TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Diplomarbeit

Master's Thesis

Migrating Business Software Applications based on IGS (Inova Q Generator System) from Windows/VB to Linux/Java

ausgeführt
unter der Anleitung von

A.o.Univ.Prof. DI. Dr. Franz Puntigam

am

Institut für Computersprachen

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

durch

Christian Putsche

Matr.Nr: 0025328
Ultzmanngasse 39/1/10
A - 1220 Wien

Wien, im März 2008 .....

Zusammenfassung

In den letzten Jahren ist vor allem im Bereich der öffentlichen Verwaltung und Behörden die Nachfrage nach Open Source Lösungen gestiegen. Dies stellt an die in diesem Umfeld tätigen IT-Unternehmen die Anforderung entweder neue plattformunabhängige Produkte zu erstellen, oder aber bereits Bestehendes zu migrieren. Diese Diplomarbeit befasst sich mit der Migration eines bestehenden Systems zur Generierung von Software-Applikationen. Die generierten DatenbankApplikationen holen sich ihre Daten aus einer SQL Datenbank, sind in Visual Studio geschrieben und laufen unter einem Microsoft Betriebssystem. Die Herausforderung an den Migrationsprozess besteht darin, diese "Monokultur" aufeinander bestens abgestimmter Komponenten in ein offenes System zu übertragen. Ausgehend von einer Analyse der vorhandenen Techniken im Bereich Software-Migration und Reengineering wird versucht, den Generator dahingehend anzupassen, dass er in Aussehen, Verhalten und Funktionalität äquivalente Programme generiert, die mit Hilfe der Sprache Java unter dem Betriebssystem Linux einsetzbar sind. Die Vorgehensweise und die dabei auftretenden Probleme und Unzulänglichkeiten werden dokumentiert und bewertet.

Abstract

In the last years the demand for Open Source solutions increased considerably, especially in the area of public management and authorities. Resulting from this, IT enterprises active in this business segment either have to provide new platform independent products or migrate already existing ones. This thesis deals with the migration of an existing system that generates software applications. The generated database applications get the needed data from an SQL database, are written in Visual Studio, and run on a Microsoft operating system. The challenge to the migration process consists in transferring this "monoculture" of well coordinated components into an open system. Outgoing from an analysis of the available technologies in the area of software migration and reengineering we try to adapt the generator in order to generate Java programs executable on Linux. The resulting application must be equivalent in appearance, behaviour, and functionality. The chosen migration approach and the problems and inadequacies arising hereby are documented and evaluated.

Acknowledgements

First of all I want to express my sincere gratitude to my supervisor at Inova Q, Mag. Wilhelm Pfendt, for giving me the chance to work on this interesting and challenging project. In the beginning he taught me the principles of effective and well-structured programming, subsequently letting me work quite independently and in charge of the project, but always giving me the support I needed. I do appreciate the technological aspects of this project, for I have received complete insight into an astonishing and elaborate software generating tool named IGS. The time spent at Inova Q was both very productive and educational and will be of great importance for my further advancement.

I am deeply indebted to my supervisor at Technical University of Vienna, Prof. DI. Dr. Franz Puntigam, whose comments and criticism significantly contributed to the quality of this thesis. His stimulating suggestions, his patience and encouragement guided me through all phases of research and writing. Without his inspiration and care the thesis would likely not have matured, and its present content reflects the influence of his invaluable feedback.

Contents

1 Introduction

1.1 Motivation

With a growing market for the Linux operating system it becomes increasingly important to software enterprises to offer their essentially Windows-based software and products also for the Linux platform. In the field of server systems, Linux is already a popular choice of IT experts. Today's server market is impossible to imagine without Linux. If stability, scalability and openness are requests to meet, there is no alternative.

In the field of business applications and desktop solutions, where Windows is still predominant, the development is just at the beginning of the possibilities. As the increasing presence of Linux in the press and a variety of current studies on the topic of using Linux in enterprises shows, the demand for an open source solution increased considerably over the last few years [Brucherseifer 2004].

The two primary key factors for the change from Windows to Linux are cost saving and freedom from licensing dominance occurring in any proprietary environment. The switch to Linux enables large and small companies world wide to reduce their most significant overhead costs with relative ease.

Resulting from this change, the ability to offer software for all relevant platforms is increasing in importance, and can be achieved by either developing new platform independent products or by migrating already existing and approved ones.

This thesis deals with the more challenging second aspect. The challenge hereby consists in migrating components designed for a proprietary system like Windows to their appropriate components for an open system (in our case Linux).

The goal of this work is to answer the following question:

> *Which problems arise when migrating typical business software applications developed for Windows to cross-platform applications equivalent in performance, appearance and functionality, and how can we address this problems?*

We will explore this topic on the basis of migrating applications generated by the IGS system from Windows to Linux. IGS is a system generating business applications with connection to databases, developed by Inova Q, a Viennese company producing customised software solutions.

Furthermore we will look at the quality of existing platform independent toolkits, the use of free software in commercial projects, the respective GUI integration, and the approach chosen for the migration process.

1.2 Introduction to IGS

The information about IGS outlined in this section originates from [Inova Q 2004]. IGS (Inova Q Generator System) is a software development tool that on the basis of a data model and well defined rules, written in the description language IFL (Inova Q Form Language), automatically generates database applications.

IGS has the goal to make customised software development:

- more accurate,
- reasonably priced,
- faster, and
- to a large extent independent from staff skilled in programming.

IGS technology tries to meet the typical requirements in the range of client-specific application development:

- High efforts in analysis to understand the consumer's business.
- Relational database as foundation.
- Reuse usually not possible or only possible in a small scale.
- The resulting application must be suitable for mission critical services.
- Frequent changes and enhancements.
- Short throughput time especially after completion of the analysis.
- No proper quality assurance with real data and simulated users is possible. In general, the product is passed directly from the software developer to the user.

IGS adresses related problems by an innovative approach. It does not generate applications to their full extend, but builds a framework for the final application. The generated application is fully operational and can be used right from the start. The application provides a GUI with the needed functionalities (Print, Copy, Paste, Delete, Search, and much more). Users can manipulate data within the range of the defined application types (see definitions later in this section). Through existing Call-In- and Call-Out-Interfaces additional functionality can easily be added by the developer.

IGS takes over laborious and intricate work like establishing a robust and error-free connection to the database and developing the GUI with its functionalities. We use the term "framework" because the generated application is standardized in its behaviour and look-alike, and to overcome this peculiarity (of generated software in general) the application provides interfaces for the software developer to do the final adjustments. The great virtue is that the software developer need not develop (and basically test) the application from scratch, but can spend his/her time doing refinements.

The currently established software development process is as follows:

1. (Pre-)requirement analysis either by the costumer, by a corporate consultant or by another consultant (e.g., an IT-consultant having insight in the particular business skills).
2. Technical analysis (Conceptual Design) by the developer.
3. Realisation of the software product followed by internal testing and integration testing (Alpha Testing).
4. Initial start up and pilot phase (Beta Testing) which leads to final application establishment.
5. Maintenance and adaptation.

Most problems hereby arise from breaks in the information flow between the individual steps. IGS tends to eliminate such breaks between steps two and three. This is achieved by generating a large part of the application completely automatically. Thus, the developer's main focus is moved to the analysis phase, and time and effort concerning programming are reduced. In addition, maintenance and adaptation become much easier.

By the use of IGS particularly complex and intricate (but constant) development steps can be automated. Therefore, even employees with little coding know-how are able to produce software products in convincing quality. Since efficient software developers are rare, IGS reduces the need of software developers in favour of analysts and consultants.

The following target groups mainly benefit from the resulting potential:

- Independent software houses that develop costumer-related software.
- IT departments of mid-sized and great companies developing for themselves and/or other companies.
- Companies with few or not so highly skilled software developers.

The essential part of IGS is the generation of program code that can directly be translated into an executable program. This program code implements on the one hand the user interface and on the other hand fundamental business logic and database access routines (3-Tier-Architecture).

At present the following target platforms and application development systems are supported:

- Microsoft Visual Studio 6.0, target platform Win32
- Microsoft Visual Studio 2005 (in progress), target platform Win32

1.3 Thesis overview

The reason for establishing this project is to enable IGS to create platform-independent products. This goal shall be achieved by the use of the programming language Java provided that this is technically and commercially feasible. As the project's target platform Linux seems to be the most suitable choice because it is not proprietary and is freely available.

A further requirement is that the application, after being generated, shall support adjustments within predefined limits and through clearly structured methods and interfaces. Therefore, the generated code must have a consistent, well-structured design. Developers must be able to make any necessary adjustments in a standardized way and (as far as possible) independent from the content of the program.

Arising from these requests there will be the following steps in the project:

Step 1 - Identification and implementation of Java base classes: In this first step we must find an adequate Java model which satisfies the functional and technical requirements of the project. At this stage in the project the internal functionality of the generator need not to be understood to its full extent. Only a slight insight should exist in what is the generator's input and what is the resulting outcome.

The automatically generated application has, based on the given input, a predetermined appearance and functionality. The initial challenge will be to examine GUI (graphical user interface) toolkits for Java, whether they meet the above stated requirements. GUI widgets such as text boxes, buttons, split-panes, and tables have to be examined. In particular the differences in behaviour and appearance between them and the currently used Visual Studio components must be clarified. If there are some differences, we have to check how we can customise the Java components to achieve that in the long run they will be similar to the given Visual Basic components.

To achieve this step's goal we will mainly make use of prototyping. The prototypes will help us to identify inconveniencies and differences and provide a basis for the development of templates in step three. At the end of this first step it has to be clear whether the desired functionality can be realised with Java and which GUI toolkit has to be used.

Step 2 - Elaboration of interfaces for manual adaptations: This second step of the project consists of elaborating the interfaces through which the interactions between manual adaptations and automatically generated code are possible. In this connection, interfaces for three types of subsystems have to be provided:

- **Data Entry Subsystems:** Data entry subsystems are subsystems for data management (input and output of data). They can access the underlying database either in reading or in writing mode.

- **Data Browsing Subsystems:** Data browsing subsystems are subsystems for data output (display and search of data). They can access the database only in reading mode.
- **Parameter Subsystems:** Parameter subsystems are subsystems which are not coupled with an underlying database.

For each of these three subsystem types we have to create *Call-in-interfaces* and *Call-out-interfaces*. The former define public methods and functions which can be invoked from outside of the generated code, the latter provide methods and functions to add manually written code to the application.

At the end of step two the fundamental architecture of the generated software shall be identified. Figure 1 shows the software development schema of IGS.

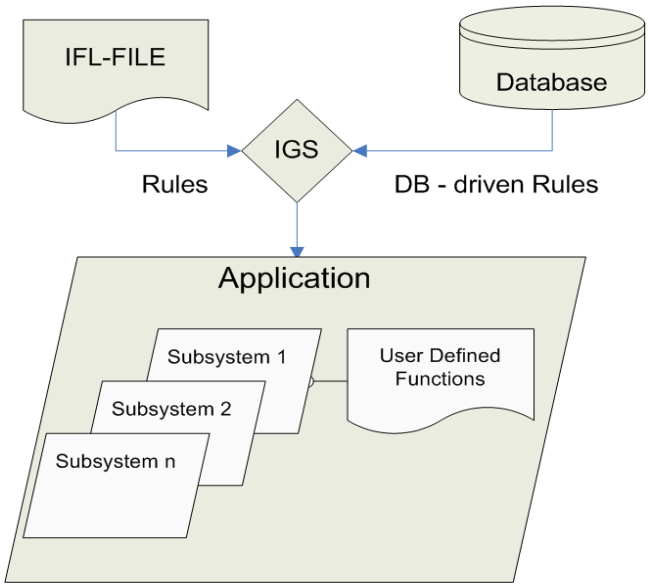


Figure 1: IGS Software Development Schema

Step 3 - Adaptation of IGS: The last step consists of two tasks. First, the templates for the particular Java controls and the above mentioned subsystems must be developed. This will be done on the basis of the prototypes elaborated in step one. These templates will be filled by IGS with the appropriate data and serve as fragments to build together the specified application.

Subsequently the used framework must be extended. It has to produce applications written in Visual Basic and also applications written in Java. In essence we will develop the Java code generator in this step. This task tends not to be too elaborate because we will only have to adapt the already existing code generator without expanding the process logic significantly.

2 Software Migration and Reengineering

This section will give an adequate overview over the most important techniques in the field of software migration and reengineering. Subsection 2.1 provides short definitions of the techniques whereas subsections 2.2 to 2.6 describe them in more detail.

2.1 Notions

Software Migration: The notion of software migration stands for the changing of basic software and/or the transformation of data into new formats. Moreover, software migration is integration of old technology into new technology by extensive use of already existing technologies [Müller 1997].

Reverse Engineering: Reverse Engineering means identification of system components and their correlations. The aim is to get the descriptions of the system in another form or on a higher abstraction level [Chikofsky et al. 1990].

Reengineering: Reengineering is examination and modification of a programming system with the goal to reestablish it in a new form and subsequently implement this form persistently [Chikofsky et al. 1990].

Prototyping: Prototyping is the process of building a model of a system. In terms of an information system, prototypes are employed to help system designers to build an information system for end users intuitive and easy enough to be used. Prototyping is an iterative process and is part of the analysis phase of the systems development life cycle [Sauter 1999].

Restructuring: Restructuring means the transformation between representation formalisms without changing the functionality, respectively the behaviour which can be observed from outside [Müller 1997]. Restructuring can either be the transformation from unstructured programs to structured programs by eliminating Goto's, or merging of multiple if's to a case construct, or the normalization of data by eliminating synonyms and homonyms.

Application Understanding: Application understanding is the identification of structures and attributes of the particular system, and comprehends the programs, databases, libraries and files being used. Another field of interest is the interdependences between the particular programs of the application [Müller 1997].

Program Understanding: Program understanding means the process of understanding the internal technical functionality of a program. This process requires understanding of the program's call hierarchy and the branches in control flow [Müller 1997]. Moreover, it should be known where data is read and where it is written, and what effects are caused by specific modifications. When we mention the term program understanding in this thesis we mean application understanding and program understanding together.

Language Conversion: The transformation of a program from one language to another conserving the genuine semantics of the former language is called language conversion [Müller 1997].

2.2 Software Migration

2.2.1 Aspects of Software Migration

Software Migration is the transformation of a software system from one into another (target) environment. It is a pure technical transformation with a strict definition of the specifications. The legacy system gives a well-defined characterization of the system functionality to be achieved [Gimnich et al. 2005]. The functionality can be verified by a regression test after the migration has been done [Sneed et al. 2004].

The gathering momentum for such migration is usually altered requirements on software systems. Like organizational requirements on business processes, enhancement of software functionality, and external requirements. These requirements lead to non-functional requirements of the system environment, which are the objects to deal with in the software migration process.

Software migration is part of comprehensive reengineering activities where the available software will be transferred to a new environment without changing persisting functionality. Thereby according to [Gimnich et al. 2005] there must be paid regard to technical aspects as well as to aspects concerning the software.

Technical aspects are:

- Changing of hardware environment (from Mainframe to Unix).
- Changing of runtime environment (changes in the system software, such as OS, DBMS).
- Changing of software architecture (from monolithic systems to multilayered architectures [Hasselbring et al. 2004]).
- Changing of application development system (changing of programming environment).

[Sneed et al. 2004] adds following points to the group of software aspects:

- Data
- Userinterfaces
- Programs

Data migration means the transfer of the stored data as well as the underlying data structures and schematas into the new system, user interface migration includes the transformation of the interaction components for the user, and program migration addresses transformation of executable program logic. Software migration caused by these two groups of aspects likely involve other software migration processes since there is a strong interdependence between the individual aspects.

Figure 2 sketches the interdependences between the particular migration types.

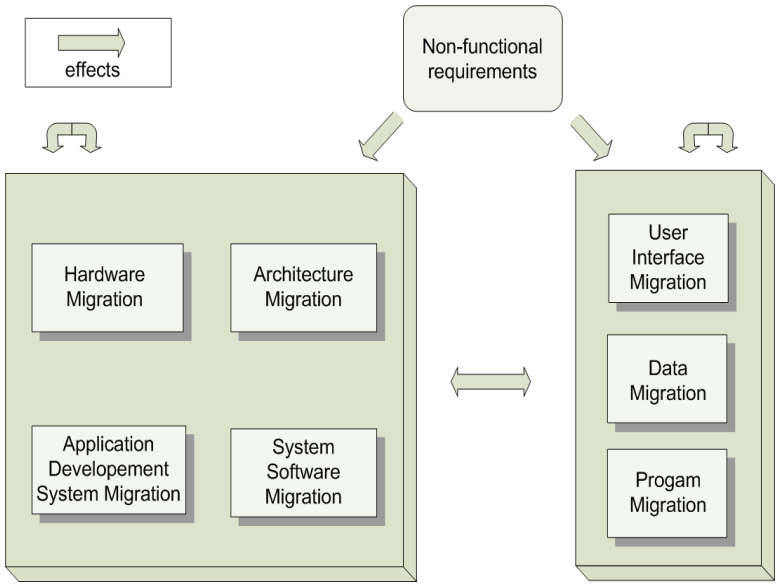


Figure 2: Migration Types

2.2.2 Steps of Software Migration

Regarding to [Gimnich et al. 2005] the software migration process consists of the following steps:

- **Identifying the migration strategy:** Basically there are three different strategies to transfer legacy systems into their new environment. The first strategy is complete redevelopment, where the system is directly implemented in the new environment. The second (possible) strategy is wrapping. Following this strategy the old system remains in the original state and the new system contains only interfaces to gain access to it. Last but not least there is conversion. Conversion transforms the old system into a new one in the target environment.

- **Defining the target environment:** Preceding the transformation we must map out the hardware environment, the system software, the application development system, the software architecture, as well as the affected aspects concerning interfaces, data, and programs.

- **Analysing the differences:** This step implies an accurate examination of the differences between legacy system and system to develop with regard to use of interfaces, access techniques on data, and the linking of executable programs through batch programs. If the transformation process makes use of language conversion, the differences between the language and the data structures have to be studied. The validation of the legacy system regarding to its general ability to be migrated is a substantial precondition for the realisation of a migration project.

- **Defining the complexity of migration:** A qualitative and quantitative analysis of the objects concerned by migration and their interdependences, as well as comparisons with migration projects already made, permits us to identify guidelines for the complexity of the planed migration project.

- **Specifying the transformation:** Ahead of the implementation of migration the necessary transformations have to be mapped out and the rules for conversion have to be defined.

- **Implementing the transformation:** This step performs the migration and takes place on basis of the decisions made in the steps before. Hereby the interrelated modules must be migrated in iterative manner. [Sneed et al. 2004] recommends beginning with data migration, then program migration and finally user interface migration.

- **Delivering the migrated system:** Basically there are two strategies for delivery of a system. The complete delivery of the whole system all at once, called Cold Turkey or Big Bang Strategy, or the incremental, step-by-step delivery, called Chicken Little Strategy. In the latter case there will be a temporary co-existence of the legacy system and newly developed system. Hence adequate synchronisation measures have to be provided.

- **Migrating the staff:** The software developers who implement the migration as well as the employees who will work later on with the migrated system have to be skilled adequately. For the software developers this means they have to learn methods and techniques of software migration and be trained in manipulation of the legacy and the target system.

- **Assuring quality:** This step guarantees the functional equivalence of legacy and target system. [Sneed et al. 2004] points this step out to be the greatest expense factor of a migration project.

2.3 Software Reengineering

Software Reengineering consists of analysing and changing an existing system with the aim to implement the system in a new, changed form. This adjustment of the system results mainly from new requirements to the software. Such reengineering objectives are facilitating the maintenance of existing software products, extracting reusable components from existing software products for incorporation into new systems or extracting design information from an existing system to bring it under the control of a new environment.

There are two different approaches to reengineering. The pure reengineering approach only restructures the system without adding new functionality. Extended reengineering first analyses and restructures the old system in order to add new functionality or change already existing functionality.

2.4 Software Prototyping

In the software prototyping process an incomplete model of the latter fully featured software program is created. This model allows evaluation on behalf of the clients and gives them a first idea of the prospective program.

The merits of software prototyping are manifold. Most important is the gain of early feedback to expose misunderstandings between clients and users on the one hand and developers on the other hand. It is possible to compare whether the developed software matches the requirements. Missing services can be detected and confusing services can be identified. Prototyping allows us to derive enhanced and more realistic software specifications. Furthermore, users receive an early available working system in the initial stage of the software developing process. Finally, a prototype allows insight into initial project estimates and points out whether deadlines and milestones can be met. In the long run software prototyping helps to avoid great expenses and difficulties in changing the finished product.

But there are some important drawbacks, too. In a software prototyping process there is often insufficient analysis. The focus on the limited prototype leads to inadequate analysis of the complete project. By focussing too much on the prototype, better solutions may be overlooked, specifications are likely to be incomplete, and therefore the final product will be poorly engineered and hard to maintain. As another drawback the user may be confused about prototype and real application. Users may expect the prototype to exactly model the final system and may become used to features finally being removed. Resulting wishes made by users can lead to uncontrolled changes in the project's scope. Moreover, these changes can occur at a time when the scope of a project is not properly defined, documented, or controlled. If developers produce a too complex prototype or loose time in debates over details of the prototype, they will delay the implementation of the final product and development time of the finished system will be raised to an excessive amount.

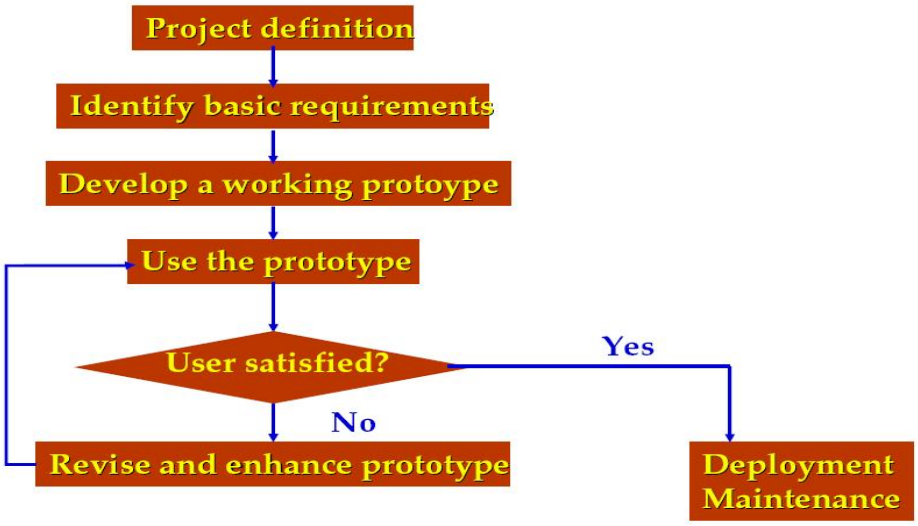


Figure 3: Prototyping Process

Due to Figure 3 taken from [Malone 2005] the process of prototyping involves the following steps:

- **Identify basic requirements:** This means determining the basic requirements including the input and output information desired.
- **Develop Initial Prototype:** The initial prototype including only the user interfaces is developed.
- **Review:** The prototype will be examined and feedback on additions or changes is provided.
- **Revise and Enhance:** Using the feedback obtained on the former steps both the specifications and the prototype can be improved.

2.4.1 Evolutionary Prototyping

Evolutionary prototyping is an approach to system development where an initial prototype is produced and redefined through a number of stages until the final system is established. [Sommerville 2000].

The major task is to make a working system available to the client/user. The development of the prototype begins with the specification requirements most easily to manage. Evolutionary prototyping is used for systems where specifications are not clear from the outset, as e.g., AI systems or user interfaces. The used techniques support rapid development of the desired output mainly through rapid system iterations. Conveniences of this approach are the possibility of quickly handing over the product to the client if necessary and a high familiarity of end-users with the product. This familiarity not only facilitates that the finished product satisfies the expectations of users, but moreover

causes the user to already establish a relationship to the system. Specification, design and implementation of the prototype are developed simultaneously.

The major drawbacks are potential maintenance problems resulting from permanent changes. These changes can corrupt the system structure and lead to an inconsistent design of the end product.

2.4.2 Throw-away Prototyping

A prototype is usually a practical implementation of the system produced only for the purpose of discovering requirements problems. Once the problems are located the prototype is discarded. The system is then developed using other development processes [Sommerville 2000].

Throw-away prototyping mainly provides an instrument to screen the specifications to see whether they are feasible, and then adapt or even discard them if necessary. In contrast to evolutionary prototyping, the development of the prototype begins with specification requirements. They are the most difficult part to implement and therefore the risk of not defining them properly is reduced.

The prototype is developed in favour to identify the initial specification. After delivery to the users and evaluation and adaptation of the requirements it is thrown away. The prototype has no affinity to the final product, since many system characteristics are not implemented yet. This type of prototype will be poorly structured and documented and therefore hard to maintain.

Figure 4 shows the two main approaches to prototyping.

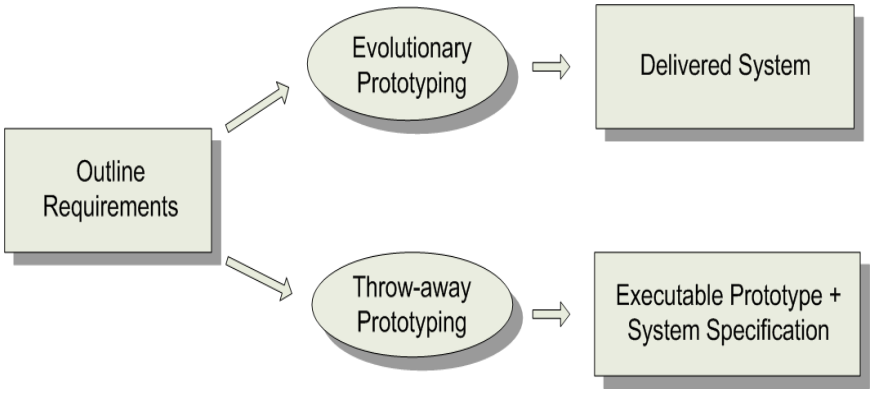


Figure 4: The two approaches to prototyping

2.5 Program Understanding

Program understanding is an essential precondition for migration and makes use of resources to a great extent. The cause is the lack of confidence with the software to migrate.

Most of the time in the migration process is needed for understanding where changes have to be made to gain the desired results. The actual implementation itself is not so intricate. The study made by Fjeldstad and Hamlen in [Fjeldstad et al. 1979] analyzes time and effort of the single activities involved in the software maintenance process.

Table 1 shows the outcome of this study. The Summary of the first three values shows that program understanding amounts to 47%, whereas modification (consisting of writing code and documentation) makes up only 25%. The remaining part is testing with a percentage of 28%. In other words, software professionals spend at least half of their time reading and analysing software in order to understand it [Corbi 1990].

ACTIVITY	RATIO
understanding request	18%
understanding documentation	6%
understanding of code	23%
implementing	19%
testing	28%
adapting documentation	6%

Table 1: Time and effort of maintenance activities

[Müller 1997] recognizes three theories to approach program understanding:

- **Bottom-Up Understanding:** In a bottom-up approach we first try to understand individual base elements of the system in great detail. These elements are then linked together to gain understanding of larger subsystems, which then in turn are linked (sometimes in many levels) until the complete system is understood. In the beginning the system is very small and then grows in complexity and completeness. Thus, starting from the source code of the legacy system we get to an abstraction on a higher level.

- **Top-down Understanding:** In a top-down approach program understanding starts from the expectation we have in the behaviour of the system. From this point of view we search for particular estimated structures and after recognizing them we will put them together as in puzzle. A top-down approach treats the software as a set of black boxes that can be reformulated for integration with other systems. The black-box approach is preferred because the technology for interfacing and integrating is developing much faster than the technology for program understanding [Weiderman et al. 1997]. However, black boxes can fail to illustrate elementary mechanisms or to be detailed enough to realistically validate the model.

- **Opportunistic understanding:** Opportunistic understanding is a join of the two former approaches. It uses both algorithmic and coding knowledge as well as knowledge about domains. The term opportunistic states that everywhere in the process of understanding the legacy program it is possible to choose the more convenient and promising approach among the two.

2.6 Language Conversion

Language conversion, or source-to-source-translation as often termed, is the translation of a program in language L1 to a functionally equivalent program in language L2, where L1 and L2 are generally different from each other.

The main reasons for language conversion are:

- Replacement of hardware or operating system.
- Desired enhancement in efficiency.
- Desired maintenance reduction of the new language.

There are two types of conversion schemas:

- Translation via transliteration and refinement.
- Translation via abstraction and reimplementation.

2.6.1 Translation via transliteration and refinement

This is a translation process in two steps where assignments in the source language are converted to assignments in the target language. The source language first is literally translated step-by-step, assignment by assignment into an intermediate program and then refined and optimized.

Benefits are the straightforwardness of the transformation schema, and the assurance of correctness and efficiency through a divide-and-conquer approach. Through locality of conversion it gains correctness and through refinement it gains efficiency since this translation makes use of optimisation techniques of the target language.

The main disadvantage hereby is missing improvement in the program code because this conversion will not considerably make use of program language constructs of the target language that are not available in the source language. To achieve improvements a broader, more global perspective is needed. Figure 5 shows the two-stage translation process.

Figure 5: Translation via transliteration and refinement

2.6.2 Translation via abstraction and reimplementation

This kind of translation is like the translation reviewed above also done in two steps. Differently from the former one this translation is not made locally but globally, and not horizontally, from source to source, but vertically from source program to a higher abstraction layer and then to the target program. Figure 6 shows the model of this approach.

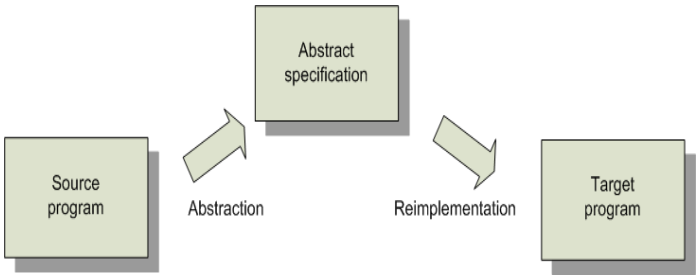


Figure 6: Translation via abstraction and reimplementation

The steps to be taken are:

- Global analysis of the source program.
- Setting up an abstract description.
- Mapping the description to the target language.

As an advantage, translation via abstraction and reimplementation eliminates the shortcomings of the source program, and the resulting program will rather be an improvement of the original one. The drawback is the large complexity of implementing the abstraction process.

According to [Terenkhov et al. 2000] the availability of constructions facilitating the expression of a solution determines how easy it is to formulate a solution for a particular problem. If such constructions are available in the desired language, they are called native language constructions. For example if a conditional problem language construct must be expressed, a language supporting conditional constructs is more convenient than one that doesn't. The latter language must simulate the conditional construct. Therefore, these constructs are called simulated language constructs. The Language conversion problem amounts to mapping its native and simulated constructs to hopefully only native constructs.

Language conversion must consider who in the end will work with and has to maintain the developed product. If these persons were familiar with the original software, the conversion must be done in a mode ensuring the new system being as similar as possible to the original system to achieve recognition of original code. If the users are new, the target language idioms must be used to assure the originated code being as natural as possible in this particular language.

3 Java

This section outlines why we choose Java as programming language (Subsection 3.1) for this project and which GUI toolkit (Subsection 3.2) is the best suitable for our needs. In Subsection 3.3 we take a closer look on the Model-View-Controller paradigm because it is fundamental when developing applications using a GUI. Subsection 3.4 explains how MVC is solved in Swing and introduces some Swing features. Subsection 3.5 covers a general programming mechanism called Reflection. Due to its great importance to this project and the fact that it is a rather advanced feature, we explain the fundamentals of reflection. In Subsection 3.6 we take a look at alternatives to Java and evaluate their feasibility.

3.1 Why Java?

The properties making the use of the programming language Java attractive are present in other programming languages too. Some of these languages are considered to be even better suited for certain types of applications than Java. But the great virtue of Java is to bring all these properties together in one language [Reilly 1999]. This benefit is the decisive factor of using Java as programming language for this project.

The main Java virtues are in detail:

- **Java is object-oriented:** This means that a program is constructed from objects. Objects are reusable software components containing both the data and the functions operating on this data. Programs using object-orientation are easier understandable and more reliable than programs written in a procedural language. Since the needed data and the functions using this data are part of the same object, it is assured that all relevant information is local.
- **Java is portable:** The Java compiler produces machine-neutral code (the so-called Java byte code) which on the target platform gets interpreted by the Java Virtual Machine (JVM). Due to this mechanism Java is platform-neutral, and as result every platform supporting the JVM can run the byte code of a Java application. The source code of the application need not be present on the host platform. The source code is protected from modification since only the byte code is available to users. But more significant is the fact that Java code can be compiled once and run on any machine and operating system combination supporting the JVM ("Write once, run anywhere"). Thus, Java can run on Unix, Linux, Windows, Macintosh, and even the Palm Pilot. Moreover, it can run inside a web browser, or a web server [Reilly 1999].
- **Java offers automatic garbage collection:** Using Java, software developers don't need to allocate and reallocate memory for data and objects manually. When memory is no longer needed reallocation is done automatically by the garbage collector. Hence, programming complexity is reduced and the problem of memory leaks is solved. By reallocating memory manually it is likely to forget some objects, and the amount of free memory available will decrease.

- **Java is secure:** Since Java was designed to support network programming, security is of great importance in Java. At the API level there are strong security restrictions on file and network access for applets. At the byte code level checks are made for obvious hacks such as stack manipulation or invalid byte code [Reilly 1999].
- **Java is simple and easy to use:** Although Java is similar to C++ it does not include its dangerous parts and is therefore safer and easier to use. Java provides no memory pointers, but uses object references instead. Multiple inheritance has been removed and replaced by a singly rooted hierarchy (with Object as the ultimate ancestor of all classes) combined with Java interfaces.

These few points outline the main features of the Java language leading to the decision to use Java as programming language for this project. A more detailed specification of the Java language is given in [Gosling et al. 1996].

3.2 Comparing GUI Toolkits for Java

A fundamental decision at the beginning of this migration project is to find the appropriate library for the graphical user interface (GUI).

For Java there exist three main GUI libraries:

- **Abstract Windows Toolkit (AWT)** is the original Java GUI toolkit.
- **Swing**, originating from AWT, is the reference toolkit for the Java 2 Standard Edition (J2SE).
- **Standard Widget Toolkit (SWT)** has been developed by IBM as part of the Eclipse platform. Eclipse is an open source integrated development environment (IDE) built using Java and SWT.

In addition there are some other Java toolkits like GTK+ and QTJambi not further discussed here. The following comparison is essentially a summary of [Feigenbaum 2006] where a well-founded overview of the topic is provided.

3.2.1 AWT

The main advantages of AWT are that it comes standard with every version of Java technology and is very stable. It has not to be installed separately and software developers can depend on it being available on any Java Runtime Environment (JRE).

For AWT Sun decided to use a lowest-common denominator (LCD) approach, and therefore AWT is very simple and limited. Only GUI components working on all host environments are used. Because of this approach we have to create some commonly used, but not generally supported components by ourselves (e.g., tables).

AWT components are thread-safe. Hence, some GUI update problems are eliminated, but the application can run slower. AWT supports automatic disposal of unused GUI components. Components can exist without a parent container or even change the parent at runtime. With regard to automatically generate an application and its belonging GUI, these are essential advantages.

The main problem of AWT is that it depends on host GUI peer controls to implement the GUI. Thus, the AWT controls map directly and without modification to the host platform's graphic interface. According to this direct mapping the GUI looks and behaves differently on different hosts, and a platform-independent application is really hard to realise.

3.2.2 Swing

Swing, being built on parts of AWT, was developed by Sun to solve most of AWT's shortcomings. It provides more sophisticated GUI components. The components were designed to be consistent across all platforms by minimizing their dependence on host controls.

Swing uses peers only for top-level components like windows and frames. All other components are emulated in pure Java code leading to a higher portability across different platforms. Due to this emulation Swing is unfortunately no more able to take advantage of hardware GUI accelerators and special host GUI operations. As a result applications using Swing can be slower than applications using AWT.

Like AWT, Swing is part of the standard Java runtime environment and has not to be installed separately. It also supports automatic disposal of GUI components, and components can exist autonomously. Unlike AWT, the components are not thread-safe.

In contrast to AWT, Swing provides many architectural features making it more powerful than AWT. The following features assure Swing to be the arguably best architecture of the three toolkits:

- Separation of model, view, and controller.
- Programmable look and feel.
- Use of renderers and editors.

A more detailed description of these features is given in the sections 3.3 and 3.4

3.2.3 SWT

SWT is tightly integrated with the native host window system (especially with Windows, but Linux and Solaris are supported as well). Nevertheless it is independent from the

host's operating system. The intention of designing SWT was to combine the advantages of AWT and Swing without their disadvantages.

In concept it is comparable to AWT, because it is based on a peer implementation. But differently from AWT, where peers can provide services to minimize the differences between hosts, in SWT peers are only wrappers on host controls. The LCD problem of AWT was solved by defining a set of controls based on native peers and creating emulated controls (as in Swing) for any controls not supplied by the particular host. Thus, SWT can be seen as a thin wrapper over the native code GUI of the host operating system. Its great advantage over the other two toolkits is that an SWT-based GUI has (differently from AWT and Swing) a host look and feel, and - even more important - a host performance.

SWT does not support automatic disposal of GUI components. The software developer is in charge of this task. Furthermore, SWT is less flexible than AWT or Swing regarding to the fact that components cannot exist without a parent container and the parent cannot change at runtime. As in Swing, SWT components are not thread-safe.

As major drawback SWT is not part of the Java runtime environment, but must be installed separately. The needed libraries differ for different operation systems like Windows, Unix, and Macintosh.

3.2.4 Selection

Each of the three considered toolkits has its advantages and disadvantages. The selection of the appropriate one is a result of the given needs and the intended users.

In our case, AWT is out of the consideration, since it does not support needed components like Tables, Trees, and Progress Bars. Thus, the decision is between Swing and SWT which are both complete and powerful enough to build full-function GUIs. SWT would be the proper choice if the application is to be developed only for one platform. There it takes advantage of its better host compability including integration with host features (like ActiveX controls under Windows). Moreover, it has better performance than Swing. But, the decisive factors for the selection of Swing are the characteristics of Swing being built completely onto Java technology and therefore being highly portable.

In Appendix C in Figure 25 and Figure 26, both taken from [Feigenbaum 2006], the most important characteristics of the AWT, SWT, and Swing libraries are summarized.

3.3 The Model-View-Controller Paradigm

> *"The Model-View-Controller (MVC) pattern separates the modelling of the domain, the presentation, and the actions based on user input into three separate classes"* [Burbeck 1992].

Model-View-Controller is a fundamental design pattern for the separation of user interface logic from business logic. The MVC architecture was introduced first in Smalltalk-80 to construct graphical user interfaces. It provides several views for one and the same data. Its main field of application are interactive systems where the main focus is on man-machine communication and hence on the GUI.

In such interactive systems the functional core and presentation are separated. In general, the functional part remains stable whereas user interfaces frequently are subject to modifications, either by expanding the functionality, by changes in the graphical user interface, or by migrating it to another platform with another look and feel. A tight coupling between user interface and functional code complicates further development since many areas in the system are affected. The configuration of the user interface at runtime must be possible.

To satisfy these requirements interactive systems are being separated in three types of components: Model, View and Controller.

The following definitions are taken from [Burbeck 1992]:

- Model
  The model manages the behaviour and data of the application domain, responds to requests for information about its state (usually from the view), and to instructions to change the state (usually from the controller).
- View
  The view manages the display of information.
- Controller
  The controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view.

The model is the central component to perform operations on data. It is either independent from the presentation of output data or from a certain behaviour of input data. The model defines the functionality of the application and contains and encapsulates its state. It responds to requests concerning the state of the application and informs the associated (registered) views and controls about changes. It provides methods for evaluation and processing of input and enables access to displayed data.

The view is the output component of the MVC architecture. One or many views represent the model which they can consult in case of changes. Input from the user is accepted and sent to the controller. The views present the information to the user. After changing essential information the according view is getting informed by the model and brought up-to-date. Every view maps to an adequate controller component.

The controller is the input component of the MVC architecture, and defines the behaviour of the application. The functionality of the model is defined in the controller. Hence, user actions are mapped onto modifications of the model, and the appropriate view for the presentation of the model is choosen. Controllers are always assigned to a specific

view. All user input accepted by this view is handled by the controller and then passed on to the model. The controller causes its view to be updated after changes.

Figure 7 shows a detailed MVC abstraction.

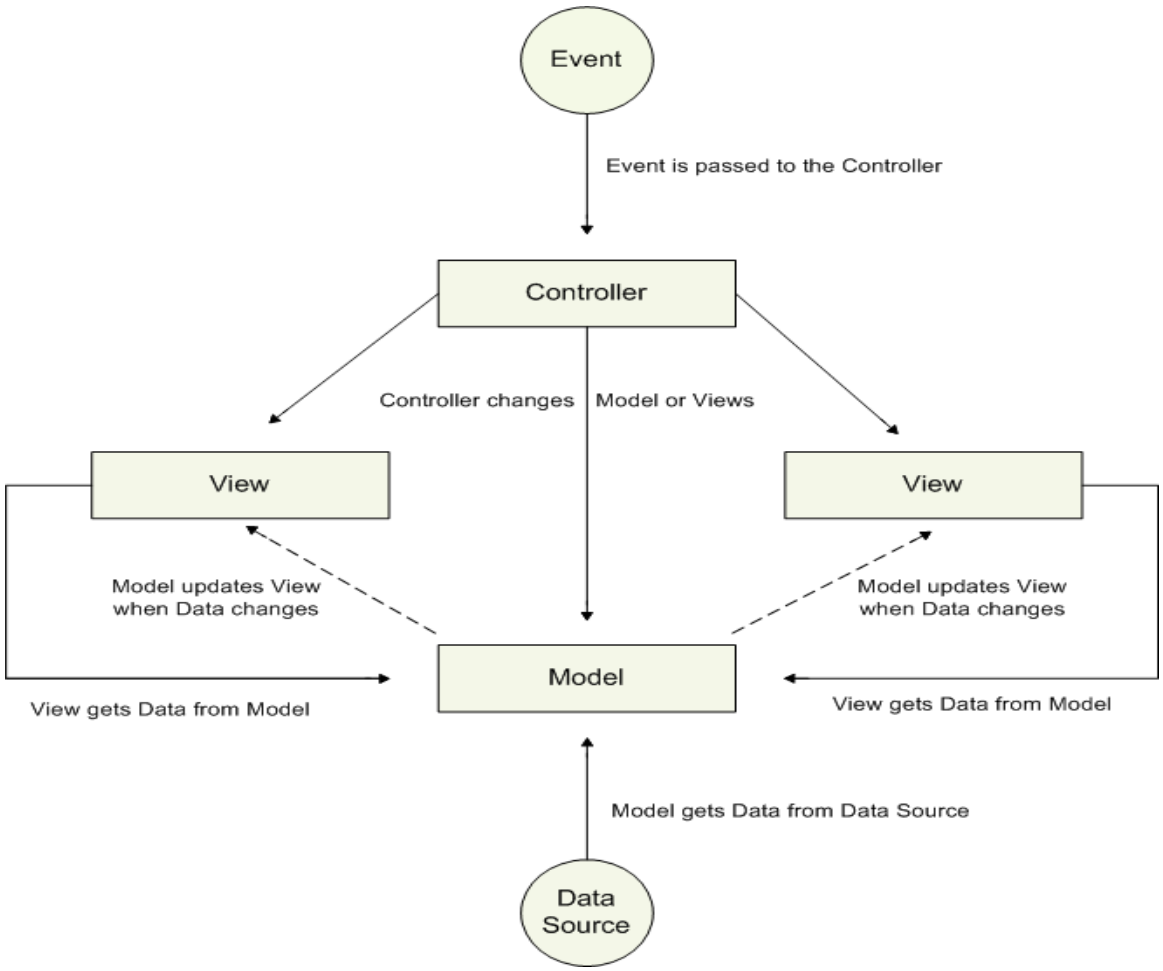


Figure 7: The Model View Controller architecture

3.4 Swing Features

3.4.1 MVC in Swing

The architecture of Swing is not strictly based on the traditional model-view-controller design, but on a common variation where controller and view are combined into a single UI object. This so-called UI delegate object is the basis for each individual Swing UI component whereas the typical MVC design pattern is used to construct entire user interfaces.

Each UI object (e.g., table, button, field, ...) has its own component-specific UI-delegate providing the group of methods for the specific component class. Since communication between view and controller is very complex, the combination of them makes component

design a lot easier. Furthermore, even when the component is in use the model, view, and controller object can be replaced. These characteristics offer great flexibility to a software developer using Swing.

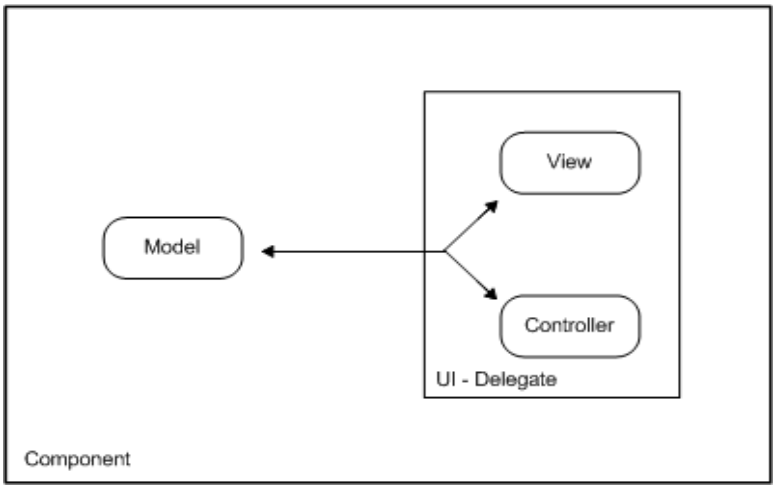


Figure 8: Swing MVC Model

As seen in Figure 8, model, view, and controller are bundled within component and their presence is masked to the observer. The component holds the three classes and hides them from the software developer. Direct access of these classes is not favoured. Many of the methods provided by the component class are wrappers passing along the method invocation to either the model or the UI delegate.

3.4.2 Programmable Look and Feel

The term "look and feel" means the appearance and the behaviour (e.g., how it reacts to input events) of an application [Krüger 2002]. A useful characteristics of Swing is the possibility to change the look and feel of an application. The look and feel of a GUI is controlled by a separate and dynamically replaceable implementation allowing us to change all or parts of it. This feature, termed pluggable or programmable look and feel, allows software developers to choose between predefined motifs for each operating system and even to write own look and feels. Though the decision for or against a certain look and feel need not be made while designing or developing the application. In fact all Swing components were designed that way giving us the possibility to change the look and feel comfortably even at runtime.

3.4.3 Renderers and editors

Components showing model content, such as lists, tables, and trees, can process model elements of almost any type. To do this a renderer or editor is mapped for each component type and model type [Feigenbaum 2006]. For example, a table column containing a specific data type value can have different code portions to deal with either the presentation or the editing of the value.

3.5 Reflection in Java

> *"As its name suggests, reflection is the ability for a class or object to examine itself. Reflection lets Java code look at an object (more precisely the class of an object) and determine its structure"* [Niemeyer et al. 2005].

According to [Simmons 2004] reflection is one of the least understood, but also one of the most powerful aspects of Java. [Sun 2007] describes reflection as commonly used by programs requiring the ability to examine or modify the runtime behaviour of applications running in the Java virtual machine. This relatively advanced and powerful technique can allow applications to perform operations which otherwise would be impossible.

With the aid of reflection it is possible to find out the constructors, methods, fields, and attributes a class has. It enables the use of classes at runtime without knowing their name beforehand. Objects of these classes can be created dynamically and access to their methods and attributes is possible. Therefore, reflection, also known as introspection, supports the development of high-flexible and generic applications.

The Reflection API allows us to operate on classes and objects in ways such as the following [Darwin 2007]:

- Load a class file into memory at runtime, knowing only its name.
- Given a class, examine its methods, fields, constructors, annotations, and so on.
- Given a class, invoke constructors and methods.
- Given a class and an instance, access fields.
- Given an interface, create proxies for it dynamically.

3.5.1 Class Class

The basis for the Reflection API are the classes *java.lang.object* and *java.lang.class*. The object class is the base class for all objects in Java, and therefore every class in Java offers methods like *toString()* or *getClass()*. In connection with reflection especially the latter is important because it returns the particular class of the object.

An object of class *Class* represents a Java class. The runtime environment creates automatically an object for this class during the first loading of a class. All objects of type class are managed internally by the runtime environment so that for any class always the same class object is used.

The class *Class* provides methods to retrieve the structure of a class, to dynamically load classes by means of their names, and to create objects.

Table 2 lists some important methods of class Class.

Method	Description
Field[] getFields()	Gets all public variables, including inherited ones.
Field getField(String name)	Gets the specified public variable, which can be inherited.
Field[] getDeclaredFields()	Gets all public and non-public variables.
Field[] getDeclaredField(String name)	Gets the specified variable, public or non-public.
Method getMethods()	Gets all public methods, including inherited ones.
Method getMethod(String name, Class...argumentTypes)	Gets the specified public method whose arguments match the types listed in argumentTypes.
Method getDeclaredMethods()	Gets all public methods declared in this class.
Method getDeclaredMethod(String name, Class...argumentTypes)	Gets the specified method whose arguments match the types listed in argumentTypes, and which is declared in this class.

Table 2: Important methods of class Class

Reflection allows us to analyse the structure of a class. Structures are returned as objects of type field, method, and so on. Each of these classes again provides methods to further analyse the structure or to gain access to it. As an example in the next subsection we will illustrate the class *Method*.

3.5.2 Dynamic method invocation

The starting point for dynamic method invocation at runtime is an object of type *Method*. We can get such an object with the methods of Class described above. An object of type Method represents exactly one method. Method itself provides various methods, but we will put the focus on one method. The *invoke()* method can be used to call the underlying object's method with specified arguments and acts in some way as a method pointer. In Listing 1 inspired by [Ullenboom 2007], a method object for class *java.awt.Point* is created and by means of its invoke() method the x- and y-value of two different point objects are dynamically set.

Listing 1: Dynamic method invocation

```
1  import java.awt.*;
2  import java.lang.reflect.*;
3
4  class DynamicMethodInvocation
5  {
6
7     public static void main(String args[]) throws Exception
8     {
9        Point p1;
10       Point p2;
11       Method method;
12
13       p1 = new Point(10,0);
14       p2 = new Point(10,0);
15       method = p1.getClass().getMethod("setLocation", int.class, int.class);
16
17       method.invoke(p1, 1, 2);
18       System.out.println(p1);    // => java.awt.Point[x=1,y=2]
```

```
19          method.invoke(p2,3,4);
20          System.out.println(p2);     // => java.awt.Point[x=3,y=4]
21      }
22  }
```

Table 3 lists the methods of class Method.

Method	Description
String getName()	Gets the name of the method.
int getModifiers()	Gets the modifiers of the method.
Class getReturnType()	Gets the return type of the method.
Class[] getParameterTypes()	Gets the parameter types of the method in the order the must be specified.
Class[] getExceptionTypes()	Gets the exception types of the method, which are the types specified in the *throws* clause.
Class getDeclaringClass()	Gets a class object of the type declaring the method.
boolean isVarArgs()	Returns whether the method allows a varying number of parameters.
Object invoke(Object obj, Object...args)	Calls the method of Object *obj* with the specified parameters *args* and returns its result. If the method expects primitive types they must passed with the aid of wrapper classes. The Reflection API converts them internally into the adequate primitive types.

Table 3: Important methods of class Method

3.6 Alternatives

This section offers a short overview about the .NET technology. The .NET framework was created by Microsoft (not only, but also) as an alternative to the Java platform. Next we will deal with Mono, which builds up on .NET, but in contrast to .NET is really open source. Since the language is still relatively new, we will describe it more detailed. With the aid of some small examples we will demonstrate Mono's language independence and the power of the concept behind it.

3.6.1 .NET

.NET, what is it about? What is so interesting on this platform and what is new in this technology? Microsoft describes .NET on its web page [Microsoft.net] as follows:

> *The .NET Framework is a development and execution environment that allows different programming languages and libraries to work together seamlessly to create Windows-based applications that are easier to build, manage, deploy, and integrate with other networked systems. Built on Web service*

> *standards, .NET enables both new and existing personal and business applications to connect with software and services across platforms, applications, and programming languages. These connections give users access to key information, whenever and wherever you need it.*

This means, confronted with a particular problem, .NET offers us the possibility to use the most suitable technology. To satisfy the above mentioned requirements, Microsoft offers three different solutions of realisation. These solutions concern web services, .NET remoting services for distributed applications in LAN networks, and the .NET enterprise services.

The latter can be viewed as a competitive product to the J2EE technology developed by Sun. The Common Language Runtime (CLR) is the most important part of .NET. The CLR is a just-in-time compiler that translates a .NET program into machine code. It is comparable with Java's virtual machine. Figure 9 (taken from [De Icaza 2005]) shows the architecture of the CLR compiler

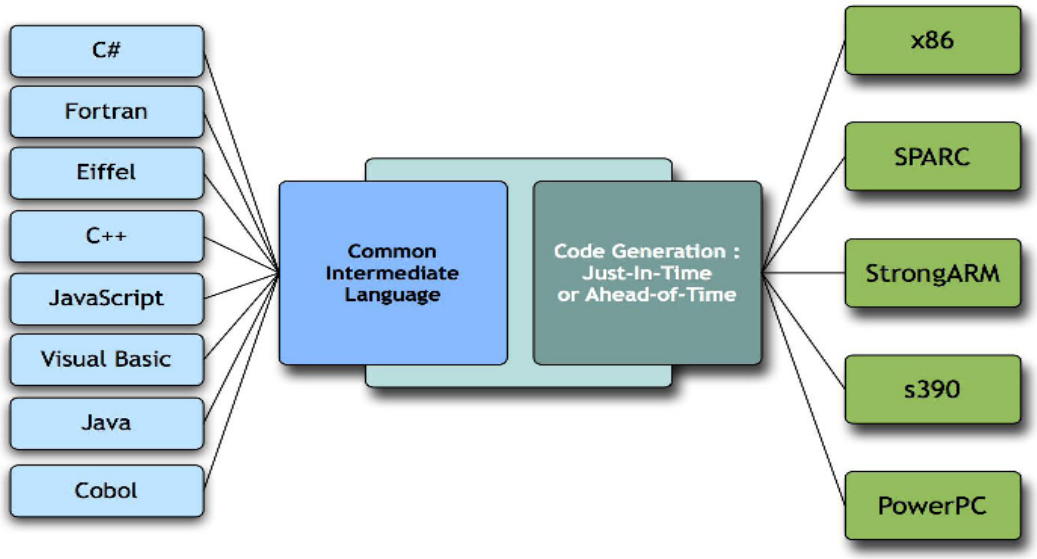


Figure 9: Common Language Runtime

With the .NET framework Microsoft uses a middleware concept and provides, at least theoretically, similar conditions for the development of platform independent software as the Java platform. Unfortunately, Microsoft uses the aspect of platform independence primarily in it's field of activity and only for own purposes. The .NET-Framework has been planned to be used only for applications on Windows operating systems.

Nevertheless, Microsoft in November, 2000 revealed it's specifications for the programming language C#, the CLR and parts of the FCL (the .NET Framework Class Library) to the standardization committees ECMA and ISO and allowed therefore at least the basic possibility of real platform independence for the .NET-Framework (though without own participation) [Easton et al. 2004].

Based on the ECMA specification there originated several projects with different objectives. The most important project among them is Mono.

3.6.2 Mono

Mono is an open source implementation of the Microsoft .NET framework. In opposition to .NET, Mono is not only available for Windows, but also for the other common OS like Linux, Unix, MacOS and Solaris [Dumbill et al. 2004].

The fact that libraries developed with Mono can be used cross-platform is only one of its advantages. Another benefit compared to the Microsoft counterpart consists in the fact that Mono supports not only the programming language C#, but also other important languages like Java, Python, and some more. Therefore, a developer can use the language appearing most appropriate to him for the particular problem, or even combine several different languages. The possibility to combine and apply together different computer languages makes Mono an effective and efficient platform for software architects.

However, Mono is not a full conversion of .NET. Elements like Windows.Forms and Visual Basic are (still) missing. But, it is worth mentioning that in the following versions of Mono the missing elements shall be added. The actual version contains additional database layers, improved XML interfaces, LDAP (Lightweight Directory Access Protocol) functionality, and with GTK# an graphics API that extends and improves the .NET framework [Dumbill et al. 2004].

In Figure 10, taken from [De Icaza 2005] the various components of Mono are shown.

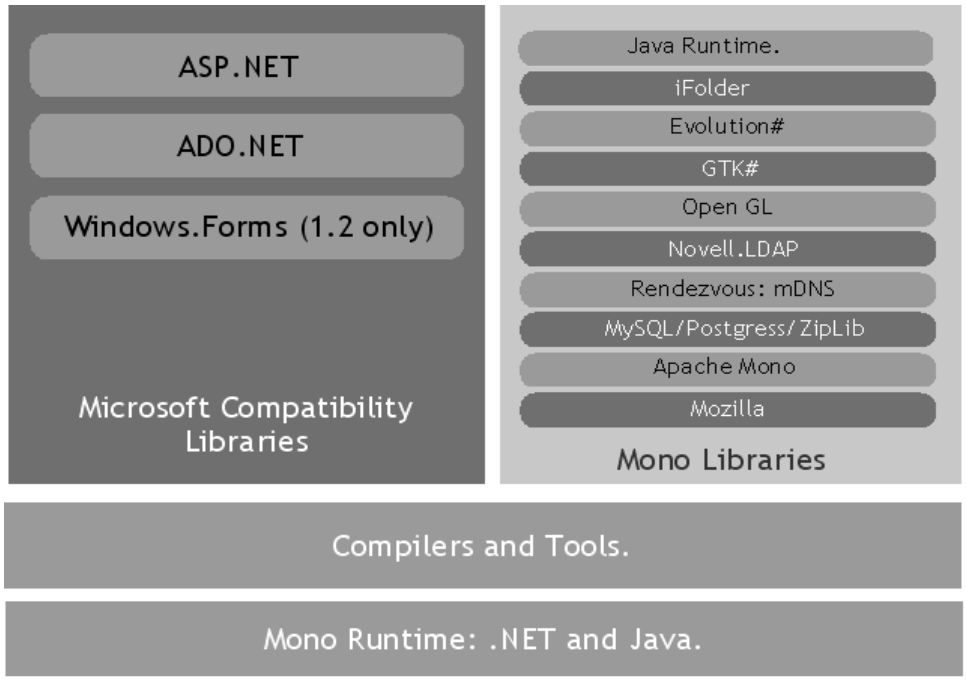


Figure 10: Elements of Mono

Mono tries to remain compatible with Microsoft .NET. Thus, with Mono it is possible to develop freely available applications which do not have licence problems when running under Windows. The porting of .NET programs from Windows to other platforms like Linux is made easier with Mono. Mono originated in 2001 from the GNU project Gnome that had similar difficulties with grown program interfaces like Windows. The idea of

language independency in the Linux world was as attractive as under Windows. A major advantage of Mono is to use programs developed for the Windows OS also under Linux. This fact increases the attraction of Gnome and Linux [Mählmann 2005].

The Open Source project Mono was launched in June, 2001 by the Ximian company. The ECMA standard was taken as premise to be able to develop Linux software cheaper and of higher quality than up to now. From the company's view in particular the amount of interoperability features spoke for the standard and against the use of the comparable Java platform. Unfortunately the porting of .NET to Mono is still only partially in a consistent and final state, as can exemplarily be seen on the non-existence of Windows.Forms [Easton et al. 2004].

Why Mono?

According to [Dumbill et al. 2004] there are various reasons to use Mono as programming language for projects.

- Brings together language features of several modern programming languages.
  While Mono's technologies are present in various languages, the conjunction of them in a single platform makes developing with Mono attractive and exciting.
- Provides a controlled environment.
  Programs are compiled into a bytecode known as the Common Intermediate Language (CIL), and run as managed code in a controlled environment.
- Reuses existing code investments.
  Mono makes it easy to reuse existing investment in code libraries. This feature is especially in our case very interesting. If we decide in a near future to switch to Mono, the work on the Java project is not lost. Moreover, we can use existing and well proven parts of the legacy system (e.g., the ADO layer).
- Fast to write, fast to run.
  Developing with Mono and C# can bring significant productivity increases, since C# can easily be understood by developers coming either from a Java or C++ background. The runtime of Mono uses a just-in-time (JIT) compiler to translate the bytecode, maximizing application performance.
- Presents cross-platform code and migration paths.
  Unlike .NET, Mono is cross-platform, supporting Windows, Linux, MacOS, and hardware including x86, PowerPC, and SPARC processors.
- Provides a choice of languages.
  The CLR is not restricted to C# alone, but provides compilers for Java, JavaScript, Basic, C, Phyton, and even Cobol.

The language independence mentioned in the last point of the enumeration is one of Mono's main advantages and increases the flexibility and expressiveness. The introduction of IKVM and the following example, taken unmodified from [Dumbill et al. 2004] give an idea about the power of this feature.

******* Begin of excerpt *******

IKVM is a runtime for Java that works by translating Java bytecode into CIL bytecode. By incorporating bridging technology, IKVM enables Java programs to make calls to Mono assemblies, and Mono code to use Java class libraries.

The follwing example demonstrates calling Mono assemblies from Java.

Listing 2 shows a C# program, which is compiled into an assembly that later on will be invoked from Java code.

Listing 2: C# Adder Class: Adder.cs

```
1  using System;
2
3  public class Adder
4  {
5    public int Add(int a , int b)
6    {
7      return a + b;
8    }
9
10   //example usage from C#
11   public static void Main(string[] args)
12   {
13     Adder a = new Adder();
14     Console.WriteLine ("1 + 2 = {0}", a.Add(1,2));
15   }
16 }
```

IKVM provides a tool called ikvmstub that converts the Mono assembly placed in an *.dll file into a *.jar file needed by the Java compiler. With the following commands the conversion takes place.

$ mcs -target:library Adder.cs
$ mono $IKVMDIR/ikvmstub.exe Adder.dll

Listing 3 shows a small Java program to invoke the Adder class. In Line 1 we see the import of the Adder class. ikvmstub changed its notation from Adder.cs to cli.Adder, which indicates the conversion by ikvmstub.

Listing 3: Java Adder Client Class: AddClient.java

```
1  import cli.Adder;
2
3  public class AddClient
4  {
5    public class AddClient
6    {
7      public static void main(String[] args)
8      {
9        Adder adder = new Adder();
10       System.out.println("1 + 2 = " + Integer.toString(adder.Add(1,2)));
11     }
12 }
```

The following commands then will compile and execute the program.

$ javac -classpath Adder.jar AddClient.java
$ MONO_PATH=$MONO_PATH:. mono $IKVMDIR/ikvm.exe -classpath .:Adder.jar AddClient

The setting of MONO_PATH in the second command is needed to instruct Mono where to look for the implementation of Adder from Adder.dll. At start-up time IKVM translates the Java AddClient program and executes it inside the Mono runtime.

The contrary use of IKVM making Java class libraries available to Mono applications is shown in a reverse implementation of the first example.

Listing 4: Java Adder Class: JAdder.java

```
1 public class JAdder
2 {
3   public int Add(int a, int b)
4   {
5     return a + b;
6   }
7 }
```

Listing 5: C# Adder Client Class: JAddClient.cs

```
1  using System;
2
3  public class JAddClient
4  {
5    public static void Main(string[] args)
6    {
7      JAdder j = new JAdder();
8      Console.WriteLine("1 + 2 = {0}", j.Add(1,2));
9    }
10 }
```

The program can be compiled and executed by the following commands. The reference to IKVM.GNU.Classpath.dll makes the Java runtime APIs available to Mono.

$ javac JAdder.java
$ mono $IKVMDIR/ikvmc.exe -reference:$IKVMDIR/IKVM.GNU.Classpath.dll -target:library JAdder.class
$ MONO_PATH=$MONO_PATH:.:$IKVMDIR mcs -r:JAdder.dll JAddClient.cs
$ MONO_PATH=$MONO_PATH:.:$IKVMDIR mono JAddClient.exe

******* End of excerpt *******

Why not Mono?

Particularly with regards to the future there exist many open questions and there is still need for discussion. The question may be raised whether there is a desire for .NET under Linux.

With some libraries and modules (e.g., web service and security) Mono extends and improves the existing .NET platform. Nevertheless, Mono receives neither consideration

nor support from Microsoft. Microsoft can change its system in such a way that Mono is not applicable any more and many developers are forced to bind themselves further to Microsoft instead of using a free solution (e.g., through changes in the graphical user interface).

It will be interesting to keep track of how Microsoft positions itself to Mono in future. Will Microsoft tolerate Mono, support or maybe even try to forbid it? Should Microsoft change its patents, Mono would have to rewrite various components of its language. This would basically mean that it does not correspond any more to the Microsoft standard which, however, is not required under Linux [Mählmann 2005].

Concluding this section it must be said that at the moment the Mono platform is still in a too early stage of development to take it actually into consideration. As long as Microsoft's position to Mono is insecure the risk in developing for the Mono platform is too great.

Mono together with .NET provides some remarkable benefits, but like Java, it will suffer from the same performance problems when trying to be completely platform independent.

In the case that applications need not to be real cross-platform but use according to the OS the adequate API, Mono takes advantage over Java on a Linux system through its use of GTK# (and maybe in future with Windows.Forms even on Windows).

4 Implementing the Connection to the DB

The first step of the migration process is the design of the connection between application and database. The resulting connection layer is a very important component within this project since the data transfer to and from the database has great influence on almost every aspect of the project.

Subsection 4.1 provides a detailed listing of the requirements the connection layer has to meet. Since according to this task we won't reinvent the wheel we will take the already existing connection layer of the legacy system as model. Subsection 4.2 sketches the components of the legacy system. In Subsection 4.3 the DAO-pattern will be introduced. Together with the existing layer this database access pattern will build the basis for the architecture of our connection layer. Subsection 4.4 outlines the database access API we choose to accomplish this task, whereas subsection 4.5 presents the resulting connection layer called IQDAO.

4.1 Specifications

The requirements concerning this task are in detail:

- Database access shall be implemented securely with regard for availability, integrity, and confidentiality of the data.
- The entry and update of data stored in databases shall be accomplished in accordance with the business rules established in software application systems. Data shall be entered and updated using software applications and business rules to protect the data from unauthorized or accidental access and to ensure security, data integrity, and accurate interpretation of the data. Data access and permissions shall be assigned within the context of the software application, ensuring the relevant business rules implemented by the software application system for normal entry and update not to be violated.
- Database access routines shall be written as independent of the platform and underlying data structure as feasible. Separating database access logic from the application logic of a software application makes it easier to relocate or restructure the database, and to re-platform the back-end services with minimal disruption of the software applications using the database. In a first step the connection layer must be able to deal with Oracle databases and Microsoft SQL Server databases. Nevertheless, for future expansions, easy access to other databases of less important (in the sense of market penetration) providers shall be facilitated.
- The emerging data volume need not be subject to other restrictions than those established by the underlying platform (e.g., hardware, operating system).
- Special attention shall be paid to the scalability and expandability of the connection layer. Future enhancements shall be facilitated without degradation of system performance.

- Efficient and straightforward access to most of the current data types.
- Transparency in the development of the connection layer for the various different databases.
- Easy adaptability and validation of entered configuration data.

4.2 The existing Connection Layer

The main goal of this task is to emulate the behaviour and functionality of the existing system written in Visual Basic and based on ADO and OLE DB as accurately as possible. First of all, the fact that these elements all have been developed from Microsoft and work together well facilitates the design of a homogeneous architecture. As second great virtue, this system is already in use by database business applications developed by Inova Q Inc. and is well proven, secure, and efficient.

Before covering the further approach the existing components will be described in a nutshell.

OLE DB: *OLE DB (Object Linking and Embedding Database) is Microsoft's strategic low-level application program interface (API) for access to different data sources. OLE DB includes not only the Structured Query Language (SQL) capabilities of the Microsoft-sponsored standard data interface Open Database Connectivity (ODBC) but also includes access to data other than SQL data. As a design from Microsoft's Component Object Model (COM), OLE DB is a set of methods for reading and writing data.* [SearchSQLServer.com]

ADO: Microsoft ActiveX Data Objects (ADO) is a set of Component Object Model (COM) objects for accessing data sources. It provides a layer between programming languages and OLE DB. ADO objects can be used in any language that supports COM and automation (i.e., Visual Basic, Visual C++, Visual Fox Pro, VBScript, etc.). ADO is able to interface not only with relational databases, but also with non-relational databases, folders, data files, and even e-mail messages. [msdn]

COM: *Microsoft COM (Component Object Model) technology in the Microsoft Windows-family of operating systems enables software components to communicate. COM is used by developers to create re-usable software components, link components together to build applications, and take advantage of Windows services. The family of COM technologies includes COM+, Distributed COM (DCOM) and ActiveX Controls.* [Microsoft.com]

ODBC: *ODBC is a standardized API, developed according to the specifications of the SQL Access Group, that allows one to connect to SQL databases. It defines a set of function calls, error codes and data types that can be used to develop database independent applications.*[Kingsley 1993]

Whereas the existing system makes use of ADO as a high-level interface to provide ease of access to data stored in a wide variety of database sources, for the new system to develop no high-level Java API exists. Java offers some technologies (like JDBC, SQLJ, Java DB, ...) to link applications to databases. For this project only JDBC will be considered since it is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases.

4.3 The DAO Pattern

According to the given requirements we have to design an architecture for the data source access. Sun in [Sun 2001] implemented a pattern being very suitable for these needs. Because, despite all SQL standardization efforts, *even within an RDBMS environment the actual syntax and format of the SQL statements can vary depending on the particular database product* [Sun 2001], the so-called *Data Access Object* pattern was designed. Since we must ensure that there is no direct dependency between application code and data access code (this would make it difficult and tedious to adapt the application to new requirements) the DAO pattern appears to be an adequate solution. The components to be developed need to be transparent to provide easy migration to different vendor products, storage types, and data source types. The DAO was designed *to abstract and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data* [Sun 2001].

The DAO provides simplified interfaces for its clients to access and work with the data source. Hiding the implementation details of the data source from the clients this pattern allows changes and adaptations without affecting the business object implementation of the clients. Thus the DAO acts as an adapter between business components and the underlying data source. Figure 11, taken from [Sun 2001] shows the class diagram for the DAO pattern.

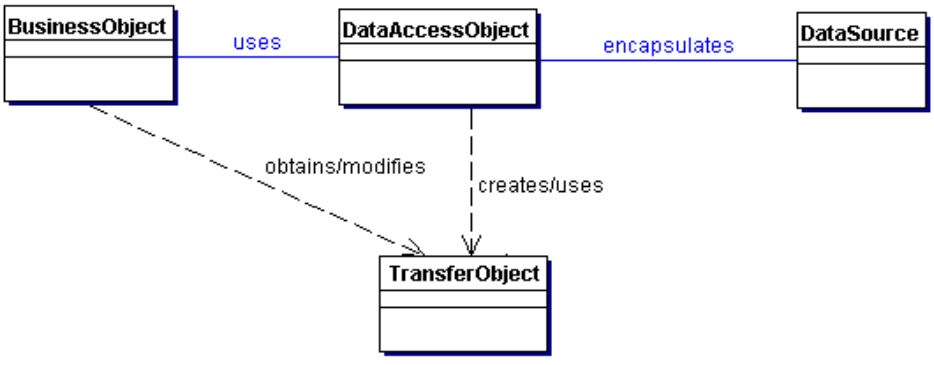


Figure 11: Data Access Object

[Sun 2001] specifies the needed components as follows:

- BusinessObject
  The BusinessObject is the object requiring access to the data source to obtain and store data.
- DataAccessObject
  The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.
- DataSource
  The DataSource object represents a data source implementation. A data source can be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository.
- TransferObject
  This object represents a transfer object used as a data carrier. The DataAccessObject can use a transfer object to return data to the client. The DataAccessObject can also receive the data from the client in a transfer object to update the data in the data source.

According to [Sun 2001] the use of the DAO pattern has the following benefits:

- Enables transparency
  Applications can use various data sources without knowing their internal specific details. The implementation details are hidden inside the DAO and therefore the access to the database is transparent.
- Enables easier migration
  Since the business application objects have no knowledge of the underlying data implementation, migration involves only changes in the DAO layer. Hence, it is easier to migrate to a different or add a new database implementation.
- Reduces code complexity in business objects
  All data access code is managed by the DAO and therefore the code complexity of the clients using the DAO is reduced. Thus, the code readability and development productivity is improved.
- Centralization of all data access in a separate layer
  Since DAO is as a separate data access layer it isolates the rest of the application from the data access implementation. Therefore, the application will be easier to maintain and manage.
- Adding of an extra layer
  The DAO creates an extra layer between client and data source.

4.4 JDBC

4.4.1 What is JDBC?

The JDBC API provides programmatic access to relational data from the Java programming language. Using the JDBC API consisting of a set of classes and interfaces, applications written in the Java programming language can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also be used to interact with several data sources in a distributed, heterogeneous environment. [Sun 2006]

4.4.2 Why JDBC?

The Java Database Connectivity (JDBC) API provides a call-level API for SQL-based database access. A wide range of underlying data sources or legacy systems can be accessed. Differences in the implementation of these systems are masked through JDBC API abstractions. Thus, it is a valuable target platform for developers who want to create portable tools or applications. An application written in Java can access virtually any data source and run on any platform with a Java Virtual Machine.

Being a call-level interface it is suitable as a layer for higher-level facilities. Its intention is to be a simple-to-use, straight forward interface upon which more complex entities can be built.

Applications written in Java and using JDBC do not directly access the database, but communicate with a so-called database manager. All instructions sent to the database manager are forwarded to a database specific driver. Through this chaining of several connectors, applications remain independent of specific databases. Figure 12, taken from [Computerbase.de], shows database access using the four possible driver types.

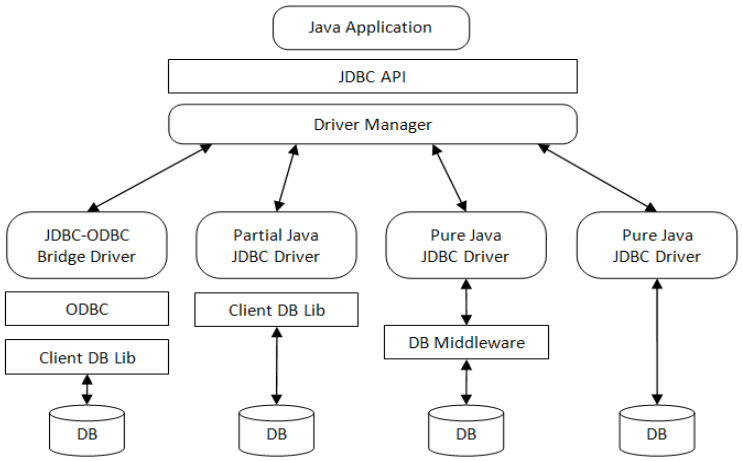


Figure 12: JDBC Database Access Schema

JDBC can be used to query and update database tables using so-called dynamic SQL statements. These are statements where, for example, the number and types of columns in the database are not statically known. With JDBC an application can construct the needed SQL statements at runtime. This benefit is essential in the case of program generation.

4.4.3 Important JDBC features

Scrollability *Scrollability refers to the ability to move backward as well as forward through an existing result set. Associated with this feature is the ability to move to any position in the result set, through either relative positioning or absolute positioning* [Oracle 2002]. The first property allows us to move a specified number of rows backward or forward from the current row whereas the second allows us to move to a specific row number, counting from either the begin or end of the result set.

Sensitivity *Sensitivity refers to the ability to detect and reveal changes made to the underlying database from outside the result set* [Oracle 2002]. Being sensitive a result set provides a dynamic view of the underlying database because it can see the changes made to the database. An insensitive result set provides a static view of the underlying database. While the result set is open no changes to the underlying database are visible. To see changes to the database a new result set must be retrieved.

Updatability *Updatability refers to the ability to update data in a result set and then (presumably) copy the changes to the database* [Oracle 2002]. Thus, updates, inserts, and deletes can be performed on the result set and copied to the underlying database.

4.5 IQDAO

4.5.1 Architecture

On the basis of the DAO pattern described in Section 4.3 and the already existing ADO connection layer we designed the iQDAO package. As the ADO data model it consists of three main components (and an additional fourth important component).

- IQConnection
  The IQConnection object enables the communication with the database and contains the basic information about the database. It provides a basis for all possible operations.
- IQDataSet
  The IQDataSet object is an enhanced ADO Recordset providing the (extended) methods of the record set and new methods to improve its behaviour. It serves to display the database data as well as to edit and evaluate them.

- IQCommand
  It performs SQL statements in form of simple queries or queries on stored procedures.

- IQJdbcWrapper
  This object is a wrapper for the needed JDBC key classes and interfaces.

Figure 13 shows the architectural framework of the iQDAO package with all classes and the sub-package IQEnums.

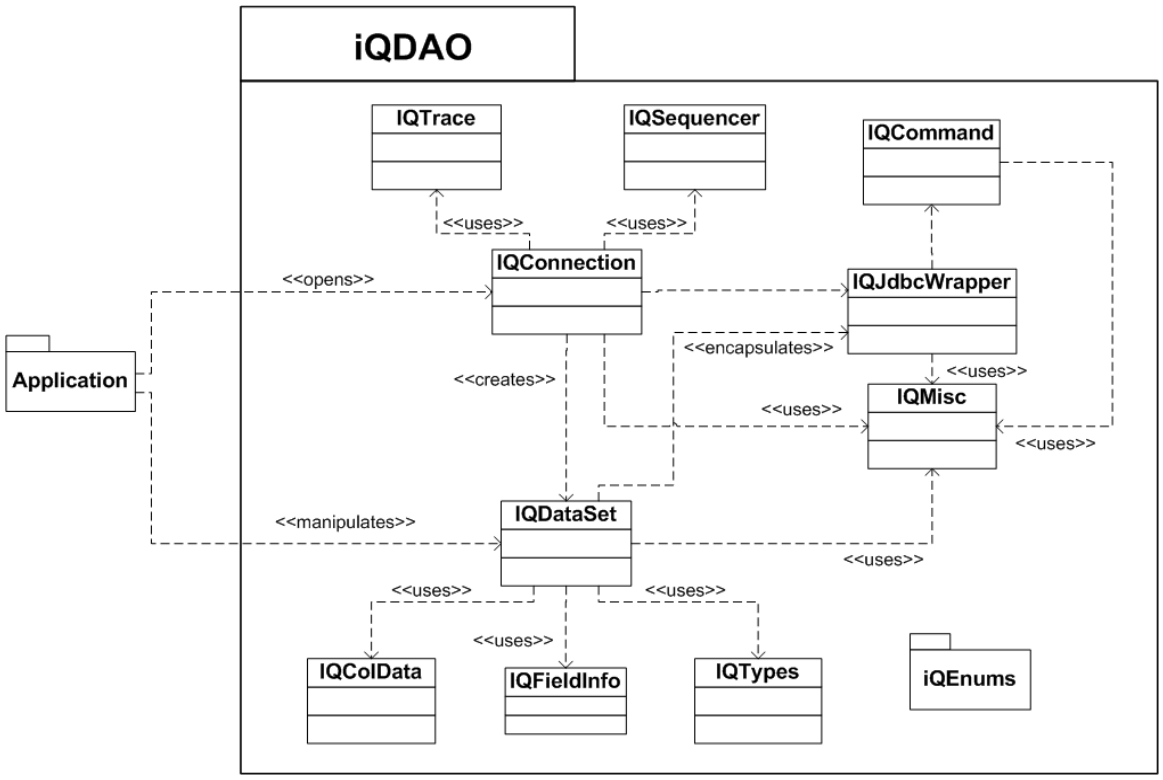


Figure 13: Architecture of iQDAO

The IQMisc class is an important utility class providing general methods used by the main classes. IQFieldInfo describes a database field and contains needed information about it. Thus, e.g., it holds the index, the name, the type, the subtype, the maximum length, the scale, the precision and the catalogue name of the field. Moreover, it provides information whether the field is nullable or not and whether the field is an ID field (it represents a primary key field in the database). The IQTypes class holds self-defined types used by the IQDataSet class. The IQSequencer class contains the program logic to create and use database sequences for various databases whereas IQTrace is in charge of logging messages to files. The main classes IQConnection, IQDataSet, IQJdbcWrapper, and IQCommand are described in more detail in the following sections.

4.5.2 IQJdbcWrapper

In essence the JDBC interface makes it possible to do the following three things:

1. Establish a connection with a data source.
2. Send queries and update statements to the data source.
3. Process the results.

The following code fragment Listing 6 gives a simple example of these three steps:

Listing 6: Simple JDBC example

```
1
2  Connection con = DriverManager.getConnection("jdbc:Driver", "Login", "Password");
3  Statement stmt = con.createStatement();
4  ResultSet rs = stmt.executeQuery("SELECT ID, Name, Age FROM Persons");
5
6  while (rs.next())
7  {
8     int iID = rs.getInt("ID");
9     String sName = rs.getString("Name");
10    double dAge = rs.getDouble("Age");
11 }
```

This simple code fragment shows the way JDBC usually is used. With Connection, Statement, ResultSet and DriverManager we need four different classes (respectively interfaces) from the JDBC package for this simple query. If we access stored procedures we even need the classes PreparedStatement and CallableStatement. And to get descriptive informations about the DBMS needed by applications to adapt to its requirements and capabilities we have to use the interface DatabaseMetaData.

Figure 14, taken from [Sun 2006] shows the interactions and relationships between the key classes and interfaces in the JDBC API.

Another drawback resulting from Listing 6 is that method calls like *getInt()* and *getString()* will not work, since neither tables nor data types are known before the generation process. For such cases there are generic methods like *getObject()*, wich can be used with any type, but needless to say that the type safety of the resulting application will be poor.

Since data types used by different DBMS sometimes vary significant [Fisher et al. 2003], JDBC defines a set of generic SQL type identifiers in the class java.sql.Types. According to [Fisher et al. 2003] another area of difficulty is that most DBMS do not conform to the standard SQL syntax or semantics for more advanced functionality (e.g., stored procedures, outer joins). The portion of SQL being truly standard should expand to include more and more functionality. Nevertheless, the JDBC API supports SQL as it is.

According to these drawbacks and incompabilities we designed the class IQJdbcWrapper to bundle the needed functionality of these classes/interfaces in one single class and to add some extended functionality.

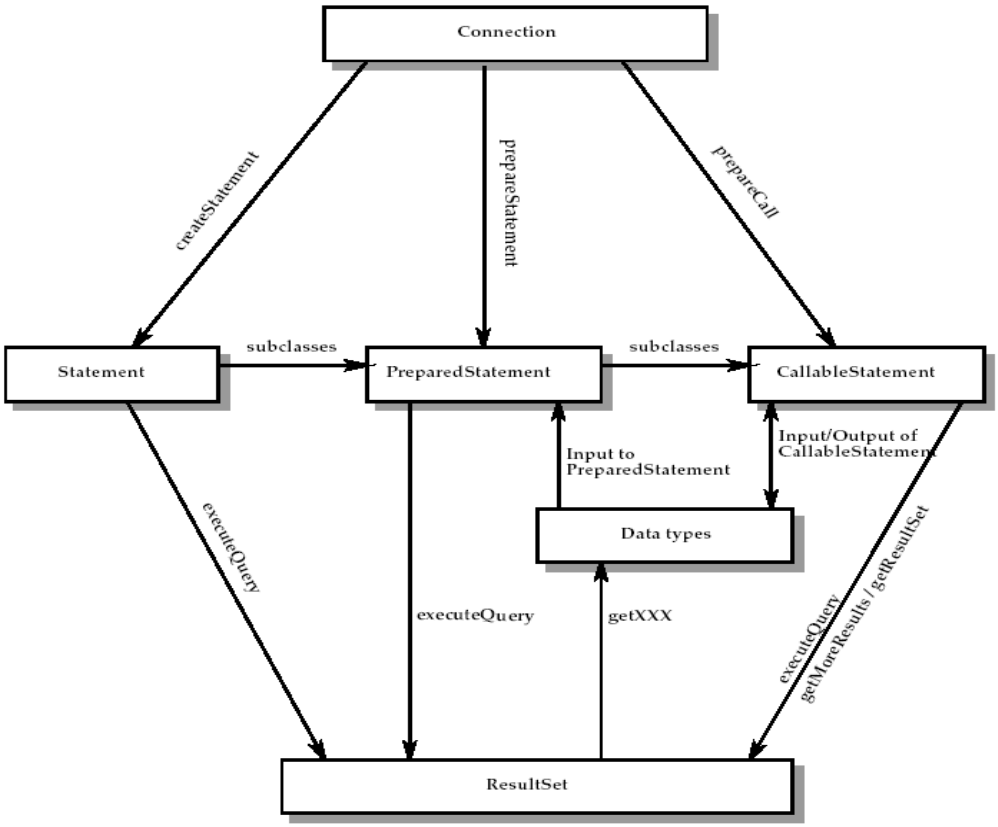


Figure 14: Relationships between major classes and interfaces in the java.sql. package

4.5.3 IQConnection

The IQConnection object opens a connection to a data source and provides access to the data. It is a basic building block in between the database and the front-end. By means of the IQConnection object a database can be accessed several times. One IQConnection can handle more subsidiary objects like IQDataset and IQCommand at the same time.

The main tasks of the IQConnection object are:

- Open and close the connection to the database.
- Execute queries, statements, procedure or provider specific text.
- Provide security (Identification, Authentification).
- Handle transactions.
- Return schema information about the data source.
- Manage self-defined operational modes (e.g. data accessing modes, editing modes, ...).

4.5.4 IQDataSet

The IQDataSet object holds a set of records from a database table. This object is the one most often used since it manipulates the data in the database. It displays the records and enables navigation through the record sets. The IQDataSet object serves to update, delete and insert data and, therefore, is the most important object for front-end applications.

IQDataSet supports two types of updating - immediate updating and batch updating. Using the first one all changes are immediately written to the database. The second type caches several changes and then sends them to the database in a single step.

As another feature, IQDataSet allows us to define different cursor types. It is possible to define dynamic and static cursors, read-only and updatable cursors, and forward-only cursors instead of scrollable cursors. Dynamic cursors allow us to see additions, changes, and deletions by other users whereas static cursors provide a static copy of the record set. Read-only cursors don't allow us to change the record set and can be used well in case of data needed to be protected from changing. Forward-only cursors support only scrolling forward through the record set and are faster then scrollable cursors.

The main tasks of the IQDataSet object are:

- Define cursor types.
- Add, delete and update data.
- Store information about fields and types.
- Provide information about the record set (e.g. position of cursor, record count, ...).
- Move the monitor within the record set (through First,Next,Previous,Last).
- Manage data type translation between database and application (e.g., translation of boolean types, handling and representation of Null values).
- Provide methods to read out Inova Q-specific information from the database (e.g., wether a field is an id field or not).
- Enable searching and sorting of data within the record set by implementing various methods.

4.5.5 IQCommand

The IQCommand object is the most specific object and is used to execute a single query against the database. The major role of this object is the handling of SQL queries and its corresponding parameters. It stores the command text, the type of the command and its parameters.

The command text contains the SQL query string or the name of the stored procedure or table. The type of this query distinguishes between a simple text query, a query to

invoke a stored procedure, and a query to target a table directly. According to this type the correct method (of the various offerd by JDBC) must be invoked. The parameters are stored together with the information about their type to assure type safety and invocation of the adequate *setObject* method.

By means of the IQCommand object, parameters can be replaced and new parameters can be appended. IQCommand provides some useful methods like a text replacement function and a function to sort the parameters in the correct order. The second method mentioned allows the software developer to add new parameters to a query without rewriting the query concerning the parameter order.

5 Prototyping the Application

The next migration step is to implement prototypes showing whether Java and Swing are capable to meet the requirements of the project. The prototypes shall be streamlined versions of the legacy application, reduced to the essentials. They should be equivalent in appearance and functionality and if possible identify potential for enhancement. They will provide the basis for automation and therefore a proper design is inevitable. In this section we describe the development of the different prototypes and the problems arising thereby. First, we will give a short introduction to IFL, the frame language used to specify the properties of an application.

5.1 Introducing IFL

IFL is a proprietary language developed by Inova Q. The following notes are mainly based on [Inova Q 2002] extended by experiences made by creating the models for the prototypes.

An IFL file serves as definition file for components and their attributes. These definitions are in a subsequent step processed by IGS. Listing 7 shows the IFL file for the first prototype and gives a general idea about its capabilities and principles of usage.

Listing 7: IFL-File for prototype 1

```
1  [Project]
2  Name=MUSTER1
3  Product="MUSTER1"
4  Description="MUSTER1"
5  Company="Inova Q Limited"
6  IRSFile=.\MUSTER1.IRS
7
8  [Database]
9  DataSource=.\MUSTER1.DDF
10
11 [Metrics]
12 PixelsX=800
13 PixelsY=600
14 TwipsPerPixelX=15
15 TwipsPerPixelY=15
16
17 [Globals]
18 DBLanguage=0
19 NumLanguages=3
20 Boolean=-1;0
21
22 [Form:MUSTER1]
23 BaseTable=T_MUSTER1
24 Caption=Musterprogramm Nr. 1
25 Style=MDIChild
26 DataLarge=False
27 Navigate=True
28 HasSearch=False
29
30 [Field:MUSTER1.MUS_DATETIME]
31 Type=Datetime
32
33 [Field:MUSTER1.MUS_TIME]
34 Type=Time
35
36 [Field:MUSTER1.MUS_OPTION]
```

```
37 ControlType=KeyPicker
38
39 [KeyPicker:MUSTER1.MUS_OPTION]
40 Type=RadioButtons
41 Values=A;0;B;1;C;2
```

An IFL file contains sections and definitions. Definitions identify values of specific attributes and are placed within sections. The definitions consist of a key, an assignment operator and a value. The key represents the attribute being identified by the definition. Depending on the attribute (*Key*), *Value* can contain either any string or a named enumeration constant. Sections like *[Project]* group the definitions into corresponding statements and tell IGS their specific meaning. In this context we can refer to sections as namespaces for definitions. Sections either contain zero, one or more definitions of an object (a form or a control) or just a definition of a file to include. The ordering of the sections is free, but following the *Object Model* (Project –> Forms –> Fields) increases understanding and maintainability.

The *[Project]* section covers general information about the project and provides the declaration of the name and the path of the IRS file to create. The abbreviation IRS denotes Inova Q Resource String. An IRS file is an access database containing the required SQL commands and descriptions of the various components used by the form. In this particular case, with regard to create a platform independent application, we will use simple text files instead of the Microsoft specific access database. The content of the text file will consist of tab delimited strings specifying the input data. Nevertheless, in the course of this project we will use the term IRS file for this kind of text files.

The *[Database]* section tells IGS where the data source defining the connection to the database is to be found. As container for these informations *Data Dictionary Files* (DDFs) are used. They describe the data in the database in terms of tables, columns, and indexes.

Within the *[Globals]* section in particular the definition of booleans is worth closer examination. The definition illustrates the database representation of boolean data. *T/F* defines boolean fields as 1-byte character fields where logical True corresponds to "T" and logical False corresponds to "F". *1/0* defines boolean fields as 1-digit numeric fields where logical True corresponds to 1 and logical False corresponds to 0, whereas *-1/0* defines boolean fields as 1-digit numeric fields where logical True corresponds to -1 and logical False corresponds to 0. To deal with these different representations the iQDao package (explained in Section 4) provides various methods.

The *[Form]* section contains definitions regarding the user interface (UI), e.g., the definition of the database table (table or updatable view) to which the form is bound. The *BaseTable* definition is obviously the most important definition since the information concerning the UI, the IRS files, and much more will be generated from the data provided by the database table.

The *[Field]* section provides additional information about controls. The term controls in this context denotes either fields in UI forms (like JTextField) or cells in tables (like JTable). For example, in lines 30 to 34 the fields MUS_DATETIME and MUS_TIME get values of another type than the one specified in the database. There they are declared as timestamp (just like MUS_DATE which doesn't change and therefore has not to be redefined here). In the UI, MUS_DATE will display only the date portion, MUS_DATETIME the date and the time portion, and MUS_TIME only the time portion of the database timestamp value.

Field MUS_OPTION is defined to be a *KeyPicker*. A *KeyPicker* is an I/O element to access, input, and update data using a key selection mechanism. If a control is declared to be of control type KeyPicker, then there has to be a *[KeyPicker]* section defining the type of the *KeyPicker*. The type of a *KeyPicker* can be one of the following: Listbox, Combobox, Search or RadioButtons. Listbox is a multiple column dropdown list box, Combobox a multiple column dropdown combo box, and Search a read-only text box of given data type plus button control (displays a toolbox-like search dialog used to pick the record). The *Values* definition is needed especially for *KeyPicker* of type RadioButtons to translate the values being present in the database to values that will be displayed in the UI.

Based on these definitions IGS automatically generates the required classes. These classes together with other (non-generated) basic classes and the IRS files make up the UI. According to the above made explanations Figure 15 shows a more detailed software development schema compared to the one shown in Section 1.3.

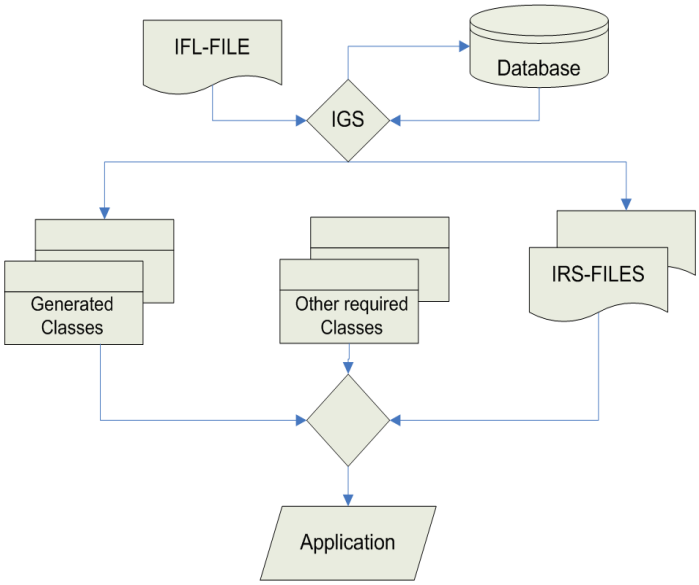


Figure 15: Detailed IGS Software Development Schema

Concluding this section two final remarks have to be made. First, the depicted sections and definitions represent only a small subset of the complete language specification of IFL. For the purpose of sketching out the basic functionality they will meet the requirements. Second, for mainly all values there exist default values being used by IGS if no explicit definition exists or an empty definition is created (e.g., *Caption* =). For instance, in the

[Form] section we have the possibility to define the appearance of the UI defining either *Presentation = Fields* or *Presentation = Grid*. The default value for this definition is Fields. Together with the other definitions such settings specify the UI for the first prototype shown in Figure 16.

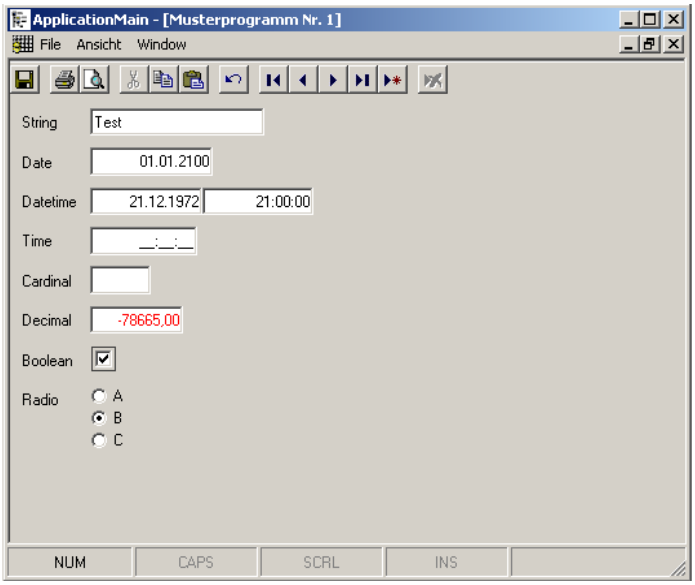


Figure 16: Prototype 1: UI with Fields

5.2 Prototyping approach

Using IGS together with the IFL file specifiations tailored for our needs we will generate three Visual Basic (VB) applications and try to convert them to equivalent Java applications using the techniques depicted in Section 2. We will make use of the VB code resulting from the generation process. Where possible we will apply simple language conversion of existing VB code. Where not possible we will treat the application as black box and try to establish the desired appearance and behaviour.

5.3 Prototype 1: UI with Fields

5.3.1 Specifications

The first prototype to develop is a UI containing fields for all columns in the particular database table and provides features to carry out operations on the supplied data. The user shall be able to move within the data tuples, updating, deleting, and adding data.

The specific requirements regarding this task are:

- All label, format, and type specifications have to be initiated at runtime, taken from information provided by the IRS files.
- The different fields have to be typed and be in responsibility for their appearance and behaviour.
- Data must be bound to a database table or result set.
- Navigation must be performed on the *ResultSet* received from the database.
- The application must support several languages and formats.

5.3.2 Architecture

The application starts by launching IQApplicationMain which subsequently invokes IQMainFrame – a container holding the various forms. In this example we have only one form, IQForm1 – an inner frame with several fields of different data types. Since the final application shall be capable to contain several forms, there has to be a common class used by all possible forms. This class implementing and providing general-purpose methods will be called IQCommon. To extract data out of the IRS files we created a class called IQNLSString. To assure each field being bound to the correct data type and being responsible for its appearance and behaviour, we designed a package named IQUserEntry. IQCommon makes use of these classes and packages and hands over the edited data and information to the particular forms. Beside these main classes there are some utility classes not mentioned here to keep the focus on the important classes. Figure 17 shows a class diagram describing the interaction of the main classes. Based on settings in the IFL file, IGS generates the main frame and the form and sets the specific values in the main class of the application.

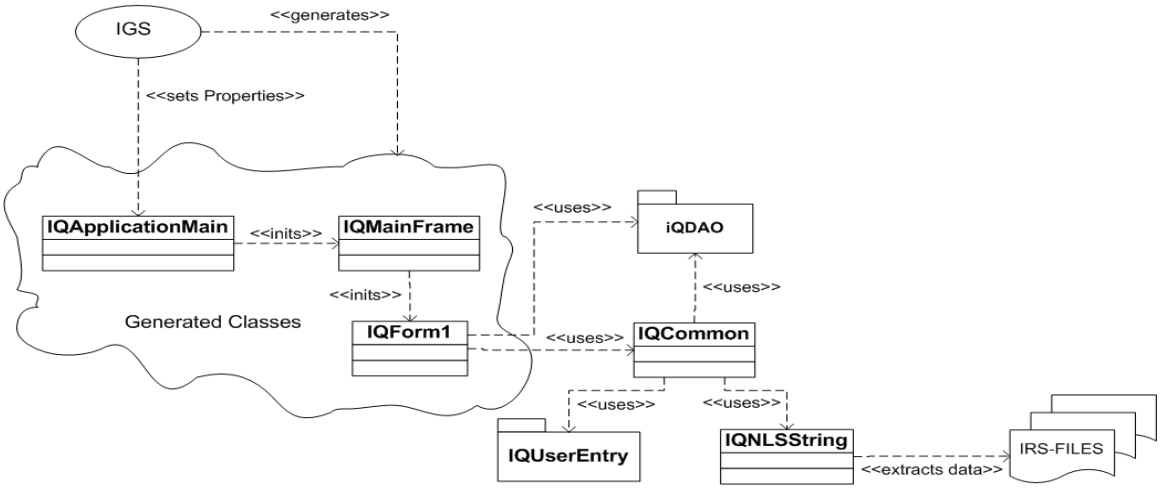


Figure 17: Architecture of prototype 1

5.3.3 Challenging Features

Typed fields. To support typed fields the enumeration class IQValueSubType from the IQDAO package has been extended. It got so-called *constant-specific methods* to facilitate the typing of the particular fields. According to [Sun 2004a] constant-specific methods are reasonably sophisticated. The best way to give each enum constant a different behaviour for some method is to declare the method abstract in the enum type and override it with a concrete method in each constant. The two new methods *getValueFieldType* and *getJavaClass* provide a static mapping between the data types defined in the IQDAO package and the Java internal types. Since we use the Swing class JTextField for the implementation of the different fields and JTextField treats all input as text, we need a method to convert the text string to an object of the specific value. This conversion will be done concerning the designated type of the particular field.

Listing 8 shows the use of such constant-specific methods in connection with the different Java date types. The Java date types are in the end all of the same IQDAO type, namely date, and we are in need of their distinction only in matters of appearance. In Listing 8, lines 45 and 46, we added definitions for two new classes simply by extending the existing java.util.Date class without providing new functionality.

Listing 8: ValueSubType enum with constant specific methods

```
1  public enum IQValueSubType
2  {
3        .
4        .
5        .
6    DATE
7    {
8      public final IQValueFieldType getValueFieldType()
9      {
10       return IQValueFieldType.DATE;
11     }
12     public final Class<?> getJavaClass()
13     {
14       return Timestamp.class;
15     }
```

```
16    },
17    DATETIME
18    {
19      public final IQValueFieldType getValueFieldType()
20      {
21        return IQValueFieldType.DATE;
22      }
23      public final Class<?> getJavaClass()
24      {
25        return Datetime.class;
26      }
27    },
28    TIME
29    {
30      public final IQValueFieldType getValueFieldType()
31      {
32        return IQValueFieldType.DATE;
33      }
34      public final Class<?> getJavaClass()
35      {
36        return Time.class;
37      }
38    },
39        .
40        .
41        .
42    public abstract IQValueFieldType getValueFieldType();
43    public abstract Class<?> getJavaClass();
44
45    public final class Datetime extends Date {}
46    public final class Time extends Date {}
47  }
```

Supporting several natural languages and formats. Several languages have to be supported by the application and according to the choosen language the appropriate date/time format and number format must be defined.

The business applications created by IGS must support the following languages:

- German
- Italian
- English
- French
- Spanish
- Portuguese

For this reason we created an enum class called IQNLSLanguage. Once again we use the formerly mentioned constant-specific methods to realize the intended behaviour. That way every enum constant has methods to return the language-specific date, time or datetime format.

Whereas the number format can easily be defined by

NumberFormat.getNumberInstance(m_enLanguage.getDefaultLocale());

for the date/time formats it was necessary to implement our own methods.

Java provides the two classes DateFormat and SimpleDateFormat to format dates. DateFormat has many class methods to obtain the default date/time formatters based on the default or a given locale, and a number of formatting styles. The formatting styles include FULL, LONG, MEDIUM, and SHORT [Sun 2004].

But while the invocation

DateFormat.getDateInstance(DateFormat.DEFAULT, Locale.GERMAN);

returns the desired result *09.08.2007* the following one

DateFormat.getDateInstance(DateFormat.DEFAULT, Locale.ENGLISH);

returns *Aug 9, 2007*, being completely insufficient for our needs because it is not a numeric date format.

Indeed it is possible to slightly affect the output of the method getDateInstance by varying the format style parameter. Unfortunately, the results for the six supported languages differ in such way that it is not possible to reduce them to a common denominator (as is seen easily on the example shown above).

For our intention the class SimpleDateFormat is more suitable. According to [Sun 2004] SimpleDateFormat is a class for formatting and parsing dates in a locale-sensitive manner. It supports formatting (date -> text), parsing (text -> date), and normalization. SimpleDateFormat allows us to choose any user-defined patterns for date/time formatting. Date and time formats are specified by date and time pattern strings.

We define these date and time pattern strings conforming to the windows region and language options. First, because they were used already by the original VB application and, second, because through the popularity of the windows OS these settings became a standard to a certain extent. Through the fixed coding of these date and time pattern strings in the constant-specific methods of the particular enum, the desired consistency for the three date display formats (date, time and datetime) is realisable in an easy, reliable, and independent manner. With the aid of the language property defined in the IFL-File the language, the format of date and time, and the format of numbers can be controlled globally.

Listing 9 gives an overview of the class IQNLSLanguage by showing the definitions for the languages German and Italian. Besides the particular constant-specific methods the class provides two common class methods to return the defined possible separators for date and time strings.

Listing 9: IQNLSLanguage enum

```
public enum IQNLSLanguage
{
      .
      .
      .
  GERMAN
  {
    public Locale getDefaultLocale()
    {
      return Locale.GERMAN;
    }

    public String getDateSeparator()
    {
      return DOT;
    }

    public String getTimeSeparator()
    {
      return COLON;
    }

    public String getDefaultDatePattern()
    {
      return "dd.MM.yyyy";
    }

    public String getDefaultTimePattern()
    {
      return "HH:mm:ss";
    }

    public String getDefaultDateTimePattern()
    {
      return "dd.MM.yyyy HH:mm:ss";
    }
  }
  ,
  ITALIAN
  {
    public Locale getDefaultLocale()
    {
      return Locale.ITALIAN;
    }
    public String getDateSeparator()
    {
      return SLASH;
    }

    public String getTimeSeparator()
    {
      return DOT;
    }
    public String getDefaultDatePattern()
    {
      return "dd/MM/yyyy";
    }
    public String getDefaultTimePattern()
    {
      return "H.mm.ss";
    }
    public String getDefaultDateTimePattern()
    {
      return "dd/MM/yyyy H.mm.ss";
    }
  }
  ,
      .
      .
      .
```

```
71   public Vector<String> getDateSeparators()
72   {
73     Vector<String> vDateSep = new Vector<String>();
74
75     vDateSep.add(SLASH);
76     vDateSep.add(DOT);
77     vDateSep.add(MINUS);
78
79     return vDateSep;
80   }
81
82   public Vector<String> getTimeSeparators()
83   {
84     Vector<String> vTimeSep = new Vector<String>();
85
86     vTimeSep.add(DOT);
87     vTimeSep.add(COLON);
88
89     return vTimeSep;
90   }
91
92   public abstract Locale getDefaultLocale();
93   public abstract String getDateSeparator();
94   public abstract String getTimeSeparator();
95   public abstract String getDefaultDatePattern();
96   public abstract String getDefaultTimePattern();
97   public abstract String getDefaultDateTimePattern();
98 }
```

Listing 10 gives a general idea about use and functionality of the IQNLSLanguage methods. Representative for the different date, time, and datetime editors we take the IQDateEditor class as example. Each field of a specific form (or in case of a table each column) gets its own editor even if there are several fields of the same data type. It is possible, as is seen in line 11, to pass a date format differing from default to the constructor. This possibility gives us the flexibility to define a different date or time format for every individual field (if desired). If this situation arises the method setFormat is invoked. This method verifies and consequently sets the format string by means of the designated separators. The class variable m_DateSep (defined either in line 18, in line 44 or in line 53) will be used to provide the desired functionality of date, time, and datetime fields by getting the separator string value from one of the earlier mentioned common methods in IQNLSLanguage. These fields are able to autonomously correct incomplete or even permuted inputs. We do so by examining the date or time lexically using the particular separator (either the separator for date or the separator for time, or even both in case of datetime fields) and subsequently setting the correct value following given rules for the specific fieldtype.

For example, assuming today's date to be 09.08.2007, the following corrections and/or completions are made:

1.1	to	01.01.2007 (date)	or	01.01.2007 00:00:00 (datetime)
1.31	to	31.01.2007 (date)	or	31.01.2007 00:00:00 (datetime)
45.12	to	01.12.1945 (date)	or	01.12.1945 00:00:00 (datetime)
1.1.19	to	01.01.2019 (date)	or	01.01.2019 00:00:00 (datetime)
1:12	to	01:12:00 (time)	or	09.08.2007 01:12:00 (datetime)
1.31 13:6	to	31.01.2007 13:06:00 (datetime)		

At a first glance these corrections and completions seem error prone, but if users get used to the logic behind, every-day work will be easier and faster.

Listing 10: Setting the format in editor

```
1
2  public final class IQDateEditor implements IQDefaultEditor
3  {
4    private SimpleDateFormat m_Format;
5    private String m_sFormat;
6    private IQNLSLanguage m_enLanguage;
7    private String m_sDateSep;
8          .
9          .
10         .
11   public IQDateEditor(IQUserEntry ucEntry, String sFormat)
12   {
13     m_enLanguage = ucEntry.getUserLanguage();
14
15     if(sFormat == null)
16     {
17       m_sFormat = m_enLanguage.getDefaultDatePattern();
18       m_sDateSep = m_enLanguage.getDateSeparator();
19     }
20     else
21       setFormat(sFormat);
22
23     m_Format = new SimpleDateFormat(m_sFormat);
24       .
25       .
26       .
27   }
28
29   .
30   .
31   .
32
33   private void setFormat(String sFormat)
34   {
35     Vector<String> vDateSep = m_enLanguage.getDateSeparators();
36     String sDateSep;
37
38     for(int i = 0; i < vDateSep.size(); i++)
39     {
40       sDateSep = vDateSep.elementAt(i);
41
42       if(sFormat.contains(sDateSep))
43       {
44         m_sDateSep = sDateSep;
45         m_sFormat = sFormat;
46         break;
47       }
48      }
49
50     // if Format String is not correct set Defaults
51     if(m_sDateSep == null)
52     {
53       m_sDateSep = m_enLanguage.getDateSeparator();
54       m_sFormat = m_enLanguage.getDefaultDatePattern();
55     }
56   }
57
58   .
59   .
60   .
61
62 }
```

The support of the above mentioned languages results in showing all captions (Labels,

Menus, MenuItems, Tooltips, ...) as well as all messages (Error messages, Warning messages, Question messages, ...) in the specific language.

In Visual Basic this differentiation can easily be done with resource files. These resource files are mainly used in larger software projects, which will be released in various language versions. As main advantage of resource files language-specific elements (captions, grafics) are not placed in program code, but are stored externally. Thus, for example, it is sufficient to link the resource file containing the resource-strings for another language to the project and all captions in the already compiled program will be adapted.

The original VB application loads the needed resource strings directly within code by means of the LoadResString(Integer idx) function. This function does nothing else than loading a character string from a *.res resource file. A required index *idx* of type integer identifies the declared data in the resource file. The LoadResString function can be used in code instead of character string constants. Storing and accessing long character strings in resource files reduces the loading time because elements can be loaded one by one, if required, instead of loading all simultaneously when loading the form object.

In this project we simulated this functionality using a class named IQRes. This class, shown in extracts in Listing 11, is initialized when launching the application and writes character strings in all needed languages to a data structure m_sRessourceStrings of type Hashmap<Integer,String>. At runtime the required string can be loaded. The Hashmap-Keys of the various languages differ only on the most significant digit. In application code it is possible to always language-independently use the same key for the same purpose. This key will later on (in the IQCommon class) be mapped correctly to the according language setting (line 99).

IQRes provides several methods to return the needed character string. LoadResString (Integer lID) in line 11 is used for standard captions (Labels, Tooltips,...) and reads entries like those illustrated in lines 63 to 71. LoadResString(String sFormName, Integer lID) in line 16 and LoadResString(String sFormName, String sFieldName, Integer lID) in line 24 send messages (Error-, Warning-, Info- and Questionmessages) to the user specifying the particular form and the field. In applications with several forms (Master/Detail applications described in Section 5.5, for example, can contain several forms) this fact makes it easier to identify the origin of the message. The Hashmap entries for this type of character strings are shown in lines 75 to 81. The method LoadResMenuString (Integer lID) in line 37 is designed for Menus and MenuItems and returns a string array of length two. The first field contains the caption for the menu whereas the second field contains the mnemonic to be defined for the menu. Lines 85 to 90 show the character string entries for Menus and MenuItems. On the basis of the &-symbol the mnemonic is defined. This &-symbol will be cut and the remaining character string will be the menu caption (shown in lines 106 to 108).

The Hashmap values in IQRes correspond exactly to the entries in the original *.res file. The methods to read out the character strings are named according to the VB method described above. As main difference between the original and the converted approach the objective in one case is a system file and in the other case a java class.

Listing 11: Ressource strings for different languages

```
1  public class iQRes
2  {
3    private final HashMap<Integer,String> m_sRessourceStrings;
4
5    public iQRes()
6    {
7      m_sRessourceStrings = new HashMap<Integer,String>();
8      Init();
9    }
10
11   public String LoadResString(int lID)
12   {
13     return  m_sRessourceStrings.get(lID);
14   }
15
16   public String LoadResString(String sFormName, int lID)
17   {
18     if(sFormName.toUpperCase().substring(0, 3).equals("FRM"))
19       sFormName = sFormName.substring(3, sFormName.length());
20
21     return  m_sRessourceStrings.get(lID).replace("$FORMNAME$", sFormName);
22   }
23
24   public String LoadResString(String sFormName, String sFieldName, int lID)
25   {
26     String s;
27
28     if(sFormName.toUpperCase().substring(0, 3).equals("FRM"))
29       sFormName = sFormName.substring(3, sFormName.length());
30
31     s = m_sRessourceStrings.get(lID).replace("$FORMNAME$", sFormName);
32     s = s.replace("$FIELDNAME$", sFieldName);
33
34     return s;
35   }
36
37   public String[] LoadResMenuString(int lID)
38   {
39     String[] sArray = new String[2];
40     String sMenuName;
41     String sMenuMnemonic;
42     String RessourceString;
43     int lPos;
44
45     RessourceString = m_sRessourceStrings.get(lID);
46     lPos = RessourceString.indexOf("&");
47     sMenuMnemonic = RessourceString.substring(lPos + 1, lPos + 2);
48     sMenuName = RessourceString.replace("&", IQConstants.EMPTY);
49     sArray[0] = sMenuName;
50     sArray[1] = sMenuMnemonic;
51
52     return sArray;
53   }
54
55   private void Init()
56   {
57     /*******************************************************/
58     /*                                                     */
59     /*                    DEFAULT (= ENGLISH)              */
60     /*                                                     */
61     /*******************************************************/
62
63     m_sRessourceStrings.put(10001, "Save");
64     m_sRessourceStrings.put(10002, "Print");
65     m_sRessourceStrings.put(10003, "Page preview");
66     m_sRessourceStrings.put(10004, "Cut");
67     m_sRessourceStrings.put(10005, "Copy");
68     m_sRessourceStrings.put(10006, "Paste");
69     m_sRessourceStrings.put(10007, "Undo");
70     m_sRessourceStrings.put(10008, "Go to first record");
71     m_sRessourceStrings.put(10009, "Go to previous record");
72     .
```

```
73      .
74      .
75      m_sRessourceStrings.put(10027, "$FORMNAME$: No record may be added currently.");
76      m_sRessourceStrings.put(10028, "$FORMNAME$: Error transfering Data to Database.");
77      m_sRessourceStrings.put(10029, "$FORMNAME$: Error moving to first record.");
78      .
79      .
80      .
81      m_sFixedStrings.put(10040, "$FORMNAME$: $FIELDNAME$ is required.");
82      .
83      .
84      .
85      m_sRessourceStrings.put(10101, "&File");
86      m_sRessourceStrings.put(10102, "&Info...");
87      m_sRessourceStrings.put(10103, "E&xit");
88      m_sRessourceStrings.put(10104, "&Window");
89      m_sRessourceStrings.put(10105, "Tile &horizontally");
90      m_sRessourceStrings.put(10106, "Tile &vertically");
91
92      .
93      .
94      .
95    }
96  }
97
98    //class IQCommon
99    public String[] LoadRESMenuString(IQNLSLanguage enLanguage, int lID)
100   {
101     lID = lID + 10000 * (enLanguage.ordinal());
102     return m_Res.LoadResMenuString(lID);
103   }
104
105   //class IQMainFrame
106   sMenuArray = m_Common.LoadRESMenuString(m_enLanguage, 10103);
107   mnuFileExit = new JMenuItem(sMenuArray[0]);
108   mnuFileExit.setMnemonic(sMenuArray[1].toCharArray()[0]);
```

Visibility of inserted data. The main problem concerning this task is that after an insert operation the inserted data is not available in the JDBC ResultSet. According to the official Sun Java documentation [Sun 2004] the *insertRow* method of the *ResultSet* class

> *"Inserts the contents of the insert row into this ResultSet object and into the database."*

Relying upon this statement, the inserted data should be visible both in the ResultSet and in the database. But [Bruce 1999] says:

> *"The ResultSet object uprs is updatable, scrollable, and sensitive to changes made by itself and others. Even though it is TYPE_SCROLL_SENSITIVE, it is possible that the getXXX methods called after the insertions will not retrieve values for the newly-inserted rows. Finally it depends on driver and DBMS."*

As recommendation he suggests:

> *To be absolutely sure that the getXXX methods include the inserted row values no matter what driver and DBMS is used, we can close the result set and create another one, reusing the Statement object stmt with the same query. A result set opened after a table has been changed will always reflect those changes."*

This suggestion can work fine with some smaller databases and custom software solutions tailored to fit their needs. But in our case we have to set up a system being able to handle database tables with data sizes unknown in advance. Thus,we cannot solve the problem in that way. If we think of database tables with several thousands of data tuples (which is usual in overall business projects) doing like suggested would lead to an unacceptable degradation of performance. If we bring to mind the objectives of IGS, referred to in Section 1.2, we will notice such a degradation to be just the opposite of our intention.

Analyzing [Oracle 2001] we finally find the statement:

> *"Internal INSERT operations are never visible, regardless of the result set type (neither forward-only, scroll-sensitive, nor scroll-insensitive)."*

Table 4 taken from [Oracle 2001] shows a summary of the visibility of internal and external changes made in an Oracle database. The term *Internal Changes* means the ability of a result set to see its own changes (DELETE, UPDATE, or INSERT operations within the result set) whereas *External Changes* are changes made from elsewhere (either from our own transaction outside the result set or from other committed transactions). The other terms used in this table are explained in Section 4.

ResultSet Type	Int. Delete	Int. Update	Int. Insert	Ext. Delete	Ext. Update	Ext. Insert
forward-only	No	Yes	No	No	No	No
scroll-sensitive	Yes	Yes	No	No	Yes	No
scroll-insensitive	Yes	Yes	No	No	No	No

Table 4: Visibility of Internal and External Changes

This behaviour leads to the conclusion that we have to regard a ResultSet as kind of snapshot. The ResultSet is actually not being updated by an Insert operation and therefore we will get inconsistencies between the data in the database and the data shown by the UI. Since we want to develop a platform independent software system, we cannot afford to depend on any driver implementations. Therefore, at this step in the project we have to adapt the implementation of the prototype in order to take care of these changed requirements. Using IGS the adaptation can simply be made by changing the definition in Listing 7, line 26 from *DataLarge=False* to *DataLarge=True* and add the statement *LargeKeys=MUS_DECIMAL*. These changes tell IGS to write code loading the data tuples one by one on the basis of the primary key identified by the *LargeKeys* definiton made above.

Since we aren't at this point yet, to achieve this behaviour we have to change the prototype's code manually. The application shall not load the whole data at the initialization of the particular form, but only the first data tuple. Then, while performing the various move operations being implemented (DoFirst, DoLast, DoNext, DoPrevious) only the requested data tuple is loaded to the system.

Therefore, we change the so-called *Master SQL Statement* used by the application. Instead of the previous simple SQL statement

"Select tblColumn1,...,tblColumnN from tbl"

loading the whole bunch of data at once we use now

"Select tblColumn1,...,tblColumnN from tbl where LARGEKEY $LargeOps$ $1".

The two parameters marked by the $-symbol are placeholders for the particular move operation and the requested primary key. This master statement will be used to create a so-called derived table. [Wells 2001] defines a derived table to be a table being created on-the-fly by a SELECT statement inside another SQL statement. It can be referenced as a regular table or view, but this referencing is possible only from the outer SQL statement. The difference between a base table and a derived table is that a base table is actually existing in the database whereas the derived table results of any table sub-queries. The following two listings, Listing 12 and Listing 13, show the differences between the old and the new version by means of the *DoNext* operation.

Listing 12: DoNext operation without DataLarge option

```
1  /*****************************
2  *  Class IQForm1       *
3  *****************************
4  /public boolean DoNext()
5  {
6    Integer lPos;
7        .
8        .
9        .
10   lPos = IQDataSet.getRow();
11   IQDataSet.GotoNext();
12
13   if(IQDataSet.getEOF())
14   {
15     IQDataSet.GotoRow(lPos);
16     return false;
17   }
18       .
19       .
20       .
21   RecordToForm();
22
23   return true;
24 }
```

Listing 13: DoNext operation with DataLarge option

```
/*****************************
*  Class IQForm1       *
*****************************/
public boolean DoNext()
{
  boolean blnOK = false;
  Vector<Object> v = new Vector<Object>();
  Integer lPos;
     .
     .
     .
  blnOK = IQCommon.GetLargeKeyValues(this, IQRecordSetOps.DONEXT, v);

  if(blnOK)
  {
    CreateRecordSets(v);
    RecordToForm();
  }

  return true;
}

/*****************************
*  Class IQCommon             *
*****************************/
public boolean GetLargeKeyValues(IQForm1 oForm, IQRecordSetOps enRsoOp,
                                                          Vector<Object> vKey)
{
  boolean blnRetVal = false;
  String sSql = oForm.getMasterSql();;
  Vector<Object> v = new Vector<Object>();
  int lCnt = oForm.getLargeKeyCount();;
  String sSel;
  IQDataSet ds = new IQDataSet();
  IQConnection cn = oForm.getConnection();

  switch (enRsoOp)
  {
  case DOFIRST:
     .
     .
     .
    break;
  case DONEXT:

    sSql = sSql.replace("$LARGEOP$", ">");

    for(int i = 0; i < lCnt; i++)
      v.add(oForm.getClass().getMethod(oForm.getLargeKeys(i) + "_GetValue",
                                       new Class[]{}).invoke(oForm, new Object[]{}));

    sSel = EMPTY;

    for(int i = 0; i < lCnt; i++)
    {
      if(sSel.length() > 0)
      sSel = sSel + ", ";
      sSel = sSel + oForm.getLargeKeys(i);
    }

    sSel = "SELECT MIN(" + sSel + ") AS LARGEKEY FROM (" + sSql + ") DERIVEDTBL";
    ds = cn.CreateDataReader(sSel, v);

    if(ds.getEOF())
      blnRetVal = false;
    else
    {
      if(IsNull(ds.getValue(0)))
      {
        blnRetVal = GetLargeKeyValues(oForm, IQRecordSetOps.DOLAST, vKey);
```

```
71        }
72        else
73          blnRetVal = true;
74      }
75      break;
76    case DOPREV:
77        .
78        .
79        .
80      break;
81    case DOLAST:
82        .
83        .
84        .
85      break;
86    default:
87      break;
88    }
89        .
90        .
91        .
92    return blnRetVal;
93  }
```

Due to the fact that we cannot move within a loaded ResultSet and have to send a query to the database for each data tuple, this solution will inevitably suffer from performance degradation, too. Furthermore, as we see in Listing 13, more time-consuming operations are required now in the Java code.

The old version shown in Listing 12 used the method *GotoNext* to move to the next tuple. The *GetRow* method in line 10 served only to store the actual row to assure that in case the next operation moves beyond the last row of the ResultSet the cursor can be positioned on the previous row. The *RecordToForm* method then displayed the choosen tuple in the UI form.

In contrast, the new version (where *DataLarge* option is set to true) is more complex. To correct the behaviour of the application we need the assistance of the *IQCommon class*, which leads to increased communication between the involved classes. In line 50 we have to use the *Reflection* mechanism described in Section 3 to invoke the adequate *XXX_getValue* method (again positioned in IQForm1). To avoid moving beyond range we even require recursion as we can see in line 70.

Despite the slowdown due to increasing complexity, this adjustment is still considerably faster than the suggestion already discussed earlier in this section.

As another slight drawback of the new version we now have database access logic in our application code. As described in Section 4, to ease maintenance all database related stuff shall be managed by the IQDAO package. That way re-platforming of back-end services could be made with minimal disruption to the application using the database. But now, having hard-coded SQL Strings in our application code, when changes are made these strings and the corresponding classes have to be adjusted.

Another challenge is caused by the fact that after an insert operation the database cursor is still positioned on the *Insert Row*. The *Insert Row* is defined in [Sun 2004] as:

> *"The insert row is a special row associated with an updatable result set. It is essentially a buffer where a new row can be constructed by calling the updater methods prior to inserting the row into the result set."*

After saving, we see in the UI the currently inserted data, but the database cursor is still positioned on the Insert Row buffer and not on the data tuple referring to the currently shown UI data. If we try to update this data now (maybe because we inserted incorrect data and want to correct it immediately), we will get a database error, since we are in a undefined state for performing updating or deleting operations. At the end of each Saving operation the application must be aware of whether an Insert operation was performed or not. If so, the application must simulate moving by going back and forth in the record set to ensure that the database cursor is positioned on the same data as shown in the UI.

Listing 14 shows the workaround for this issue.

Listing 14: Simulating Movement to leave the Insert Row

```
1  public boolean DoSave()
2  {
3    Vector<Object> v;
4      .
5      .
6    *** Here the Saving is done ***
7      .
8      .
9    if (m_blnAdding)
10   {
11     //Simulating DoPrev() and DoNext() to move away from the InsertRow
12     v = new Vector<Object>();
13     m_Common.GetLargeKeyValues(this, IQRecordSetOps.DOPREV, v);
14     CreateRecordSets(v);
15     RecordToForm();
16     v = new Vector<Object>();
17     m_Common.GetLargeKeyValues(this, IQRecordSetOps.DONEXT, v);
18     CreateRecordSets(v);
19     RecordToForm();
20
21     m_blnAdding = false;
22   }
23     .
24     .
25     .
26   return true;
27 }
```

5.4 Prototype 2: UI with Table

5.4.1 Specifications

The second prototype is a UI where the data is placed in a table. The basic process logic remains the same as in Prototype 1, but will be adapted to comply with new or changed requirements. Move operations are not required any more since the whole data is loaded and shown at once. The user again must be able to update, delete, and insert data.

The main specific requirements regarding this task are:

- Data must be unbound now (unaffected from the database model of the underlying data source).
- The columns of the table have to be typed.
- Columns can be hidden at runtime.
- Columns must be able to support KeyPicker components.
- Changes must not be written to the database straight away. They have to be retained and sent to the database only when a Saving operation is performed. Nevertheless, the table model and the model of the underlying database must be kept consistent.
- Undo of changes must be possible.
- The system performance must be satisfying.

Since the data is not bound to a database table or result set, the handling of the system is more efficient and flexible. We are able to display a column of data not originating from the data source, we can fill the UI table with data from different sources or even hide columns of data not to be viewed. Thus, we have several ways to display data. Several result set fields can be mapped to one form control. In the opposite direction we gain advantage of having complete control over the data being added to the database. As price for this flexibility unbound UI components have to be synchronized with the underlying database programmatically, and vice versa.

Through appropriate typing of the particular columns we have full control over their formatting. This characteristic includes control of text formats (such as Dates, percent, currency), word wrapping, alignment, borders, checking of boundaries for input values, and much more.

By means of the KeyPicker component the table provides multi-column ComboBoxes displaying a collection of objects. The ComboBox dropdown list can be formatted as a table with the power to define style attributes on each column independently or make each column share the same attributes.

Figure 18 shows the generated VB prototype for this task.

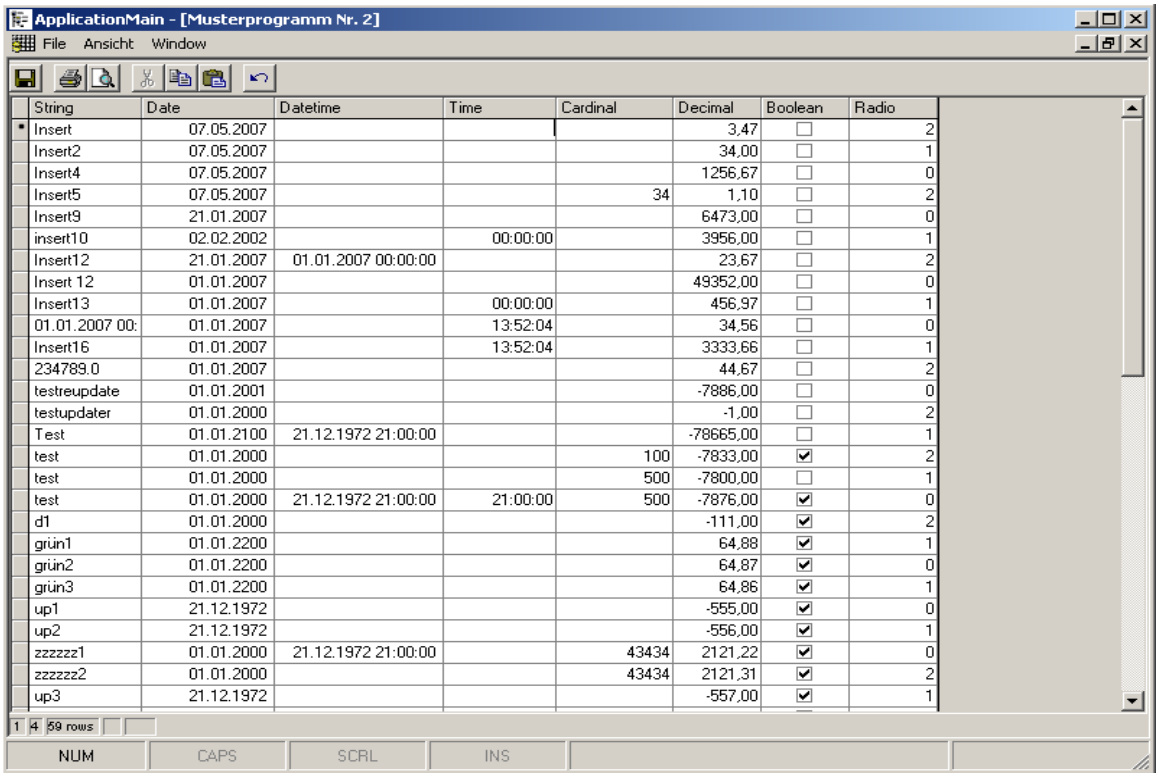


Figure 18: Prototype 2: UI with Table

5.4.2 Architecture

The architectural framework consists essentially of the same classes as in Section 5.3.2. The IQCommon class has got new methods to initialize the UI table's naming and typing declarations. But, in contrast to the former approach it makes no longer use of the IQUserEntry package mainly because the behaviour of the table cells differs from the fields implemented in Prototype 1. As we see in Figure 19 the particular form (here IQForm2) is now connected with its IQTable class doing the main part of the view process logic. Corresponding to Swing's UI-Delegate paradigm (see Section 3.4.1) the table object uses a table model object to manage the actual table data. For performance reasons the cells in Swing tables are not implemented as stand-alone components [Geary 2002]. Instead, for all cells containing data of the same type a given cell renderer is designated. The cell renderer performs the task to appropriately format the data in each cell. As soon as the user starts to edit a specific cell the predefined cell editor takes control over the cell and monitors its editing. In our case, if needed, the editor makes use of a document class. Due to [Sun 2004] this class serves as a container for text, supporting editing and providing notification of changes. For each type of column we provide the particular renderer, editor, and document class. Figure 19 shows the architecture of Prototype 2.

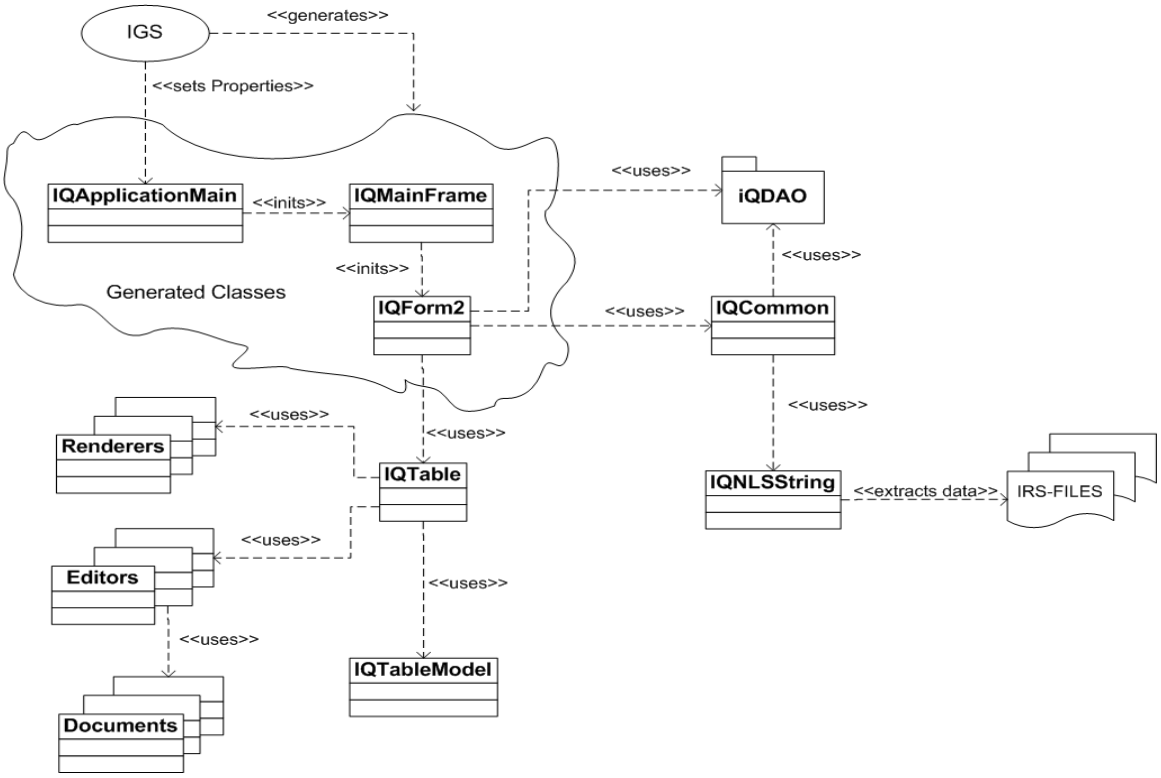


Figure 19: Architecture of Prototype 2

5.4.3 Challenging Features

Initializing the table automatically. All information concerning the table must be taken from a tab-delimited text file and the UI view must generate itself dynamically. Table 5 shows a shortened version of the IRS file containing the most important parameters to build the UI of the application. Each row in the file stands for a tuple in the database and will be mapped to a column of the table. The information is used to define the data type, the table header's name, if a value is required in the particular column, the alignment and format of the viewed values, and the length of the column.

FORMNAME	GRIDNAME	COLUMNID	IGSTYPE	FIELDNAME	REQUIRED	ALIGNMENT	LENGTH	FORMAT
IQFORM2	GRID00	0	3	MUS_STRING	0	0	20	
IQFORM2	GRID00	1	2	MUS_DATE	-1	1	10	
IQFORM2	GRID00	2	7	MUS_DATETIME	0	1	18	
IQFORM2	GRID00	3	8	MUS_TIME	0	1	8	
IQFORM2	GRID00	4	0	MUS_CARDINAL	0	1	6	####0
IQFORM2	GRID00	5	1	MUS_DECIMAL	-1	1	10	#####0.00
IQFORM2	GRID00	6	6	MUS_BOOLEAN	-1	1	1	0
IQFORM2	GRID00	7	0	MUS_OPTION	-1	1	3	#0

Table 5: IRS File for Table Definitions

The extracting and storing of the information from the text file is done by the class IQNLSString. The following method *InitializeGrid* is stated in IQCommon and is invoked

by IQForm2. As parameter it gets the specific IQTable and the name of the Form. The name is needed to allow us to locate the adequate text file, since there can be more than one form in the final business application. In Listing 15 line 12, *m_NLS* is the IQNLSString object providing the extracted data. With this data a vector of type IQStructuredStorage is filled. The IQStructuredStorage class is a user defined type designed to hold the data in a structured way and provide methods to easy access the data. In lines 17 to 54, the specification of each column in the database table is mapped to the conforming column in the application.

Assigning column editors. In listing 15 beginning with line 33, we try to define the desired editor on basis of the type and subtype set by the IQDAO package. Here we run into a remarkable pitfall caused by the fact that when in Java a *fireTableStructure* event is fired the table column model is recreated from the data model. This recreation clears any existing columns before creating the new columns based on information from the model. The *DefaultTableModel* we are extending by our IQTableModel does not support adding of columns directly. Instead, it notifies its listeners of its complete change. This information causes the table to throw away the old table columns and create new ones. All the information stored in this table columns is lost. To turn this behaviour off we have to call *JTable.setAutoCreateColumnsFromModel(false)*. The slight drawback we have to bear is that modifications concerning table columns must be defined explicitly by hand now. In turn we will have full control over the behaviour of the columns.

Listing 15: Initializing the table

```
1  protected void InitializeGrid(IQTable tTable, String sFormName)
2  {
3    int lColCnt;
4    Vector<IQStructuredStorage> vCol;
5    String sTableName = tTable.getName();
6    int lColumnType;
7    String sFormat = null;
8    int lLength;
9
10   sTableName = sTableName.trim().toUpperCase();
11
12   vCol = m_NLS.getGridDef(sTableName);
13   lColCnt = m_NLS.getGridFieldCount();
14   tTable.setColumnCount(lColCnt);
15
16
17   for(int i = 0; i < lColCnt; i++)
18   {
19     tTable.setColumnHeader(i, vCol.elementAt(i).getValue("FIELDNAME").toString());
20     tTable.setTableColumn(i, getTableColumnSizeFactor((Integer)vCol.
21                                                elementAt(i).getValue("IGSTYPE")));
22
23     tTable.setColumnClass(i,getValueSubType((Integer)vCol.elementAt(i).
24                                       getValue("IGSTYPE")).getJavaClass());
25
26     lColumnType = (Integer)vCol.elementAt(i).getValue("COLUMNTYPE");
27
28     if(!m_Misc.isEmptyNull(vCol.elementAt(i).getValue("FORMAT").toString()))
29       sFormat = vCol.elementAt(i).getValue("FORMAT").toString();
30
31     lLength = (Integer)vCol.elementAt(i).getValue("LENGTH");
32
33     if(lColumnType == IQControlType.USERENTRY.ordinal())
34     {
35       tTable.setControlType(i, IQControlType.USERENTRY);
36       tTable.setColumnEditor(i, sFormat, lLength,getValueFieldType((Integer)vCol.
```

```
37                  elementAt(i).getValue("IGSTYPE")), getValueSubType((Integer)vCol.
38                  elementAt(i).getValue("IGSTYPE")));
39      }
40      else if(lColumnType == IQControlType.CHECKBOX.ordinal())
41      {
42        tTable.setControlType(i, IQControlType.CHECKBOX);
43        tTable.setColumnEditor(i, sFormat, lLength, getValueFieldType((Integer)vCol.
44                  elementAt(i).getValue("IGSTYPE")), getValueSubType((Integer)vCol.
45                  elementAt(i).getValue("IGSTYPE")));
46      }
47      else if(lColumnType == IQControlType.KEYPICKER.ordinal())
48      {
49        tTable.setControlType(i, IQControlType.KEYPICKER);
50        tTable.setColumnEditor(i, sFormat, lLength, getValueFieldType((Integer)vCol.
51                  elementAt(i).getValue("IGSTYPE")), getValueSubType((Integer)vCol.
52                  elementAt(i).getValue("IGSTYPE")));
53      }
54    }
55  }
```

Varying column widths

> *"By default, all columns in a table start out with equal width, and the columns automatically fill the entire width of the table."*[Sun 2004]

This behaviour is not what we really want. Our intention is that the width of each column conforms to the values within the column. Unfortunately, this is not possible since the only way to fit each column to its greatest value requires a walk through all of its values. For performance reasons we reject this solution. But again we can make use of the dedicated type of a column. As we see in Listing 15 line 20, we create a method *getTableColumnSizeFactor* returning an integer factor depending on the type of the column. On setting the table column we call the *TableColumnModel.setPreferredWidth* method with a defined basic size multiplied with the regarding factor. Once again, this approach is only possible if the automatic recreation of the table is turned off.

Storing the data. One of the most important considerations regarding this task is to define the place where the data shall be stored. The application must be able to deal with a great bunch of data and still show sufficient performance. For this reason a *Vector* (which usually is the first structure to come into mind) is no proper container to store the data into. A *Vector* is basically a thread-safe array (it is synchronized) and grows dynamically to allow us to add new elements. This synchronization overhead makes access to vectors very slow [Dale et al. 2002]. Because of that an *ArrayList* is the better alternative. An *ArrayList* is a non-synchronized *Vector*. It is faster than a *Vector* and cleaner to use since it need not support the legacy methods of the *Vector* class. The following Listing will give an idea of the performance gap between a *Vector* and an *ArrayList*.

Listing 16: Performance comparison between Vector and ArrayList

```
1  import java.util.*;
2
3  public class PerformanceTest
4  {
5    public static void main(String args[])
6    {
7      ArrayList<Object> oArrayList = new ArrayList<Object>(1000000);
8      Vector<Object> oVector = new Vector<Object>();
9      Iterator iterator;
10     long startTime = 0;
11     long endTime = 0;
12
13     startTime = System.currentTimeMillis();
14
15     for (int i = 0; i <= 999999; i++)
16     {
17       oVector.add(new Object());
18     }
19
20     endTime = System.currentTimeMillis();
21     System.out.println(
22          "Load Vector with 1.000.000 objects: " + (endTime - startTime));
23
24     startTime = System.currentTimeMillis();
25     iterator = oVector.iterator();
26
27     while(iterator.hasNext())
28     {
29       Object o = iterator.next();
30     }
31
32     endTime = System.currentTimeMillis();
33     System.out.println(
34          "Iterate through Vector with 1.000.000 objects: " + (endTime - startTime));
35
36     startTime = System.currentTimeMillis();
37
38     for (int i = 0; i <= 999999; i++)
39     {
40       oArrayList.add(new Object());
41     }
42
43     endTime = System.currentTimeMillis();
44     System.out.println(
45          "Load ArrayList with 1.000.000 objects: " + (endTime - startTime));
46
47     startTime = System.currentTimeMillis();
48     iterator = oArrayList.iterator();
49
50     while(iterator.hasNext())
51     {
52       Object o = iterator.next();
53     }
54
55     endTime = System.currentTimeMillis();
56     System.out.println(
57          "Iterate through ArrayList with 1.000.000 objects: " + (endTime - startTime));
58   }
59 }
```

The test shown in Listing 16 was performed on a 2,8 GHz Pentium 4 processor under Windows XP SP 2 using java build 5.0. Comparing the outputs (showing the mean times of running this program a hundred times) in Table 6 demonstrates that loading the elements from an *ArraList* is quite three times faster than loading them from an *Vector*. The performance difference concerning the iteration is even greater. Iterating through the elements of an *ArrayList* is almost four times faster than iterating through

the elements of a *Vector*.

Description	Time
Load Vector with 1.000.000 objects	445
Iterate through Vector with 1.000.000 objects	213
Load ArrayList with 1.000.000 objects	153
Iterate through ArrayList with 1.000.000 objects	57

Table 6: Mean output after running program 100 times

Identifying operational modes. According to the requirement that consistency has to be guaranteed although data must be written to the database only at particular times we introduce different operational modes. We have to distinguish between original rows in the UI table, updated rows, and newly inserted or deleted rows. We obtain this distinction by implementing appropriate data structures handling this challenge. The IQEvent enum denotes the type of operation to perform and sets the IQRowOperationMode enum to mark the state of the row. The user defined data structure IQRowOperation is a container for the the row's actual state and its corresponding database index. The application undergoes a remarkable change from its VB counterpart. Due to Swings inherent UI-Delegate logic we need only three events, namely *AddNew*, *Change*, and *BeforeDelete* to perform this task compared to the various events the VB application needs. Each time a table cell looses the focus Swing fires an event to invoke the method *setValue* of the specified table model and to cause repainting of the view. Thus, in this method implemented in our IQTableModel class, we put the whole logic concerning state validation and setting of appropriate modes.

Deleting data. The main difficulty was the implementation of deleting. The complexity hereby arises from the different indices between the data in the model and the data in the database. Since the model is responsible to provide the data for the view, the data index of the model must always be the same as the data index of the view. After performing several deletes there will be no more compliance between the indices of the database on the one hand and the indices of model and view. To solve this problem each row in the UI table (view) is linked with two different data structures. First, the *ArrayList* holding the data of the row, and, second, a *Vector* of type IQRowOperation holding the mode and the index of the row. The final delete of such a row in the database will be put into effect by the index of the particular IQRowOperation value and be uncoupled from model and view.

The fact that each row is aware of its state and index is again of great value when executing a saving operation. There will be an iteration over all rows (and conforming to the actual state) the affected rows (all rows no more being original state) will be stored in ascending order. The deletes performed in the database will be added to a counter variable and subtracted on subsequent database accesses within this operation to ensure correct indexing.

Listing 17 demonstrates the just mentioned features on basis of a shortened version of *SaveGrid*. This method (stated in the IQCommon class) gets the specific IQTable object,

the IQDataSet object to perform the operations on, and a Vector of type IQRowOperation that is used to get the original indices of rows already deleted from the model. The *AffectedRecords* method invoked in Line 13 is responsible for some important prework to be done. It clears out the unnecessarily deleted rows. Those rows have been added and then deleted, and therefore shall be ignored further on. After the cleaning it merges the remaining deleted rows with the other affected rows (updated or inserted rows) and sorts them in ascending order. The required operation to accomplish then depends on the IQRowOperationMode of the particular row. Line 26 shows the mapping of the different indices between model/view and database by using the *lIndex* value of the IQRowOperation value and incorporating indices of database rows already being deleted.

Listing 17: Saving operation

```
1  protected boolean SaveGrid(IQTable oTable, IQDataSet ds,
2                             Vector<IQRowOperation> udtDeletedRows)
3  {
4    IQRowOperationMode enStatus;
5    int lNumAffectedRows;
6    boolean blnOK = true;
7    int lDeletedRows;
8    int lActualRow = -1;
9    Vector<IQRowOperation> udtAffectedRows =  new Vector<IQRowOperation>();
10   IQRowOperation udt = new IQRowOperation();
11
12   enStatus = IQRowOperationMode.NONE;
13   lNumAffectedRows = AffectedRecords(oTable, udtDeletedRows, udtAffectedRows);
14
15   lDeletedRows = 0;
16
17   if(lNumAffectedRows > 0)
18   {
19     for(int i = 0; i < lNumAffectedRows; i++)
20     {
21       switch (udtAffectedRows.elementAt(i).enMode)
22       {
23       case DELETED:
24         udtDeletedRows.clear();
25         enStatus = IQRowOperationMode.DELETED;
26         lActualRow = udtAffectedRows.elementAt(i).lIndex - lDeletedRows;
27         ds.GotoRow(lActualRow);
28         ds.DeleteRec();
29         lDeletedRows++;
30         enStatus = IQRowOperationMode.NONE;
31       break;
32       case UPDATED:
33         enStatus = IQRowOperationMode.UPDATED;
34         lActualRow = udtAffectedRows.elementAt(i).lIndex - lDeletedRows;
35         ds.GotoRow(lActualRow);
36         ds.EditRec();
37
38         blnOK = (Boolean)oTable.InvokeCheckRules(lActualRow);
39
40         if(blnOK)
41         {
42           oTable.InvokeFormToRecord(enStatus, lActualRow);
43           ds.UpdateRec();
44           enStatus = IQRowOperationMode.NONE;
45         }
46         else
47         {
48           enStatus = IQRowOperationMode.UPDATED;
49           return false;
50         }
51
52         udt.enMode = enStatus;
53         udt.lIndex = lActualRow;
54         udt.lRow = udtAffectedRows.elementAt(i).lRow;
```

```
55            oTable.ChangeRowData(udtAffectedRows.elementAt(i).lRow, udt);
56          break;
57          case INSERTED:
58            .
59            .
60            .
61            .
62  }
```

Inserting data. For insertion of data we append an additional empty row to the UI table view. Analogously to the JDBC buffer for inserting we name this additional row *InsertRow*. To create the InsertRow we provide a method called *MakeInsertRow* that gets called in the beginning when the table is being populated and every time when the user wants to insert a new row. Anytime the users inserts a value in the InsertRow a new InsertRow is created. Since every insert into a table cell invokes the *setValue* method of the table model, to obtain the desired behaviour we only have to check in this method if the user is on the actual insert row and create a new InsertRow if necessary. Thus, we obtain the equivalent behaviour as the sample application. This insertion logic allows us to easily distinct between "old" (coming from database) and "new" (newly inserted) rows. Respectively we can define the proper operational mode (either *Change* or *AddNew*).

Explicitly invoking setValue. The table model defines the data to be viewed in the table and writes successfully edited data back to the model using the already mentioned method *setValue(Object value)*. This writing is done every time the cell looses the focus in the table. At first glance this behaviour seems satisfactory. Nevertheless, it is the cause for many problems arising in the field of implementing application logic in user interfaces containing a table. For instance, the invocation of *setValue* doesn't work if the user clicks somewhere outside of the table, e.g., a button or the table header (is not part of the table). If we want to save the modifications by clicking on the button performing the saving operation, the table-related cursor is still positioned in the cell. Hence, *setValue* was not invoked and the table model has still the old value for this cell. The value actually being saved to the database is not the value viewed in the table, but the value in the model.

Another pitfall is described by the following example. A cell in the last row of the table view has the focus. Now we decide to delete another row. We can do this by choosing the row in the row header (is not part of the table, too) and pushing the *Delete button*. Though we moved around in our application we didn't click somewhere in the table and therefore the cell in the last row still owns the table focus. Now we want resume editing this particular row and click in another cell of the row. We will get an *ArrayOutOfIndexException* since setValue is invoked now with the former row index not existing anymore. To avoid such inconveniences the application logic must be analyzed and the methods *JTable.editCellAt* and *AbstractCellEditor.fireEditingStopped* have to be implemented in the table and editors to explicitly force the table model to write data to the model and, therefore, guarantee consistency with the view.

Update of newly inserted data. Once again we have the problem that newly inserted data is not visible in the current result set. The solution introduced in Section 5.3.3 on page 62 can't be adopted for the table prototype, since the table must show all result tuples at once and moreover navigation by means of buttons (DoFirst, DoNext,...) is not realised. Therefore, we need a different solution of this problem. Refreshing the table data with the actual data from the database will not work - not only because of performance reasons, but mainly because the position of the newly inserted data depends on database inherent logic and, therefore, it is not possible to bring it in accordance with the position in the table view. The newly inserted data always must remain at the end of the table. Since JDBC offers nothing to add a new record to the existing result set, we have to build our own "result set" for newly inserted data. We do so by creating a new class named IQInsertedData as inner class of IQRecordSet. The data structure to simulate a result set will be an ArrayList<HashMap<String,Object>> with the name of the database table field (corresponds to the UI table column name) as key. Any operation on database data performing not on the original result set will be directed to the database by means of this data structure, as shown in Figure 20.

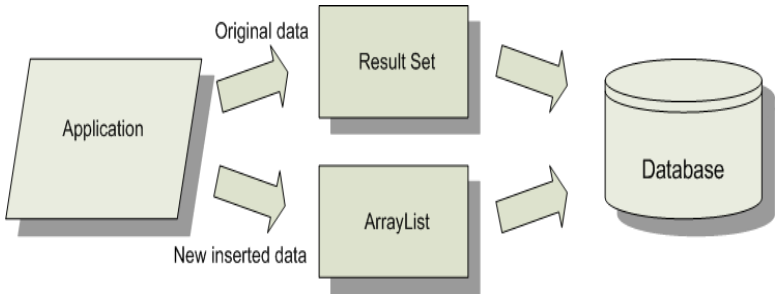


Figure 20: Simulating a ResultSet

First, we have to make preparations in the application data to be able to distinguish between original data and new data. We introduce three new row operation modes, namely *SavedInsert*, *UpdatedAfterInsert* and *DeletedInsert*. The first one, SavedInsert is used to differ between data being only inserted and data being already saved. For updates and deletes we need these extra operation modes since the application must know the adequate path to the database to use. The usual one for operations concerning original data or the one through IQInsertedData.

As long as data have not been saved they exists only in the UI view and model and operations on these data take place only there. After being saved the data exist in the database and every operation must be performed there, too. A table row holding new data is in Inserted mode. When this data must be saved it is written to JDBC's Insertrow buffer and to a temporary buffer in IQInsertedData. Only when the data were saved to the database and thereby were checked to be correct, they are written in the ArrayList and the mode is changed to SavedInsert. If then these data must be updated, we not only have to change the values in the ArrayList, but also the values in the database. Otherwise we would miss the verification of the data on the part of the database (primary key violations, not null values,...). To find the currently inserted row in the databases again we need *LargeKeys* (see Section 5.3.3 on page 62). Using the primary key value(s) we create a new jdbc PreparedStatement

Update sTableName set values.... where sPrimaryKey1 = pkValue1 [... and sPrimaryKeyN = pkValueN]

to update the particular row. Again we must not write the update to our data structure until the database update succeeds.

Listing 18 shows the used methods to insert data in the database. The method FormToRecord (stated in the particular Form class) gets the index of the row to manipulate and its designated IQRowOperationMode. Depending on the mode it performs the desired operation. As we see in line 11 and in line 42 before an insert or update (after insert) operation is done the data is stored in a temporary buffer. In case of the insert operation the method Insert (stated in IQRecordSet) stores the data written to the jdbc InsertRow buffer to the database (line 76). If no exception occurs in line 77 the method InsertRow stated in IQInsertedRow is invoked. However, the index of the row in the UI is not the same as in the ArrayList, which holds only the currently inserted data. Therefore, we defined a class variable m_lInsertedRowCount initialized to zero and being incremented every time an insert operation is to be performed. The method InsertRow finally does nothing else then copying the values from the temporary buffer to the ArrayList.

Listing 18: Inserting data to the database

```
1  public boolean FormToRecord(IQRowOperationMode enFlag, Integer lRow) throws IQException
2  {
3    boolean blnRetVal = true;
4    Object oValue;
5
6    try
7    {
8      // Insert
9      if(enFlag.ordinal() == 1)
10     {
11       m_oMasterRecordSet.AddTempRow();
12
13       oValue = MUS_STRING_tf_GetValue(lRow);
14       m_oMasterRecordSet.setValue("MUS_STRING", oValue);
15       m_oMasterRecordSet.InsertTempData("MUS_STRING", oValue);
16                          .
17                          .
18                          .
19       oValue = MUS_OPTION_tf_GetValue(lRow);
20       m_oMasterRecordSet.setValue("MUS_OPTION", oValue);
21       m_oMasterRecordSet.InsertTempData("MUS_OPTION", oValue);
22     }
23
24     // Update rows from the original recordset
25     else if(enFlag.ordinal() == 0)
26     {
27       if(lRow < m_oMasterRecordSet.getInitialRecordCount())
28       {
29         m_oMasterRecordSet.setValue("MUS_STRING", MUS_STRING_tf_GetValue(lRow));
30                                    .
31                                    .
32                                    .
33         m_oMasterRecordSet.setValue("MUS_OPTION", MUS_OPTION_tf_GetValue(lRow));
34       }
35     }
36
37     //Update newly inserted rows
38     else if(enFlag.ordinal() == 5)
39     {
40       if(lRow >= m_oMasterRecordSet.getInitialRecordCount())
41       {
42         m_oMasterRecordSet.AddTempRow();
```

```
43
44          m_oMasterRecordSet.InsertTempData("MUS_STRING", MUS_STRING_tf_GetValue(lRow));
45          .
46          .
47          .
48          m_oMasterRecordSet.InsertTempData("MUS_OPTION", MUS_OPTION_tf_GetValue(lRow));
49
50          m_oMasterRecordSet.UpdateInsertedRow(FORM2_TBLNAME, lRow);
51        }
52      }
53
54      //Delete newly inserted rows
55      else if(enFlag.ordinal() == 6)
56      {
57        if(lRow >= m_oMasterRecordSet.getInitialRecordCount())
58        {
59          m_oMasterRecordSet.DeleteInsertedRow(FORM2_TBLNAME, lRow);
60        }
61      }
62    }
63    catch(IQException e)
64    {
65      blnRetVal = false;
66      throw(e);
67    }
68
69    return blnRetVal;
70  }
71
72
73
74  protected void Insert() throws SQLException
75  {
76    m_rs.insertRow();
77    m_udtInsertedData.InsertRow(m_lInsertedRowCount);
78    m_lInsertedRowCount++;
79  }
80
81
82
83  private void InsertRow(Integer lPos)
84  {
85    m_InsertData.add(lPos,new HashMap<String, Object>());
86
87    for(int i = 0; i < m_TempData.size(); i++)
88    {
89      m_InsertData.get(lPos).putAll(m_TempData.get(0));
90    }
91  }
```

Listing 19 shows the update of inserted data. Here we get the correct index by subtracting the original record count stored in IQRecordset class variable m_lRecordCount from the UI row index. In opposition to the requirement that no SQL code must appear hard coded (see Section 4.1) here we must make an exception to meet the needs of this application. We create a new PreparedStatement object used only to perform the update operation. We take the values to update from the temporary buffer and again do not write the data to the ArrayList (line 51) until the database update has succeeded.

Listing 19: Updating inserted data

```
1  protected void UpdateInsertedRow(String sTableName, int lPos)
2                                        throws SQLException, IQException
3  {
4    //to get the position in the m_InsertData udt
5    int lIndex = lPos - m_lRecordCount;
6
7    // to update the value in the DB
```

```
 8    PreparedStatement pstmt = null;
 9    String sUpdateString = "UPDATE " + sTableName + " SET ";
10    int lLast = m_TempData.get(0).size() - 1;
11
12    for(int i = 0; i < lLast; i++)
13    {
14      sUpdateString = sUpdateString + m_vFieldNames.elementAt(i) + " = ?, ";
15    }
16
17    sUpdateString = sUpdateString + m_vFieldNames.elementAt(lLast) + " = ? ";
18    sUpdateString = sUpdateString + " WHERE " + m_vPrimaryKeys.elementAt(0) + " = ?";
19
20    for(int i = 1; i < m_vPrimaryKeys.size(); i++)
21    {
22      sUpdateString = sUpdateString + " AND " + m_vPrimaryKeys.elementAt(i) + " = ?";
23    }
24
25    pstmt = m_ActiveConnection.prepareStatement(sUpdateString);
26
27    for(int i = 0; i <= lLast; i++)
28    {
29      Object oValue = m_TempData.get(0).get(m_vFieldNames.elementAt(i));
30      int lType = m_Misc.getValueType(oValue);
31      this.setObject(pstmt, i+1, lType, oValue);
32    }
33
34    for(int i = 0; i < m_vPrimaryKeys.size(); i++)
35    {
36      Object oValue = m_InsertData.get(lIndex).get(m_vPrimaryKeys.elementAt(i));
37      int lType = m_Misc.getValueType(oValue);
38      this.setObject(pstmt, lLast + 2 + i, lType, oValue);
39    }
40
41    // Update value in DB
42    pstmt.executeUpdate();
43
44    if(pstmt != null)
45    {
46      pstmt.close();
47      pstmt = null;
48    }
49
50    // Update row in ArrayList
51    m_InsertData.get(lIndex).putAll(m_TempData.get(0));
52
53  }
```

5.5 Prototype 3: Master/Detail UI

5.5.1 Specifications

A so-called Master/Detail UI is an application having a master form that can be linked to several detail forms. The IFL-file specifies the database tables needed for this purpose and the values of the master table to which the details are linked. There can be more than one detail since every one can be linked to another value. As another specific feature the appearance of the individual forms (components) can be configured. By means of a fraction parameter it is fixed how much vertical place each form gets when the application starts. Moreover, it is possible to place the different forms on different pages. These pages can be choosen by clicking on the page header. In such a case some fields of the master form can be displayed in an exposed way and are therefore visible

even when the remaining master data is covered (e.g., by selecting another page). This approach eases the assignment of detail data to the appropriate master data. The master form by definition can contain only fields, but no grid (due to the 1-to-many relation between master and details).

The requirements regarding to these specifications are:

- It must be possible to define the fraction of each form.
- Placing one (or more) forms on different pages.
- The source of an error (e.g., by writing data to the database) must be definitely identifiable.
- The focus behaviour must be adapted/expanded.

Figure 21 shows the generated Master/Detail VB prototype.

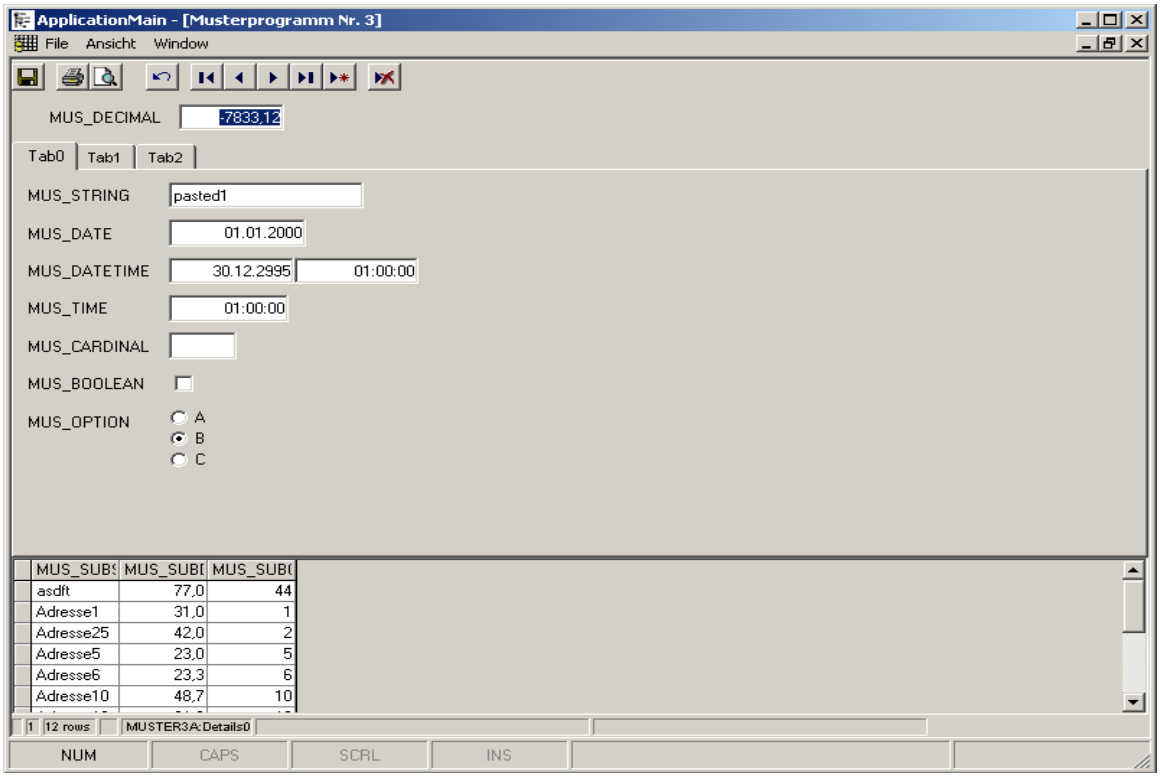


Figure 21: Prototype 3: Master/Detail UI

5.5.2 Architecture

There is not much to say about the architecture of this prototype. The Master- and Detailforms are mainly made up of the two forms developed before. Each form (besides the master form that can only be a form with fields) is either a form with fields or a form

with a table. These already designed forms can be taken without modifications from the previous prototypes and altered to a slight extend to satisfy the above mentioned requirements. The only architectural change to be made is an internal one that affects the class IQForm. The container layout has to be changed because it was not designed to be used with more than one form and misses the flexibility to arrange the various forms according to the stated fraction.

5.5.3 Challenging Features

Handling multiple details. Until now the application logic of the IQForm class was designed for only one form. Now this class must manage and deal with multiple (n) forms. In the case of forms with fields this is not a great problem, since these fields are only created with the aid of the IQCommon class, but are handled entirely in the IQForm class. Regarding forms with tables the whole thing gets more complicated because the IQTable class in collaboration with its table model manages the data by itself and forwards only the needed information to the IQForm class.

Figure 22 shows the object diagram of a master/detail form. To build forms with fields oForm invokes oCommon where the specific IQUserEntry objects (here oUserEntry1 to oUserEntryN) are instantiated. Everything else concerning these generated fields is later on handled in oForm itself. The IQUserEntry objects need not know anything about oForm. In case of the IQTable forms (here oTable1 to oTableN) the application logic is split between oForm and the oTable objects (directly accessed by oForm and vice versa).

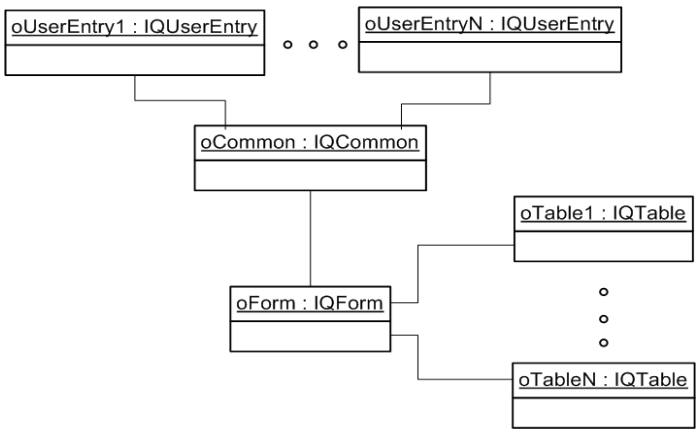


Figure 22: Handling multiple tables

Some methods in oForm, which will be invoked by various occurrences of the oTable objects, are designed to cope with requirements of one particular oTable object. As solution of this problem oForm provides the methods for each oTable in a separate specific occurrence. All the methods needed by oTable objects are named in a consistent way as *method name + two-digit numeric ending*. The ending is used to declare the detailed index of the method. The constructor of the IQTable class is changed to that effect that it now gets besides the IQForm object parameter a String sDetailsIdx as second parameter assigning the detail index of the IQTable object. That way each IQTable object gets

its unique name consisting of *GRID + sDetailsIdx*. This solution ensures a well defined one-to-one relationship between a IQTable object and its methods in IQForm.

Listing 20 shows the changed constructor of IQTable and the method InvokeCheckRules which invokes, as the name says, the method CheckRules in IQForm. This invocation is done by using the reflection mechanism described in Section 3.

Listing 20: Reflection in IQTable

```
1  public class IQTable extends JTable
2  {
3    private String m_sTableName;
4    private String m_sDetailsIdx;
5    private boolean m_blnIsDetail;
6    .
7    .
8    .
9
10   public IQTable(IQDefaultForm frmParent, String sDetailsIdx)
11   {
12     m_oForm = frmParent;
13     m_sDetailsIdx = sDetailsIdx;
14     m_blnIsDetail = false;
15     .
16     .
17     .
18
19     if(!m_sDetailsIdx.equals("00"))
20       m_blnIsDetail = true;
21
22     m_sTableName  = "GRID" + m_sDetailsIdx;
23     this.setName(m_sTableName);
24   }
25
26   public boolean InvokeCheckRules(int lRow) throws IQException
27   {
28     boolean blnRetVal = false;
29
30     try
31     {
32       blnRetVal = (Boolean)(m_oForm.getClass().getMethod("CheckRules" + m_sDetailsIdx,
33               new Class[]{Integer.class}).invoke(m_oForm, new Object [] {lRow}));
34     }
35     catch (Exception e)
36     {
37       //should never be caught
38     }
39
40     return blnRetVal;
41   }
42 }
```

Identify error source / set focus correctly. These two tasks are closely related since when an error occurs the form containing the particular error source needs to gain the focus. Focus handling in Java is to my regards (mainly based on experiences made in the course of this project) a very intricate field. Therefore, we tried to follow a different path in this matter. The identification of the error source (and the form containing this source) is due to the specification explained above (each form has its own methods) the least problem. Errors should occur only in methods checking predefined rules (like CheckRulesXX, where XX stands for the form index) or methods trying to write data to the database (like FormToRecordXX). To enable the focussing of the particular form the IQCommon class provides a special data structure of type Hashtable<Integer,

JComponent>. Every time a form or container is created its HashValue is stored as Key in the Hashtable and the corresponding parent container is stored as value. This way the focus can easily be defined top-down in recursive manner without depending on sophisticated Swing focus handling mechanisms.

5.6 Evaluation

The major task in creating the prototypes was the development of basic controls as well as testing and debugging the database connection layer.

The connection layer is in an almost final evolution state. The first real applications will presumably show some inadequacies, but to my opinion they will not be vital. By now the development of the connection layer is completed in a satisfactory manner.

The controls represent the basic components for the generation of applications. Due to this reason much effort was taken to develop them as sophisticated as possible. At this point in project the evaluation of the controls with regard to their capability to be used in challenging business software applications is a very important task. The outcome of this evaluation will be decisive for the continuation of the project. The developed controls are namely Textbox, Datebox, Numberbox, RadioButton, Checkbox and Grid. Except of the last one they are structurally sound and show the desired performance at runtime. The grid control in form of JTable and its model forms the weak point of this matter.

Apart from the fact that Java GUI controls are emulated (see Section 3.2.2 for details) and therefore will never reach the performance of native VB controls, JTable suffers additionally from its overloaded design and its high abstraction.

With "overloaded design" we mean that JTable has the capacity to do almost everything, but its features are established only to a small extend. First, this provides too much overhead in proportion to the task to accomplish, and, second, it takes much effort to develop satisfactory solutions. Standard functionality must be implemented manually and laborious whereas the VB grid offers many features implicitely (e.g., Clipboard, Datepicker, ...). Another major difference is that the VB Grid is more restricted in its use, but therefore it is also more target-oriented and faster.

The second drawback regarding performance it the high abstraction of the Java approach. Due to the UI-Delegate paradigm (Section 3.4.1), which separates the view (JTable) and the data model(JTableModel), there is a communication overhead leading to performance degradation.

Although the grid is not state-of-the-art regarding performance and behaviour of highly sophisticated business applications we continue proceedings as intended, but keep at the back of our mind that subsequent to this migration process an improvement of the actual solution must take place. To try it in the course of this project would exceed the predetermined project time frame and is therefore out of scope. By now the methods and interfaces needed by the generator are developed and well established and therefore the subsystem architecture and the adaptation of IGS can be done without inconveniences.

5.7 Concluding Remarks

The development of the prototypes was the main effort of this project. With the controls being present in the IQUserEntry package and the grid control consisting of IQTable, its model IQTableModel and the corresponding editors, renderers, and documents we now have all the controls needed by IGS. For each of the prototypes we used the same form class with the effect that this class has been extended and refined for the prototyping process and is now general enough to meet the requirements of the different subsystem types. On the basis of the controls and the form in a next step we can develop the templates required for the generation process.

Moreover, during this task we developed and refined general (not to be generated) classes each application will have to make use of (e.g., IQCommon, IQNLSString,...), and we already designed a container layout that meets the requirements for use with IGS. Apart from small refinements according to requested changes in the controls and forms these classes and the architectural design are almost in a final development stage.

Appendix D shows the Java prototypes for the different subsystems developed during this task.

6 Subsystem Architecture

6.1 Overview

All of the generated UI form-classes will share significant parts of the overall application functionality. Thus, we divided the entire application into subsystems that are already strongly encapsulated. Each subsystem consists of a form class plus a convenient support class.

With the aid of the prototypes we designed so far we will build templates of each control (e.g. DateField, Table,...) in the next section. These templates serve as fragments for IGS and will be put together to build up the generated container class (form) of the appropriate type (SearchForm, Master/Detail-Form,...). Together with a generated form IGS will provide an EventHandler class as support class. The support class contains method bodies for the predefined events and can be expanded by own methods if needed. If a new run of IGS is necessary, e.g., because an additional control is needed, IGS overwrites only the form classes and leaves the support classes in their original state. This approach offers the flexibility of generating the application as often as we want until we are convinced of the result without loosing the modifications we made.

Each form provides call-in-interfaces in form of public methods and public "getters" and "setters" which are methods to assign or read the value of a property. These interfaces are designed to be used by handler classes of other subsystems within the application and give the developer the flexibility to gain access to other subsystems if needed. However, the access is restricted to the setting of property values and ordinary execution of some defined methods. With "ordinary execution" we mean that the developer can enforce their execution without changing the ordinary behaviour of the method. Well directed and more flexible modifications are only possible by means of the call-out-interfaces. At well defined points the form class invokes particular methods in the supporting EventHandler class. All parameters are passed by reference to allow their modification or to set the appropriate value (e.g. if the parameter acts as flag).

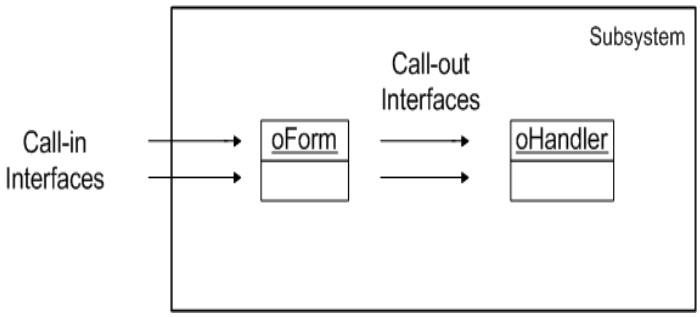


Figure 23: Subsystem architecture

Figure 23 shows the architecture of a subsystem. As we can see the call-in-interfaces can be invoked from outside the subsystem by handler of other subsystems, whereas the call-out-interfaces reach only the handlers assigned to the form.

6.2 Call-In-Interfaces

As we already mentioned in the previous section the call-in-interfaces consist of public methods and public "getters" and "setters". Though they are all methods we make this semantic separation to distinct between methods that perform operational code and methods that only set or get properties in form of class variables.

6.2.1 Operational methods

All of these methods are public and return a boolean value indicating whether they were successful or not. In the latter case the cause can be a failure during the execution of the method, or cancelling the execution by means of the designated handler. Cancelling is possible since each method triggers an OnBeforeXXX event in the beginning of the method body and an OnAfterXXX event in the end (XXX stands for the name of the method) where a flag to cancel the execution of the method can be set to the appropriate value. The following enumeration lists the methods choosen to establish the call-in-interfaces and gives a short explanation of their purpose or usage.

- DoSave
  Saves the data of the subsystem persistently.
- DoPrint
  Invokes the subsystem's printing functionality. It gets an integer indicating the print mode (either Print or Page Preview) as parameter.
- DoAddNew
  Puts the subsystem in add new mode. This method is used in subsystems containing a functionality to add new Data objects.
- DoSearch
  Invokes the subsystem's search functionality. Used only if applicable, i.e., if there are search dialogues.
- DoDelete
  Deletes the data of the subsystem.
- DoRefresh
  Refreshes (mass) data (mainly in tables). Used only if there is a query or refresh button on the toolbar.
- DoPick
  Performs all functionality that is required when the user selects the item from a search form.
- DoUnload
  Performs cleanup if necessary.

- FormToRecord
  Puts all data being visible in the form into a batch job to the record set.

- RecordToForm
  Loads all data prevailing in the record set to the form. This is done when the form is loaded for the first time or an operation like DoNext is performed.

- CreateRecordSets
  Creates the record set by querying the database.

Listing 21 shows the DoAddNew method as example for a call-in-interface. In line 19 the OnBeforeAddNew event is triggered and according to the boolean flag the code from line 22 to line 29 will be executed.

Listing 21: Public method DoAddNew

```
1
2  public boolean DoAddNew() throws IQException
3  {
4      boolean[] blnCancel = {false};
5
6      if(m_blnAdding)
7      {
8        ShowErrorMessage(m_Common.LoadRESString(m_sFormName, m_enLanguage, 10035));
9        return false;
10     }
11
12     if(!m_blnNoRecord)
13       ValidateControls();
14
15     if(m_blnChanged)
16       if(!ShowQuestionMessage(m_Common.LoadRESString(m_sFormName, m_enLanguage, 10030)))
17         return false;
18
19     raiseOnBeforeAddNew(blnCancel);
20
21     if(!blnCancel[0])
22     {
23       m_blnAdding = true;
24       UnlockAllControls();
25       m_oMasterRecordSet.AddRec();
26       DefaultsToForm();
27
28       raiseOnAfterAddNew();
29     }
30
31     return !blnCancel[0];
32 }
```

6.2.2 Get- and Set-Methods

The job description of these methods is to write or read values being assigned to a property (class variable) of the form object. Depending on their usage properties can have one of the following modes:

- Write only (WO)
  Only the Set-method of the property is public.

- Read only (RO)
  Only the Get-method of the property is public.
- Read/Write (RW)
  Both the Set-method and the Get-method are public.

Table 7 gives an overview over the properties with assigned public methods. These properties must be provided by each form object.

Property	Type	Mode	Description
vConnections	Vector of IQConnection objects	WO	The Database connections of the subsystem.
oNLSString	IQNLSString object	WO	The Strings (NLS Strings and SQL Statements) associated to the application. The NLSString object contains the user information including the user's calendar. Only this calendar can be used to perform calendar calculations.
enFormType	IQSubsys-FormType	RO	Defines the type of the form. See Appendix E for details.
enUseType	IQSubsys-UseType	RO	Defines the use type of the form. See Appendix E for details.
iRequestedWidth	Integer	RO	The width in twips requested by the subsystem's visible area. See Appendix E for details.
iRequestedHeight	Integer	RO	The height in twips requested by the subsystem's visible area. See Appendix E for details.
iConnectionCount	Integer	RO	The number of connections requested by the subsystem.
sCaption	String	RO	String containing the form's caption to display at runtime.
vFields	Vector of controls	RO	
iFieldCount	Integer	RO	Number of items in the controls vector.

Table 7: Public methods for properties

6.3 Call-Out-Interfaces

The aim of the call-out-interfaces is to provide a possibility for the software developer to change the generic behaviour of the generated application. As we already mentioned this functionality is realised with the aid of a self designed event handling mechanism. At well defined points of the application events will be fired and the assigned support class (event handler) will catch the event and invoke the appropriate method. After initial generation of the application the bodies of these event handler methods are empty. The software developer is in charge of filling them with code to achieve the desired results if needed.

For each of the following events (referred to as <Event>) there is an according On-Before<Event> and an according OnAfter<Event>. The events have the same parameters with the exception that the OnBefore<Event> always has an additional boolean parameter (boolean[] blnCancel) which is set to "False"' when the event is raised by the subsystem. If set to "True" by the handler of the event, any follow-up action may not take place.

To enable their modification the according parameters must be passed as reference variables. In Java all types except scalar primitive types are reference types. To allow notification of change we make use of flags represented by such primitive types. Though for these types there exist wrapper classes that encapsulate the value of the primitive type internally and represents it as object, in this connection the wrappers are of no use for us. Due to various reasons (e.g., thread-safety, speed, usage in hashtables,...) such wrappers are immutable. To deal with this restriction instead of a primitive type we use an array of this type having length 1. Since an array is an object and , therefore, a reference in the support class, we can change the value of its only field and read the changed value in the form class. Unfortunately, this immutability holds also for the String type. Instead of String we must use StringBuffer which implements a mutable sequence of characters. The length and the content of the sequence can be changed through certain method calls.

This approach is closely related to the programming paradigms of aspect-oriented programming (AOP). The first and most popular general-purpose AOP language, AspectJ is described in [Xerox 1998].

Table 8 shows the events that are handled in the support class.

<Event>	Parameters	Description
Save	N/A	Before and after saving the data of the subsystem.
Print	int iPrintMode	Before and after the subsystem's printing functionality is invoked. Print = 0, Page Preview = 1
AddNew	N/A	Before and after the subsystem is put into add new mode.
Search	Object oWhich	Before and after the subsystem's search functionality is invoked. This parameter is meaningful only in cases where the subsystem contains more then one search form. In other cases, Null is passed.
Delete	N/A	Before and after Data deletion. Mandatory for DM use type (see Appendix E) if applicable, i.e. if there is a functionality to delete data.
Query	StringBuffer sSQL, Vector<Object> vParams	Mandatory for DM use type (see Appendix E) if applicable, i.e., if there is a query or refresh button on the toolbar. sSql is a parameterized SQL command. vParams is an array containing the needed paramters.
Refresh	N/A	Before and after mass refresh. Mandatory for DM Use Type (see Appendix E) if applicable, i.e. if there is a query or refresh button on the toolbar.
Pick	N/A	Before and after a search result is picked
GotoFirst	N/A	Before and after data has been retrieved from the database to the subsystem.
UserToolbar-ButtonClick	String sKey	When the user presses the toolbar item, the OnBeforeUserToolbarButtonClick is raised. sKey = The key of the toolbar item that has been clicked.
Show	N/A	OnBeforeShow is raised at the very initialization of the subsystem. OnAfterShow is raised when initialization of the subsystem is finished, i.e., the subsystem is ready to react on any method.

Table 8: Subsystem Events

As example for a call-out-interface event handler in Listing 22 we show the method OnBeforeSave. Its only parameter is a boolean flag indicating if the method DoSave in the according form class shall be executed. This example is based on the assumption that we have a UI with values coming from a database view. The data of the database view consists of data taken from the two tables *PERSONS* and *STUDENTS*. Since Oracle database views cannot be updated, the generated DoSave method in the form class will fail due to the fact that IGS always takes the data source as target of the Save operation. Thus, we have to write our own save logic and set blnCancel to true (see line 90 and line 94). In line 19 we retrieve the value *studpersid* which is the primary key for both, the *PERSONS* and *STUDENTS* table, and links together this two tables. With the aid of this value we create two IQDataSet objects, one for each table. According to the prefix of each control (either "PERS" or "STUD") we save its value in the corresponding table. In line 86 we load the updated record set to the form to ensure that the data in the UI is bound again to data in the database.

Listing 22: Method OnBeforeSave

```
1  public void OnBeforeSave(boolean[] blnCancel)
2  {
3    IQNLS nls = new IQNLS();
4    IQConnection con = new IQConnection();
5    IQDataSet dsStud = new IQDataSet();
6    IQDataSet dsPers = new IQDataSet();
7    Vector<Object> vParams = new Vector<Object>();
8    Vector<Object> v = new Vector<Object>();
9    double PERSPERSID_SEQ;
10   Object oBuffer;
11   String sControlName;
12
13   try
14   {
15     if(m_frmForm.CheckRules())
16     {
17       con = m_frmForm.getConnections(0);
18
19       vParams.add(0, m_frmForm.MUS_STUDPERSID_GetValue());
20       dsStud = (IQDataSet)con.CmdExecute(nls.GetSqlCmd(SQL_STUD_SAVE), vParams);
21       dsPers = (IQDataSet)con.CmdExecute(nls.GetSqlCmd(SQL_PERS_SAVE), vParams);
22
23       if(m_frmForm.getAdding())
24       {
25         PERSPERSID_Seq =  con.SeqNextVal("PERSONENDATEN");
26         dsPers.AddRec();
27         dsPers.setValue("PERSPERSID", PERSPERSID_SEQ);
28       }
29       else
30         dsPers.EditRec();
31
32       for(int i = 0; i < m_frmForm.getControlCount(); i++)
33       {
34         sControlName = m_frmForm.getControl(i).getName();
35
36         if (sControlName.substring(0, 3).equals("PERS"))
37         {
38           oBuffer = m_frmForm.getControl(i).getValue();
39
40           if(m_Common.isDate(oBuffer.toString()))
41           {
42             oBuffer = m_Common.Date2Number(oBuffer);
43           }
44
45           dsPers.setValue(sControlName, oBuffer);
46         }
47       }
48
```

```
49        dsPers.UpdateRec();
50        dsPers.DoClose();
51
52        if(m_frmForm.getAdding())
53        {
54          dsStud.AddRec();
55          dsStud.setValue("STUDPERSID", PERSPERSID_SEQ);
56        }
57        else
58          dsStud.EditRec();
59
60        for(int i = 0; i < m_frmForm.getControlCount(); i++)
61        {
62          sControlName = m_frmForm.getControl(i).getName();
63
64          if (sControlName.substring(0, 3).equals("STUD"))
65          {
66            oBuffer = m_frmForm.getControl(i).getValue();
67
68            if(m_Common.isDate(oBuffer.toString()))
69            {
70              oBuffer = m_Common.Date2Number(oBuffer);
71            }
72            if(!m_Common.getControl(i).getName.equals("STUDPERSID")
73            {
74              dsStud.setValue(sControlName, oBuffer);
75            }
76          }
77        }
78
79        dsStud.UpdateRec();
80        dsStud.DoClose();
81
82        v.add(0, m_frmForm.getControl("STUDPERSID").getValue());
83
84        if(m_frmForm.CreateRecordSets(v))
85        {
86          m_frmForm.RecordToForm();
87        }
88      }
89
90      blnCancel[0] = true;
91    }
92    catch (IQException e)
93    {
94      blnCancel[0] = true;
95      ShowErrorMessage(m_Common.LoadRESString(m_frmForm.getName(),
96                       m_Common.getLanguage(), 10035));
97    }
98  }
```

7 Adapting IGS

The main task of the last step in our project is the development of templates for the particular subsystems and controls. This task will be rather straightforward since the prototypes developed so far are reproductions of generated applications. Therefore, they already cover the requirements concerning design, functionality, and method of application pretty good. So the effort will consist mainly of extracting the parts needed for the templates and converting them to generalizations. These generic templates will finally be filled by IGS with appropriate data and serve as fragments of the specified application.

Subsection 7.1 shows the process of creating such templates by means of an example covering the whole area of application. Subsection 7.2 overviews briefly the topic of conditional compilation because this technique is essential for generating programs saving time and memory.

The completion of the generator adaptation process will be the enhancement of the generator itself, concerning its ability to "talk" Java. To gain this functionality small adaptations to the generator have to be made without expanding the process logic significantly. These slight changes affect mainly the syntactical differences on invocations of methods, written either in Java or in VB, and will not be further discussed here. Figure 24 shows the different layers of IGS. The layers for the database connection and for the positioning and building of controls remain untouched, whereas within the code generation layer the method calls have to be adapted for to work with Java.

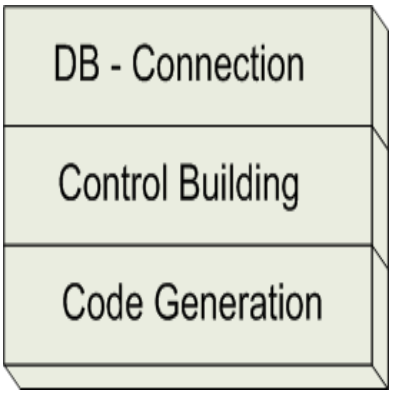


Figure 24: IGS Layers

7.1 Creating Templates

IGS makes use of various templates to put together the final application. These templates consist of generic code fragments in which the appropriate information will be inserted by means of simple textual substitution. The *$* symbol acts as identification mark for the generator to recognize positions where replacements have to take place, either by insertion of another template or the insertion of constants (e.g., names). In the latter case IGS will scan the created IRS file to find the according entry. For each application three IRS files are generated - one for the grid definitions (see Section 5 for details), one

for the SQL commandos, and one for the constants (names, captions, tooltips, ...). In Appendix F the IRS file for the constants of prototype 1 is shown.

To build the application IGS uses three different types of templates:

- Form templates
- Subsystem templates
- Control templates

7.1.1 Form Templates

Each application consists of one *MainForm* and one or more (sub)forms. MainForm is of type JFrame and the forms are either of type JInternalFrame (incorporated within a JDesktopPane) or JDialog. For each of these three java window classes a specific template exists since their behaviour differs slightly from each other.

- JFrame
  is a window with decorations such as a border, a title, and button components to close or iconify the window. Applications with a GUI usually include at least one frame [Sun 2004].
- JInternalFrame
  Most commercial business applications are Multiple Document Interface (MDI) applications. This means that there is one large *desktop pane* that holds all other windows. The other windows can be iconified (minimized) and moved around within this desktop pane, but not moved outside of it. Furthermore, minimizing the desktop pane hides all windows it contains as well. Swing introduced MDI support by means of two main classes. The first, JDesktopPane serves as a holder for the other windows. The second, JInternalFrame acts mainly like JFrame, except that it is constrained to stay inside JDesktopPane. JInternalFrame is a lightweight object that provides many of the features of a native frame, including dragging, closing, becoming an icon, resizing, title display, and support for a menu bar [Hall et al. 2001].
- JDialog
  is an independent sub window meant to carry temporary notice apart from the main Swing Application Window. Most Dialogs present an error message or warning to a user, but Dialogs can present images, directory trees, and anything compatible with the main Swing Application that manages them. A Dialog can be modal. When a modal Dialog is visible, it blocks user input to all other windows in the program [Sun 2004].

The following listing is taken from the form template. Together with listings 24 and 25 it shows a continguous example for the functionality of the generator. The excerpt contains the definition of the two methods *DoLoad* and *DoShow*. In lines 6 and 19 code of the appropriate subsystem templates (and subsequently the according control templates) is included, whereas in lines 7 and 28 simple name substitution will take place.

Listing 23: Form Template

```
1  //frmSubsystemTemplate.IFT DoLoad Method Code Begin
2  private void DoLoad(IQDefaultForm oForm) throws IQException
3  {
4     String sCaption = m_Common.LoadIRSString("$FORMNAME$", $STRIDCAPTION$);
5     //frmSubsystemTemplate.IFT #INITIALIZE# Code Begin
6     $INITIALIZE$
7     $USERCONTROL$.Resize();
8     //frmSubsystemTemplate.IFT #INITIALIZE# Code End
9  }
10 //frmSubsystemTemplate.IFT DoLoad Method Code End
11 //frmSubsystemTemplate.IFT DoShow Method Code Begin
12 private void DoShow()
13 {
14    boolean blnCancel = false;
15
16    try
17    {
18       //frmSubsystemTemplate.IFT #SHOW# Code Begin
19       $SHOW$
20       //frmSubsystemTemplate.IFT #SHOW# Code End
21    }
22    catch(IQException e)
23    {
24       int lErr = e.getErr();
25       String sSource = e.getSource();
26       String sDesc = e.getDescription();
27
28       String sErrMsg = m_Common.LoadRESString("$FORMNAME$", m_enLanguage, 10054);
29       String sErrDetail = lErr + "\n" + sSource + "\n" + sDesc;
30       try
31       {
32          ShowErrorMessage(sErrMsg, sErrDetail);
33       }
34       catch (IQException e1)
35       {
36          //doNothing
37       }
38    }
39 }
40 //frmSubsystemTemplate.IFT DoShow Method Code End
```

7.1.2 Subsystem Templates

Each form of the application is assigned to exactly one subsystem type. As specified in Section 6 there are three types of subsystems. According to the definition of the application type (done in the IFL file) at the positions pointed out in Listing 23 the appropriate code of the (type-)consistent subsystem template will be included. Therefore, the subsystem template (as well as the control template) is divided into sections. These sections enable the identification and qualification of the code so that the generator always knows how to address this code.

The subsystem template is classified into the following sections:

- [Events]
- [Variables]
- [Initialize]

- [Show]
- [Routines]

In [Events] the according event handler class for the particular subsystem is constituted. [Variables] holds the declaration of all variables (variables of form object and of all needed control objects). [Initialize] will be used (as seen in listing 23) to implement the method that initializes the form and all corresponding controls. [Show] creates the method to show the form. In [Routines] all other methods of the form and the control classes are defined.

Listing 24 shows the [Initialize] section of a subsystem template. The section definition tells IGS to include also the code of the [Initialize] section of all controls specified for the according application.

Listing 24: Subsystem Template

```
1  [Initialize]
2  if($FORMNAME$_LARGE)
3  {
4      $INITLARGEKEYS$
5  }
6
7  $INITMASTERSQL$
8
9  if($FORMNAME$_NUMDETAILS > 0)
10 {
11     m_Common.IGSSetupDataEntryToolbar(tbTB, imgToolBar, false, true);
12 }
13 else
14     m_Common.IGSSetupDataEntryToolbar(tbTB, imgToolBar, false, false);
15
16 if (!$FORMNAME$_NAVIGATE)
17 {
18     tbTB.get(TBB_FIRST).setVisible(false);
19     tbTB.get(TBB_PREV).setVisible(false);
20     tbTB.get(TBB_NEXT).setVisible(false);
21     tbTB.get(TBB_LAST).setVisible(false);
22 }
```

7.1.3 Control Templates

IGS creates an application of the appropriate type according to the specifications, and controls for the fields/grids of the application according to the used database tables. The types of the database fields define the types of the controls. Nevertheless, in the specification (in IFL file) a different type can be assigned to a control.

Controls are:

- Textbox
- Datebox
- Numberbox

- Radio
- Checkbox
- Grid

For each of these controls there are templates which are classified into the following sections:

- [Variables]
- [Initialize]
- [Routines]
- [GetValue]
- [SetValue]
- [GetCaption]
- [Lock]
- [SetFocus]

The first three sections will be included in the subsystem sections of the same name. The other sections are defined explicitly (not in the [Routines] section) because they will be included several times.

Listing 25 shows the code of the [Initialize] section of a textbox control.

Listing 25: Control Template

```
1  [Initialize]
2  InitializeTextbox($PANEL$, $FIELDNAME$, $LBLNAME$, $ALIGNMENT$, $FORMNAME$,
3                    $STRIDCAPTIONS$, $STRIDCOMMENTS$, $HASBROWSEFORM$,
4                    $TOUPPERCASE$, m_vLabels, m_s$FIELDNAME$_Nm);
5
6  If $FIELDNAMES$_HASBROWSEFORM
7  {
8     m_$FIELDNAME$SF = new IQCFSearch();
9     m_$FIELDNAME$SFLoaded = false;
10 }
```

Listing 26 shows the generated code resulting from the substitution steps explained above.

Listing 26: Generated Code

```
1  // frmMDIChildTemplate.IFT DoLoad Method Code Begin
2  public void DoLoad() throws IQException
3  {
4     String sCaption = m_Common.LoadIRSString("frmMUSTER1", 23);
5     //frmMDIChildTemplate.IFT #INITIALIZE# Code Begin
6     m_Common.InitializeTextbox(m_UserEntryPane, m_MUS_STRING_tf,
7     m_MUS_STRING_lbl, m_sFormName, 14, 0,false,false, m_vLabels,
8     m_sMUS_STRING_Nm);
9     //Datebox.ICT Initialize Code Begin
```

```
10      m_Common.InitializeDateBox(m_UserEntryPane, m_MUS_DATE_uc,
11      m_MUS_DATE_lbl, m_sFormName, 15, 1,m_vLabels, m_sMUS_DATE_Nm);
12      //Datebox.ICT Initialize Code End
13      //Datetimebox.ICT Initialize Code Begin
14      m_Common.InitializeDateTimeBox(m_UserEntryPane, m_MUS_DATETIME_uc,
15      m_MUS_DATETIME_lbl,m_sFormName,16, 2, m_vLabels, m_sMUS_DATETIME_Nm);
16      //Datetimebox.ICT Initialize Code End
17      //Datebox.ICT Initialize Code Begin
18      m_Common.InitializeTimeBox(m_UserEntryPane, m_MUS_TIME_lbl,
19      m_sFormName, 17, 3, m_vLabels, m_sMUS_TIME_Nm);
20      //Datebox.ICT Initialize Code End
21      //CardinalBox.ICT Initialize Code Begin
22      m_Common.InitializeCardinalBox(m_UserEntryPane, m_MUS_CARDINAL_tf,
23      m_MUS_CARDINAL_lbl, m_sFormName, 18, 4, m_vLabels, m_sMUS_CARDINAL_Nm);
24      //CardinalBox.ICT Initialize Code End
25      //DecimalBox.ICT Initialize Code Begin
26      m_Common.InitializeDecimalBox(m_UserEntryPane, m_MUS_DECIMAL_tf,
27      m_MUS_DECIMAL_lbl, m_sFormName, 19, 5, m_vLabels, m_sMUS_DECIMAL_Nm);
28      //Decimalbox.ICT Initialize Code End
29      //CheckBox.ICT Initialize Code Begin
30      m_Common.InitializeControl(m_UserEntryPane, m_MUS_BOOLEAN_chk,
31      m_MUS_BOOLEAN_lbl, m_sFormName, 20, 6, m_vLabels, m_sMUS_BOOLEAN_Nm);
32      //CheckBox.ICT Initialize Code End
33      //RadioButtons.ICT Initialize Code Begin
34      m_vRadioButtons.add(m_MUS_OPTION_rb1);
35      m_vRadioButtons.add(m_MUS_OPTION_rb2);
36      m_vRadioButtons.add(m_MUS_OPTION_rb3);
37      m_Common.InitializeRadio(m_UserEntryPane,m_vRadioButtons,m_MUS_OPTION_bg,
38      m_MUS_OPTION_lbl, m_sFormName,10, 21, 13, m_vLabels, m_sMUS_OPTION_Nm);
39      //RadioButtons.ICT Initialize Code End
40
41      boolean[] blnCancel = {false};
42
43      raiseOnBeforeShow(blnCancel);
44
45      if(FRMMUSTER1_LARGE)
46          m_sLargeKeys.add("MUS_DECIMAL");
47
48      m_sMasterSQL = m_Common.GetSqlCmd(1);
49
50      if(FRMMUSTER1_LARGE)
51      {
52                  if(FRMMUSTER1_NAVIGATE)
53                  {
54                      raiseOnBeforeGotoFirst(blnCancel);
55
56                              if(!blnCancel[0])
57                              {
58                                  DoFirst(false);
59                                  raiseOnAfterGotoFirst();
60                              }
61                  }
62      }
63      else
64          CreateRecordSets();
65
66      raiseOnAfterShow();
67      //frmMDIChildTemplate.IFT #INITIALIZE# Code End
68  }
69  //frmMDIChildTemplate.IFT DoLoad Method Code End
```

7.2 Conditional Compilation in Java

The code in the templates makes use of conditional compilation. It selects particular sections of code to compile, while excluding other sections. These conditional compilation statements are designed to run at compile time, not at run time.

We designed the templates this way to save memory capacity and fasten execution time of the final program. Moreover, we are able to compile several different versions of our program with different features present in the different versions.

Since Java has no pre-processor and hence no construct like *#ifdef* can be used, there is a constricted form of conditional compilation. The following program fragment will not be compiled because the result of the if-statement is always false. This behaviour is urgently recommended in the Java language specification, but not obligatory for the compiler manufacturer.

Listing 27: Conditional Compilation in Java

```
1 if (false)
2 {
3    //doSomeThing
4 }
```

This behaviour of the java compiler is in opposition to its postulation to accept only reachable assignments. According to the java language specification the compiler should reject all assignments that are not reachable and signal an error.

In technical sense assignments are not reachable in loops whose condition is false at compile time, and assignments lying after *break*, *continue*, *throw* or *return* assignments that are accessed unconditionally. The only exception of this rule is the above mentioned option of a constantly false branch which can be used for conditional compilation.

Listing 28 shows a small Java program to test the behaviour of the compiler according to conditional compiling. We made use of the dissassembler *javap* to examine the resulting compiled CompileTest.class file. Javap takes the class and dumps information about its methods to standard out. It doesn't decompile the code into Java source code, but it disassembles the byte code into the byte code instructions defined by the Java Virtual Machine specification.

The output shown in Listing 29 is a little cryptic, but we can easily see that there's no assembly code for the method Donothing() and, therefore, the compiler acts as required.

Listing 28: CompileTest

```
1  public class CompileTest
2  {
3
4     private final boolean $LARGEKEYS$ = false;
5
6     public void Donothing()
7     {
8        if($LARGEKEYS$)
9           System.out.println("This shouldn't happen!!");
10    }
11
12    public static void main(String[] args0)
13    {
14       CompileTest t = new CompileTest();
15       t.Donothing();
16    }
17 }
```

Listing 29: Dissasembled CompileTest

```
Compiled from "CompileTest.java"
public class CompileTest extends java.lang.Object{
public CompileTest();
  Code:
    0:   aload_0
    1:   invokespecial   #12; //Method java/lang/Object."<init>":()v
    4:   aload_0
    5:   iconst_0
    6:   putfield        #14; //Field $LARGEKEYS$:Z
    9:   return

public void Donothing();
  Code:
    0:   return

public static void main(java.lang.String[]);
Code:
    0:   new     #1; //class CompileTest
    3:   dup
    4:   invokespecial   #23; //Method "<init>":()v
    7:   astore_1
    8:   aload_1
    6:   invokevirtual   #24; //Method Donothing:()v
    9:   return
```

8 Results

The objective of this project was to check the general feasibility of migrating the IGS generator. In the long run the generator shall produce Java applications that are equivalent in appearance, behaviour, functionality, and performance to the VB applications. It is evident that in the course of this project such a complex task can only provide a proof of concept and a basis for a commercial product.

The porting/migration of an existing well proven application cannot be compared with a complete new product development. Java, which we have chosen as target language to migrate the system, is a completely object-oriented programming language allowing the same program to be executed on multiple operating systems. Although not designed as a successor for successful languages like C++ or Delphi, it offers astonishingly efficient language elements and a great variety of tools to work with. On account of the fact that the Java compiler generates byte code which is translated by a Interpreter on all systems and every hardware in the same way, Java programs are theoretically executable on every operating system. Thereby, Java remains architecture-neutral and, hence, independent of the platform.

However, mainly because of platform independence, graphical user interfaces pose a considerable problem in Java. A user interface can look quite differently and moreover behave different on different operating systems. Since the intention of this project was to migrate business software applications, the graphic user interface is a very important component, and the success of the migration will be measured by its quality.

Therefore we can say:

> *The central issue of this project was to develop portable and platoform-indedpendent graphic user interface components that comply with the defined requirements.*

In this section we discuss, above all, the question whether porting IGS is practicable and appropriate. Thus, in Subsection 8.1 we will look whether and to which extend the above mentioned issue can be solved. Subsection 8.2 gives a brief overview of the time the particular project segments took and tries to bring them in relation to the insights we made throughout this project. Subsection 8.3 summarizes open work.

8.1 Developing GUIs with Java

We recall the four properties to be kept in the migration process:

- Appearance
- Behaviour

- Functionality
- Performance

To keep appearance, behaviour, and functionality was no big problem. Thereby we understand "appearance and behaviour" to mean that users get the same information and the same kinds of interaction on each operating system. We do not require the same "look and feel". A company that uses, e.g., Linux as operating system, will be used to the fact that its applications look different than those on a Windows operating system. With the change of programming languange from VB6 to Java the emulation of functionality is pretty easy to realise, since the complexity and power of the fully object-oriented language Java exceeds the possibilities of VB6 to a great extent. Consequently, the system performance remained as the only unsolved problem.

According to [Steyer 2003] there are several approaches to realise GUIs:

- We take a look at all platforms that should be supported, identify the common components, and then use the smallest common denominator.
- The second approach again examines all relevant platforms and then uses an API which encloses all possibilities of the base UIs (a sort of the union amount of all possibilities of the base UIs).
- We use the prevailing API on the according platform.
- We implement the components completely in pure Java code. The appearance and the behaviour of the GUI components adapt themselves automatically to every supported system platform.

All mentioned attempts have their advantages and disadvantages. The first solution is relatively easy to implement and can be supported easily on all platforms. Nevertheless this approach will lead to considerable restrictions compared to the base UI of the particular platforms.

Indeed, with the set union of all possibilities of the base UIs (as in the second alternative) there are no restrictions of efficiency, but it can come to conflicts within the chosen realisation. The system will be much more complicated and perhaps not to be realised adequately on all platforms.

The third idea brings the least changes for users. Using the Java UI would provide the same performance as with applications using the system or native API. The disadvantage of this approach (and, actually, also of the first two) is that it is not really independent of the used platform. We can not take into consideration all possible UIs (under Linux there exist several). If we want to be really platform independent, we must provide a solution to this fact. Even more unfavourable is the fact that this approach is very inflexible to innovations and, in addition, constrained to the host API. When there are changes of the host API our components must be adapted.

The fourth approach is the one we have chosen by the use of Swing. As already mentioned Swing extends AWT considerably. However, the use of Swing causes also problems, in particular bad performance and the need of many resources. Swing applications are generally slower than comparable applications based on the pure AWT API. As mentioned in Subsection 3.2.2, Swing uses peers only for top-level components like windows and frames. All other components are emulated in pure Java code and therefore Swing does not make use of hardware GUI accelerators and special host GUI operations. In addition, Swing applications also need more resources from the processor and main memory.

To conclude this section it must be said that the slogan "Write once, run anywhere", created by Sun Microsystems to illustrate the cross-platform benefits of Java, does not really hold, especially for graphical user applications. Java's main drawback is that developers are often restricted to using the lowest common denominator subset of features which are available on all platforms. This may hinder the application's performance or prohibit developers from using platforms' most advanced features.

8.2 Statistics

This section is intended to give the reader an overview of the efforts we made in the project. We do this on basis of the time we took for each particular project segment. The project duration was in sum 1430 hours (or 36 weeks, or 9 months). Table 9 shows time and effort of the particular segments of the project.

ACTIVITY	HOURS	RATIO
orientation	70	4,9%
implementing data transfer	280	19,6%
implementing user interface	390	27,2%
implementing subsystem architecture	140	9,80%
adapting IGS	90	6,30%
testing	200	14,00%
documentation	260	18,20%
SUM	1430	100,00%

Table 9: Time and effort of project activities

For reasons of better measurement we divided the implementation into four different segments. The first of these segments is the implementation of the data transfer. The basic connection layer was done in about half the time stated, but finally took about 20% of total time because of many refinements and corrections we had to do in the prototyping step. The second segment is the implementation of the UI. The time we took for this segment covers mainly the time we spent for prototyping without the adaptions we had to make on the connection layer. As we can easily see with about a third of the total time this was the major task of the project. Nevertheless the UI is still (in opposition to the connection layer) not in a sufficient state of development, due to performance reasons and some open tasks (described in Subsection 8.3). This fact helps to undermine the conclusion we made in the former section that UI development is the central issue of this

project. The third implementation segment is the development of subsystem architecture. The last segment is the adaption of IGS which consisted mainly in the development of generalised templates. These last two tasks were rather straightforward and therefore the effort was small compared with the former two. The orientation phase was a short pre-work where we examined if Java and JDBC are generally able to satisfy the functional and technical requirements of the project. Testing and documentation was done during every step of the project and made up about 30% of project effort.

8.3 Future Work

To be ready for every day use several important and elaborate functionalities are still missing. Their implementation was never scheduled for this work. They will be the topic of future adaptions.

The print and page preview functionalities take great effort to implement due to their complexity. To realise printing we can make use of an existing JDK framework. Regrettably it is not really highly evolved and therefore does not offer many features. Page preview in Java is even worse. Because JKD offers nothing in this application area, page preview functionality has to be designed from scratch.

Another open task for the future is the implementation of reporting (mainly in XML) which is an essential property for business applications. Here Java reveals its full strength. It provides a bundle of Java XML programming APIs (e.g., JAXP) or implements direct support of XML parsers like Xerces with its Xerces2 Java Parser.

The third open task to accomplish is the revision and refinement of the Grid control consisting of JTable, JTableModel, and its renderers, editors, and documents. As already mentioned in Section 5.6 the Grid control in its actual state is not practicable for real world solutions.

Apart of these three major tasks there remain some smaller refinements to be done. Examples are designing and implementing further controls providing specific (relevant, but not vital to functionality) features (e.g., Clipboard, Datepicker, multi-column Combo- and Listboxes).

9 Conclusions

In the course of this project we tried to show the possibility to migrate business software applications designed for proprietary systems to equivalent cross-platform applications. With continuing advance in project it became more and more evident that if the UI plays an important role, the migration can't be done without loss of performance.

We showed that this is not a particular problem of the chosen platform or language, but a general problem lying in the nature of things. At the moment and with actual prevailing software technologies UI applications designed for a particular system will be reasonable faster than applications designed for several different systems. Since in this project performance is a key factor of the resulting software products, the migration was not able to meet the requirements.

Java applications generated with IGS will be developed and distributed by Inova Q in the future. Due to the (yet) existing performance handicap of GUI applications written in Java, their application area will be restricted to specific business segments. Applications dealing with the maintenance of data, common order entry solutions, and user management applications are likely to be realised, whereas complex applications in the field of logistics or CRM (Costumer Relationship Management) solutions are not feasible at the moment.

A Abbreviations

Abbreviations

ADO ..... ActiveX Data Objects
AI ..... Artificial Intelligence
AOP ..... Aspect Oriented Programming
API ..... Application Program Interface
AWT ..... Abstract Windows Toolkit
B2B ..... Business 2 Business
CIL ..... Common Intermediate Language
CLR ..... Common Language Runtime
COM ..... Component Object Model
CRM ..... Costumer Relationship Management
DAO ..... Data Access Object
DBMS ..... Database Management System
DDF ..... Data Dictionary Files
ECMA ..... European Computers Manufacturers Association
FCL ..... Framework Class Library
GNU ..... Gnu is not Unix
GTK ..... Gimp Toolkit
GUI ..... Graphical User Interface
ID ..... Identification
IDE ..... Integrated Development Environment
IFL ..... Inova Q Frame Language
IGS ..... Inova Q Generator System
IRS ..... Inova Q Ressource String
ISO ..... International Standards Organization
IT ..... Information Technology
J2EE ..... Java2 Enterprise Edition
J2SE ..... Java2 Standard Edition
JDBC ..... Java Database Connectivity
JIT ..... Just-in-time
JRE ..... Java Runtime Environment
JVM ..... Java Virtual Machine
LAN ..... Local Area Network
LCD ..... Lowest Common Denominator
LDAP ..... Lightweight Directory Access Protocol
MDI ..... Multiple Document Interface
MVC ..... Model View Controller
ODBC ..... Open Database Connectivity
OLE DB ..... Object Linking and Embedding Database
OODBMS .... Object Oriented Database Management System
OS ..... Operating System
RDBMS ..... Relational Database Management System
SPARC ..... Scalable Processor Architcture

SQL ..... Structured Query Language
SWT ..... Standard Widget Toolkit
UI ..... User Interface
VB ..... Visual Basic
XML ..... Extensible Markup Language

B List of Figures, Tables and Listings

List of Figures

List of Tables

Listings

C Comparison between AWT, Swing and SWT

Function/Role/Aspect	AWT	Swing	SWT (style)
Display static text	Label	JLabel	Label, CLabel
Display multi-line static text	Multiple Labels	Multiple JLabels or JLabel with HTML content	Multiple Labels or Label with newlines
Display multi-line formatted static text	Multiple Labels with different fonts	JLabel with HTML content	Multiple Labels with different fonts
Single-line text entry	TextField	JTextField	Text(SWT.SINGLE)
Multi-line text entry	TextArea	JTextArea	Text(SWT.MULTI)
Display an image	N/A	JLabel	Label
Display text and image	N/A	JLabel	CLabel
ToolTip pop-up help	N/A	setToolTip on component, subclass JToolTip	setToolTip on control
Styled text entry	N/A	JEditorPane	StyledText
Select from list of items	List	JList	List
Simple push button with text	Button	JButton	Button(SWT.PUSH)
Simple push button with text and/or image	N/A	JButton	Button(SWT.PUSH)
Drawing area; possibly for custom controls	Canvas	JPanel	Canvas
On/off check box	CheckBox	JCheckBox	Button(SWT.CHECK)
Radio selection	CheckBoxGroup	ButtonGroup and menus	Group and Menu
Select from a drop-down list	Choice	JComboBox	Combo, CCombo
Enter text or select from a drop-down list	N/A	JComboBox	Combo, CCombo
Scrollable area	ScrollPane	JScrollPane	Create Scrollable subclass
Top level windows	Dialog, Frame, Window	JDialog, JFrame, JWindow	Shell with different styles
Generic window	Window	JWindow	Shell
Frame window	Frame	JFrame	Shell(SWT.SHELL_TRIM)
Dialog window	Dialog	JDialog	Shell(SWT.DIALOG_TRIM)
Menu	Menu	JMenu	Menu
MenuItem	MenuItem	JMenuItem	MenuItem
Menu shortcuts	Generic keystrokes	same as AWT	host dependent mnemonics and accelerators
Pop-up menu	PopupMenu	JPopupMenu	Menu(SWT.POPUP)
Menu bars	MenuBar	JMenuBar	Menu(SWT.BAR)
Display an insertion caret	N/A	Caret	Caret
Web browser	N/A	JTextPane (HTML 3.2)	Browser (via embedded browser)
Embed control in web page	Applet	JApplet	Host control (ex. OLE)
Generic container of other controls	Panel	JPanel	Composite
Generic container of other controls with a border	Panel (if drawn manually)	JPanel with a Border	Composite(SWT.BORDER)

Figure 25: Comparing AWT, Swing and SWT(1)

Generic container of other controls with a border and title	N/A	JPanel with a TitledBorder	Group
Radio button (one of set on)	Checkbox	JRadioButton	Button(SWT.RADIO)
Control extent of radio buttons	CheckboxGroup	RadioButtonGroup	Group
Arrow buttons	N/A	JButton with image	Button(SWT.ARROW)
Supports int'l text orientations	via ComponentOrientation	same as AWT	Many components support styles for this
Focus Traversal	Policy and Manager objects	same as AWT	Next on control
Custom dialogs	Dialog subclass	JDialog subclass	Dialog subclass
Access to system events	EventQueue services	same as AWT	Display services (less robust than AWT)
System access dialogs	FileDialog	JColorChooser, JFileChooser	ColorDialog, DirectoryDialog, FileDialog, FontDialog, PrintDialog
Display simple message dialog	N/A (must subclass Dialog)	JOptionPane static methods	MessageBox with numerous styles
Display simple prompting dialog	N/A (must subclass Dialog)	JOptionPane static methods	N/A (classes exist in JFace to do this)
Layout managers	BorderLayout, CardLayout, FlowLayout, GridLayout, GridBagLayout	AWT plus BoxLayout, CenterLayout, SpringLayout	FillLayout, FormLayout, GridLayout, RowLayout, StackLayout
Basic drawing control	Canvas	JPanel	Canvas
Basic drawing	Graphics and Graphics2D objects - Basic shapes and text, arbitrary Shapes and Strokes, Bezier, fills, etc.	same as AWT	GC object - Basic shapes and text
Drawing transforms	Affine, composites	same as AWT	N/A
Off screen drawing	BufferedImage, drawImage	same as AWT	Image, drawImage
Double buffering	Manual	Automatic or manual	Manual unless provided by host control
Printing	PrintJob and PrintGraphics	same as AWT	draw to Printer device
Custom colors	Color	same as AWT	Color
Custom fonts	Font, FontMetrics	same as AWT	Font
Cursors selection	Cursor	same as AWT	Cursor
Image features	load from file, create dynamically, extensive edits	same as AWT	load from file, create dynamically, basic edits
Input automation	Robot	same as AWT	N/A
Display a tool bar	N/A	JToolBar	ToolBar, CoolBar
Display a progress bar	N/A	JProgressBar	ProgressBar
Divide space between areas	N/A	JSplitPane	Sash or SashForm
Display tabbed areas	N/A	JTabbedPane	TabFolder, CTabFolder
Display tabular info	N/A	JTable	Table
Format table columns	N/A	TableColumn	TableColumn
Display hierarchical info	N/A	JTree	Tree
Select from range of values	N/A	JSlider	Slider
Select from discrete range of values	N/A	JSpinner	Scale
Access to the base display	Toolkit, GraphicsConfiguration, GraphicsDevice	same as AWT	Display
Add items to the system tray	N/A	N/A	Tray
Key: N/A - Not available. In many cases, this feature can be created, with varying degrees of difficulty, by creating custom controls or containers of controls or by other custom programming.			

Figure 26: Comparing AWT, Swing and SWT(2)

D Java Prototypes

D.1 Prototype 1: UI with Fields

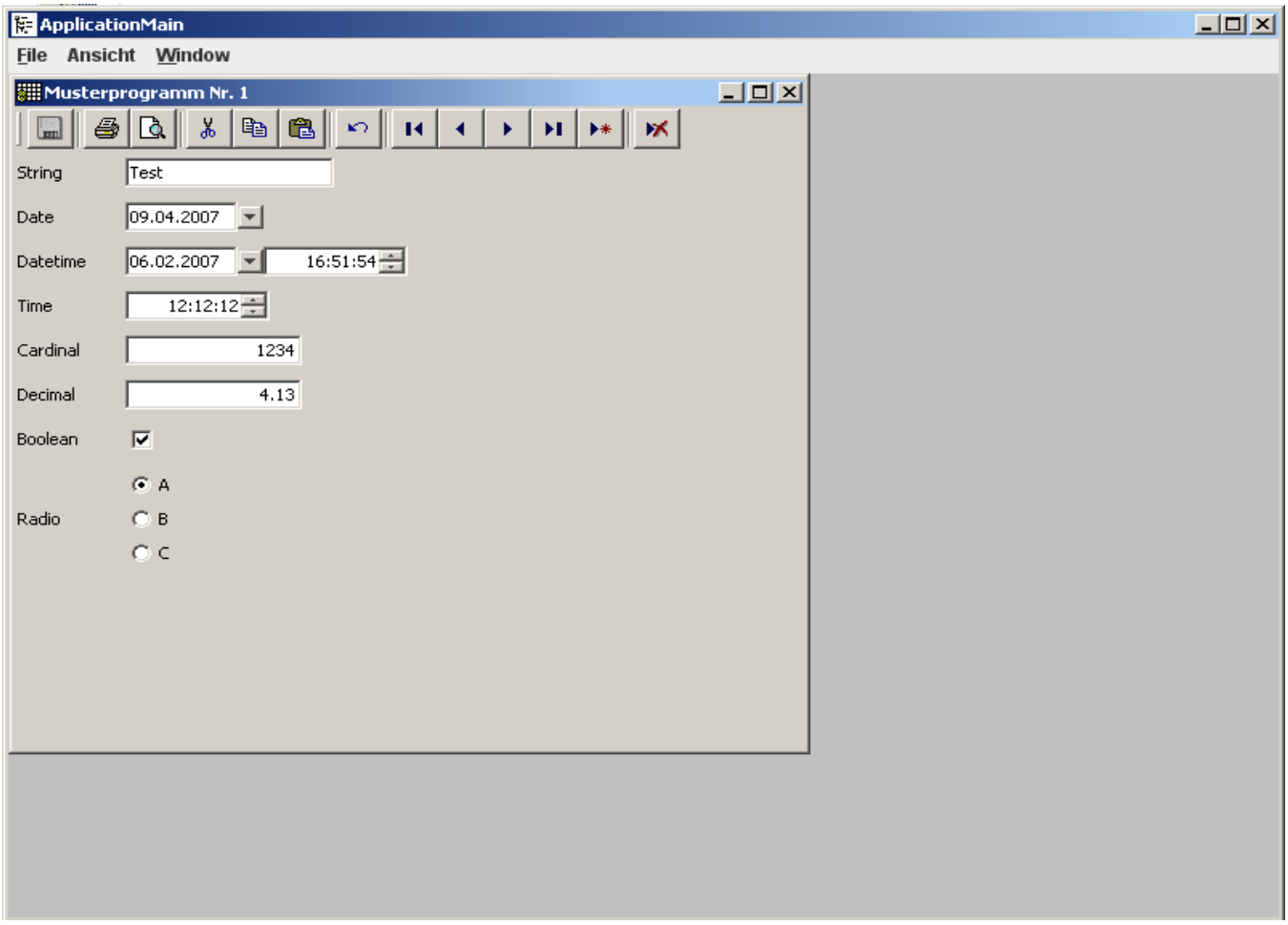
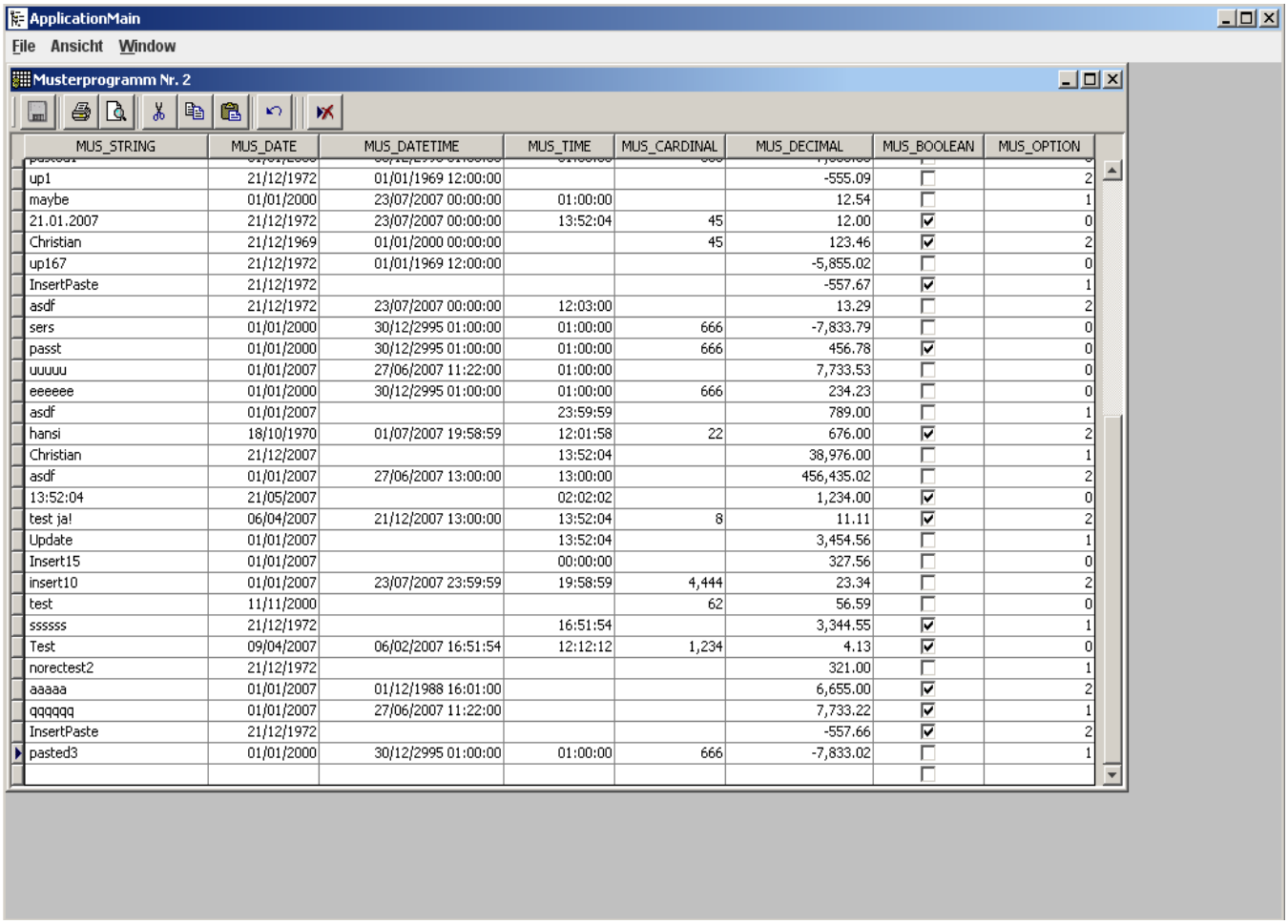


Figure 27: Prototype 1 Java: UI with Fields

D.2 Prototype 2: UI with Table



MUS_STRING	MUS_DATE	MUS_DATETIME	MUS_TIME	MUS_CARDINAL	MUS_DECIMAL	MUS_BOOLEAN	MUS_OPTION
up1	21/12/1972	01/01/1969 12:00:00			-555.09	☐	2
maybe	01/01/2000	23/07/2007 00:00:00	01:00:00		12.54	☐	1
21.01.2007	21/12/1972	23/07/2007 00:00:00	13:52:04	45	12.00	☑	0
Christian	21/12/1969	01/01/2000 00:00:00		45	123.46	☑	2
up167	21/12/1972	01/01/1969 12:00:00			-5,855.02	☐	0
InsertPaste	21/12/1972				-557.67	☑	1
asdf	21/12/1972	23/07/2007 00:00:00	12:03:00		13.29	☐	2
sers	01/01/2000	30/12/2995 01:00:00	01:00:00	666	-7,833.79	☐	0
passt	01/01/2000	30/12/2995 01:00:00	01:00:00	666	456.78	☑	0
uuuuu	01/01/2007	27/06/2007 11:22:00	01:00:00		7,733.53	☐	0
eeeeee	01/01/2000	30/12/2995 01:00:00	01:00:00	666	234.23	☐	0
asdf	01/01/2007		23:59:59		789.00	☐	1
hansi	18/10/1970	01/07/2007 19:58:59	12:01:58	22	676.00	☑	2
Christian	21/12/2007		13:52:04		38,976.00	☐	1
asdf	01/01/2007	27/06/2007 13:00:00	13:00:00		456,435.02	☐	2
13:52:04	21/05/2007		02:02:02		1,234.00	☑	0
test ja!	06/04/2007	21/12/2007 13:00:00	13:52:04	8	11.11	☑	2
Update	01/01/2007		13:52:04		3,454.56	☐	1
Insert15	01/01/2007		00:00:00		327.56	☐	0
insert10	01/01/2007	23/07/2007 23:59:59	19:58:59	4,444	23.34	☐	2
test	11/11/2000			62	56.59	☐	0
ssssss	21/12/1972		16:51:54		3,344.55	☑	1
Test	09/04/2007	06/02/2007 16:51:54	12:12:12	1,234	4.13	☑	0
norectest2	21/12/1972				321.00	☐	1
aaaaa	01/01/2007	01/12/1988 16:01:00			6,655.00	☑	2
qqqqqq	01/01/2007	27/06/2007 11:22:00			7,733.22	☑	1
InsertPaste	21/12/1972				-557.66	☑	2
pasted3	01/01/2000	30/12/2995 01:00:00	01:00:00	666	-7,833.02	☐	1
						☐	

Figure 28: Prototype 2 Java: UI with Table

D.3 Prototype 3a: Master/Detail UI

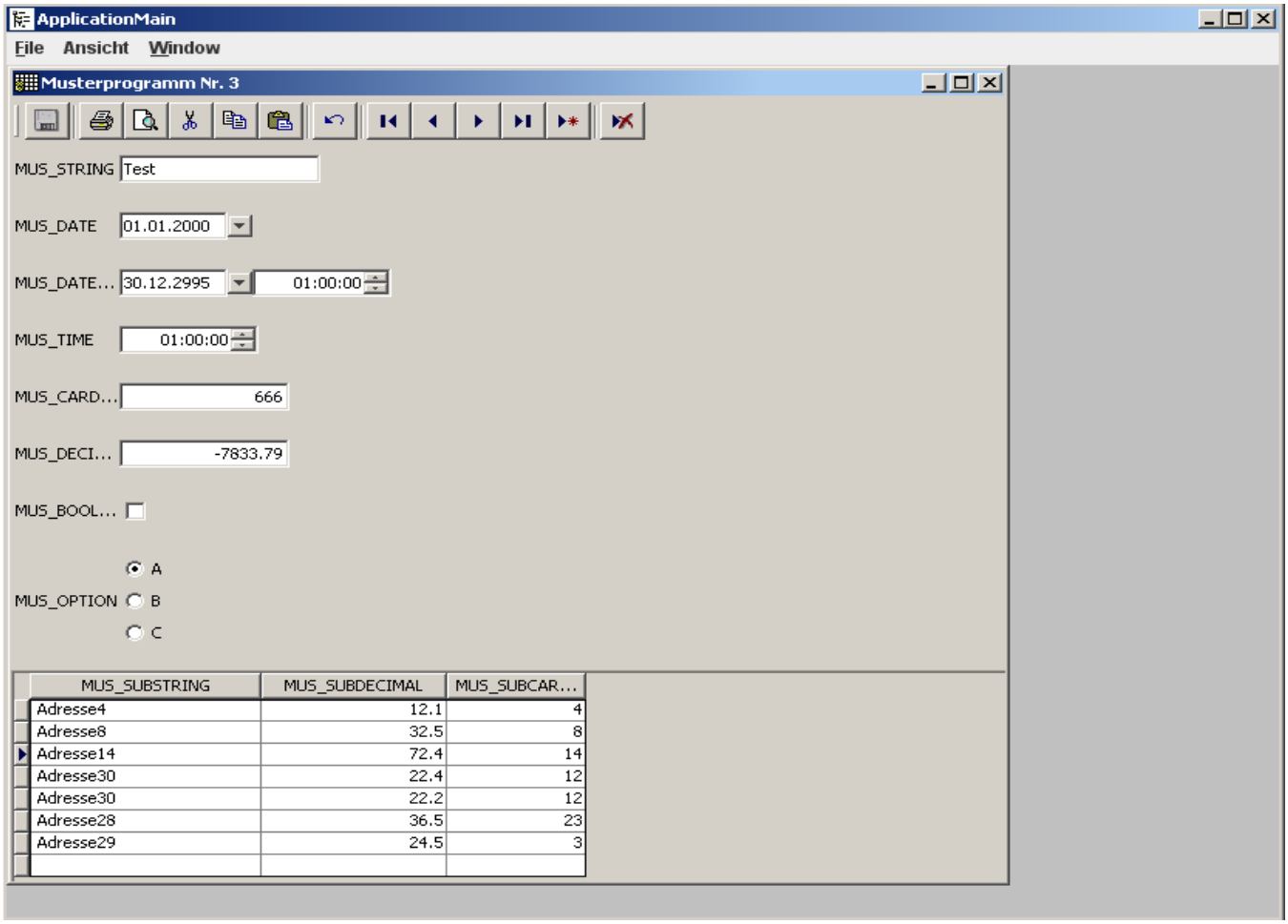


Figure 29: Prototype 3A Java: Master/Detail UI

D.4 Prototype 3b: Master/Detail UI with Pages

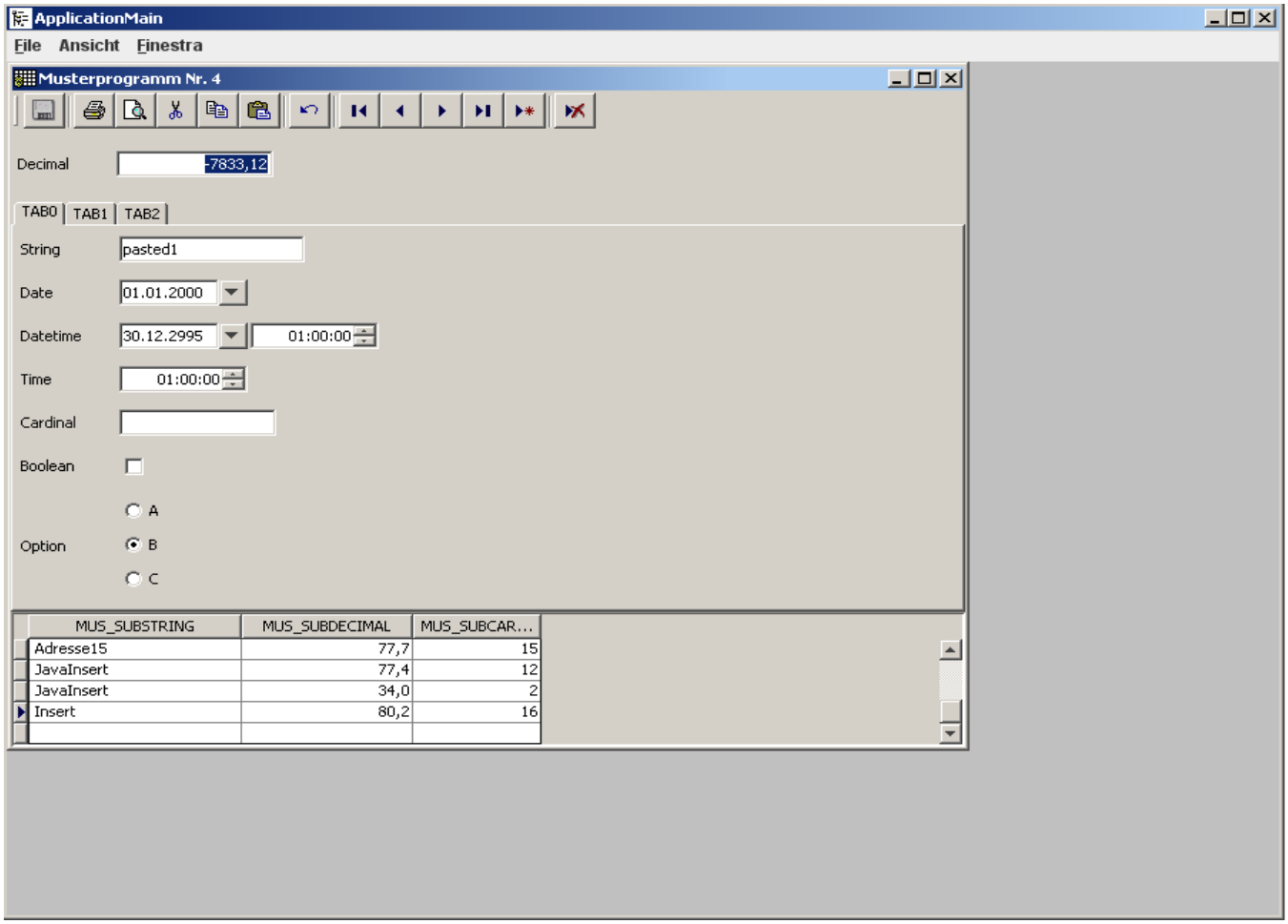


Figure 30: Prototype 3B Java: Master/Detail UI with Pages

E Subsystem Type Specification

E.1 IQSubsysFormType

Enumeration	Value	Description
IQSFT_GENERIC	0	The subsystem can handle all form types. The caller will call the resize handler as the callers client area size is changed for the subsystem. The iRequestedWidth and iRequestedHeight properties are honoured, if possible, by the caller.
IQSFT_MDICHILD	1	The subsystem is targeted to an MDI Child Form. The caller will call the resize handler as the callers client area size is changed for the subsystem. The iRequestedWidth and iRequestedHeight properties are honoured, if possible, by the caller.
IQSFT_DIALOG	2	The subsystem is targeted to a fixed-sized dialogue. The caller will not resize the object as the callers client area size is changed for the subsystem. The iRequestedWidth and iRequestedHeight properties are either honoured by the caller or the subsystem will not be loaded.
IQSFT_TOOLBOX	3	The subsystem is targeted to a fixed-sized toolbox window. The caller will not resize the object as the callers client area size is changed for the subsystem. The iRequestedWidth and iRequestedHeight properties are either honoured by the caller or the subsystem will not be loaded.

Table 10: IQSubsysFormType

E.2 IQSubsysUseType

Enumeration	Value	Abbrev.	Description
IQSUT_OTHER	-1	O	The subsystem is used for other purposes (e.g., a dialog asking for parameters for a report to be created).
IQSUT_DATAMANAGEMENT	0	DM	The subsystem is used for entry and maintenance of data.
IQSUT_DATABROWSE	1	DB	The subsystem is used to browse for data.

Table 11: IQSubsysUseType

F IRS File for Constants

FORMNAME	STRINGID	STRINGDATA	FIELDNAME
frm$APPLICATION$	0	Copyright© Inova Q Limited 2007 - All rights reserved.	$COPYRIGHT$
frm$APPLICATION$	1	MUSTER1	$DESCRIPTION$
frmAPPLICATIONMAIN	0	Ansicht	ANSICHT
frmAPPLICATIONMAIN	1	ApplicationMain	$CAPTION$
frmMUSTER1	0	String	MUS_STRING$TOOLTIP$
frmMUSTER1	1	Date	MUS_DATE$TOOLTIP$
frmMUSTER1	2	Datetime	MUS_DATETIME$TOOLTIP$
frmMUSTER1	3	Time	MUS_TIME$TOOLTIP$
frmMUSTER1	4	Cardinal	MUS_CARDINAL$TOOLTIP$
frmMUSTER1	5	Decimal	MUS_DECIMAL$TOOLTIP$
frmMUSTER1	6	Boolean	MUS_BOOLEAN$TOOLTIP$
frmMUSTER1	7	Alle	MUS_BOOLEANV00
frmMUSTER1	8	Ja	MUS_BOOLEANV01
frmMUSTER1	9	Nein	MUS_BOOLEANV02
frmMUSTER1	10	A	MUS_OPTION0
frmMUSTER1	11	B	MUS_OPTION1
frmMUSTER1	12	C	MUS_OPTION2
frmMUSTER1	13	Radio	MUS_OPTION$TOOLTIP$
frmMUSTER1	14	String	MUS_STRING
frmMUSTER1	15	Date	MUS_DATE
frmMUSTER1	16	Datetime	MUS_DATETIME
frmMUSTER1	17	Time	MUS_TIME
frmMUSTER1	18	Cardinal	MUS_CARDINAL
frmMUSTER1	19	Decimal	MUS_DECIMAL
frmMUSTER1	20	Boolean	MUS_BOOLEAN
frmMUSTER1	21	Radio	MUS_OPTION
frmMUSTER1	22	Musterprogramm Nr. 1	$CAPTION$

Table 12: IRS File for Constants

References

[Bruce 1999] Jonathan Bruce, Jon Ellis, and Maydene Fisher. JDBC API Tutorial and Reference(Java Series).Addison Wesley, 1999

[Brucherseifer 2004] Eva Brucherseifer. Softwaremigration Linuxtag 2004. Whitepaper basysKom GbR, 2004

[Burbeck 1992] Steve Burbeck, Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC),

[Chikofsky et al. 1990] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 7(1), January 1990.

[Computerbase.de] http://www.computerbase.de/lexikon/Java_Database_Connectivity

[Corbi 1990] T. Corbi. Program understanding: Challenge for the 1990's. IBM Systems Journal, 28(2):294-306, 1989.

[Dale et al. 2002] Nell B. Dale, Daniel T. Joyce, Chip Weems. Object-Oriented Data Structures Using Java. Jones and Bartlett Publishers, 2002.

[Darwin 2007] Darwin Ian, Java's Reflection API (Article on Beautiful Code), http://beautifulcode.oreillynet.com/2007/08/javas_reflection_api.php, August 2007

[De Icaza 2005] Miguel de Icaza. Mono at ApacheCon Novell. Jan 2005. Research Report

[Dumbill et al. 2004] Dumbill Ed, Bornstein Niel M., Mono: A Developers Notebook™. O'Reilly , July 2004

[Easton et al. 2004] Easton M., King J. Cross-Platform .NET Development: Using Mono, Portable.NET and Microsoft .NET, Heidelberg 2004

[Feigenbaum 2006] Barry Feigenbaum. SWT, Swing or AWT: Which is right for you?. Article on IBM DeveloperWorks (http://www.ibm.com/developerworks/grid/library/os-swingswt/), February 2006

[Fisher et al. 2003] Maydene Fisher, Jon Ellis, and Jonathan Bruce. JDBC API Tutorial and Reference, Third Edition, June 2003

[Fjeldstad et al. 1979] R.K.Fjeldstad and W.T.Hamlen. Application Program Maintenance Study - Report to our Respondents. In Proc. GUIDE 48, Philadelphia, 1979.

[Geary 2002] David M. Geary. Graphic Java 2.0: die JFC beherrschen. Prentice Hall, München, 2002.

[Gimnich et al. 2005] Rainer Gimnich und Andreas Winter. Workflows der Softwaremigration. http://www.uni-koblenz.de/ winter/papers/gimnichwinter2005.pdf.

[Gosling et al. 1996] James Gosling and Henry McGilton. The Java Language Environment, Sun Microsystems Whitepaper, May 1996

[Hall et al. 2001] Marty Hall and Larry Brown. Core Web Programming (2nd Edition), Chapter 14(Basic Swing). Prentice Hall, 2001

[Hasselbring et al. 2004] W. Hasselbring, R. Reussner, H. Jaekel, J. Schlegelmilch, T. Teschke, S. Krieghoff. The Dublo Architecture Pattern for Smooth Migration of Businees Information Systems: An Experience Report. 26th ICSE, 117 - 126, Edinburgh, 2004.

[Inova Q 2002] IGS - Inova Q Generator System. Inova Q Technology White Paper TD-02.450, August 2002

[Inova Q 2004] Erneuerung des Entwicklungsprozesses mit IGS. Inova Q Technology White Paper TD-04.300,January 2004

[Kingsley 1993] Kingsley Idehen. Open Database Connectivity Without Compromise. ODBC Technical Whitepaper, 1993

[Krüger 2002] Guido Krüger. Handbuch der Java-Programmierung, 3. Auflage. Addison-Wesley,Deutschland, 2002

[Malone 2005] Thomas Malone. Managing Software Development, Slides to Lecture Information Technology Essentials, 2005, ocw.mit.edu/.../15-561Spring-2005/5CC9E1B4-1703-40E9-928E-35C3342C143A/0/lecture12.pdf

[Mählmann 2005] Mählmann Lars. Untersuchung von Mono als Plattform für Webservices auf mobilen Endgeräten. Diplomarbeit, Hochschule für Angewandte Wissenschaften Hamburg. April 2005

[Microsoft.com] Component Object Model Technologies http://www.Microsoft.com/com/default.mspx

[msdn] Ado Programmer Guide, http://msdn2.Microsoft.com/en-us/library/ms805098.aspx

[Microsoft.net] Microsoft .NET Oveview http://www.microsoft.com/net/Overview.aspx

[Müller 1997] Bernd Müller. Reengineering - Eine Einführung. Teubner, Stuttgart, 1997.

[Niemeyer et al. 2005] P. Niemeyer, J. Knudsen. Learning Java - Third Edition. O'Reilly , May 2005

[Oracle 2001] Oracle Corporation. Oracle9i JDBC Developer's Guide and Reference,Release 1 (9.0.1),Part Number A90211-01, 2001

[Oracle 2002] Oracle Corporation. Oracle9i JDBC Developer's Guide and Reference,Release 2 (9.2),Part Number A96654-01, 2002

[Reilly 1999] David Reilly. Inside Java: The Java Programming Language. Article on Java Coffee Break (http://www.javacoffeebreak.com/articles/inside_java/index.html)

[Sauter 1999] http://www.umsl.edu/ sauter/analysis/prototyping/proto.html

[SearchSQLServer.com] http://searchsqlserver.techtarget.com/sDefinition/0,,sid87_gci214419,00.html

[Simmons 2004] Robert Simmons jr. Hardcore Java, Secrets of Java Masters. O'Reilly 2004

[Sneed et al. 2004] H. Sneed, H. Hasitschka, M. Teichmann. Software Produktmanagement, Wartung und Weiterenwicklung bestehender Anwendungssysteme. Dpunkt, Heidelberg, 2004.

[Sommerville 2000] Ian Sommerville. Software Engineering, 6th Edition, Addison Wesley, 2000

[Steyer 2003] Ralph Steyer, Java 2 - M+T Pocket, Das Programmier-Handbuch, ISBN: PDF-3-8272-6106-6, Markt&Technik, 2003

[Sun 2001] Sun Microsystems, Inc. Core J2EE Patterns - Data Access Objects, http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

[Sun 2004] Sun Microsystems, Inc. Java™ 2 Platform Standard Edition 5.0 API Specification, http://java.sun.com/j2se/1.5.0/docs/API/, 2004.

[Sun 2004a] Sun Microsystems, Inc. http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html

[Sun 2006] Sun Microsystems, Inc. JDBC 4.0 Specification, JSR 221, Lance Andersen, November 7, 2006

[Sun 2007] Sun Microsystems, Inc. The Java™ Tutorials - Trail: The Reflection API. https://cis.med.ucalgary.ca/http/java.sun.com/docs/books/tutorial/reflect/index.html, 1995 - 2007

[Terenkhov et al. 2000] Andrey A. Terenkhov, Chris Verhoef. The Realities of Language Conversion. IEEE Software 17(6), 2000

[Ullenboom 2007] Christian Ullenboom, Java ist auch eine Insel. Programmieren mit der Java Standard Edtition 6, 6. aktualisierte und erweiterte Auflage, Gallileo Computing, 2007

[Weiderman et al. 1997] N. Weiderman, J. Bergey, D. Smith, S. Tilley. Approaches to Legacy System Evolution, Technical Report CMU/SEI-97-TR-014, Software Engineering Institute, Carnegie Mellon University,Pittsburgh, 1997

[Wells 2001] Garth Wells, Code-centric: T-SQL programming with stored procedures and triggers, Apress, Berkely, CA, 2001

[Xerox 1998] Xerox Corporation, The AspectJ™ Programming Guide.1998-2001 Xerox Corporation, 2002-2003 Palo Alto Research Center, http://www.eclipse.org/aspectj/doc/released/progguide/index.html