability of the code which will have to be introduced. An opportunity is the knowledge of the developers about the component and the access to the reengineering knowledge from the Software Evolution and Architecture Lab.[15]

3.4 Detailed Reengineering Definition

The collected knowledge about the ticket sales component enables to define detailed reengineering priorities allowing a goal oriented proceeding in the detailed model capture phase of the reverse engineering process as the last step of preparation for the forward engineering. Setting the reengineering goals and deriving the propositions of how to achieve them is influenced by the beneath listed internal (*i.e.*, discovered in the program) and external (*i.e.*, given by stakeholders of

[15] http://seal.ifi.unizh.ch

the component) constraints.

- *The state of the component.* The ticket sales component is in a decayed state: An outdated GUI, a god class containing business and GUI control logic, considerable bug fixing activities across the component, insufficient code documentation, outdated user documentation, and a missing comprehensive testing framework.

- *What is most valuable to the companies customers?* The productivity of a ticket sales clerk. It depends on the GUI usability, the communication speed with the database, and the speed by which tickets are printed out. For the GUI part, it is important to have a quick and convenient access to the most commonly used features. For the communication part, stability and the time lag are important. The *Interview during Demo* pattern (Section 3.2.1 and discussions with the responsible developer indicated that improving the communication part is less urgent than the GUI part. The GUI starts up too slowly and does not incorporate the look and feel users expect of todays window based applications.

- *How to reduce the maintenance effort?* The GUI cannot be changed with the JWS environment anymore due to memory problems. Changes need to be done by hand. The `form_clCache` class contains evidence in the form of comments, that a lot of bug related changes have been undertaken. The business logic and the new GUI code needs to be structured maintenance friendly.

- *How to enable a successful longterm evolution of the ticket sales component?* The core asset of the system to be preserved for the future is the business logic. The business logic has to be restructured. Changes to the GUI have to be done easily again via a WYSIWYG editor.

- *Time constraints.* The time period given for the reverse engineering, forward engineering propositions, and prototype code fragments is six months. The state of the system implies that is not possible to consider the detailed reengineering for all of the already revealed problems the component suffers from.

The following reengineering goals that regard the above stated constraints are proposed:

- Introduction of a new GUI

- Adjustment for JDK 1.5 compliance of the whole component

- Remodularization of the controller

- Introduction of a test framework

- Improvement of the documentation of the code

- Introduction of a technical documentation of the component including the documentation of the business processes

It is yet unclear how and in which order the goals are going to be achieved. The purpose of the *Detailed Model Capture* pattern set in the next section aims at refine the understanding of the whole system and revealing details that allow to set up a reengineering plan for the stated goals.

3.5 Detailed Model Capture

The applied analysis patterns allowed to develop a static and functional perspective on the code. The emphasis was on the understanding of the component as a whole. We continue to work goal oriented after having stated the reengineering goals in Section 3.4. Starting to think about how the solution will be implemented becomes necessary to work efficiently toward goals.

Working through the *detailed model capture* pattern set is an iterative process. Newly discovered aspects of the component that are relevant to achieve the goals have to be included in capturing the model by using the patterns again. Iterations complicate the mapping of the observations made into a thesis given that it has to be well structured. The not strictly sequential proceeding has to be kept in mind when reading the *detailed model capture* section.

The focus of the this pattern set is to understand the runtime behavior of the program and to build up a detailed model of the ticket sales component parts that will be important for the reengineering. The patterns chosen to apply are *Step through the Execution*, *Tie Code and Questions*, and *Look for the Contracts*. While applying the *Step through the Execution* pattern, the *Tie Code and Questions* pattern is used to write questions, notes, and answers directly as comments into the source code to keep track of the understanding and allow synchronization with the code. The *Look for the Contracts* pattern concentrates on finding the implicit and explicit rules in the code for the interaction between the classes within the ticket sales component and the communication with the server.

3.5.1 Step through the Execution

The *Step through the Execution Pattern* aims at understanding where objects are instantiated at runtime and how they collaborate (*i.e.*, which methods are called and which parameters are passed). We use this pattern to locate the methods containing controller and business logic involved when GUI features are used. The understanding of the logic at the method granularity level allows to develop initial ideas for refactorings.

The component runs under JDK 1.1.8 together with the `jws.zip` file containing the JWS classes. JDK 1.1.8 does not come with the Java debugger interface[16] required by the Eclipse debugger which makes setting breakpoints and stepping through the execution in the debug mode under Eclipse impossible. Running the component with higher JDKs produced runtime errors at the startup of the component. The reason is that the event handling in the JWS classes is implemented to be compliant with the original event handling mechanism of Java which has been changed in higher JDKs. Nevertheless, the component can be run with the required JDK 1.1.8 in Eclipse.

We decided to follow the execution path with the built into the code debug statements given out to the console when the debug parameters are set at program startup. The risk of using the built-in debug system is that it is not clear whether all the relevant code sections for the understanding do contain debug statements. The work is slowed down because the methods shown in the console need to be searched manually in the code since no breakpoints can be set due to the missing debugging interface in the JDK 1.1.8.

The generated debug output data typically includes the class name, the name of the called method, a number of its parameter names and their values. They are written to the console and also into a log file depending on the *debug* integer parameter value given at the program startup. At the end of each executed feature chosen in the GUI, a small number of debug statements that were also written to the console has been separately noted to have reference points available for

[16] JDK 1.2 and higher versions implement the JDPA (Java Platform Debugger Architecture) interface required by modern IDEs. Sun Microsystems did not provide JPDA retrofits for the 1.1.x JVMs. http://www.visi.com/g̃yles19/fomserve/cache/169.html (27.4.2005).

the mapping of features onto the code. The log file has then been taken to manually follow the debug traces in the code. It turned out that the debug data in the log file is inappropriate to follow the execution path. The analysis of the log file showed that the debug output is either too detailed or does not at all cover the business logic we are interested in. The debug statements are not systematically distributed in the code. They have been added to the code when problems during the implementation arose, but not when the code worked as expected. The built-in debug statements allowed to locate a part of the code that has been executed, but the information obtained is not detailed enough to see all the involved logic on the method level relating to a feature. The initial console output was useful to identify the rough relations between the features executed in the GUI and the location of the corresponding functionality in the code. The built-in debug statements focus on the debugging of the communication logic, but we are interested in the business logic since our goal is to allocate business logic source code to GUI features. The understanding of the communication is not a priority. We were forced to consider other options to find out about the runtime behavior of the program.

Alternatives to Follow the Execution path

The built-in debug statements did not allow to develop the necessary understanding of the program to proceed with rearranging the logic. We were no able to develop a sufficient mental model of the component as a preparation for further reengineering steps. The debugging restrictions imposed by the JDK 1.1.8 thwarted our program understanding. We considered alternatives since this reengineering step is crucial. Our goal is to understand where the logic of the GUI features is located in the code on the method level. The list beneath discusses options.

- **Port to a newer JDK**
  - **Description:** Port the ticket sales component to a JDK higher than 1.1.8. The JWS class files need to be decompiled and workarounds to the code generating errors in JDKs higher than 1.1.8 have to be created.
  - **Advantages:** Debugging support with a modern IDE becomes available. Early migration to the current JDK.
  - **Disadvantages:** It is unclear how big the effort is going to be since the reasons for the errors that arise in the JWS class files are not well understood. Considerable time will be spent on the reengineering of JWS GUI code that will be substituted.
  - **Assessment:** This solution is not further considered because of the risk of failure and too much time will have to be spent on the old GUI code.
- **Application Profiling**
  - **Description:** Use a profiler to collect runtime data of the ticket sales component.
  - **Advantages:** Detailed information of a program run is generated.
  - **Disadvantages:** Profilers focus on performance aspects of a program and create too much low level memory information when profiling a program. A lot of the available profilers connect to the Java Virtual Machine Profiler Interface (JVMPI) which has been introduced only since JDK 1.2.
  - **Assessment:** Profiling under JDK 1.1.8 is done by using the `-prof` option when running a program which creates a `java.prof` file. A look at existing `java.prof` files of the ticket sales component showed that the files contain too much information and that is not in a convenient information source for our purposes. This solution is not further considered.

- **Use a logging framework**
  - **Description:** Introduce a logging framework to the ticket sales component.
  - **Advantages:** Systematic and customized information of the runtime behavior of the program.
  - **Disadvantages:** Considerable effort for preparing the code to be logged. Decreases the programs performance (even if the log mode is switched off).
  - **Assessment:** The open source logging framework Log4J[17] has been studied to evaluate the fit of a logging framework for our goal. This solution is not further considered because the setup of the Log4J framework takes too much time and requires an understanding of the ticket sales component superior to the one we currently have. This solution is not further considered.
- **Use another current IDE Framework**
  - **Description:** Load the classes of the ticket sales component into an alternative IDE.
  - **Advantages:** The whole ticket sales component can conveniently be managed.
  - **Disadvantages:** Time needed to obtain and install the IDEs.
  - **Assessment:** The popular Java IDEs all require JDK 1.2 and higher because their built-in debuggers are based on the JDPA. Nevertheless, we decided to load the ticket sales component into the *Java Studio Enterprise 8* IDE from Sun Microsystems. The result was as expected, even the IDE of the company that offers the Java programming language does not provide any workaround to the JDK 1.1.8 debugging deficiencies. This approach was not successful.
- **Use the JDK 1.1.8 Debugger**
  - **Description:** Debug with the Java Debugger (JDB) that comes with the JDK 1.1.8.
  - **Advantages:** Small effort to run the program in debug mode.
  - **Disadvantages:** JDB is console based.
  - **Assessment:** We made a test run with JDB. The command line based control of the JDB does not scale well for the size of our component. We did not find any suitable JDB front end GUI to simplify the interaction with JDB. This approach was not successful.
- **Use the original GUI designer**
  - **Description:** The GUI of the ticket sales component was built with Java Workshop editor which came with a specially adapted debugger. Load the project into the Java WorkShop and debug it.
  - **Advantages:** Small effort to run the program in debug mode.
  - **Disadvantages:** The responsible developer told us, that the whole ticket sales component has never been runnable in the Java Workshop.
  - **Assessment:** Only the GUI has been created with Java WorkShop. Trials to debug the whole component did not work. The documentation of the Java Workshop is not available anymore.
- **Use print statements**

[17] http://logging.apache.org/log4j/docs

- **Description:** Set `System.out.println(String)` statements at interesting positions in the source code.
- **Advantages:** Forces to get in touch with the source code. Allows to flexibly set print statements at interesting points in the code at the desired level of granularity.
- **Disadvantages:** Consumes a lot of time.
- **Assessment:** We started out to wire the `form_clCache` logic god class with print statements at the beginning of each method. The results written to the console of a test run were promising.

Another option is to change to a UNIX environment and use the data display debugger[18] (ddd) which supports the JDK 1.1 JDB. We already spent plenty of time to evaluate the above listed options. We decided to stop to look for tools that help us to accomplish this reengineering step. Tools are comfortable, but they tend to take more time than expected to get them running and to get acquainted to them. The goal of this pattern is to understand what the code does, which means that the code sections of interest have to be carefully studied anyway. The use of a tool does simplify the detection of code that is of interest, but not the understanding. We decided to proceed in the stepping through the execution with adding print statements to the code. We are interested in customized analysis results for the improvement of our understanding of the code. The method to use print statements fits best to achieve our goal.

Print Statements

We decided to allocate features to the code with the built-in debug system as good as possible and then refine our understanding of the code by taking irrelevant debug statements away and adding print statements to expected places of interest in the code. We iteratively ran the ticket sales component and executed different functionalities the GUI offers until we had identified the overall program flow. After each GUI input operation, we studied the debug information printed out to the console. We focused on console output suspected to execute business logic. Debug statements that are related to the client-server communication have been removed to reduce the amount of output information. Listing 3.1 shows a snippet of the debug information for the GUI feature *load the hall map of a chosen event* after irrelevant debug statements have been removed and before additional ones have been added. The output delivers enough information to roughly identify the zones affected in the code.

```
MultiColListGroup selectedRow listPos=20 list.entries()=21
form_clCache dialogOpen=false newdialogOpen=true value=0
form_clCache ok_Editlist=VORSTELLUNGSLISTE
Setze pltze zurck
KeyListGroup reset:
KeyGroup reset:
KeyListGroup reset:
KeyGroup reset:
KeyListGroup: setOnGroup: selectedKey=0
*************** Set Vector **************
KeyGroup extractEntries!
KeyGroup extractEntries item=Selbstabholung
KeyGroup: key=0
```

[18] http://www.gnu.org/software/ddd

```
KeyListGroup set ListItems
KeyListGroup set ListItems=1
KeyListGroup searchEntriesDB: 1 item found setEntry 0
KeyListGroup setEntry index=0 numberOf=1 item=Selbstabholung
KeyListGroup: setEntry: selectedKey=0
KeyListGroup: setEntry: selectedItem=Selbstabholung
ShortKeyListGroup: setEntry: selectedItem=Selbstabholung
...
```

Listing 3.1: A snippet of debug information obtained when loading the hall map for an event.

To refine our understanding of the program flow, we iteratively added additional print statements to the methods in the control and logic classes identified to be involved in a feature. There were also statements that had to be removed again because they have been set in methods that are called a lot of times or in methods constituted of navigation code. The print statements in the methods constituting of navigation code had to be added to find logic methods. For the logic methods involved in *load the hall map of a chosen event*, results are summarized in Table 3.15. Listing 3.2 shows the debug information produced by `form_clCache` for the *load the hall map of a chosen event* feature. The example shows, that even when debug statements are selectively set to only spot executed business logic, the amount of information produced implies that a lot of code is involved in the execution of the feature. A lot of methods in `form_clCache` implement methods declared in interfaces. They are called in group controller classes derived from the JWS `Group`.

```
form_clCache.ok_list
form_clCache ok_Editlist=VORSTELLUNGSLISTE
form_clCache.init_vorstellung
form_clCache.resetInfoText
form_clCache.verkauf_clear
form_clCache.check_berechtigung
form_clCache.add_ValueKeyListDefault sv teil
form_clCache.slectPictureErmaessigung
form_clCache.select_pictureChoice
form_clCache.getSonderkondVector
form_clCache.switch_vorstellungstyp
form_clCache.init_plan_vorst
form_clCache.select_canvas_dialog
form_clCache.setCursor
form_clCache.syncToolkit
form_clCache select_eventupdate
form_clCache iCanvasId=102
form_clCache newCanvasTime=392221299
```

Listing 3.2: A snippet of the debug information with added print statements obtained for the `form_clCache` class when loading the hall map for an event.

Listing 3.3 shows the code of a method in the `form_clCache` class involved when executing the feature *load the hall map of a chosen event*. Data needed for the query on the database is added to a vector that is sent to the server. The code listing gives an impression of the code style of the ticket sales component and how the ODP communication is implemented.

```
public void select_pictureChoice(int iSKId) {
   if (form_clCacheprinttrace)
   { System.out.println("form_clCache.select_pictureChoice");}
   clearParameter();

   add(new OdpString("SELECT_LOVKEY"));
   add(new OdpInt(iLangCode));
   add(new OdpString(data.SQLConstant.substChar)); // SearchKey
   add(new OdpString(data.SQLConstant.substChar)); // SearchItem1
   add(new OdpString("PREISEVENT"));
   add(odpiVorstellungNum); //VorstNum
   add(new OdpInt(iSKId));
   add(new OdpInt(1)); // iStartPos Damit Event-Sonderkondition
   //und gleichzeitig die platz_seq ein - beim Aufrufer!!

   start(ObjectOdp.TRANSFER_SYNCHRON);

   if (Debug == 1)
      Odp.message("OdpCache ValueKeyList");

   try {
      recvErrorList();
      fGUI.pictureChoice.setVector((OdpVector)get(1));
   } catch(CacheException e) {
      showError(e.odpvkError,false);
   }
}
```

Listing 3.3: Code of the method `select_pictureChoice` in the `form_clCache` class.

Table 3.15 summarizes the insights gained of the *load the hall map of a chosen event*. The feature is divided into steps and the most important methods related to a step are listed. The called methods in other classes than `form_clCache` have been identified by manually following the execution path of methods in `form_clCache`. This procedure is cumbersome handwork, but good for the understanding of the code because we are forced to get in touch with important code. `form_clCache` is a singleton and larger GUI elements constituting of several groups have few instances at runtime. This eases the understanding of the runtime behavior, especially when polymorphic methods are involved.

The *Step through the Execution* pattern should always be applied in an organized manner. Getting distracted by code irrelevant to achieve a defined understanding goal can cost a lot of time. A goal can be to identify a data flow or to allocate a feature to code. Stepping through the execution with print statements allowed to verify the program flow assumption identified in Section 3.3.1 and to better understand the `form_clCache` logic.

We tested features with the method described in this section until we sufficiently identified the program flow pattern and key logic methods in the `form_clCache` class. We realized that in a refactoring phase, the `form_clCache` class has to be split up into coherent pieces as a preparation

Step	Related methods
Loads the information of the selected event into a vector.	ListGroup.selectItem() ListGroup.selectedRow()
Reset information in GUI elements	form_clCache.selectedRow()
Retrieve values from the database based on the selected event	KeyListGroup.setOnGroup KeyListGroup.searchEntriesDB ClientCache.select_ValueKeyList form_clCache.select_canvas_dialog
Set the received values in the corresponding GUI elements	KeyGroup.setVector KeyGroup.extractEntries MapVJCanvasShadow.newCanvas

Table 3.15: The *load the hall map of a chosen event* feature split up into steps with the corresponding logic methods.

for more detailed refactorings.

The class hierarchies and the code of the component are better understood because we were forced to go deep into the code. The business and communication logic of the component is mainly in the `form_clCache` and the GUI logic is additionally distributed in the JWS groups.

The use of an UML sequence diagram to represent the feature example given in this section is inappropriate because we are mainly interested in the business logic code of the feature since are too many methods calls to have a clearly laid out on such a diagram. An automated sequence diagram creation is not appropriate either because there are too many possible program execution paths.

An Aside: UML Reverse Engineering Tools

UML offers sequence, state, and collaboration diagrams for dynamic modeling of how elements of programs interact at runtime. There are UML tools such as EclipseUML from Omondo and UML Together from Borland that offer automated reverse engineering of UML sequence diagrams. Nevertheless, those tools are expensive and can produce reverse engineered diagrams that are incorrect or incomplete.[19] This is because the conceptual problems related to correct reverse-engineering of sequence diagrams are complicated and require advanced static analysis techniques.

Another argument against the use automatically reverse engineered UML behavior diagrams is that for architecturally considerate refactoring steps, a solid understanding of the code is necessary. Automatic generation of diagrams lessens the important contact with the code for the understanding of the system and tends to bias the view on the system. Still, sequence, state, and collaboration diagrams are good means to capture the gained insights. The tools supporting the reverse engineering of the UML diagram types in question can be used after the runtime analysis of the code to capture the observations made in diagrams serving as architectural views.

3.5.2 Look for the Contracts

The *Look for The Contracts* pattern is applied to understand the implicit and explicit contracts classes of interest demand to interact with each other. We use this pattern to investigate the Java interface classes of the ticket sales component. We want to learn more about the communication between the GUI and the controller to be able to decide how much work needs to be done to replace the GUI.

[19] http://nomad.cse.ohio-state.edu/red/ (21.4.2006).

Java Interfaces

Figure 3.9 shows a class and interface (purple squares) hierarchy view including the interface implementations generated with Creole. Not all of the 21 interfaces of the component are visible. Creole was not able to show `form_clCache` with its implementing interfaces. Our goals are to understand the interface hierarchy, the purpose of the interfaces, and to locate the classes that use them to access methods in the implementing classes. We studied each interface, the implementing classes, and the class(es) using it. The results for part of the interfaces are summarized in Table 3.16.

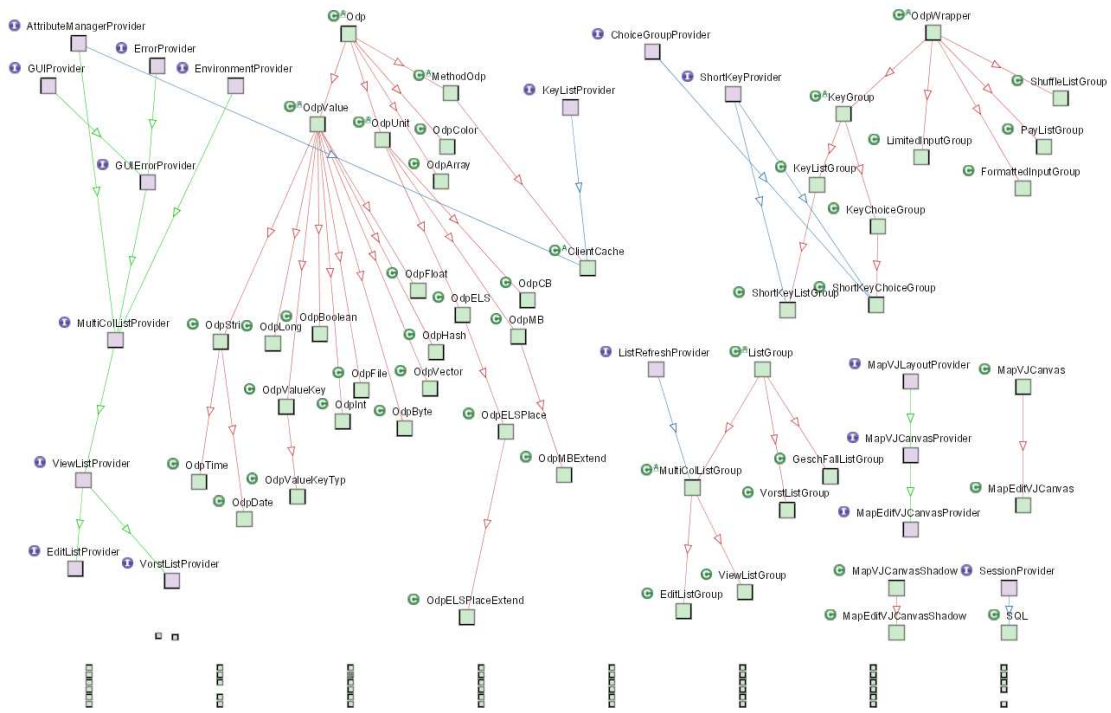

Figure 3.9: Visualization of class hierarchies, interface hierarchies, and interface implementations in the ticket sales component.

Interface	Implemented by	Used by	Purpose
`MapVJCanvasProvider`	`form_clCache`	in various ODP classes and classes related to maps	Defines the methods for operations to handle Canvas events (click, update)
`EditListProvider`	`form_clCache`	`EditListGroup`	Database interface for `EditListGroup`.
`ListRefresh Provider`	`form_clCache` `MultiColListGroup`	`KeyListGroup`	Workaround access to close a dialog window due to runtime problems under Windows95.

continued on next page

continued from previous page

Interface	Implemented by	Referenced by	Purpose
`ShuffleList Provider`	`form_clCache`	`ShuffleListGroup`	Database interface for `ShuffleListGroup`.
`TimeoutProvider`	`form_clCache`	`form_clCache` `TimeoutThread`	Interface and using class are defined in the `form_clCache.java` file. To wait for the user password at login.
`GUIProvider`	`form_cl`	Used by 8 classes	Interface for error output handling.
`VorstListProvider`	none	none	Database interface for `VorstListGroup`, unused
`MultiColList Provider`	none	`FormattedInput Group`, `ListGroup`	Database interface.
`MapVJLayout Provider`	none	none	unused
`ErrorProvider`	none	none	Database interface for error codes.
`ChoiceGroup Provider`	`ShortKey ChoiceGroup`	`ChoiceGroup`	Called when a choice in a list has been made.
`KeyChoiceListener`	none	none	unused

Table 3.16: Interfaces, their implementing classes, their referencing classes, and their purpose.

Interface hierarchies make the code harder to read because the method declarations inherited from super interfaces also have to be considered when studying an interface. We assume that the interface hierarchy has been introduced in parallel to the hierarchically ordered customized GUI subgroups. This explains why a part of the interfaces is not used. Most of the interfaces access server communication functionality in `form_clCache`. The methods declared in interfaces implemented in `form_clCache` can be taken as an indication to extract methods into new classes.

3.5.3 Detailed Model Capture Summary

Stepping through the execution with a graphical debugger was not possible. Alternatives such as JDB or application profiling have been considered, but did not function or were too extensive. Under the given circumstances, following the execution path was done by activating the built-in print-line oriented debug system and adding additional print-line statements at points of interest. The results allowed to better understand the runtime behavior of the ticket sales component and its logic. Studying the Java interfaces allowed to identify the contacts the classes support. The goals stated in Section 3.4 are not changed since the detailed model capture has not revealed more deficiencies.

3.6 Reverse Engineering Results

The externally and internally imposed constraints discussed in Section 3.4 led to the formulation of the reengineering goals and a goal directed appliance of the reverse engineering patterns. Based on the analysis results of the patterns, Table 3.17 and Table 3.18 list the loose classes and the classes in the `odp` package that are subject to reengineering or will be substituted. Criteria for classes to be changed are (at least one has to be fulfilled): LOC, method size (in terms of LOCM), local use of JWS data types, accessing JWS objects, or being derived from an JWS class. The dealing *amputation means* that the class will be substituted with class(es) of the new GUI. It is possible that a logic code of amputated classes can be reused. The dealing *type adaption* implies that the class either contains a JWS data type that needs to be substituted or accesses a JWS object which needs to be changed to refer to a mapper or directly to corresponding classes used by the new GUI depending on the reengineering propositions developed in Chapter 4. None of the criteria are fulfilled by the classes in the packages responsible for electronic payment, administration, printing, and the `data` package.

Class	Dealing	Description
`form_clRoot`	amputate	Substituted by new view class set.
`form_clOps`	amputate	Substitute with new controll class set.
`form_clCache`	refactor	Splitting into smaller coherent pieces
`form_cl`	amputate	reuse part of its logic
`form_clMain`	amputate	Substitute with new main class.
`KartendruckCache`	type adaption	Adapt to be usable without JWS.

Table 3.17: Reengineering dealing for the loose classes.

Class	Dealing	Description
`ClientCache`	type adaption, refactor	Adapt to be usable without JWS. Move misplaced fields.
`OdpCB`	type adaption	Adapt to be usable without JWS.
`OdpMB`	type adaption	Adapt to be usable without JWS.
`OdpUnit`	type adaption	Adapt to be usable without JWS.

Table 3.18: Reengineering dealing for the ODP classes.

The classes forming JWS groups in the `group` package are all subject to changes. All classes derived from the `Root` and the `Operations` classes are amputated and substituted by classes of the new GUI. All descendant classes from the `Group` class are part of the controller and need to be substituted depending on the structure imposed by the new GUI. Local control logic in the classes of type `Group` can be reused. The interfaces can be left aside during the forward engineering and only the ones still needed in the goal architecture have to be in the new version.

Chapter 4

Reengineering Options

The understanding we have gained of the ticket sales component enables develop a plan for the reengineering under the given constraints (Section 3.4). This chapter discusses the options on how to proceed and evaluates alternatives of technologies to use. Prototype examples are created to verify the feasibility of the chosen reengineering path.

4.1 Rebuild from Scratch?

Approximately 40 percent of the code will replaced at the end of the this reengineering project. The question arises: Why not build the whole system from scratch with a new communication protocol? The reasons against this option are:

- The controller logic is complex, valuable, and tightly coupled with the communication protocol. A new implementation of the business logic is a risk of failure.
- The human resources for a complete rebuild of the component and the server are not available.
- It is undetermined how the customers of the company would react: How will the acceptance of completely new solution be? Since the company works closely with its customers, the risk of a flood of incompatible requirements emerges.
- The risk of failure when migrating a system incrementally is smaller than it is when setting up new project.
- A stepwise reengineering deepens the understanding of the controller logic for the documentation and further optimizations.
- The communication protocol is stable and performs sufficiently, there is no urgent need to change it.

4.2 Alternatives to Java

The next big step in the evolution of the ticket sales component will be the substitution of the implemented ODP communication protocol. After this step, only the business logic of the controller will remain from the current code, the GUI controll logic and communication code will be

substituted. This change intent is an opportunity to question the implementation language of the component. Why stay with Java? The alternative to Java is the .Net framework from Microsoft. The reasons to stay with Java technologies—even though they do not have a good client-side reputation—are:

- The companies personal has more experience with Java technologies than with the .Net Framework.
- The .Net framework is windows operating systems centric, the company wants to offer its customers the possibility to run the ticket sales component on different platforms.
- There are many Java open source projects that can be of use for the ticket sales component.

4.3 Graphical User Interfaces

The following sections discuss technical and cognitive principles for GUIs and evaluate Java GUI API frameworks.

4.3.1 GUI Principles

The GUI is an important component of many software systems. The development of a GUI is a significant undertaking and the GUIs source code often comprises a substantial portion of the overall source code [HKR03] of the program. Nowadays, the GUI typically represents 60% of the overall source of a program [Mem02]. It is easy to write a GUI with the today available editors, but they often generate code that is hard to maintain. A GUI has to be user friendly and easy to maintain. To allow the user to work comfortably and efficient with a GUI the following usability aspects have to be considered:[1]

- Simplicity: Keep the GUI simple and straightforward.
- Support: Offer informative feedback.
- Familiarity: Build on prior knowledge of the user (from the real world, with other applications or operating systems).
- Obviousness: Make objects and their controls visible and intuitive.
- Encouragement: Make actions predictable and reversible.
- Satisfaction: Reflect the results of actions immediately. Any delay intrudes on the tasks of the users and erodes confidence in the system.
- Availability: Make all objects available at all times (no cancel in an operation when additional information is needed).
- Safety: Users have to be protected from making errors.
- Versatility: Allow users to choose the method of interaction that is most appropriate to their situation.

[1] http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/6 (2.5.2006).

A lot of literature discusses the cognitive aspects of GUIs. The general usability principles listed above are the most important ones for our purposes. The new GUI of the ticket sales component needs to consider them.

There is less current literature available that discusses the implementation of GUIs: While principles regarding good GUI design are well-established, principles regarding which programming structures to use with them are not [BH04]. Nevertheless, object-oriented GUI design in practice consists of positioning widgets in a WYSIWG editor that creates the view code automatically and implement the handling of its events in the controller. Oftentimes the separation of concerns is already proposed by the code the editors create. The strict realization of the MVC pattern is crucial for the future maintainability of a GUI. It is important to keep the size of the classes maintainable in all layers. This means when new features are added to the GUI, the refactoring potential in the controller has to be assessed since the view often evolves at the expense of the controller (which happened in our case). Periodic refactoring of the controller and the view keeps the code from degenerating.

4.3.2 GUI Creation Frameworks

GUI creation frameworks for enable the rapid construction of complex GUIs. The two most current Java GUI creation frameworks are Swing and the Standard Widget Toolkit (SWT). We evaluate them and chose the one fitting better for the new GUI of the ticket sales component. The AWT (Abstract Window Toolkit) is not considered in this evaluation because it is a simple toolkit with limited GUI components, layout managers, and events. "This is because Sun Microsystems decided to use a lowest common denominator approach for AWT. Only GUI components defined for all Java host environments are used. As a result, some commonly used components such as progress bars are not supported."[2] An alternative approach is to describe GUIs in XML documents that are parsed at runtime and rendered into GUI widgets, for example the open source project Thinlet.[3] This concept is very promising, but does not seem mature yet for large GUIs because of the restricted widget set and lacking documentation.

Swing, SWT, and JFace

Swing is the primary Java GUI library from Sun Microsystems that aims to look and feel the same on all platforms [CM04]. The Swing library is part of the Java Foundation Classes (JFC). The architecture of Swing follows the MVC pattern. Swing has a complete set of GUI components. They do not rely on the native widgets of the operating system. Swing components are painted using graphic primitives such as lines, rectangles, and text[4] Swing emulates native components, but their behavior is not always the same which confuses users. The emulation of native components reduces the performance of Swing GUIs.

SWT is the GUI toolkit developed by IBM for its Eclipse IDE. It is open source and has an abstract API that is platform independent and uses platform dependent implementations to wrap around native components. The SWT API and SWT behavior are consistent on all supported platforms. The SWT classes make direct calls to the operating system using JNI. SWT provides a rich set of widgets, wrapping native components if they are available. Otherwise, it emulates the components using Java code [CM04].

Each of the two toolkits is complete and powerful enough to build complex GUIs. Swing is generally superior to SWT when used without JFace. Swing has the advantage of being built into Java technology, is completely portable, and has a better architecture. Swing also has the

[2] http://www-128.ibm.com/developerworks/opensource/library/os-swingswt/ (3.5.2006).
[3] http://www.thinlet.com
[4] http://www.onjava.com/pub/a/onjava/2004/02/18/desktop.html (3.5.2006.)

advantage for advanced graphical applications. SWT has the advantage of being implemented as a native application which increases the performance and native compatibility of SWT based GUIs.[5] Table 4.1 lines up the advantages and disadvantages of SWT and Swing.

	SWT	**Swing**
Advantages	Use of native components has brings good performance User friendly native look and feel	Uses MVC Platform independent
Disadvantages	Must include SWT dynamic link libraries (DLL) for the targeted platform Fewer literature available compared to Swing	Performance not as good as equivalent native program Behavior never completely synchronized with native platform due to emulation

Table 4.1: Line up of SWT and Swing properties.

Our biggest fear when using Swing is a poor performance for the ticket sales application. The main concern with SWT is the lack of MVC support. JFace is a framework on top of SWT that allows to implement the MVC pattern and offers more sophisticated widgets. JFace is independent from the window system in both, its API and implementation, and is designed to work with SWT without hiding it.[6] We decided to use SWT together with JFace for the implementation of the new GUI. There are GUI editors that allow the generation of Swing and SWT/JFace code, for example WindowBuilder Pro.[7] This gives us the option to easier switch to Swing if SWT causes unexpected problems.

4.4 Unit Tests

The ticket sales component classes to be reengineered do not have a systematic and repeatable test set. Before we can start with the refactorings, we need to establish a test framework. On the class level, unit tests for testing controller methods will be established before each refactoring. The Junit[8] test framework will be used to test methods of the controller classes. On the component level, GUI tests offered will be implemented to validate the system as a whole. GUI tests take the perspective of the end user and can be seen as a quality test. There are two general styles of GUI test automation tools:[9]

- Record: The user records a set of actions on the GUI under test, and the tool is able to replay those actions later. This approach is simplistic, but fragile (*i.e.,* tests break easily if the GUI changes).
- Programmatic: The user writes code describing the interaction with the GUI under test. The approach is more robust and flexible, but time to program the tests has to be investigated.

We will use GUI automation tests of the record style. If the tests prove too fragile, we will change to the programmatic style. The unit tests together with the functional GUI test set will result in a regression test suite. It will fulfill the reengineering goal of a test framework.

[5] http://www-128.ibm.com/developerworks/opensource/library/os-swingswt/ (5.52006).
[6] http://www-128.ibm.com/developerworks/opensource/library/os-ecgui1/ (5.5.2006).
[7] http://www.windowbuilderpro.com
[8] http://junit.sourceforge.net
[9] http://wiki.java.net/bin/view/Javapedia/TestingGUIApplications (30.5.2006).

4.5 Evaluation of the Proceeding

Migrate Systems Incrementally pattern recommends to deploy new functionality in frequent increments. The *Always Have a Running Version* pattern suggests to keep the component functional during the reengineering as much as possible. The two patterns are most easily respected by keeping the current JWS GUI and incrementally adding the SWT GUI to the component. A parameter given at startup decides which GUI is loaded. A mapper between the main controller and the two views needs to be installed to route and transform the accesses between the views and the controller.

4.5.1 SWT and JWS together?

The SWT documentation[10] does specify which JDK is at least needed to to run the SWT libraries. The SWT newsgroup states that that SWT only requires a JRE starting from 1.1.8.[11] We implemented a small GUI component with the SWT API classes and added them to the ticket sales component to verify the coexistence of the JWS and SWT API class sets under JDK 1.1.8. The window contains frequently used widgets and ran successfully under the JDK 1.4.2. We added the classes to the ticket sales component, and tried to deploy the code together with the current SWT JAR under JDK 1.1.8. The result was a runtime error because the SWT class files in the swt.jar library have been crated with a more recent version than JDK 1.1.8. We then tried to compile the SWT source files with the JDK 1.1.8. This was not successful either, because the current SWT source files use Java methods that have been added to the JDK after 1.1.8. We decided to abort this path because the next step would have been to compile older SWT source files and hope that they match with the JDK 1.1.8. Even if this worked, the problem that current SWT GUI editors create non JDK 1.1.8 compliant code remains. The option of migrating incrementally while keeping the JWS GUI toward the new SWT GUI under JDK 1.1.8 is abandoned because we run into too complicated compatibility problems. The validation showed the weakness of open source software: The documentation is often not sufficient and the feasibility therefore has to be verified with with a small implementation example.

4.5.2 View or Controller first?

The crucial reengineering steps are the implementation of the SWT GUI and the restructuring of the controller. The impossibility of the parallel existence of JWS and SWT libraries restrains the reengineering proceedings. We are left with the choices of either first restructure the controller under JDK 1.1.8 or first implement the SWT GUI and then connect it with the controller. The logic in the controller works sufficiently reliable, its main problem is the maintenance due to its size and complexity. The GUI loads slowly and does not correspond with todays expectations of GUIs. A new GUI is more valuable to the customer than the refactored controller. We therefore implement the new GUI before we refactor the control logic. The missing graphical debug support and unit tests also discourages to first refactor the model. A lot of methods that are candidates for refactorings are of return type `void` and setting up unit tests for them is inefficient because they can only be tested whether they trow exceptions. The implementation of the new GUI has to be stepwise, that is subcomponents of the new GUI have to be built and wired to the controller to verify their functioning to avoid the risk of failure that comes with a "big bang" migration. Existing GUI components (*e.g.*, the pop-up window to choose an event) can be taken as entities to reengineer. On the code level, those entities correspond to a set that consists of a `Root`, `Group`,

[10] http://www.eclipse.org/swt/docs.php (30.4.2006).
[11] eclipse.platform.swt newsgroup, thread: Java and SWT libs compatibility (30.4.2006)

and `Operations` class. This approach also gives us the opportunity to better understand the controller logic by looking at the parts used by an entity that is reimplemented. First creating the view is also the fastest way to a higher JDK with debug support.

4.5.3 Mapper or Direct Access to the View?

The non amputated controller logic in `form_clCache` cannot use JWS types to store data to be able to run with the SWT GUI. This leaves us with two possibilities to set up the communication between the controller and the new SWT view:

- **Wire the SWT GUI directly with the controller.** All view accesses in the controller need to be substituted with the corresponding accesses to SWT GUI elements and the used JWS types need to substituted in the controller.
- **Communicate via a mapper.** Substitute the used JWS data types in the model and make mapper classes for the JWS classes accessed in the model. The mapper classes delegate the model accesses to the right SWT element in the view.

Wire the SWT GUI directly to the controller increases the risk of failure because two large steps are made (*i.e.,* creating the whole SWT GUI, and wiring it with the controller) before the component is able to run again. The controller is out of order for the time the new GUI is wired. We therefore chose to create a mapper. This means more work, but allows to verify that the controller can run without the JWS classes. In a first step, we substitute the JWS data types used in the controller with standard JDK data types. We then create a mapper that will consist of the set of JWS classes that are accessed by the controller. The mapper will delegate the accesses from the controller to the view (the mapper therefore knows JWS classes). The result is a JWS-free controller that can via the mapper access the JWS GUI. We are then ready to gradually implement and test the new SWT GUI, duplicate the JWS mapper and adapt it for the SWT GUI. Mappings to the not yet implemented SWT GUI parts are substituted with dummy accesses. The accesses from the controller to the view remain the same, but the transformation of data and the routing to the view are adapted to the SWT GUI structure. The mapper solution allows to have a running version during the construction of the SWT GUI and incrementally migrates to the target component. The events passed to `form_clCache` from the view do not have to mapped because the old AWT `Event` class is still available in JDK 1.5 for backwards compatibility. The considerations made in these sections lead to the beneath listed proceeding for the core reengineering work:

- Substitute the JWS data types in the controller with standard JDK data types and constructs thereof
- Create mapper classes for the used JWS classes in the controller classes
- Fill the mapper classes with view methods accessed by the controller
- Fill the methods in the mapper with data transformations where necessary and link them to the corresponding method in the JWS GUI.
- Test the component with newly added mapper
- Incrementally create the SWT GUI.
- Incrementally adapt the mapper for the SWT GUI. Yet unused mapper methods either return dummy values or are empty in the case of a setter mapping.
- Deploy the SWT ticket sales component

- Refactor the controller
- Remove the SWT mapper
- Deploy the refactored SWT ticket sales component

4.6 Prototype the Target Solution

We have to verify the feasibility of the critical steps, that is the substitution of JWS data types in the controller, the implementation of the mapper, and the controller refactorings. A prototype examples of the solution have to be implemented. This also allows to asses the amount of work needed and its complexity.

4.6.1 Feasibility of JWS Type Substitutions

JWS data types locally used in controller classes that are not amputated have to be substituted with a JWS free alternative. We selected the JWS type `AttributeList` used in the `OdpUnit` class which stores a list of another JWS type, `Attributes`. The sparse JWS documentation of the two types has been studied to understand their purpose. We then searched all of its used methods in the component. We created the two types and added them to the component. To test their correctness, we ran the JWS and the substitute types in parallel, that is whenever an instance of the original type is created or accessed, we performed the same operations with the substitute types. Print statements have been added to the code after operations on the JWS and the substitute types are performed to verify the correctness. The implementations of the substitute types is feasible. The code listed in Section A.1.

4.6.2 Feasibility of the Mapper

The accesses to JWS view objects in the controller classes that are refactored need to be found and mapped. The JWS GUI view and the new SWT view code need to be able to function with the same controller classes that are refactored. The mapper classes need to have the same class hierarchy relations as the accessed JWS classes to conform to polymorphism (specialized objects can be held by more general object types).

We created at prototype mapper for the accessed JWS types and instances used in the `odp` package. The ODP classes that contain or access JWS objects are listed in Table 3.18 in Section 3.6. For each ODP class that accesses objects of a JWS type, we created a mapper class when a JWS object in the view was accessed or used for comparison (*i.e.*, `instanceof`). We had to respect the JWS class hierarchy for the mapper classes. We filled up the mapper classes with the signatures of JWS methods used by the ODP classes. The methods contain the original JWS method that accesses the view and type transformations for the cases where a JWS type is returned to the ODP classes from the view or sent by the ODP classes to the view. The methods are poorly described in the JWS API documentation, it is not always clear how the JWS types store data, and how the different JWS classes that form a group interact with each other. We added comments to methods that where not perfectly understood for the case of buggy behavior. The mapper classes reference the JWS view objects. A main mapper class has been created to hold the used mapper classes. The main mapper class is instantiated right after the `form_clRoot` class to enable the references to it. Section A.2 lists an example of a mapper class.

4.6.3 Feasibility of the Refactorings

The refactoring of the business logic which is mainly breaking big classes and methods into smaller cohesive classes and relate them to each other consists of small steps. The refactorings that are used are well documented in the literature and Eclipse offers automated refactoring features to support the process. Given the possibility of rapid edit-compile cycles in conjunction with testing and automated refactorings reduces the risk of running into hard refactoring problems.

The `form_clCache` class holds a lot of responsibilities. Code can be removed from the class to split it up by either moving it to a collaborating class or into a new class. The majority of the attribute names of the `form_clCache` class do not clearly indicate the purpose and comments are rare. The method names do not always describe its behavior well.

We identified the method `public void besuchsstatistik()` in `form_clCache` to be executed when the string `sSubMethodTyp` passed by the event handler is equal to `BESUCHS STATISTIK` as shown in Listing 4.1. Before the a method can be moved, the target class has to be created and made visible in the originating class.

```
if (sSubMethodTyp.equals("BESUCHSSTATISTIK")) {
        besuchsstatistik();
    }
```

Listing 4.1: An extract of the `method(String sSubMethodTyp,Event evt)` code calling a method candidate for refactoring.

Section A.2 lists the resulting class. The automated refactorings produce sufficient results. Eclipse only has problems with enlarging the visibility scope of methods accessed via the past references of the class(es) they are defined in.

Prototyping the critical reengineering steps showed their feasibility. The refactoring example (discussed in detail in Section A.2) showed that a refactoring enables further refactorings which slowly approaches the code to the goal architecture. The rule of thumb to apply is to start with evident refactoring candidates and make refactorings that have have a clearly positive impact on the goal architecture.

Chapter 5

Reengineering Plan

A reengineering plan has to be developed based on the goals set in Section 3.4, the information revealed about the ticket sales component during the reverse engineering, and the discussed options in Chapter 4. The plan states what needs to be done and gives templates for the implementation.

5.1 Overview

Figure 5.1 shows the goal architecture for the ticket sales component. The GUI constitutes of a hierarchy of SWT composites. A composite is the SWT pendant to a JWS group. Each composite has its own event handler which calls the desired methods in the set of operation classes that controll view, application model and communication. Communication logic and the application model are separate components to foster software reuse. The overview to the reengineering plan including milestones is given in Table 5.1. During the controller refactoring, JUnit tests have to be introduced whenever code is regrouped, changed or added. This allows to grow a test base incrementally and rapid edit-compile-test cycles. Insufficiently commented code during the forward engineering has to be commented to be used by JavaDoc. During the reengineering, the extension of the component will be suspended and maintenance of the business logic will be reduced to a minimum.

5.2 Substitute Local JWS Elements in Controller

Table 5.2 shows all non amputated controller classes that contain locally used JWS data types. The JWS types to be substituted are listed together with their substitutes. The data types that are sent to the view are transformed back into JWS types in a temporary mapper class. An implementation example of a custom types is given in Section A.1.

5.3 Create JWS Mapper

Figure 5.2 shows the mapper hierarchy that has to be created for the JWS types. The hierarchy for the custom-made classes derived from the JWS `Group` class in the `group` package also need to be constructed (a part of them is shown in Figure 3.7 in Section 3.3.1) and added to the hierarchy. The "M" at the end of each class name indicates "mapper class" to avoid mistakes. Only the view

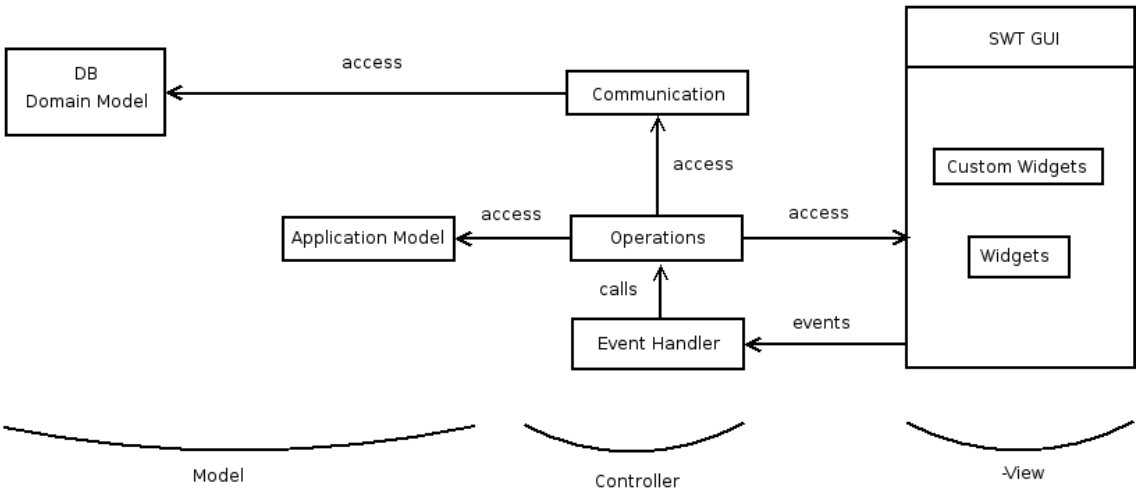

Figure 5.1: The architecture of the reengineered ticket sales component.

classes accessed in the non amputated controller classes need to be created, but the ordering of the original JWS hierarchy has to be respected for polymorph functions in the controller. The classes only need to contain the methods used in the controller, mostly setters and getters. If JWS types have to be returned or obtained, the mapper also needs to make the type conversion. A part of the JWS classes in the controller are only used in conditional statements to find out of what type a received object is. Section A.2 gives an example of an implemented mapper class.

The mapper classes are declared in a main mapper class which also sets references to the corresponding elements in the `form_clRoot` class. Its instantiation follows right after the `form_clRoot` class has been created in the `form_cl` class.

5.4 Build SWT GUI

We create the new GUI from top to bottom, that is we start with substituting the JWS main panel, proceed with the definition of its sub-panels, until we are down to the basic widgets a panel holds. `form_clRoot` contains all elements the GUI consists of. The class holds the information about the relations between the panels. It is not recognizable from the names of the widgets to what part of the GUI they belong (widgets containing text visible to users receive the text from the database to enable the support several GUI languages). We use the Java Workshop editor to visualize the `formcl_Root` class. This allows us to associate the names of GUI elements with their look. The iterative procedure to reimplement the GUI is as follows:

- In the Java Workshop editor, chose the panel that is closest to the root panel of the JWS panel hierarchy.
- Look up the name of the panel and which widgets are added to it in the `form_clRoot` class.
- Implement the corresponding SWT widgets in a SWT composite
- Find the events in the `form_clOps` class that are handled by the substituted panel
- Reimplement the corresponding event handlers in a separate class (Listener interfaces)

Step	Description	Milestone
Substitute JWS elements in controller	The controller classes need to be freed from locally used JWS data types. Substitutes for JWS elements can be fitting Java types, or custom JDK type based constructs.	**M1:** The controller is free of locally used JWS types. The component can still be run under JWS.
Create mapper	A class hierarchy analog the JWS classes in the controller that access the view has to be created. The mapper classes have to be filled up with all accessed view methods in the controller. Fill the methods in the mapper with data transformations where necessary and link them to the corresponding method in the JWS GUI.	**M2:** The component runs properly under JWS with the mapper.
Build SWT GUI	Incrementally create SWT GUI. The GUI elements are connected with the mapper for JWS. The mapper is adapted in a way that the controller and the already implemented view parts can run together. Automated tests of style record have to be created.	**M3:** The ticket sales component has a new SWT GUI, follows the MVC pattern and can be run under JDK 1.5. A new version of the component can be deployed.
Refactoring of the controller	The `form_clCache` controller class is split up. New classes containing data and their handling methods are created. Misplaced methods are moved to classes containing features that use them most. Large methods are split up.	**M4:** The god class is split up into manageable pieces. The code of the component is reengineered.
Create technical documentation	The architecture of the component is described, UML diagrams are given, and the use cases are related to the code.	**M5:** A documentation of the component is established on the Wiki system of the company.
Update user documentation	The outdated use cases are updated	**M6:** The user manual is synchronized to the current state of the ticket sales component.

Table 5.1: Overview of the reengineering plan including milestones.

Class	JWS Type	Substitute Type
OdpUnit	AttributeList	see Section A.1
ClientCache	AttributeManager	Custom type
form_clCache	ColumnList	Object[]

Table 5.2: List of JWS types in the controller that need to be substituted.

- Connect the composite to the mapper

The result after each iteration is a SWT composite class that connected to the mapper and an event handler class directly accessing the `form_clCache` class. Code from custom JWS groups

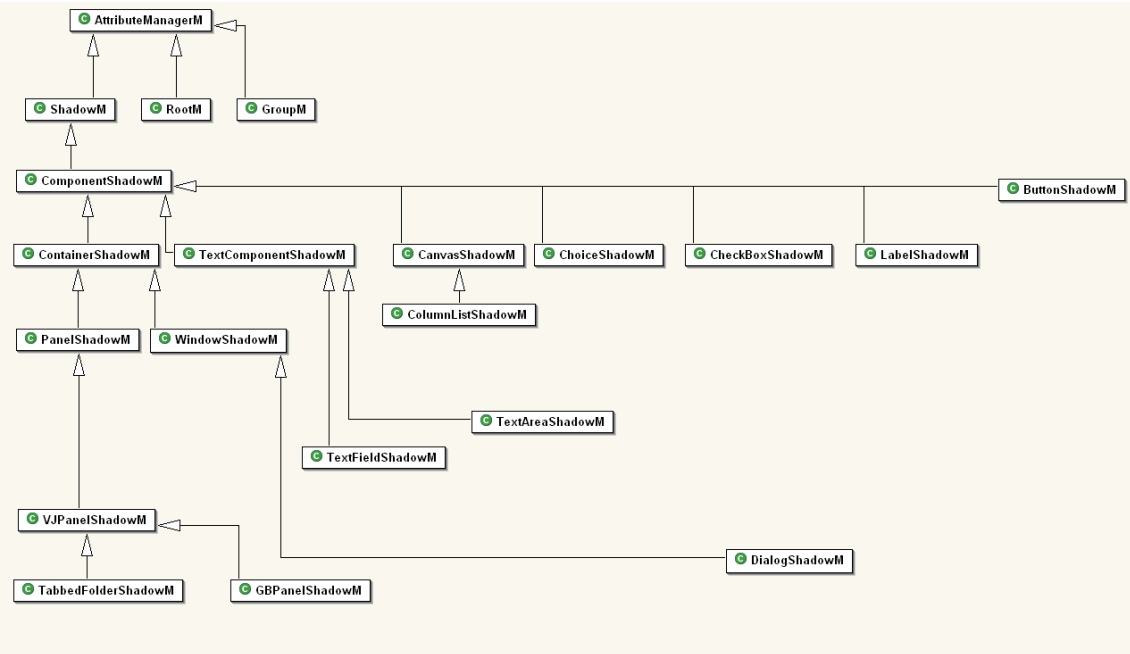

Figure 5.2: The class hierarchy of the JWS mapper classes.

can be reused.

The resulting GUI will look similar to the existing one. The main difference is that events are now handled in the controllers of the composites and are not delegated up the GUI hierarchy anymore to a main event handler.

5.5 Controller Refactoring

A mixture of static and dynamic observations has to be made to be able to find couplings of methods and attributes that can be moved into other classes. The size of the class and the limited ability to follow the execution path make it hard to give a perfect plan of how to split up the class. The refactoring reality is to start out with moving out clearly coupled refactorings and then identify more cohesive methods and attributes as the understanding of the code improves due to the newly added GUI and the already made refactorings.

The goal is to split up the `form_clCache` class into smaller classes that are responsible for either communication with the server, holding the state attributes[1] of the application model, or executing methods called by the event handler. The first step is to split up the operations of `form_clCache` into classes responsible for a subject, for example managing client data. We used the criteria listed beneath to find methods that work on the same subject.

- Method names
- The `method( String sSubMethodTyp, Event evt)` conditional method in `form_clCache` which selects the methods to be executed upon the arrival of an event from the view.

[1] Data Containers a bad smell, but we prefer to have the application model separated from other controller components.

Considering the events is helpful to see to which GUI feature the code is coupled if the names of the methods do not allow deduce their purpose. Table 5.3 summarizes the identified subjects related methods resulting from the dynamic and static observation of the `form_clCache` class. If a method has several signatures, all methods with the same name belong to the same subject.

Subject	**Method(s)**
Subject	**Method(s)**
Application startup	`checkClientVersion, initApplication, login`
Payment	`getBKZahlung, receiveCardData, bkMessage, getSqlString, getTerminalNummer, getThisPaymentConnection, handleZahlung global_bkZahlung, mkl_aktiv, receiveCardData, retourPreis, retourPreisMain`
VerkaufController	`init_verkauf_dlg`
Ticket barcodes	`processBarCode, receiveBarCode`
Customer management	`check_email, check_string, delete_kunderelation, delete_werbemittel, getMediumRessortVector, insert_kunderelation, iud_kunde, iud_kunderelation, iud_kunde, iud_werbemittel, reset_kunde, select_kunde, select_kunderelation, select_werbemittel`
List management	`ok_List, remove_ShuffleList, select_List`
Ticket logic	`check_kombiticketPreisGruppe, click_kombiticket`
Ticket sales	`checkZahlart, click_alleskombiticket, getSonderkonditionVector, getSortedEnumeration, getSortedVector, gutschein, init_verkauf_dlg, readVerkaufListeDialog, select_kombiticket, select_kombiticketvorstellung, setVerkaufListeDialog, verkauf, verkaufcancel, verkaufja, verkaufnein, verkaufsliste`
Hall map controll	`click, clickProductPlaces, fillLeft, getAndclickNextFreePlaces, getNextFreePlaces, select_canvas, setze_status`
Merchandise	`clickArtikel, select_ArtikelPreis`
Vector sorting	`compareStatusELS`
Type converters	`convert, StringtoOdpVector`
Database operations	`do_commit, do_rollback`
Guided tour	`fillLeft, fuehrungsKalender`
Window control	`getDialogOpen, methodGUI, resetInfoText, switch_kundenmaske`

continued on next page

continued from previous page

Subject	**Method(s)**
Application model	`getPGColor`
Ticket printing	`kartendruck, loadProgress, printException, printKarte`
Orders	`select_bestellung, select_bestellung_detail, select_bestellungs_liste, storno_bestellung, wandle_bestellung`
Subsripttions	`vorst_abo_storno, vorst_abo_zuo, wahlabobuchen`

Table 5.3: Subjects representing new classes to split up the `form_clCache` class.

Not all methods are listed in Table 5.3 for the following reasons:

- They can be substituted by SWT widget controll methods
- They represent dead code
- The information they retrieve from the JWS groups can be obtained by SWT typed events.
- Their purpose is unclear or they cannot clearly be associated with a subject

The methods in Table 5.3 have to be moved first before deciding what to do with the non associated methods. The classes they are moved to have to be derived from `ClientCache` to ensure the functionality of methods that communicate with the server. The proposed move method refactorings will make further refactorings visible and clarify the unassociated methods of the `form_clCache` class. The methods have to be moved without consideration of their interfaces. The interfaces can be split up or are not needed anymore because the functionality they access has moved to an architecturally more purposeful place. Attributes that are only coupled to one subject have to be moved into the newly created subject classes. Attributes used by many methods have to be moved to the application model. After the methods in Table 5.3 have been moved into new classes, the communication logic have to be separated from the operation methods. This is done by extracting the communication code into classes. The operation methods that need to access the server will call communication methods and passing the parameters used for the communication protocol. Listing 5.1 shows the communication with the server. A vector to hold data to be transferred to the server via socket communication is first cleared. The parameters for the database query are added to the vector and the communication is started. Each extracted communication class will hold the same communication code structure as shown in Listing 5.1. The communication classes can be merged together to and a `case` statement is used to set the hard coded string as the first element of the vector. A for statement is used to load the vector with the passed values from the method call of the operations component.

```
clearParameter();
add(new OdpString("WANDLE_BESTELLUNG"));
add(new OdpInt(iLangCode));
add(new OdpString((String)fGUI.
BestellnummerTextField.get("text")+":"+verkauftyp));

start(ObjectOdp.TRANSFER_SYNCHRON);
setCursor(Frame.DEFAULT_CURSOR);
```

```
recvErrorList();
```

Listing 5.1: Use case of the ODP communication protocol.

Table 5.4 shows methods that are misplaced, they access data in an other class and can be moved the accessed class. Only methods that are not moved to amputated classes are listed. The methods accessing amputated classes need to be moved and adapted to the substituting SWT composite.

Method	Target class
`getCmdLineArgs, getFontButtonSize, getFontSeatSize, getRefreshTime`	`ParamTable`
`getColorScheme`	`OdpColor`

Table 5.4: Methods that need to moved to an existing class or a substitute thereof.

The `method(String sSubMethodTyp,Event evt)` method in `form_clCache` which under JWS handles all events of non custom GUI widgets is substituted with a set of SWT event handling classes that handle events issued by the composite classes when the new GUI is connected to the operations component.

The *move method* refactorings can be done with the Eclipse *move* refactoring feature. Typically, not all referred variables are visible in the target class. They have to be adapted manually, either by making them public or by the creation of getter and setter methods for them. A unit test for the method that is moved has to be created before each refactoring and the test set has to be run after each change. Additionally, it is advisable to run the whole ticket sales component after each refactoring and verify at runtime that the moved functionality has not been broken. Figure 5.3 shows the UML class diagram of a part of the classes incorporating the subjects proposed in Table 5.3. Only moved methods are shown, not inherited methods to implement from the abstract superclass `ClientCache`. They have been moved successfully for prototype purposes and to collect experience with the Eclipse refactoring functionality.

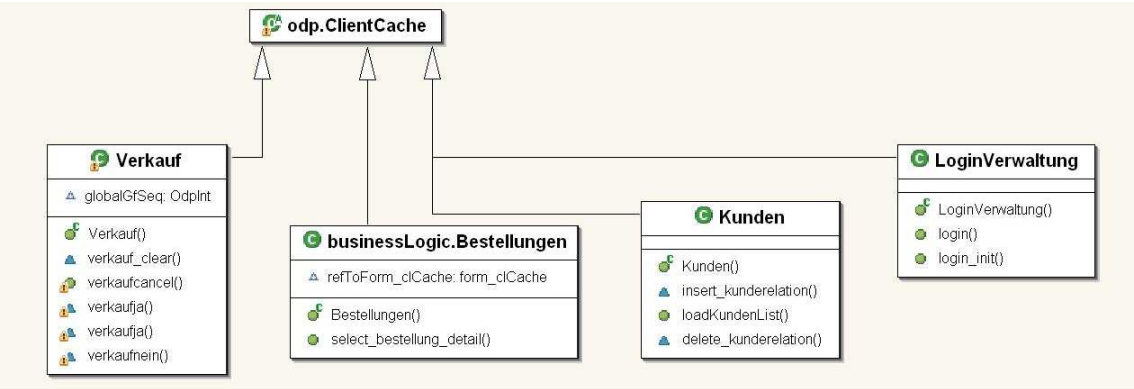

Figure 5.3: UML class diagram of classes that contain methods moved from the `form_clCache` class.

The coarse refactoring is done after the `form_clCache` has been split into communication, application model and operation classes. The new structure of the ticket sales component will imply further refactorings to optimize it. A part of the moved methods are too big in terms of LOC, refactorings to split them are necessary. Large methods typically access the view to get input data from a form or to display results obtained from the database. Such code sequences

are candidates to be extracted into a method. The refactoring has to be stopped when the target controller architecture shown in Figure 5.1 has been achieved and the classes constituting it are have become more comfortable to maintain, that is if they contain a reasonable amount of LOC and are well structured.

`ClientCache` contains variables to set the authorization of features to a user at program startup. They have to be moved to a class in the application model.

The controller refactoring described mainly consists of *move method*, *extract method*, and *move field* refactoring steps. Eclipse masters all of them sufficiently, but there still remains handwork to do. The procedure to move a method is:

- Move attributes(s) hat are used by the method in a new class
- Change the initialization methods of the originating class to refer to instances of the new class
- Move the method(s) in the new class
- Update the method accesses to the new class

5.6 Documentation

The last reengineering steps are to establish a technical documentation of the ticket sales component and to update the user manual. The goal of the technical documentation is to give a quick overview of the component to developers and maintainers who get in touch with it for the first time. The documentation has to contain architectural views and use cases of the component. UML diagrams such as state, sequence, and collaboration have to be used to explain the ticket sales component.

Chapter 6

Conclusions and Outlook

6.1 Contribution

This thesis applied techniques and tools to an object-oriented legacy system to develop a reengineering plan for the company. The thesis can be used as practical reference for the organizational proceeding in client-side software reengineering projects. The work delivers detailed examples on how to execute the steps of the reengineering process.

6.2 Experiences Made

Reengineering should be done on a regular basis, and not when it becomes unavoidable. Legacy system reengineering is labor intensive and complex due to the various aspects that have to be considered. The reengineering work done with tools proved that they are of use for the reengineering process, but they are not crucial for success. The key factors are the organization of the project, reengineering experience, and technical knowledge. The most difficult parts were selectively acquire a sufficient understanding the system and make the right reengineering decisions that shape the future of the system. The applied pattern-oriented reengineering framework proved to be effective to coordinate the activities involved in reengineering. The today available features of commercial, academic, and open source reengineering tools have a lot of potential to be improved. As software systems tend to grow bigger and more complex, the need of tool assistance needs to rise to successfully reengineer them.

6.3 Outlook

The company will implement the proposed changes of the ticket sales component. Fine-grained reengineering steps that make the component more maintainable will be undertaken, such as detection of duplicated code and transformation of long conditional statements to polymorphisms to use the strength of object-oriented programming.

Appendix A

Source Code Listings

A.1 JWS Substitute Data Types

Listing A.1 shows the substitute type for the JWS data type `Attribute` which is needed for the substitute type `AttributeList` implemented in Listing A.2.

A.2 Mapper Class

Listing A.3 shows a mapper class for the JWS type `Group`.

A.3 Move Method Refactoring Example

The `StatistikController` class listed in Listing A.4 to hold the moved method `besuchs-statistik` is derived from the `ClientCache` communication class. In a further refactoring step, the communication has to be separated from the view access. The `besuchs statistik` method can be changed to obtain a parameter which pases the data needed from the view. The `besuchsstatistik` method can be changed to return `OdpVector` that holds the results from the data base which enables to move the `showTextDialog` out of the communication method. The `form_clCache` reference given to `besuchsstatistik` is then needed anymore, since the `recvErrorList()`, `add(Odp odp)`, and `get(int i)` methods are inherited from `Client Cache`.

Note: The example contains the JWS `AttributeMmanager` class because it has not been substituted in the `form_clCache` class in the mapper prototype.

```
package jwsSubstitutes;

/*
 * Substitute Type for JWS Atrribute
 * Needs to store: ( "java.lang.String", odpsTitle.value, 6)
 */
public class Attribute {

   public String attString = null;
   public Object attrOjbect = null;
   public int attrInt = 0;

  Attribute(String attStringValue, Object attrOjbectValue,
  int attrIntValue){
   attString = attStringValue;
   attrOjbect = attrOjbectValue;
   attrInt = attrIntValue;
   }
   // Sets the value of the attribute.
    public void setValue(Object value){
       attrOjbect = value;
    }

    public Object getValue(){
       return this.attrOjbect;
    }
}
```

Listing A.1: Substitute type for the JWS type `Attribute`.

```
import java.util.Hashtable;

/*
 * Substitute Type for JWS AtrributeList
 */

public class AttributeList {

   private Hashtable attributetable = new Hashtable() ;

   public Attribute get(String name){
      return ((Attribute)attributetable.get(name));
   }

   public String resolveAlias(String name){
      if (attributetable.containsKey(name))
         return name;
      else return null;
   }

   public void add(String name, String type, Object defaultValue,
   int flag){
       Attribute m = new Attribute(type, defaultValue, flag);
       attributetable.put(name, m );
    }

   public void add(String name, String type, Object defaultValue){
      add(name, type, defaultValue, 0);
   }
}
```

Listing A.2: Substitute type for the JWS type `AttributeList`.

```
import java.awt.Frame;

/**
 * Mapper class for
 * sunsoft.jws.visual.rt.base.Group
 */

public class GroupM extends AttributeManagerM {

   public sunsoft.jws.visual.rt.base.Group jwsGroupReference;

   GroupM(sunsoft.jws.visual.rt.base.Group jwsGroupRef){

      jwsGroupReference = jwsGroupRef; }

   /**
    * JWS Group.getFrame:
    * Returns the first frame found while
    * traversing up the group tree.
    * If no frame is found, then the top
    * level frame is returned.
    *
    */
    public Frame getFrame(){
       return jwsGroupReference.getFrame();
    }

public void set(String key, Object value){

   jwsGroupReference.set(key, value);
}

public Object get(String key){

   return(jwsGroupReference.get(key));
}

}
```

Listing A.3: A mapper class containing methods to delegate the ODP method accesses to the view.

```
package businessLogic;
import form_clCache;
import odp.CacheException;
import odp.ClientCache;
import odp.ObjectOdp;
import odp.OdpInt;
import odp.OdpString;
import odp.OdpVector;
import sunsoft.jws.visual.rt.base.AttributeManager;

public class StatistikController extends ClientCache {

   /*
    * ... abstract inherited methods are not shown
    /*

   public void besuchsstatistik(form_clCache cache)
    throws CacheException {
      if (cache.form_clCacheprinttrace)
      { System.out.println("form_clCache.besuchsstatistik");}
      cache.clearParameter();
      cache.add(new OdpString("BESUCHSSTATISTIK"));
      cache.add(new OdpInt(iLangCode));
      cache.add(new OdpInt((Integer)cache.
      fGUI.kundenShortKeyListBestellung.get("selectedKey")));

      cache.start(ObjectOdp.TRANSFER_SYNCHRON);
      cache.recvErrorList();
      cache.showTextDialog((OdpVector)cache.get(1));
   }

}
```

Listing A.4: The newly created `StatistikController` class which holds the moved method `besuchsstatistik`.

Appendix B

List of Used Software Tools

- **Eclipse** http://www.eclipse.org
- **Creole:** http://www.thechiselgroup.org/
- **CodeCrawler** http://www.iam.unibe.ch/∼scg/Research/CodeCrawler/
- **jFamix** http://jfamix.sourceforge.net/eclipse/index.html
- **Omondo UML** http://www.omondo.com/
- **Java Metrics** http://sourceforge.net/projects/metrics

References

[Arn94] Robert S. Arnold. Software Reengineering: A Quick History. volume 37(5), pages 13–14. Communications of the ACM, 1994.

[BH04] Judith Bishop and Nigel Horspool. Developing principles of GUI programming using views. In *Proceedings of the 35th technical symposium on Computer science education (SIGCSE-04)*, volume 36, 1 of *SIGCSE Bulletin*, pages 373–377, New York, 2004. ACM Press.

[CC90] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. In *IEEE Software*, pages 23–32, Amsterdam, The Netherlands, September 1990. IEEE Computer Society.

[CM04] N. Chen and K. K. Ma. Java's future: Challenge and opportunity. *IT Professional*, 6(4):19–27, 2004.

[DDN03] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 340 Pine Street, Sixth Floor, San Francisco, CA 94104, USA, 2003.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, Massachusetts, 1999.

[FPG03a] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and Relating Bug Report Data for Feature Tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 90–99, Victoria, B.C., Canada, November 2003. IEEE Computer Society Press.

[FPG03b] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 23–32, Amsterdam, Netherlands, September 2003. IEEE Computer Society Press.

[HKR03] Christopher J. Howell, Gregory M. Kapfhammer, and Robert S. Roos. An examination of the run-time performance of gui creation frameworks. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 171–176, New York, NY, USA, 2003. Computer Science Press, Inc.

[Lan03] Michele Lanza. CodeCrawler – A Lightweight Software Visualization Tool. In *Proceedings of VisSoft 2003 (2nd International Workshop on Visualizing Software for Understanding and Analysis)*, pages 51–52. IEEE Comptuer Society Press, 2003.

[Lan04] Michele Lanza. CodeCrawler — Polymetric Views in Action. In *Proceedings of ASE 2004 (19th IEEE International Conference on Automated Software Engineering)*, pages 394–395, 2004.

[Mem02] Atif M. Memon. Gui testing: Pitfalls and process. *IEEE Computer*, 35(8):87–88, 2002.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The Story of Moose: an Agile Reengineering Environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press.

[otICS98] Software Engineering Standards Committee of the IEEE Computer Society. IEEE Std. 1219-1998: IEEE Standard for Software Maintenance. New York, USA, 1998. IEEE.

[PGFL05] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing Multiple Evolution Metrics. In *Proceedings of the ACM Symposium on Software Visualization*, pages 67–75, St. Louis, Missouri, 2005. ACM Press.

[RR02] Claudio Riva and Jordi Vidal Rodriguez. Combining Static and Dynamic Views for Architecture Reconstruction. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2002.

[Szy98] Clemens A. Szyperski. *Component Software*. Addison Wesley, 1998.