



TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

Master's Thesis

A Reusable Toolkit for Rapid Development of Small Business Applications

carried out at the
Information Systems Institute
Distributed Systems Group
Vienna University of Technology

under the guidance of
Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Harald C. Gall

by
Thomas Bleier, thomas@bleier.at
Hauptstraße 1, 7372 Weingraben
Matr. Nr. 9420329

To my parents who made all this possible for me

Abstract

In the world of software, many people talk about software reuse. While there are several theoretical initiatives for increasing the percentage of reused software components, software developers currently use little or none of them in their daily work. There are many reasons for this: one of them is the fact that successful software reuse across application domains is hard to achieve.

This master's thesis provides an application of software reuse principles to one application domain, the domain of small business systems. During the course of this thesis a toolkit is developed that can be used to create applications that have a high reuse percentage. That will enable short development times and better software quality. The purpose of this toolkit is to facilitate rapid development of customized applications for small businesses. Effective reuse is a driving force for this to work.

In this document the design and the implementation of the small business toolkit is described in detail, starting with a domain analysis and a definition of the requirements. Then the toolkit components are designed and implemented. Finally application development with the toolkit is analyzed and the benefits of using it are evaluated.

Zusammenfassung

In der Informatik wird viel über Software Wiederverwendung gesprochen. Von den theoretischen Überlegungen zur Wiederverwendung von Software wird aber in der Praxis wenig angewandt. Einer der Gründe dafür ist sicherlich, daß effiziente Wiederverwendung von Software über Anwendungsdomänen hinweg meist sehr schwer zu erreichen ist.

In dieser Diplomarbeit werden Prinzipien der Software Wiederverwendung für die Erstellung von Software für Klein- und Mittelbetriebe angewandt. Im Rahmen der Arbeit wird ein Toolkit beschrieben, daß zur Entwicklung von Anwendungen mit einem hohen Anteil an wiederverwendeten Komponenten führen soll. Diese Anwendungen können damit schneller und in besserer Qualität erstellt werden. Der Zweck dieses Toolkits ist die Unterstützung von Rapid Application Development (RAD) von maßgeschneiderter Software. Effiziente Wiederverwendung ist hierbei von besonderer Bedeutung.

Dieses Dokument beschreibt die Entwicklung des Toolkits, beginnend mit einer Domänenanalyse und der Festlegung der Anforderungen. Danach folgt das Design und die Implementierung der Komponenten des Toolkits. Abschließend wird die Verwendung der erstellten Bauteile und die Vorteile, die durch diese Verwendung erzielt werden können, untersucht.

Acknowledgments

I would like to thank Harald Gall for advising this master's thesis. Some time ago there was just an idea, and with this master's thesis it became reality. His ongoing support and valuable comments made it possible for me to get from this idea to the work you currently read.

Thanks also to my intended wife, Sonja, who had to accept that I paid my attention to this master's thesis instead of the preparations for our soon wedding.

*Inventing is a combination of brains and materials.
The more brains you use, the less material you need.*

Charles Franklin Kettering, inventor of the battery
ignition and the electrical starter.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goal	1
1.3	Overview	2
1.4	Software reuse	3
2	Domain Analysis	5
2.1	Domain Analysis	5
2.2	The application domain	6
2.3	Overview of applications in the SBA domain	7
2.4	Sample applications	8
2.5	Sample standard applications	11
2.6	Architecture of the sample applications	12
3	Requirements	15
3.1	Functional requirements	15
3.2	Non-Functional Requirements	20
3.3	The development environment	21
3.4	Reuse levels	22
4	Toolkit Architecture	25
4.1	Framework	25
4.2	Application development	25
4.3	Rapid application development	27
4.4	Development support	28
5	The toolkit and its components	31
5.1	Introduction	31
5.2	Description template	31
5.3	Data entry	32
5.4	Document printing	37
5.5	Master data administration	39
5.6	Report generation	47
5.7	Data export	53
5.8	Standard components	56
5.9	Application settings management	67
5.10	Other parts of the application	72

5.11 Utility components	74
6 Toolkit overview	77
6.1 Characterization	77
6.2 Classification	77
6.3 Class diagram	81
7 Toolkit Evaluation	85
7.1 Overview	85
7.2 Applications	85
7.3 Case study	86
7.4 Reuse benefit analysis	88
8 Summary and Conclusions	93
A Sample source documentation	95
A.1 TboRepPropFrameBase Class Reference	95
A.2 TboRepPropFrameEnabledBase Class Reference	97
A.3 TboRepPropFrameDateRange Class Reference	98
A.4 TboRepPropFrameDBLookup Class Reference	102
A.5 TboRepPropFrameSortOrder Class Reference	104
B Glossary	107
C Bibliography	109
D Internet references	113

List of Figures

2.1	The focus of the toolkit	6
2.2	Parts of a typical application in the SBA domain	7
2.3	The “4+1” process view for this domain.	12
2.4	The “4+1” logical view for this domain.	13
2.5	The “4+1” development view for this domain.	13
2.6	Reusable parts in the development view.	14
3.1	Sample of the data for a compound record	16
4.1	Toolkit usage scenario with a version control system	29
5.1	Activity diagram for the simple record entry form	33
5.2	Class diagram for the compound record entry pattern	35
5.3	Class diagram for the master data edit pattern	40
5.4	Action diagram showing the actions in the master data edit list form	41
5.5	Class diagram for the report properties pattern	49
5.6	Class diagram for the data export pattern	54
5.7	Class diagram for the application settings manager pattern	69
6.1	The class diagram of the toolkit, part 1	82
6.2	The class diagram of the toolkit, part 2	83
6.3	The class diagram of the toolkit, part 3	84
7.1	Case study — master data set list dialog	86
7.2	Case study — automatically generated edit form	87
7.3	Case study — customized master data edit form	87
7.4	Case study — example code for the master data edit functionality	88
7.5	NOCs distribution in the applications	90
7.6	LOCs distribution in the applications	91

List of Tables

4.1	Development process adaptations for reuse based software development . . .	26
6.1	Classification — abstract components	78
6.2	Classification — concrete components	79
6.3	Classification — design patterns	81
7.1	Overall comparison of the implementations	89
7.2	Reuse benefit — new classes vs. reused classes	89
7.3	Reuse benefit — built from scratch vs. derived classes	90

Chapter 1

Introduction

This chapter gives a short introduction into the motivation behind this master's thesis, its goal, a summary of the chapters and a brief overview about software reuse in general.

1.1 Motivation

For medium- to large-sized companies there are several well known applications that those companies use for their day-to-day business. Those software systems include modules for nearly every part of an organization. These modules can be customized to fit the specific needs of a company. For small- to medium-sized companies, those applications are not suitable. Most often they are far too complex for the structure of such companies, which makes them expensive in both deploying and maintaining the software. So these companies are required to use off-the-shelf applications. Those applications make it hard or even impossible to change the software to fit the organization's processes, but instead require the change of the business processes to fit the software. While this situation is not satisfying, most small businesses simply cannot afford custom-made applications and thus have no other opportunity. This master's thesis should enable rapid development of such applications, and thus make them affordable for small companies.

1.2 Goal

The goal of this master's thesis is to create a toolkit that allows fast development of small business applications.

Before creating such a toolkit, some preparations have to be made. First the application domain has to be analyzed to identify reusable entities. These reusable entities have to be categorized and specified in detail. Then the components of the toolkit have to be designed. Special effort should be made to ensure that the toolkit is extensible in the future.

The next phase is the implementation and testing of the components of the toolkit. Finally, it is interesting to see how the toolkit can be used in practice and what benefits can be achieved by creating applications based on that toolkit.

One requirement for the toolkit is that it should be based on the Borland¹ C++ Builder integrated development environment (IDE), which includes a component library called VCL. The toolkit should make use of those components and the component architecture of VCL. The final applications should use a standard Windows GUI as the front-end and a RDBMS (InterBase² from Borland) as the database back-end. Those requirements should be supported by the toolkit.

1.3 Overview

The thesis is organized into eight main chapters. The first chapter (*this* chapter) explains the reason why this thesis was written and the goal that should be reached. It also gives a short overview of software reuse in general.

The second chapter is dedicated to the domain analysis of the application domain. An introduction into domain analysis in general is given, and the application domain is outlined. Then some sample applications for that domain are examined and the architecture of such applications is analyzed.

Chapter three provides a requirements analysis for the final toolkit. Functional and non-functional requirements are discussed and the development environment is investigated. Finally the different levels of reuse that are possible in this environment are outlined.

The next chapter describes the architecture of the toolkit. The toolkit is embedded in an integrated development environment, and this chapter discusses the place of the toolkit in this environment and the usage scenarios resulting from that place.

In chapter five the toolkit is described. For every reusable element in the toolkit the properties and usage are outlined using a description template. Chapter six summarizes the toolkit description by providing an overview of the components from different viewpoints.

Finally chapter seven is dedicated to an analysis of the benefits gained by using the toolkit in building applications. This is done by examining two applications and comparing them to stand-alone built versions.

The last chapter summarizes the work with conclusions and directions for future work. The appendix contains a sample of the source code documentation that was built using an automated documentation tool, the glossary and the bibliography.

¹<http://www.borland.com/>

²<http://www.interbase.com/>, <http://firebird.sourceforge.net/>

1.4 Software reuse

Software reuse is told to provide big advantages for software development such as shorter development times, better quality of the developed products and better maintainable software products. While these ideas are more than 30 years old (initiated at the famous 1968 NATO Conference by Doug McIllroy [Mci69]), component-based development is not yet the technology of choice for many software development projects. Too many difficulties are involved and prevent this technique from revolutionizing the software industry until now.

Nevertheless many researchers all over the world are working on improving this situation. New methods are being developed and circumstances that promote or inhibit effective software reuse are analyzed. A paper by David C. Rine [Rin97] describes success factors for software reuse. Apart from a few management issues they include:

- a product line approach
- architecture for reuse with standardized interfaces and data formats
- a common architecture across the product line
- design for manufacturing (design with reuse)
- domain engineering
- state-of-the-art tools and methods
- high-level reuse (not just code)

Most research in this field is oriented towards big software companies. The approaches most often also include management guidelines and recommendations for changes to the business process of the software development company. Only few authors, such as [Ara94b], note that point explicitly.

In this master's thesis these principles of effective software reuse are applied to *small* applications. Most of them (especially the non-management issues) can more or less easily be applied to software development of applications in the range of a few to several man months. Nevertheless they should provide similar benefits as for big applications.

Chapter 2

Domain Analysis

This chapter provides an introduction into the field of domain analysis and analyzes the application domain. Sample applications for that domain are investigated and their architecture is described.

2.1 Domain Analysis

Domain Analysis is the process of extracting common parts of applications in a domain to provide a foundation for reuse.

According to [Sam97], [Ara94a] and others Domain Analysis involves the following activities:

- Domain definition and preparation
- Data collection
- Data analysis and classification
- Evaluation

Before a domain can be analyzed it has to be clearly defined. This is done by specifying its width (where the domain ends and another one begins) and its depth (which sub-domains should be included). The definition of the domain also determinates the information sources for further investigation of the domain.

In the next step data about the domain has to be collected. There are different methods for doing this such as analyzing existing applications, talking with domain experts or reviewing specific literature. Such data serves as the foundation for finding reusable parts in the domain.

The collected data has to be classified to find commonalities among different applications. Candidates for reuse can be found on different abstraction levels. These candidates have to be further refined and generalized, which leads to the specification of a reusable toolkit for the domain.

After the analysis process is finished, it has to be evaluated. This is an iterative process which refines the specification created in the previous step.

2.2 The application domain

The application domain chosen for this thesis is the field of small business applications (SBA). The term small refers both to the size of the applications itself as well as to the size of the businesses for which those applications are created. The term business applications means applications a company needs for its business, especially in the financial and accounting field.

Examples for applications in that domain would be:

- Invoice management
- Order management
- Stock keeping
- Costing
- Accounting
- Tax management
- Payroll systems

Each of these application areas can be considered a sub-domain of the application domain. As shown in Figure 2.1, those sub-domains have parts which are specific to them and other parts which overlap with other sub-domains. This toolkit focuses on the overlapping part, but should be designed to allow integration of sub-domain specific parts. Thus it provides basic functionality for most sub-domains of the SBA domain, but it can also contain aspects which are specific to a few or to one special sub-domain.

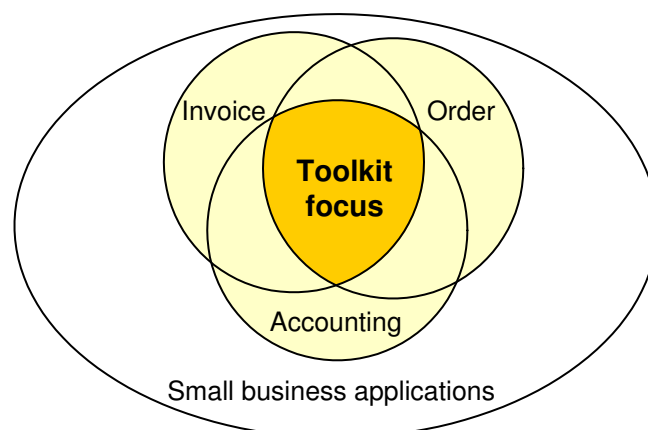


Figure 2.1: The focus of the toolkit

2.3 Overview of applications in the SBA domain

Experience shows that typical applications in this field can be divided into the following parts:

- Data entry
- Documents (e.g. invoice, delivery note)
- Master data administration
- Reports

Beside that parts, the applications also contain components that can be found in many applications in other domains. Those include standard components, utility classes and others. While a great part of those components is readily available (standard user interface widgets, for example) there may be components that are not available or that can be specifically adjusted to the needs in the SBA domain. The domain specific parts of an application are based on that standard components and utility classes. Figure 2.2 shows the relation of those application building blocks.

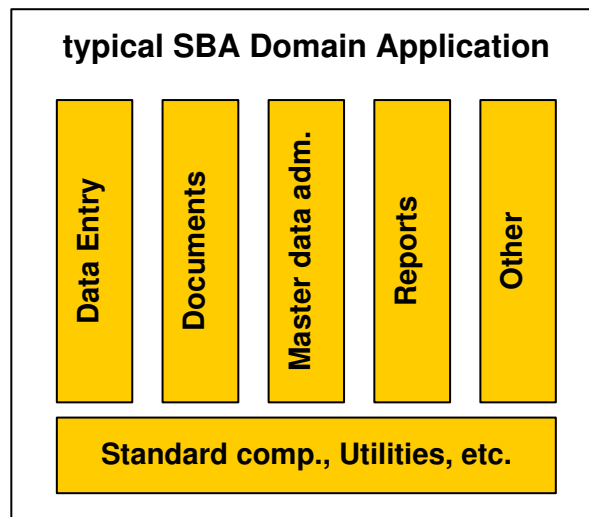


Figure 2.2: Parts of a typical application in the SBA domain

Most applications in that domain have to do with document handling, in one or in the other way. These documents can be invoices, delivery notes or any other kind of vouchers or receipts. That makes the data entry part of the software the most critical one. In many applications, only a few dialogs are used most of the time. For example, an invoice management application contains a dialog in which the invoices are entered. This dialog is used nearly the whole day. So, handling of this dialog should be as comfortable as possible.

Even in the days of the propagated paper-less office computer applications have to print documents, especially in this application domain. Often these documents have special

requirements such as special paper sizes, printers, etc. After a document has been printed, it has to be stored permanently to allow later retrieval of the data. This can be necessary for printing a duplicate of the original document, or for summarizing the data for reports.

Most applications in that domain have some sets of data that are relatively constant over time. Those data records have to be entered initially when deploying the application, but do not change often afterwards. But, of course, that data has to be maintained. For the invoice management application that could be the customer database or the article database. When creating an invoice, the user wants to select a customer from the database, and choose the articles which should be charged, but the system should automatically know the customer's address and the properties of the chosen articles.

Since a business application should store all data entered, the user wants to view reports summarizing the data filtered by some specified criteria. It should be easy for the user to get the report with the data he needs, and it should be easy for the application developer to create a custom report for a company.

That application structure requires a different user interface concept than typical document-oriented applications such as word processors, spreadsheets, etc. The user of the applications works task-oriented instead of document-oriented.

2.4 Sample applications

In the following a few sample applications of the application domain are analyzed. The analysis of these applications should show the commonalities of different applications. The analyzed applications are:

- An invoice application for a production company
- A production monitoring application for a production company
- An invoice application for a wood trader
- An invoice application for a joiner
- A costing application for a freight forwarding agency

2.4.1 Invoice application for a production company

The first application is used for creating invoices for a production company. The company sells goods which they produce on their own. On each delivery, the delivered goods are either paid cash, which requires an invoice for each delivery, or the goods are delivered on account with a delivery note. Then, later, an invoice is created which contains one or more deliveries. The application should print the delivery notes and the invoices,

and should take care of creating invoices for all not-yet-charged deliveries on a user's request.

The application has the typical structure described in section 2.3. From the main menu the user can reach a dialog in which he can enter the data for a cash invoice, allowing him to select customers and articles from a database. This dialog is also used to create delivery notes, which depends on the preferences of the customer. After an invoice is created, it is printed and stored. Another menu item is used to create an invoice for open deliveries. After selecting this command the system prints out invoices for all customers which have delivery notes that are not yet charged.

The user also has the possibility to correct an invoice or a delivery note if it should contain a mistake. Correcting also includes canceling of an item, which excludes the invoice from the reports or excludes the delivery from getting charged with the next invoice.

Customers and articles, along with their properties, are stored in a database. The application allows the user to edit, delete or create a customer or article. The article also includes pricing information, which can depend on the customer or on the customer group (discounts).

2.4.2 Production monitoring application

This application is used to monitor the production of goods in an agricultural production company. It is a very simple application that records daily production of goods, and later allows the user to view statistics of the production and control if everything is running well.

As before the application shows the structure described in section 2.3. The main dialog in this application is the dialog in which the amount of produced goods is entered. The user selects some properties of the production and enters the amount of goods produced. This data is stored in a database.

Documents are not printed in this application, because all data is stored in the database and no receipt is needed for each record. There are some master data items which have to be maintained, but these are only a few different record sets. One key element of this application are the reports. When desired the user can view and print statistics and graphs showing the production progress and various other data.

2.4.3 Invoice application for a wood trader

The second invoice application described here was designed for a wood trader. The special requirement for this application was the price calculation of the articles. The trader calculates prices by length, by area or by volume, depending on various factors. In the end, he wants to know the volume of the complete shipment to estimate the needs for the transport vehicle.

This application also consists of the typical elements described in section 2.3. The main dialog is used for entering data for invoices and delivery notes. Customers and articles are stored in the database, and the user can select them. For each article, a price and the corresponding price unit is suggested. This suggested price is fetched from the database. The user can now take over this suggested price, or he can decide to charge another price or another price unit. If the users changes the price unit, the system automatically converts the price to the new unit. If desired, the changed price can be stored for future usage when creating another invoice for this customer.

When creating an invoice, the user can also select delivery notes which are not yet charged and add them to the invoice. Other menu items allow the cancellation of invoices or delivery notes, and maintenance of the customer and article databases.

Upon request the user can view different statistics and reports showing for example the sold amounts of different articles. Another requirement for this application was the ability to export invoice data for processing by the accounting application. This accounting application had a defined format for importing invoices, which had to be produced by the invoicing application.

2.4.4 Invoice application for a joiner

The third invoice application looked at here should be created for a joiner. The requirements are very simple, nonetheless existing applications do not fulfill them.

The joiner produces different pieces of furniture on customer request. When charging the produced items, the invoice should contain a description of each item. This description should be free-form text, but can get long (up to a whole page, for example). So the invoice application must be capable of handling such a long text for each invoice item. It would be easiest to do that with a standard office word processing application, but the invoices should be stored in a database, which each item categorized by different properties, so that it will be possible to create reports on that data.

Beside invoices the joiner also wants to create offers, order confirmations and delivery notes. It should be possible to easily take over a receipt for the next step, for example to create an order based on a previously entered offer.

In this application we also find the typical structure defined section 2.3. The main dialog is used for creating the invoices (and the other documents) and entering items. After a document is finished, it has to be printed. This application also contains a database of master data records, which have to be maintained. And, last but not least, the user wants to view reports and statistics showing the sales figures.

2.4.5 Costing application for a freight forwarding agency

This application grew from the need for supervising the gasoline consumption of the different trucks of the freight forwarding agency. Data for gasoline consumption came

from different sources, mostly electronically, which had to be stored in a single database so that it was possible to create reports showing the detailed consumption of each truck.

Later on, the application grew to a complete costing application, including all costs and profits concerning the freight forwarding business. Costs are entered from different sources, partly electronically, partly manually, and stored in the database. Later, statistics and reports can be generated.

In this application we also find typical components of the application domain. There are some few dialogs which are used for data input. Since the primary output of the applications are statistics, there are no documents printed. The application also contains a database of master data records, which has to be maintained. And, from time to time, reports and statistics have to be printed.

2.5 Sample standard applications

In this section we are going to take a look at standard off-the-shelf applications in the domain of small business applications. The applications are

- Universe FreeLine from EasySoft¹
- PC Kaufmann 2000 from Sage KHK²

2.5.1 Universe FreeLine from EasySoft

This software product is a typical application for a trading company. It contains functions for invoice management, order management and stock keeping. It is a typical general purpose application that contains rich functionality for the application domain, but is fixed by design to various assumptions about the business process of the user.

This application can also be divided into the four parts data entry, documents, master data administration and reports. The data entry part contains various dialogs for entering the different documents like offers, orders, delivery notes and invoices. The documents can also be printed.

Other parts of the software contain the administration of customers, articles and other master data record sets. The application normally also allows the generation of various reports and statistics, which was disabled in the tested version.

¹<http://www.easysoft.de/>

²<http://www.sagekhk.de/>

2.5.2 PC Kaufmann 2000 from Sage KHK

The PC Kaufmann suite contains many different applications for nearly all facets of a small company. Since the application is not trade specific, it contains many different options and variations to cover most user's needs. On the other hand this makes it harder to find that parts of the application that a specific user needs.

The applications of the suite can also be divided into the four parts described in section 2.3. There are a few data entry dialogs, one for each part of the application. Each part also allows printing of documents. The master data administration functions are commonly used by the whole application suite. And, of course, many different statistics and reports can be viewed and printed.

2.6 Architecture of the sample applications

The description of the architecture follows the “4+1 view model” described in [Kru95]. For the chosen application domain, some of the views are very simple. Since the applications are not distributed, the process view is reduced to the model in Figure 2.3. The main application is a single process, which utilizes a relational database management system, which usually has its own process.

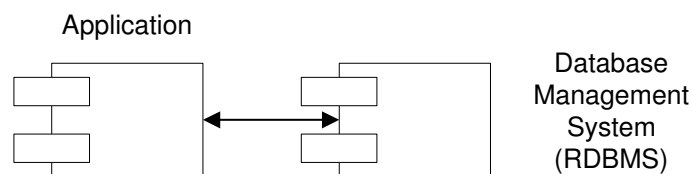


Figure 2.3: The “4+1” process view for this domain.

The physical view is similar to the process view. The RDBMS can reside on a “node” of its own, which in that case is the database server, or it can be installed on the same computer where the application is running. Since all aspects of interprocess communication and similar things are handled by the underlying database connectivity software, they are not further discussed here.

So the logical view, the development view and the scenarios are left for further investigation. While the process and the physical view are the same for all applications in the domain, the following views differ for each application. Nevertheless they have some things in common that can be discussed here.

The logical view of the applications shows the four main parts of typical applications introduced in section 2.3. For the user, these are the main components of such an application. Each of these components can be broken into smaller-grained subcomponents, which build the foundation for the toolkit described in this thesis. Figure 2.4 shows a diagram of the logical view of an application in the small business domain.

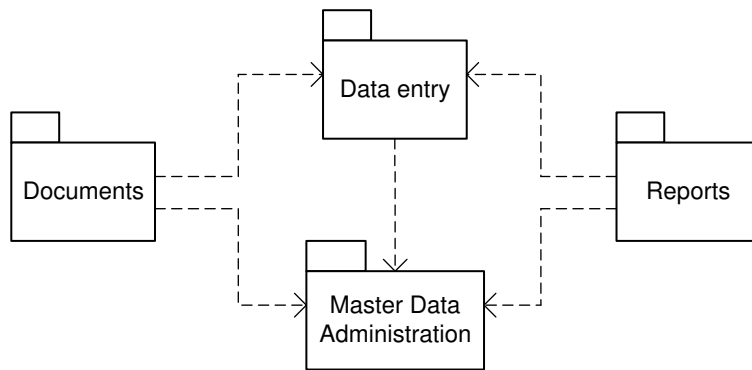


Figure 2.4: The “4+1” logical view for this domain.

The development view depends, as the name implies, on the actual development environment. For an architectural description, some common aspects of the development architecture are given here.

Usually the subsystems are somewhat similar to the components described in the logical view. Each of those subsystems is normally organized in a hierarchy of layers. Since today most applications use a relational database management system, this is at the same time the database layer, which is at the bottom of the architecture. The database management system itself is not part of the application, so the application includes a layer for interfacing to the RDBMS.

On top of this layer is the business logic. It contains functionality for preparing data and provides a interface to the next layer, the presentation layer. This layer contains the user interface. The picture in Figure 2.5 shows the development view of applications in the small business domain.

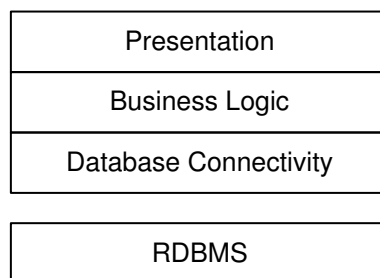


Figure 2.5: The “4+1” development view for this domain.

While this is the usual layering for business system, it also can be used to show possibilities for reuse in such a system. The RDBMS and database connectivity are off-the-shelf components and thus reused between projects. The purpose of this toolkit is to provide reusable elements in the upper two layers. Figure 2.6 shows the reusable parts in the different layers.

The scenarios, which are the “+1” part of the views, describe the interaction of the

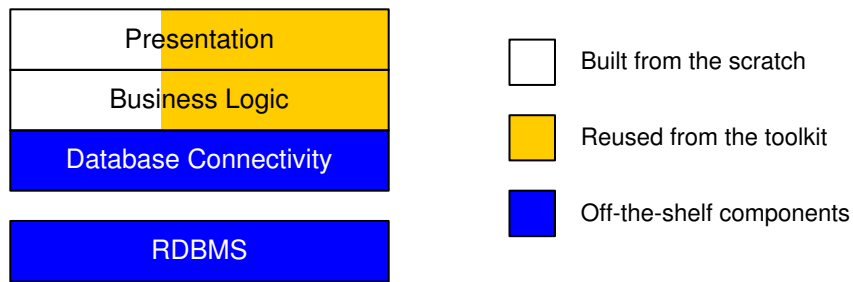


Figure 2.6: Reusable parts in the development view.

components in the other views. They describe the interactions for specific use cases and are thus not further investigated in this architectural overview.

Chapter 3

Requirements

In this chapter the functional and non-functional requirements of the toolkit are analyzed. Then an overview of the development environment is given and the different levels of reuse for this toolkit are defined.

3.1 Functional requirements

Based on the examination of typical domain applications in the sections [2.4](#) and [2.5](#) we can define the requirements of an application in the SBA domain.

Since this requirements analysis does not focus on a specific application, but instead on a domain of applications, the requirements are not specific but instead are typical requirements which an application in that domain may have. The requirements are organized by the application parts found in section [2.3](#).

3.1.1 Data entry

The data entry part is the key interface to the user and often contains only a few dialogs. Data input is record-based, which means that all properties of the record have to be entered before the record is saved. Depending on the application it is also possible to edit previously saved records, delete records or load stored records as a template for newly created records. A record contains various properties, and can also have a complex structure. There may be some different record types, but all records of a given type have the same structure.

The following classes of record types can be categorized:

- Simple record
- Compound record

The simple record class is quite obvious. The items of the record are properties, which are values of a simple data type like a number or a string. The number of properties in a record is not limited, but each property is an atomic unit which can not be divided into subcomponents. An example for a record type of this class would be a record in an accounting application. Each booking has various properties, but can not be further divided.

A record in the second class of record types consists of some data for the record itself, and a set of subitems. The properties for the record are equivalent to the simple record class. The set of subitems contains records of other types. These types belong to the simple record class. It is possible that a record can contain various different subitem record types, but these are defined in the specification for each record type. An example for a compound record would be a typical invoice, where there are some properties for the invoice itself (such as customer, date, etc) and some subitems, which also have properties on their own (article, amount, price). Different types of subitems may be subitems which contain normal article data or subitems which contain explanatory text. Figure 3.1 shows an example of a document (an invoice) with a compound record to illustrate the previous description.

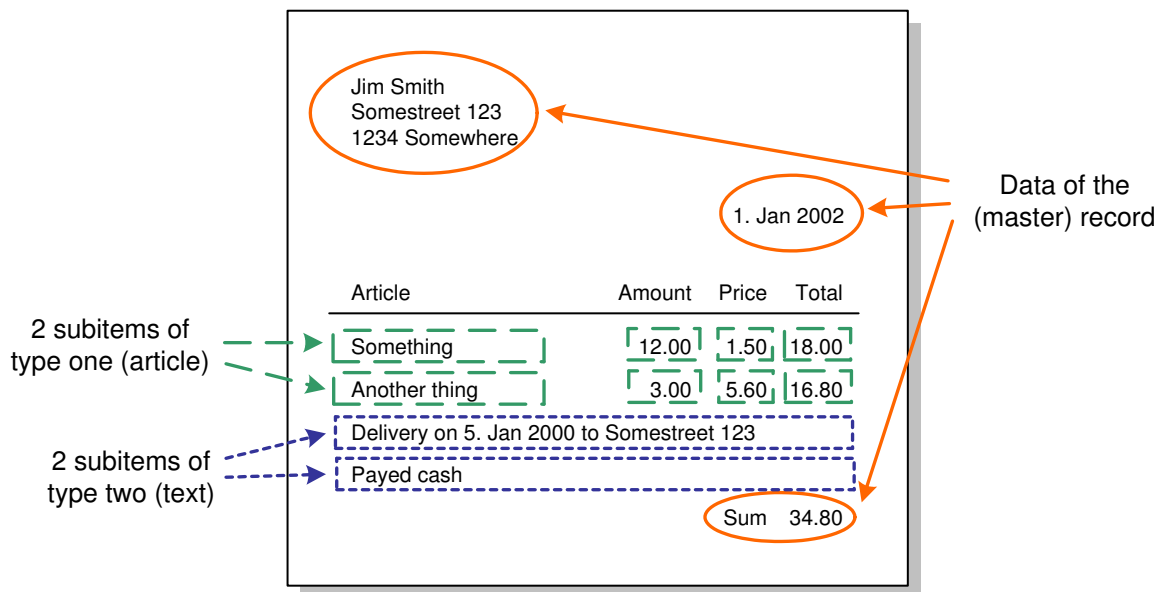


Figure 3.1: Sample of the data for a compound record

It may be possible that an application requires another class of record types. Thus the toolkit should be designed to allow integration of other record classes.

3.1.2 Document printing

When data for a record is entered, a document has to be printed (most of the time). The structure of the document is equivalent to the record itself. It can contain simple

properties, which are printed in a form-like document. If the record is a compound record, the document contains a part which shows the properties of the record and another part which contains the list of subitems. A typical document of this kind would be an invoice (somewhat similar to Figure 3.1).

In contrast to a typical office application the application in this case should “know” (or more exactly should have predefined) all necessary options to print a document (such as printer settings, number of copies, etc). Similar to those office applications it should be possible to generate a print preview, if some application requires that.

3.1.3 Master data administration

Every application in the application domain has to manage some master data sets. These data sets can be categorized into the following classes of master data record types:

- Single item records
- Key/value records
- Property set records

A single item record’s value is just the value itself, which can be any data type, but most often will be a text string. These single item records are most often used to categorize other records. The easiest representation for this type of master data records is a list. Examples for single item master data record sets would be the list of customer groups, article groups or some other categorization depending on the special application.

Key/value records have two properties: a key, which is used to locate the specific record, and a value, which is associated to that specific key. Both key and value can be of any data type, but they are single entities and not compound data types. Those master data items are often used if the value of an item is hard to read, so that it is easier to select a record by some easy-to-remember key. Another application of that class of record types would be to provide a mapping between some values.

In a master data set with property set records each record has some properties associated. The set of properties is fixed for the record set, but the values can be defined for each record. Most of the time one property is defined to be the “key” property, which makes this type an extension of the key/value type in which the value consists of more than one item. The data type of a property can be a simple type such as a number or a string, or it can be a reference to a record in another master data set. An example for this type of master data record set would be the customer database.

When editing master data records, it is obvious that typical database constraints have to be enforced. For example it must not be allowed to delete a record that is referenced from another record. Most of the time this would be the task of the underlying database

management system. Sometimes, however, it may not be possible to enforce such constraints on the database layer, and in that case the application is responsible for that.

Other parts of an application often require the selection of a record from one master data record set. This operation should be supported by the toolkit, since this is very common. While it seems easy to provide this functionality, the quality of a record selection dialog depends on the filtering methods it offers. When, for example, the user has to select a customer from a customer database with a few hundred to thousand records, it is essential that he has some means of cutting down the selection list. Different ways of doing that are conceivable, so the toolkit has to support a general way of implementing specific record selection dialogs.

When managing master data records, a common requirement is to be able to print a list of a record set, probably filtered or sorted by some specific criteria. This functionality is described in detail in the next section which describes the report generation part. What should be supported by the toolkit is to allow easy creation of a list for some master data set.

As mentioned in section 3.1.1 it should be possible to easily integrate support for another class of master data records if some application should require it.

3.1.4 Report generation

A toolkit for that application domain also has to include support for generating reports. When examining different reports, they can be categorized into the following classes:

- Simple lists
- Comparison reports

A simple list is the easiest, but also the most often needed form of a report. It shows the record set in form of a table, where each line represents a record and the properties of the records are arranged in columns.

Especially in the business domain it is interesting to compare different reports to analyze various effects. Most often that would be comparing reports for different periods of time. Even though it would be possible to compare two reports by hand, it is often desirable to have the comparison done on the report itself. The toolkit should support this kind of reports.

Common to all classes of reports are the following requirements:

- Filtering
- Sorting
- Grouping

Often a user wants to reduce the report to interesting parts of the whole data set, which is done by filtering. Sometimes it is also desirable, especially for long reports, to specify the sorting so that reports are easier to interpret. Another often needed feature in the business domain is the grouping of report data by some criteria to see the cumulative results of different items.

3.1.5 Data export

Many applications in this domain require exporting data to other applications. These other applications are often legacy applications that have their own data format. Since data formats are very different between various legacy applications, a general solution is not possible here.

However, there are some commonalities which allow the creation of some basic operations and design patterns that can be reused to create specific applications.

Another use case for data export is the usage of application data in standard office applications. For example, a common requirement is to use addresses in the customer database for creating letters with a word processor. Support for creating such functionality should also be included in the toolkit.

3.1.6 Other requirements

Some other functions required in many applications, as mentioned in section 2.3, are:

- Program control
- Application settings management
- Utility classes and functions

If an application is more complex than a single dialog it has to include some form of a program control interface, such as a main menu or something similar. Since the main element of an application in this domain is not a document like in most office applications, the typical user interface found in that applications (document area with main menu and toolbar) is not suitable here. The main element of business applications is the business process which they are built for, so the user interface should provide some easy way to find the functions a user needs.

Even if an application is tailored to a company's business processes, every application has some aspects that should not be fixed by design but instead can be modified while using the application. The user should be able to change them if he needs. The toolkit should provide a simple way to manage those application settings. It also should support easy creation of settings dialogs for the developer, so that he does not have to put much effort on that.

Every application also uses some low-level utility classes and functions. These are special candidates for reuse since they often have a simple and well defined functionality.

Most of the time they are also not specific to any application. The toolkit should also contain any such classes or functions needed in developing applications for the domain, and should be extensible to include other classes later on.

3.2 Non-Functional Requirements

Typical non-functional requirements of the applications in that application domain include:

- Easy to use interface
- Fast and comfortable data entry
- Low operation costs
- Low development costs
- Interfaces to other software
- Typically no complex architectures
- Tailored to the needs of the organization

One of the basic requirements of today's computer applications is an easy to use interface. This is especially true for applications in this domain, since users in small businesses are often not very computer literate, and thus need special support by the application to ensure flawless operation. In large companies, where a user typically has one (or some few) primary application(s) on which he works all day, it is easier for the user to get used with the peculiarities of the applications and avoid making mistakes. In small businesses, using the application (and often even using the computer) is not the main task of the user, so there must be paid special attention to this fact.

With today's graphical user interfaces usage of a new application is relatively simple if someone is used to the basic concepts of a GUI. By using the mouse someone can easily reach most of the functions needed for some task. While this is important for beginners, it is also very time consuming. After a while most users will be faster by using keyboard shortcuts and similar functions to speed up operation. Since time is a primary concern in today's business, a computer application should favor an efficient usage. This may sound contradictory to the previously mentioned requirement for an easy to use interface, but in fact it is not. An application should provide an easy to use interface for beginners, but should allow faster modes of operation for experienced users. While this should be a requirement for all computer applications, it is of special concern in the field of small business applications.

Since most small businesses often do not have dedicated computer maintenance personnel, another requirement for an application in this field is that it does not require extensive maintenance. This includes the underlying technologies such as the database system and the application environment, as well as the application itself.

Care should be taken to use technologies which require low maintenance effort. The application itself should be self-maintained, so that the user is not bothered with maintenance tasks. The used database management system, InterBase, fulfills these requirements.

The term small businesses implies also a small budget of those companies. This makes the development costs of an application a primary concern in this area. By focusing on a reuse based development model it should be possible to achieve short development times and thus low costs, but nevertheless create applications that are tailored to a businesses' needs.

Since the applications normally cover only part of an organizations business processes, interfaces to other applications are needed. The ideal situation would be to use standard applications where they fit the needs, and use customized applications where needed, with all of them interfaced to each other so that data can be passed from one application to the other. Since most standard off-the-shelf applications define interfaces, but they are most likely specific to the application and not standardized, the customized application has to deal with that interfaces and provide the "glue" to work together.

The applications in the investigated application domain typically cover one specific part of the business process of a company. This makes the applications less complex then integrated one-for-everything business applications. This also includes requirements such as distributed operation or multi-user operation. In this field the classical client/server architecture (client application with an RDBMS back-end) is sufficient to provide the functionality needed. Moreover, complex multi-tier architectures would be contradictory to the required simplicity mentioned in this section.

The key advantage of the applications developed using this toolkit is that they are tailored to the needs of the organization. Unlike with standard off-the-shelf applications the functions and requirements can be designed in cooperation with the users of the applications. This gives the user all functions that he needs for this business, but omits functionality that is not needed and may be distracting to the user.

3.3 The development environment

For this thesis the chosen development environment is Borland C++ Builder. It contains a component based class library and an application framework, which is called VCL, Visual Class Library. The toolkit developed should be based on that class library.

The VCL is an abstraction of the Microsoft Windows API. But, unlike Microsoft's MFC, it is not tightly coupled to the native C interface, but instead provides more abstractions that makes it easier to use and speeds up application development.

The key element in the VCL is the component. Components are classes that define a public interface. By subtle extensions to the C++ language, these components can

be used within the IDE similar to ActiveX controls in Visual Basic, for example. A component publishes properties, methods and events. Properties can be changed by the IDE's object inspector, and map to class variables or set/get function calls. An application that uses a component can define an event handler for a particular event and execute some custom code when the event fires.

The development environment includes many standard controls as components. Additionally, a huge amount of third party components are available, from many different vendors. Choosing an available component has to be done carefully, because if the source code is not available, maintenance and for example switching to a new version of the development environment raises problems¹.

Today most data-intensive applications use a relational database as its back-end. While SQL is an ANSI standard and thus should be portable across different vendors, practical usage shows that this is not true apart from very simple statements. The chosen database for this master's thesis is InterBase, which is a RDBMS with a long history, owned by Borland but now also developed by the community as an open source product. Whenever possible, however, the toolkit should not be bound to any specific database vendor. This is also supported by the VCL, which provides an abstraction for database access to different back-end database systems.

The database access is realized by specific data access components for each back-end. The VCL provides a common interface to those database layers and thus allows easy creation of portable solutions. From the application point of view the primary access component is a TDataSet descendant. This component represents a dataset with rows and columns, for example the result of a SQL query. There are specialized versions of the standard GUI controls (Edit field, Listbox, etc) which provide operations directly on a database. These components access a TDataSet via a TDataSource component.

The VCL provides a framework for development of GUI applications, on which this toolkit should be based. It also provides methods for designing reusable components, which should be used as far as possible. C++ Builder also provides the C++ standard template library (STL).

Since the methods and techniques used by the VCL are comparable to those used by most other application frameworks such as Java Swing or QT it should be possible to port the toolkit to another application framework without too much effort. On the other hand, this is not of primary concern for this work.

3.4 Reuse levels

The toolkit created in this thesis provides reusable elements on different levels. The following levels can be distinguished:

¹...which was a painful experience for the author.

- Classes
- Forms²
- Frames³
- Reports⁴
- Components (more specific: *VCL* components)
- Design patterns

The basic reuse level, available in all object-oriented languages in the same form is the reuse of classes. In the special case of the VCL toolkit class reuse is of course also possible, and moreover other reuse levels (forms, frames, reports and components) are also some special kind of class reuse. Classes can be reused by using an instance of a class in an application, or by deriving a class from a base class and using that derived class (which then can be customized) in an application.

Forms in the VCL are special kinds of classes, which are derived from the base class `TForm`. A form includes the definition of the form with all its properties and components (the resource) and all code that belongs to that form. The code is represented by class methods. Event-handlers connected to the form or to components on the form are also class methods. Forms can also be inherited, which makes reuse of forms easy by providing a general version of the form which is derived and modified for each application.

Some sort of a special kind of forms are *Frames*, which can contain controls like normal forms but on the other hand can be put on another form like a normal control. With frames it is very easy to reuse an abstraction where a few controls are working close together.

There are many report generator tools available for Delphi/C++ Builder. One of the first, and the one included in every Delphi and C++ Builder package is QuickReport⁵. This is also the tool used in this toolkit. When creating a report with QuickReport, the report component is placed on a form. That report component handles all aspects of report generation. So, another reuse level, *Reports*, is used, which is a QuickReport component on a form. This way it supports all reuse mechanisms described previously.

The VCL is built around components. A component is also just a special kind of class that is derived from the VCL class `TComponent`. Components are stored in component libraries, which can be included in a project. There are two different sorts of components: visual and non-visual components. A visual component is some kind of user-interface widget, that provides some functionality visible to the user of the final

²A form in the VCL is a dialog with its associated resources and code.

³A frame is a special kind of form that can be placed in another form.

⁴In this toolkit QuickReport, a report generator for Delphi and C++ Builder is used for creating reports. A report in this case is a form with a report component on it.

⁵<http://www.qusoft.com/>

application. A non-visual component provides some functionality which is not directly visible to the user. By using non-visual components operations can be encapsulated within a component, configured at design-time with the IDE's object inspector and used at runtime like a normal object.

The highest level of reuse in this context would be to abstract the cooperation of some classes and to document that. This is exactly what design patterns are. For this thesis abstractions of cooperating classes in the SBA domain should be found and documented.

There are of course other levels of reuse one can imagine. One classic example when it comes to C++ are templates. Since C++ Builder is a standard ANSI C++ compiler, it also supports templates. On the other hand the VCL does not use templates (because it is written in Object Pascal which does not support templates), so the whole design of the framework is not based on templates. That is why templates are *currently* not used in the toolkit — simply because for the design of the current requirements templates are not necessary. If, in the future, one reusable component is best designed as a template, it can be added without problems.

Chapter 4

Toolkit Architecture

This chapter describes the overall architecture of the toolkit. Its position in the development environment is outlined and usage scenarios for the toolkit are discussed. Then various support tools for development with the toolkit are described.

4.1 Framework

The toolkit itself is based on the VCL class library, which provides a framework for building GUI applications. The development environment, Borland C++ Builder, includes a GUI builder that supports the VCL classes to allow visual development of the user interface. In this environment (VCL and the IDE) there is also some support for non-visual components. The VCL provides standard components such as user interface objects, database access objects and other general purpose objects.

The toolkit is embedded into this framework. It provides enhanced components (visual and non-visual) that are specific for the application domain. It also provides reusable building blocks that implement larger functional parts which can be reused in an application, such as forms, frames or design patterns.

4.2 Application development

When creating software there are several steps involved. While there are many different process models for that (refer to [Sam97] for an overview and [Oes98] for a business-oriented example), they include (of course simplified) the following steps:

- Analysis
- Design
- Implementation
- Test

- Maintenance

The first step when creating an application is to analyze the requirements. Based on the requirements a design is made that specifies the functionality of the final application. That design is implemented using some set of tools. The implementation is tested to verify if it fulfills the requirements. After the software was deployed and is in use, it has to be maintained to adjust the application to changes in the requirements and correct bugs not found before.

For an efficient reuse process nearly all of these steps have to be adapted [Sam97]. If reuse is just applied in the implementation phase, the benefits will be poor because design decisions could have been made that prevent the usage of an existing component. So when designing an application part, the designer has to take into account what components are already available, and, if possible, base the design around these components.

Process step	Adaption
Analysis	Look for business processes that are similar to already designed patterns. While reuse does not have a direct effect on the analysis, it can help greatly by providing examples for processes in the domain.
Design	Base the design around already existing components where possible. Try to split up more complex areas so that at least parts of them can be designed with reused components. Use design patterns where applicable. When designing new components, try to create a design that can be reused afterwards.
Implementation	Use the (reused) components according to the design. For newly designed parts, try to split them up into a generic (and thus reusable) and a specific part.
Test	Use the test routines and test results from the reused parts to optimize testing. It is also possible to reuse test procedures or code.
Maintenance	Benefit from the fact that if some bug is fixed in a reused part for one application, the correction spreads to other applications. On the other side, be aware that if something is changed in a reused part for one application, it may have an effect on other applications.

Table 4.1: Development process adaptations for reuse based software development

With such a design the implementation is reduced (in the ideal case) to creating “glue” code between existing application building blocks. This also reduces the test effort, be-

cause the reused components are already tested. Table 4.1 shows the impact of a reuse-based software development model on each of the steps mentioned before.

4.3 Rapid application development

The usual RAD (rapid application development) process of implementing an application (after the design is done) with C++ Builder can be roughly described with the following steps:

1. Add a new module (which can be a form¹ or a datamodule²) to the application.
2. Place visual (for forms) and non-visual (for forms and datamodules) components on the new module, according to the specified functionality.
3. Set the properties of the components to customize them according to the specified functionality.
4. Implement functionality that cannot be achieved by using and customizing components by writing code for event handlers of components.

If an application was designed using the small business toolkit as mentioned in section 4.2, the implementation process is changed:

1. Add a new module to the application, or, where possible, create a module that inherits from a module of the toolkit.
2. Adjust the functionality of the inherited module and eventually add additional components, where possible higher-level components of the toolkit.
3. Where necessary, add additional functionality by writing code and using standard components.

Since RAD tools promote a point-and-click application development style, special care has to be taken to create a good design and to prevent quick and dirty hacks. This is the case for every application development process that uses RAD tools, but is especially important if application parts should be reused afterwards. Thus the developer has to show some extra portion of discipline to withstand the temptation of possible shortcuts.

¹A form is the C++ Builder equivalent of a dialog or window.

²A datamodule in C++ Builder is used for implementing data access, business rules and similar things. It is some kind of form that is not visible to the user and contains only non-visual components.

4.4 Development support

Application development is not only done with a compiler and a GUI builder, but also with some support tools. For different parts of the software life cycle, different tools have to be used. When focusing on reuse, some tools become especially valuable, for example version control systems. The tools used to develop this toolkit are described in the following.

4.4.1 Version control

While a version control system is beneficial for every application development process, in the current case it is an essential part of the process. The toolkit is designed to provide a base repository of components, but also to be extended in the future. So when developing an application, the toolkit itself may also be modified. The following scenarios are envisioned:

- If an error in the toolkit is found while testing an application, the error should be corrected in the toolkit, but not with a workaround in the application.
- If an application has some special requirement that may also be needed by some other applications, it can be included in the toolkit.
- If during development of an application the developer decides to change the toolkit³, it should be possible to do that.

All these changes that are made for one application may also affect other applications built with the toolkit. Especially if the interface of some part of the toolkit needs to be changed⁴, it may be desirable (because of economical or various other reasons) to not include the changes in the other applications at this point, but maybe later on.

These problems are best solved by including version control into the toolkit. Thus each application can use its own local copy of the toolkit, and changes to the toolkit can be integrated with each application when needed. On the other hand, the repository provides access to the different versions of the toolkit for reference purposes. Figure 4.1 shows this usage scenario for multiple applications using the same toolkit.

For this toolkit, the open source version control system CVS⁵ is used. In contrast to most other version control systems, it provides a non-locking style of development, which is ideal in this context. Its open source license of course also has economical benefits. For the development of this toolkit BorCVS⁶ [Ble01] is used, which is a plugin that makes CVS commands available within the C++ Builder IDE.

³which should not be common practice, but may happen occasionally.

⁴which, of course, should also be the exception and not the rule.

⁵<http://www.cvshome.org/>

⁶<http://borcvs.sourceforge.net/>

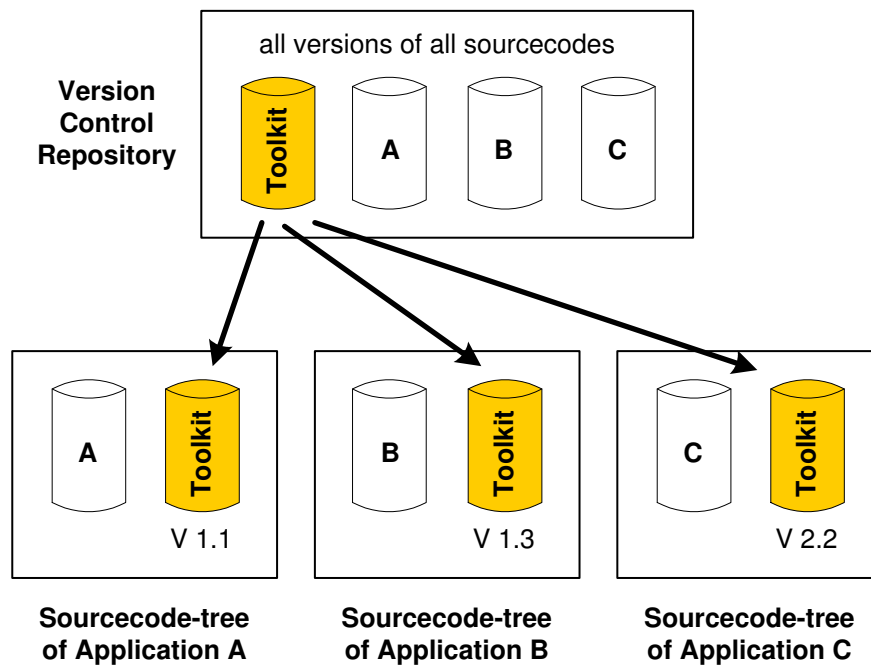


Figure 4.1: Toolkit usage scenario with a version control system

4.4.2 Source documentation

When using libraries of components, good documentation is essential. Since it has to be up-to-date and thus constantly maintained, it is advisable to create the documentation together with the code. This provides a single source for the source code and documentation, and makes maintenance easier because the developer is reminded that there is a documentation he has to update to the latest changes in the code.

There are several tools available that process comments in the program source code and produce documentation from them. One famous one is the JavaDoc utility included in the Java Development Kit. Other tools provide similar functionality for other languages, but the principles are the same.

For this work, a tool named doxygen⁷ is used. It is an open-source documentation tool that supports C++, Java, IDL and C as source languages. Documentation can be generated in various formats such as HTML, \LaTeX , RTF and so on. The tool automatically generates representations such as cross-references, class hierarchy diagrams and so on.

⁷<http://www.doxygen.org/>

4.4.3 Testing

While testing is important for every application, it is more important for a toolkit since the toolkit is used in many applications. Recent developments in software testing have produced automatic testing methods, which are also especially suitable for testing a toolkit.

For this toolkit the testing framework of the Extreme Programming methodology by Kent Beck [Bec99] is used where applicable. There is a port of the JUnit⁸ toolkit to Delphi named DUnit⁹, which was adapted for use with C++ Builder for developing test code for this toolkit.

For some parts of the toolkit this is a very useful tool. For example a non-visual component can be tested automatically with the DUnit framework. For other parts of the toolkit, however, creating the test methods would be very expensive. In this case conventional testing methods are used. There are also some automatic testing applications available for GUI testing, which may be evaluated and integrated into the toolkit in the future.

⁸<http://www.junit.org/>

⁹<http://http://dunit.sourceforge.net//>

Chapter 5

The toolkit and its components

This chapter describes the elements of the toolkit. First, a description template is developed, which is then used to describe each reusable element in the toolkit.

5.1 Introduction

Based on the preparations described in the previous chapters the toolkit is now designed. In the following each component of the toolkit is described. The description is organized by the application parts defined in section 3.1. The components included in each application part are on different reuse levels, as described in section 3.4. This chapter provides a detailed description of each component, overviews from various points of view are given later in Chapter 6.

5.2 Description template

Each element of the toolkit is described using the following template. It provides a structured method of outlining the characteristics of a reusable item. The template itself was inspired by the patterns description in [Gam94]. Section 5.2.1 provides a description of the template items and section 5.2.2 shows an example usage of the template for a hypothetical class.

5.2.1 Reusable Item

Name: Name of the item in the source code (only for classes, forms, frames, reports and components).

Purpose: Short statement about the purpose of the item.

Level: Reuse level for this item (as mentioned in section 3.4), one of: Class, Form, Frame, Report, Component or Design pattern.

Description: Description of the functionality of the item.

Parent: Parent class (only for Classes, Forms, Frames, Reports and Components).

Usage: Description of the usage of the item in an application. Not applicable for design patterns.

Customization: How can the item be customized when used in an application. Not applicable for design patterns.

Requires: Conditions under which the item can be used.

Provides: Functionality that this item provides to other reusable items in the toolkit and to the developer.

Friends: Other items of the toolkit which are used together with this item.

5.2.2 A very useful example class

Name: TFoo

Purpose: Provides some very useful methods for every application.

Level: Class

Description: TFoo contains various very useful methods, for numerical manipulations, string operations and weather forecast which are useful in every application. The methods are:

- Add(n1,n2): adds two numbers
- Concat(s1,s2): concatenates two strings
- Weather(day): weather forecast for the given day

Usage: An application has to create an object of class TFoo. Then it can use its methods.

Customization: There are no customizable parameters for TFoo.

Requires: The class requires the numerical libraries. It also depends on the weather forecasting library by XYSoft.

Provides: String concatenation functionality that is used by TBar.

Friends: TFoo is often used with TBar for weather applications.

5.3 Data entry

The data entry part contains components for building record-based data-entry dialogs. It supports simple records and compound records, which are records that contain subitems.

5.3.1 Simple record entry form

Name: TboDeSimpleRecFrm

Purpose: Provides the foundation for building data entry forms for simple records.

Level: Form

Description: This form provides the functionality for entering record data for simple records (records with no subitems). The user specifies all properties for one record and then saves the record. After that the form is ready for entering the next record (or it can be closed automatically after entering one record, depending on the application). The typical sequence of actions is shown in Figure 5.1.

Parent: TForm

Usage: After subclassing this abstract form, the fields of the record have to be put on the form using data-aware controls. The datasource for those controls is provided on the form. While that is enough for creating a data entry form, any other additional functionality may be added. For example there are methods that can be overridden to add additional initialization code or code that validates user input before the record is saved.

Customization: The form can be customized by changing its properties in the object inspector. Additionally some properties can only be set at runtime, and the virtual methods of the object can be overridden.

Requires: an editable TDataSet where the records can be stored.

Provides: The simple record entry form with entry log (5.3.2) is derived from this form.

Friends: uses a comfort keyboard handler (5.8.1) and a button panel (5.8.9).

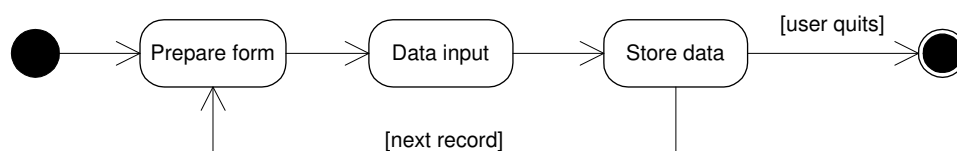


Figure 5.1: Activity diagram for the simple record entry form

5.3.2 Simple record entry form with entry log

Name: TboDeRecLogFrm

Purpose: Provides the foundation for building data entry forms for simple records. The user can modify or delete records previously entered in the current session.

Level: Form

Description: In addition to the simple record entry form (5.3.1), this form provides a list of entered records. Every time a record is saved, it is added to this list. Later the user can modify or delete a record by selecting it from the list.

Parent: Simple record entry form (5.3.1)

Usage: After subclassing this abstract form, the fields of the record have to be put on the form using data-aware controls, similar to 5.3.1. In addition a database key value object (5.11.1) has to be provided for locating the records after they are saved. The form also has an abstract method for defining the list item that shows the record in the list, which has to be overridden by a derived class.

Customization: similar to 5.3.1. Additionally the columns for the list of records should be defined.

Requires: an editable TDataSet where the records can be stored. Also requires a key value object (5.11.1) for the data set.

Provides: -

Friends: see 5.3.1

5.3.3 Compound record entry

Purpose: Provides an abstraction for creating compound record entry forms.

Level: Design pattern

Description: When editing compound records, an application has to handle the properties of the record and those of the subitems. The record properties are edited similar to the simple record entry form (5.3.1). The user just enters the desired values for the properties.

For the subitems a different edit style has to be used. Since many subitems can be associated to one record a list of subitems is provided. This list should be able to handle different types of subitems, since this is a common requirement for business applications (for example a receipt that contains receipt items and description text lines). A user should be able to add a subitem to the list (which is the most common operation), edit a subitem and delete a subitem. Sometimes it should also be possible to change the order of subitems.

A subitem in the list is represented by an object. This object is responsible for displaying the subitem in the list, and for providing methods for storing and loading its properties. For editing the subitem properties several different styles are possible: the data can be entered in fields on the form of the main record, or each subitem type can have its own edit form. This depends on the application and thus is not handled in this pattern.

The class diagram for this pattern is shown in Figure 5.2.

Requires: an implementation of the classes of the pattern by subclassing the abstract base classes for the form and for each subitem.

Provides: A collaboration style for master/detail record edit forms.

Friends: This pattern consists of the compound record entry form (5.3.4) and the abstract subitem class (5.3.5).

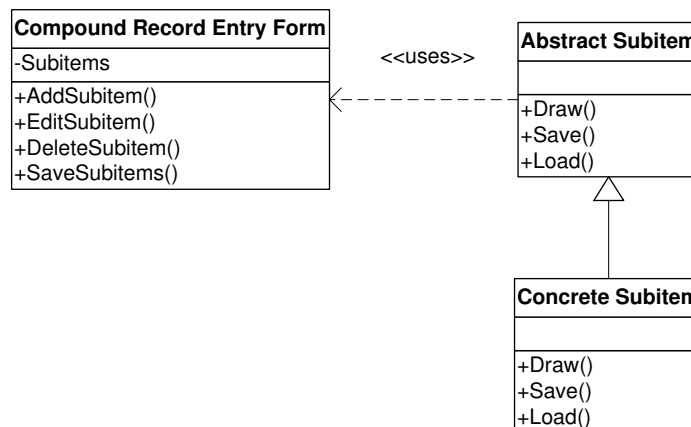


Figure 5.2: Class diagram for the compound record entry pattern

5.3.4 Compound record entry form

Name: `TboDeDetailRecFrm`

Purpose: An abstract form for creating compound record entry forms.

Level: Form

Description: Implements the form for the pattern described in 5.3.3. The form provides operations for handling the properties of the record (similar to 5.3.1) and for handling the subitems. The subitems are displayed in a `TListBox`. For each type of subitem, a specialized class is used to draw the visual representation of the subitem. This class is also responsible for storing and loading the subitem data. Since different edit styles are possible for subitems, the form provides a general framework but does not implement the user interface for editing subitems.

Parent: TForm

Usage: A subclass has to be derived from this form, and the fields of the (master) record have to be placed on the form using data-aware controls. A TDataSource for those fields is provided. For every subitem type, a subitem class has to be implemented. Then the user interface for handling the subitems has to be designed. The form provides some general methods for that like adding a subitem to the list of subitems.

Customization: The form can be customized by changing its properties and adding event handlers. There are some virtual methods for the various actions which can be overridden and extended. Part of the functionality for handling subitems (for example displaying) is handled by the subitem classes and has to be customized there.

Requires: A TDataSet for the (master) record. For each type of subitem a specialized subitem class has to be implemented.

Provides: A framework for master/detail record entry functionality.

Friends: Abstract subitem class (5.3.5), also uses a comfort keyboard handler (5.8.1).

5.3.5 Abstract subitem class

Name: TboDeDetSubBase

Purpose: Abstract class for subitems of a compound record.

Level: Class

Description: Implements the interface for subitems in the compound record entry pattern (5.3.3). A subitem has to be able to draw itself as an entry in a listbox and to store and load the subitem data from a dataset. This interface has to be implemented by concrete subitem classes. The subitem class also holds the data for a subitem until it is saved to a database, so it has to provide fields for that.

Parent: TObject

Usage: Derive a class from this abstract class. A derived class has to override the abstract methods and implement the required functionality. It also has to define attributes for the data fields of the subitem.

Customization: Beside the abstract methods other virtual methods can be overridden to extend or customize the functionality of the object.

Requires: A dataset where the data of the subitem can be stored and loaded.

Provides: An interface for subitems in the compound record entry pattern (5.3.3) and some basic functionality for storing the subitem properties in a dataset.

Friends: Uses a database key value object (5.11.1) to identify the master record to which this subitem belongs to.

5.3.6 Subitem class with text representation

Name: TboDeDetSubText

Purpose: Implementation of the abstract subitem class (5.3.5) for subitems that have a text representation.

Level: Class

Description: Implements part of the abstract interface of the abstract subitem class (5.3.5) and provides functionality for displaying a text string for the subitem. For a concrete subitem, the part of the interface that handles storing/loading the subitem data still has to be implemented in a subclass. This subclass also has to implement an abstract method that generates a string from the values of the subitem fields. This string is then used for drawing the subitem in the listbox.

Parent: Abstract subitem class (5.3.5)

Usage: Derive a class from this abstract class. The derived class has to implement the method for generating the text representation of the subitem. It also has to define attributes for the data fields of the subitem. Additionally it has to implement the functionality for storing/loading the record data.

Customization: A subclass can pass various parameters to the constructor of this class to define the overall layout of the text representation (e.g. multiple lines or not). It can also override virtual methods to customize the behavior of the class.

Requires: See 5.3.5.

Provides: In addition to the functionality provided by its base class (5.3.5), this class provides the functionality to display a string in a listbox item.

Friends: See 5.3.5.

5.4 Document printing

The document printing components provide abstractions for creating documents such as invoices or other kinds of receipts and vouchers. They support simple documents, which contain data from a single record and compound documents, which contain data from a record and its subitems.

5.4.1 Simple document printing

Name: TboVoSimpleQrpt

Purpose: Provides basic functionality for creating simple documents.

Level: Report

Description: This class is a basic Report which can be used to create a simple document (a document that contains data from a record with no subitems). The class provides support for selecting a single record in a dataset and printing a document for this record.

Parent: TForm

Usage: Derive a subclass from this form and add the report fields for the document. A dataset has to be provided to retrieve the record for the document.

Customization: The properties of the report can be modified in the object inspector. Additionally, virtual methods of the form can be overridden to customize the functionality of this methods.

Requires: a dataset for retrieving the data for the document and a SQL dataset helper (5.11.3) for filtering the dataset.

Provides: a framework for creating document reports.

Friends -

5.4.2 Compound document printing

Name: TboVoCompoundQrpt

Purpose: Provides basic functionality for creating compound documents.

Level: Report

Description: In addition to the functionality provided by the simple document printing report (5.4.1), this report provides help on implementing compound document reports. It inherits the functionality for printing a document from a single database record from its parent. This record is the master record in this case. In addition, it provides support for printing the subitems of the master record.

Parent: Simple document printing 5.4.1

Usage: Derive a subclass from this form and add the fields for the master record. Similar to its parent a dataset has to be provided to retrieve the master record. Additionally one or more datasets for the subitems are needed. For each type of subitem, a document line has to be defined that contains the fields for this subitem.

Customization: See 5.4.1.

Requires: In addition to the requirements of its base class (5.4.1) this class requires dataset(s) for the subitems.

Friends -

5.5 Master data administration

The components for master data administration include functionality for managing master data sets, both simple record sets, which contain only a few fields, and more complex record sets, which require a list and an edit form for processing. This part also includes components for selecting a record from a master data set in various ways.

5.5.1 Simple master data record set

Name: TboMdSimpleFrm

Purpose: A form for editing simple master data record sets.

Level: Form

Description: Provides a dialog for editing simple record sets. A simple record set is one that only has a few fields. The records are displayed in a grid, and can be edited in-place. The dialog provides methods for adding, editing and deleting records.

Parent: TForm

Usage: In its simplest form, the class just needs a TDataSet that contains the records to be edited and it can be used. If some aspects of the form need to be customized, a subclass can be used.

Customization: For standard usage the class can be used directly and customization is limited to modifying properties at runtime. For greater modifications or extensions a subclass of this class can be used and customized by setting properties of the form and its controls and by overriding events and virtual methods.

Requires: A dataset for the database operations.

Provides: A ready-to-use form for editing simple record sets.

Friends: Uses a button panel 5.8.9.

5.5.2 Master data edit

Purpose: Provides an abstraction for creating edit forms for managing master data sets.

Level: Design pattern

Description: When editing master data sets with more than a few fields, this is often done using a two-form approach. When the user chooses the option to edit the data set, he gets a list of all records in this data set. That list displays only some fields that gives the user an overview of the records in the dataset and allows him to identify the record he wants to modify. In this list a record can be selected.

When the user chooses to add a new record or modify an existing record, a detail form is shown which allows him to change the values of the record's fields.

This design pattern can be used to create such a solution. The toolkit provides an abstract list form and an abstract edit form. An application subclasses this abstract forms and customizes them. Figure 5.3 shows the class diagram of this pattern.

Requires: For the design pattern concrete subclasses of the abstract list and edit forms need to be implemented.

Provides: A collaboration style for list and edit forms.

Friends: This pattern consists of the master data edit list form (5.5.3) and the master data edit form (5.5.4)

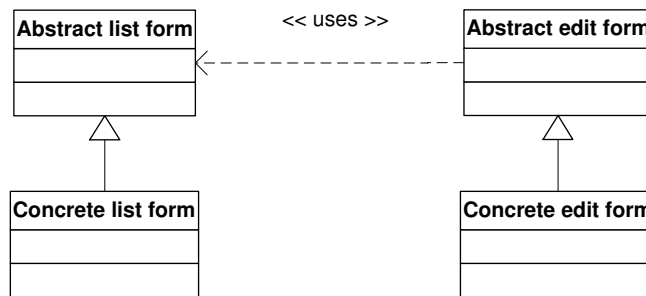


Figure 5.3: Class diagram for the master data edit pattern

5.5.3 Master data edit list form

Name: TboMdListFrm

Purpose: An abstract form for selecting a record from a master data set and executing edit operations.

Level: Form

Description: Provides a form which allows the user to select a record from a master data record set and apply the usual edit operations (add a new record, edit an existing record, delete a record) on it. An activity diagram for the actions that can be performed with this dialog is shown in Figure 5.4.

The form uses two datasets for operation, one for displaying the list and one for performing the edit operations. For editing a record a master data edit form (5.5.4) is required.

Parent: TForm

Usage: In a subclass derived from this form the abstract methods have to be overridden. They have to provide the datasets, the SQL dataset helper objects (5.11.3) and the edit form. This edit form has to be derived from the abstract master data edit form (5.5.4) and provide a user interface for editing a record.

Customization: The properties and events of the form can be modified in the object inspector. In addition to the abstract methods that have to be implemented in a derived class, the virtual methods can be overridden to modify or extend their functionality.

Requires: One SQL dataset helper object (5.11.3) for the list dataset and one for the edit dataset. Additionally an edit form is required.

Provides: A framework for editing master data sets.

Friends Is used together with the master data edit form (5.5.4) to implement the master data edit pattern (5.5.2). The form uses a button panel (5.8.9), a comfort keyboard handler (5.8.1), the database key value object (5.11.1) and SQL dataset helper objects (5.11.3).

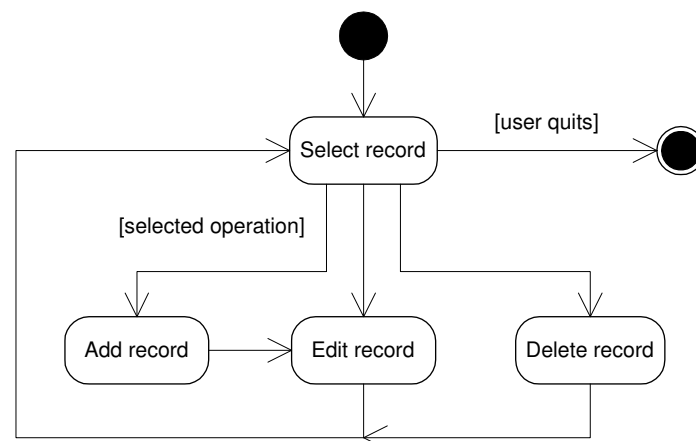


Figure 5.4: Action diagram showing the actions in the master data edit list form

5.5.4 Master data edit form

Name: `TboMdEditFrm`

Purpose: An abstract form for editing a master data record.

Level: Form

Description: Provides abstract functionality for editing the contents of a record. After changing the fields of the record the user can select the “OK” button, which saves the changes made, or the “Cancel” button, which reverts the record to its original state.

Parent: `TForm`

Usage: For each type of master data record a form has to be derived from this form. Data-aware controls have to be added for the fields of the record. They have to be connected to the datasource which is provided by the form. Before the form can be used, a dataset that contains the record to be edited has to be assigned to the dataset property of the form.

Customization: In a derived form the properties of the form can be modified. There are also some virtual methods that can be overridden, for example the `Validate()` method which can be used to check if the record data is valid before it is stored.

Requires: A dataset that contains the record to be edited.

Friends: Is used together with the master data edit list form (5.5.3) to implement the master data edit pattern (5.5.2). The form uses a button panel (5.8.9), and a comfort keyboard handler (5.8.1).

5.5.5 Auto-generated master data edit form

Name: `TboMdEditAutoFrm`

Purpose: A master data edit form that automatically generates the edit controls based on the record to be edited.

Level: Form

Description: This form is derived from the master data edit form and thus can be used together with the master data edit list form (5.5.3) to implement the master data edit pattern (5.5.2). In addition to the functionality provided by its base form it automatically creates edit controls for all fields in the dataset that should be edited. It contains an algorithm to “guess” the properties for the edit controls and can be used for master data sets with no special requirements.

This component is especially helpful for creating prototypes. It can be used in a first version of an application to provide a sample implementation, and later be replaced by a custom edit form derived from the master data edit form (5.5.4). If

all requirements are fulfilled by this component, it can also be used by the final application.

Parent: Master data edit form ([5.5.4](#))

Usage: The form can be used in the same way as its parent. When the form is displayed, the edit controls are generated automatically.

Customization: Since the form automatically creates all controls, customization is currently limited. The properties of the form and the controls can be modified at runtime, though.

Requires: A dataset that contains the record to be edited. The fields have to be correctly defined in this dataset since the form derives various settings for the edit controls from the field definitions.

Friends: See [5.5.4](#).

5.5.6 Record selection form

Name: `TboMdSelBaseFrm`

Purpose: Provides abstractions for selecting a record from a record set.

Level: Form

Description: Master data sets are often referred to when editing other records. Then one specific record has to be selected from a master data record set. This abstract form provides the functionality for easy creation of such selection forms. It does not implement the selection process itself, because subclasses can implement different methods of selecting a record, depending on the requirements of the application and the complexity of the master data record set.

When using this abstract form as a starting point, different methods of selecting a record can be implemented that share a common interface. That makes it easier to change the record selection method later on.

The basic interface this form provides is a function that shows the selection form and returns the key value (using a database key value object, see [5.11.1](#)) of the record the user selected.

Parent: `TForm`

Usage: Use a subclass derived from this class (e.g. [5.5.7](#) or similar) or derive your own subclass and implement a record selection user interface.

Customization: A subclass should add controls to implement a record selection user interface. It can also override virtual methods to perform actions for different events.

Requires: A dataset that contains the records. Some of the implementations may also require a SQL dataset helper object ([5.11.3](#)) for implementing some functionality.

Provides: An interface for record selection forms.

Friends: Uses a button panel (5.8.9), a database key value object (5.11.1) and a SQL dataset helper object (5.11.3).

5.5.7 Record selection with simple list

Name: TboMdSelListFrm

Purpose: Implements a record selection form (5.5.6) where the user can select the record from a list.

Level: Form

Description: This form implements the record selection interface. The user can select a record by choosing it from a list. The list shows all records of a data set using a DBGrid.

Parent: Record selection form (5.5.6)

Usage: For the simplest usage one has to provide a dataset and can create an instance of this class. To extend the functionality a subclass can be created that implements further actions to help the user to find a record.

Customization: When using this class directly there is not much to customize. If desired the properties of the form or controls can be set at runtime. In a subclass properties can be modified in the object inspector, and virtual methods can be overridden to extend the functionality.

Requires: A dataset with the records to select from.

Provides: An implementation of the record selection form interface (5.5.6), and a base for other record selection forms.

Friends: See 5.5.6.

5.5.8 Record selection with list and one filter

Name: TboMdSelFilter1Frm

Purpose: Implements a record selection form (5.5.6) where the user can select the record from a list and the list can be filtered by one filter criteria.

Level: Form

Description: This form is an implementation of the record selection interface. The user can select a record from a list, and the list can be filtered. The filter value is looked up in another dataset. This allows the user to narrow the list of displayed records and find the desired record more easily.

Parent: Record selection with simple list (5.5.7)

Usage: The class can be used directly by creating an instance. Several properties such as the dataset for the record, the dataset for the lookup values and the field names for the lookup procedure have to be defined. It is also possible to derive a subclass from this class and implement additional functionality.

Customization: When using the class directly, some properties can be specified mostly to set up the lookup procedure. When deriving a subclass properties can be changed in the object inspector and virtual methods can be overridden.

Requires: Similar to the record selection form (5.5.6) this class requires a dataset with the records to select from. Additionally a dataset for the lookup values has to be provided, and a SQL dataset helper object (5.11.3) for the record dataset.

Provides: An implementation of the record selection form interface (5.5.6), and a base for other record selection forms.

Friends: See 5.5.6.

5.5.9 Record selection with list and two filters

Name: TboMdSelFilter2Frm

Purpose: Implements a record selection form (5.5.6) similar to the one described in 5.5.8 but with two filters.

Level: Form

Description: This form is very similar to the record selection form with one filter (5.5.8) but provides a second filter for narrowing the displayed record set.

Parent: Record selection form with one filter (5.5.8)

Usage: Similar to 5.5.8.

Customization: Similar to 5.5.8.

Requires: Similar to 5.5.8, but additionally a second dataset for the lookup values for the second filter is required.

Provides: An implementation of the record selection form interface (5.5.6), and a base for other record selection forms.

Friends: See 5.5.8

5.5.10 Record selection with list and text search

Name: TboMdSelSearchFrm

Purpose: Implements a record selection form (5.5.6) where the user can select the record from a list and the list can be reduced to records that contain some specific text.

Level: Form

Description: This form is an implementation of the record selection interface. The user can select a record from a list. This list can be filtered to show only the records that contain some text in one of their fields. This allows the user to narrow the list of displayed records and find the desired record more easily.

Parent: Record selection with simple list (5.5.7)

Usage: The class can be used directly by creating an instance and providing a dataset for the records to select from. It is also possible to derive a subclass from this class and implement additional functionality.

Customization: When using the class directly there is not much to customize. If desired some properties of the form can be changed at runtime. When deriving a subclass properties can be changed in the object inspector and virtual methods can be overridden.

Requires: Similar to the record selection form (5.5.6) this class requires a dataset with the records to select from, and a SQL dataset helper object (5.11.3) for that dataset.

Provides: An implementation of the record selection form interface (5.5.6), and a base for other record selection forms.

Friends: See 5.5.6.

5.5.11 Record selection with incremental search

Name: TboMdSelIncSearchFrm

Purpose: Implements a record selection form (5.5.6) where the user can find the record using incremental search on some field of the record.

Level: Form

Description: This form is an implementation of the record selection interface. The form contains a list of search words, one for each record. These search words are retrieved from one field of the record. To find a record the user enters the search word, and the form constantly refines the search to display the record that matches the currently entered search word most exactly.

Parent: Record selection form (5.5.6)

Usage: To use this form a subclass has to be created. On this sub-classed form data sensitive controls should be placed to display information about the currently selected record. The subclass also has to specify the field of the dataset that contains the search words.

Customization: The derived form can be customized by setting properties in the object inspector and by overriding virtual methods to extend the functionality for some actions.

Requires: A dataset with the records to select from. One field of the dataset has to be specified to contain the search words.

Friends: See [5.5.6](#).

5.6 Report generation

This part of the toolkit contains components for creating reports from a dataset, both manually by specifying the data on the report and automatically by creating a report with the information found in the dataset. It also includes components for creating dialogs for specifying the properties of a report.

5.6.1 Basic report template

Name: `TboRepBaseQrpt`

Purpose: A basic template for creating reports

Level: Report

Description: This class provides a basic template for creating reports. It is based on the QuickReport report generation tool and contains a header and a footer line with some system information (page number, report title, etc). It also includes some support for handling the dataset and a SQL dataset helper object ([5.11.3](#)) and for cooperation with a report properties form ([5.6.4](#)).

Parent: `TForm`

Usage: A subclass has to be derived from this class and define the fields for the report.

Customization: In the subclass the properties of the report can be modified, and some virtual methods can be overridden to add additional functionality at various points.

Requires: A dataset that contains the data for the report and a SQL dataset helper object ([5.11.3](#)) for that dataset. Also requires a report properties form for the report.

Provides: a basic template for creating reports.

Friends: Uses a SQL dataset helper object ([5.11.3](#)).

5.6.2 Auto-generated list report

Name: TboRepAutoListQrpt

Purpose: An automatically generated list report for a dataset.

Level: Report

Description: Based on the basic report template ([5.6.1](#)) this class provides the functionality for automatically generating a list from the records of a dataset. The report is created based on the field definitions of the dataset. It contains columns for all visible fields in the dataset.

Parent: Basic report template ([5.6.1](#))

Usage: To use this report a dataset has to be assigned to the class. When the report is created, the columns for the fields are generated. The class contains an algorithm to create the report controls for the fields of the dataset automatically, based on the settings of the field components.

Customization: A few properties can be set to customize the look of the report. If desired, a subclass can be derived from this class and further customized by setting properties and overriding virtual methods.

Requires: See [5.6.1](#). For the automatic generation of the report to work correctly the properties of the fields in the dataset have to be defined.

Provides: A method for automatically creating a list report for a dataset.

Friends: See [5.6.1](#).

5.6.3 Report properties (sort, filter)

Purpose: Provides an abstraction for creating forms for adjustment of reports (sort order, filtering).

Level: Design pattern

Description: When creating reports, users want to customize the report by specifying the order in which the records are printed or by applying filters that restrict the records printed. Usually those customization forms have to be created from scratch for each report, because each report has different things to customize.

This pattern allows easy creation of such adjustment forms. A form is used as a container for the different properties to adjust. For each type of property a frame has to be developed, and that frame can be easily added to the adjustment form. Some commonly used frames are included in the toolkit. The frame is responsible

for allowing the user to select options for the report, and for applying the selected options to the dataset which is used for creating the report.

Figure 5.5 shows a class diagram for this pattern.

Requires: Property value selectors (see 5.6.6) for each type of customizable property of the report. Also requires a dataset and a SQL dataset helper object (5.11.3) for applying the selected properties to the report.

Provides: A collaboration style for a property container and the property value selectors.

Friends: Uses a report properties form (5.6.4) and property value selectors (5.6.6).

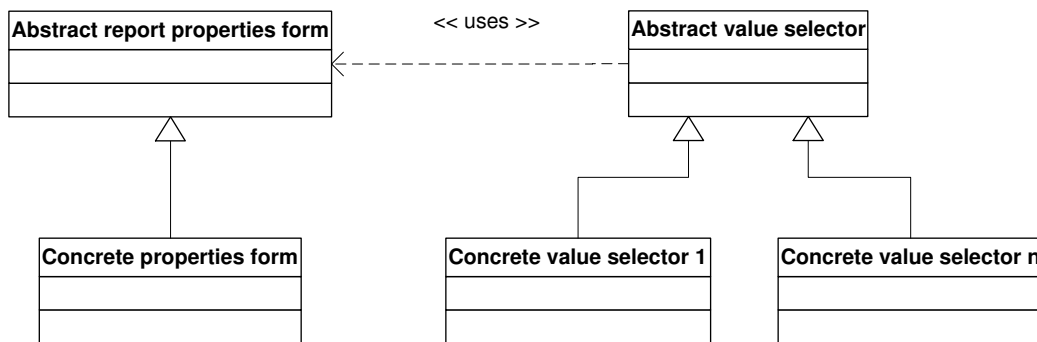


Figure 5.5: Class diagram for the report properties pattern

5.6.4 Report properties form

Name: TboRepPropFrm

Purpose: An abstract base form for creating report properties forms according to the report properties pattern (5.6.3).

Level: Form

Description: This form is used in the report properties pattern (5.6.3) as a container for the property value selectors (5.6.6). It also contains the user interface for handling a report (printing, previewing and printer setup).

Parent: TForm

Usage: The form has to be customized for the specific report tool by creating a subclass. Then an instance has to be created for each report and the corresponding property value selectors can be added.

Customization: To customize the behavior of the form (beside from adding property value selectors) a subclass has to be derived. There properties can be changed and virtual methods can be overridden to modify the form.

Requires: This class requires a SQL dataset helper object (5.11.3) for the dataset of the report. Also a specialized property value selector (5.6.6) is required for each type of property to be defined for the report.

Provides: A report properties form that can be customized for a specific report tool.

Friends: Uses a button panel (5.8.9) and a SQL dataset helper object (5.11.3).

5.6.5 Report properties form for QuickReport

Name: TboRepPropQuickrepFrm

Purpose: Provides an implementation of the abstract report properties form (5.6.4) class for QuickReport, a report generator for Borland Delphi/C++ Builder.

Level: Form

Description: This form implements the abstract methods of its base class for the Quick-Report tool.

Parent: Report properties form (5.6.4)

Usage: Use like a report properties form (5.6.4) together with a QuickReport.

Customization: If this form should be customized a subclass has to be derived. In this subclass properties can be changed and virtual methods can be overridden.

Requires: See 5.6.4. Additionally a QuickReport object is required.

Provides: See 5.6.4.

Friends: See 5.6.4.

5.6.6 Property value selector

Name: TboRepPropFrameBase

Purpose: An abstract frame for defining the value of a property (e.g. sort order, filter, etc).

Level: Frame

Description: Subclasses of this abstract class are used by the report properties form (5.6.4) for implementing the report properties pattern (5.6.3).

This class defines the interface for handling a report property. Such a property is defined by the user and thus the user input has to be validated. When the report is executed, the user selection has to be applied to the dataset that is used to generate the report.

Parent: TFrame

Usage: For each type of report property, a subclass has to be derived from this class. Those subclasses have to provide an implementation of the interface defined here. This is done by implementing the abstract methods.

Customization: A derived subclass can override the virtual methods of this class.

Requires: -

Provides: An interface for the frames in the report properties pattern (5.6.3)

Friends: Uses a SQL dataset helper object (5.11.3) for applying the user selection to the dataset.

5.6.7 Property value selector with enable/disable functionality

Name: TboRepPropFrameEnabledBase

Purpose: Refines the property value selector interface (5.6.6) for properties that can be enabled or disabled.

Level: Frame

Description: This frame adds the functionality for enabling/disabling the property to the property value selector (5.6.6). It contains a checkbox which allows the user to include a property selection in the report or not.

Parent: Property value selector (5.6.6)

Usage: Usage is similar to its parent (5.6.6).

Customization: See 5.6.6

Requires: See 5.6.6

Provides: See 5.6.6

Friends: See 5.6.6

5.6.8 Property value selector for a date range

Name: TboRepPropFrameDateRange

Purpose: Implements the property value selector interface (5.6.6) for selecting a date range.

Level: Frame

Description: This frame implements the interface defined in 5.6.6 and allows the user to enter a date range for the report. The date range is specified by a start date and an end date. These dates can be entered directly into the corresponding edit controls. The frame also provides a quick selection combo box that allows selection of a date range from a list of predefined ranges (months, years).

Parent: Property value selector with enable/disable (5.6.7)

Usage: The component is used by creating an instance of the frame and adding it to a report properties form (5.6.4).

Customization: Simple customizations can be done by supplying parameters to the constructor (e.g. the specifications for the quick selection combo box). For further customization a subclass can be derived.

Requires: -

Provides: an implementation of the property value selector interface (5.6.6).

Friends: See 5.6.6

5.6.9 Property value selector for a database value

Name: TboRepPropFrameDBLookup

Purpose: Implement the property value selector interface (5.6.6) for selecting a value from a dataset.

Level: Frame

Description: This frame implements the interface defined in 5.6.6 and allows the user to select a record from a dataset. Some field of that record is later used to filter the report dataset so that only records that match that field are included in the report.

Parent: Property value selector with enable/disable (5.6.7)

Usage: Is used by creating an instance of the frame and adding it to a report properties form (5.6.4). The properties for that database lookup are specified in the constructor of the frame.

Customization: The specifications for the database lookup are set in the constructor. For further customization a subclass can be derived.

Requires: a dataset with the lookup values.

Provides: an implementation of the property value selector interface (5.6.6).

Friends: See 5.6.6

5.6.10 Property value selector for the sort order

Name: TboRepPropFrameSortOrder

Purpose: Implements the property value selector interface (5.6.6) for selecting the sort order.

Level: Frame

Description: This frame implements the interface defined in 5.6.6 and allows the user to select the order in which the data records of a reports are sorted.

Parent: Property value selector (5.6.6)

Usage: The component is used by creating an instance of the frame and adding it to a report properties form (5.6.4). The sort orders which are allowed for the report are specified in the constructor.

Customization: For this frame there is not much to customize. The sort orders are defined in the constructor. For further customization a subclass can be derived.

Requires: -

Provides: An implementation of the property value selector interface (5.6.6).

Friends: See 5.6.6

5.7 Data export

The data export part contains components that help in creating functionality to export data to other applications such as legacy applications which require text files or word processing applications.

5.7.1 Data export pattern

Purpose: Provides an abstraction for creating data export interfaces.

Level: Design pattern

Description: For interfacing to legacy application data export interfaces are needed. This pattern describes how such an interface can be created.

The data records are exported from a dataset. A data set iterator (5.7.2) is used to process every record in a dataset. Since the legacy applications often require a special file format, another class, for example the fixed record size data file writer (5.7.3), is used to write this record format.

Figure 5.6 shows the class diagram for this pattern.

Requires: A custom implementation of the data set iterator that uses a specific record writer class to create the data file.

Friends: A record writer class for fixed size records is provided in the toolkit (5.7.3)

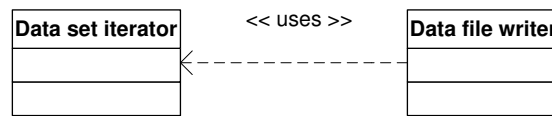


Figure 5.6: Class diagram for the data export pattern

5.7.2 Data set iterator

Name: TboIntDataSetIterator

Purpose: Process every record in a dataset

Level: Component

Description: For the data export pattern (5.7.1) a class that processes every record of a dataset is needed. This component provides an implementation of such a class.

Parent: TComponent

Usage: First a dataset has to be assigned. Then a method can be called to process every record in a dataset. The processing is done by calling a virtual method that can be overridden in a subclass or by calling an event handler.

Customization: A derived class can override virtual methods to modify the functionality of the class.

Requires: A dataset that contains the record to be processed.

Provides: A iterator for the data export pattern (5.7.1).

Friends: -

5.7.3 Fixed record size data file writer

Name: TboIntWriteFixedRecord

Purpose: Provides methods for writing data to a text file with a fixed record size.

Level: Class

Description: Legacy applications typically have special file formats for data import files. One often found format are text files that have a fixed record size, which is in this case the line length. This component supports the creation of such text files. It provides methods for writing various data types (string, integer, decimal number, date) to such a file.

Parent: TObject

Usage: Create an instance of this class, provide a filename to the constructor and start writing to the file. On each end of one record (or line), call the appropriate method to write the end-of-line marker. It automatically checks if the record has the pre-defined length.

Customization: In a subclass the methods for writing some data type can be overwritten to change the format of the exported data.

Requires: -

Provides: An implementation of a export file writer for the data export pattern (5.7.1).

Friends: -

5.7.4 Data export to Word

Name: TboIntExRecWord

Purpose: Provides the functionality for exporting data from a single record to Microsoft Word.

Level: Component

Description: Small business applications often include data that is also needed in other applications. One example is the export of the data of one record to a word processing application. This is often needed for example for customer records, where the address for a letter can be taken from the customer database. This component provides an implementation of this functionality for Microsoft Word.

Parent: TComponent

Usage: The component exports the data of a record to word by setting bookmarks that are named after the database fields to the values of the record. Everything needed is the dataset that contains the record.

Customization: A derived class can override the virtual methods to customize the way the data for a record is processed.

Requires: A dataset with the record to be exported. Also requires that a Word document has been prepared with the appropriate bookmarks.

Provides: A simple way of exporting data to a Word document.

Friends: Uses a database key value object (5.11.1) for locating the record.

5.8 Standard components

This part contains components that extend the functionality of the user interface widgets provided by C++ Builder and the VCL. These components can be used in any application (in any domain), but are tailored to the needs of the SBA domain. Also included are components for handling the permanent storage of arbitrary properties in a general way.

5.8.1 Comfort keyboard handler

Name: TboKeyHandler

Purpose: Makes dialog-handling with a keyboard more comfortable and provides a method of implementing keyboard shortcuts.

Level: Component

Description: Provides the following functions for creation of efficient¹ data entry dialogs:

- Maps the *plus* and *minus* key of the numerical keypad to the *tab* and *shift-tab* functionality (next/previous control) for easy input of numerical data.
- Provides events which are called when an operator button on the numeric keypad is pressed (plus, minus, multiply, divide).
- Provides events for the function keys (pressing a function key is mapped to an event for easy handling in an application).
- Provides customizable mapping of function keys to buttons (pressing a function key clicks the button).

Parent: TComponent

Usage: To use the component it has to be placed on a form. The plus/minus mapping can be enabled or disabled by setting the corresponding properties. The other functionality is used by implementing event handlers for the corresponding events.

Customization: is done by setting properties. For further customization a subclass can be derived that overrides virtual methods.

Requires: a form (derived from TCustomForm) where the component is placed. Captures the keyboard events of that form at runtime.

Provides: transparent handling of keyboard functionality.

Friends: -

¹“efficient” in this context means “fast input via keyboard”

5.8.2 Abstract comfort edit field

Name: `TboComfortEdit`

Purpose: An abstract class for implementing “comfortable” edit fields.

Level: Component

Description: Provides an abstract base class for implementing edit fields that provide the following functionality:

- Validity checking of data before the user leaves the control.
- Handling of empty data. When the user does not enter data, the control can either accept the empty value, provide a default value instead or show an error message and do not allow the user to leave the control until some data was entered.
- Custom display format for formatting the data.
- Clear the control when the user enters it.

Parent: `TCustomEdit`

Usage: Since this is an abstract class it cannot be used directly. Create a subclass instead and provide an implementation for the abstract methods.

Customization: is done in the subclass by overriding virtual methods and setting properties.

Requires: -

Provides: a generic framework for creating comfortable edit fields.

Friends: -

5.8.3 Comfort numeric edit field

Name: `TboNumEdit`

Purpose: An edit field for numbers which is implemented based on the comfort edit field ([5.8.2](#)).

Level: Component

Description: In addition to the features provided by its base class ([5.8.2](#)), this component provides the following functionality for input of numerical data:

- Accepts both . (dot) and , (comma) for separation of the decimal part (for different keyboard layouts of the numeric key block).
- Verifies input to ensure only valid numbers are entered.
- Provides a configurable number of decimal digits.

- In addition to providing a custom display format, the number can be displayed as a currency (system settings).
- Has the ability to handle floating point values as integer by shifting the comma (e.g. “123.45” is internally handled as “12345”).
- Provides direct access to the input value as an integer or float.
- Provides a default value that can be used for example when the user enters no value.
- A range for the value can be set (minimum and maximum value).

Parent: Abstract comfort edit field ([5.8.2](#))

Usage: Can be used like a normal edit control. By setting the additional properties the extended functionality can be configured.

Customization: is done by setting the properties of the control.

Requires: -

Provides: a replacement for the standard edit control for numbers.

Friends: -

5.8.4 Comfort db-aware numeric edit field

Name: `TboDBNumEdit`

Purpose: A database aware version of the comfort numeric edit field ([5.8.3](#)).

Level: Component

Description: This component is a database-aware version of the comfort numeric edit field described in [5.8.3](#). It provides the same functionality as its parent component but it has to be bound to a `TDataSource` component for storing its value in a database field.

Parent: Comfort numeric edit field ([5.8.3](#))

Usage: This component is used like a normal database edit control. See [5.8.3](#) for a description of the additional functionality.

Customization: See [5.8.3](#)

Requires: a `TDataSource` for database access.

Provides: an enhanced version of the database edit control specialized for numbers.

Friends: -

5.8.5 Comfort date edit field

Name: TboDateEdit

Purpose: An edit field for dates which is implemented based on the comfort edit field (5.8.2).

Level: Component

Description: In addition to the features provided by its base class (5.8.2), this component provides the following functionality for input of dates:

- Accepts both . (dot) and , (comma) for separating the parts of the date (for different keyboard layouts of the numeric key block).
- Verifies input to ensure only valid date values are entered.
- Provides direct access to the input value as a TDateTime value.
- Provides a default value that can be used when the user enters no value or when he presses the space key. This default value can be set to be the current date.
- The user input can be limited by providing the first and the last day (a range) that can be entered.
- If just a day or a day and a month is entered, the other parts of the date can be get from the default date.
- The user can enter the current date by pressing the “h” key.

Parent: Abstract comfort edit field (5.8.2)

Usage: Can be used like a normal edit control. By setting the additional properties the extended functionality can be configured.

Customization: is done by setting the properties of the control.

Requires: -

Provides: a replacement for the standard edit control for date values.

Friends: -

5.8.6 Comfort db-aware date edit field

Name: `TboDBDateEdit`

Purpose: A database aware version of the comfort date edit field (5.8.5).

Level: Component

Description: This component is a database-aware version of the comfort date edit field described in 5.8.5. It provides the same functionality as its parent component but it has to be bound to a `TDataSource` component for storing its value in a database field.

Parent: Comfort date edit field (5.8.5)

Usage: This component is used like a normal database edit control. See 5.8.5 for a description of the additional functionality.

Customization: See 5.8.5

Requires: a `TDataSource` for database access.

Provides: an enhanced version of the database edit control for date values.

Friends: -

5.8.7 Mapping db-aware listbox

Name: `TboDBMapListBox`

Purpose: A database-aware listbox that maps the selection to a value (just like the `TDBRadioGroup`)

Level: Component

Description: The standard database-aware listbox control allows the selection of a value for a database field from a list. The drawback of this control is that it stores the same value that is displayed in the database. An often needed feature is to use another value for the database, based on a mapping between display values and database values. This component provides this functionality. It provides a “Values” list which corresponds to the “Items” list, similar to the `TDBRadioGroup` component. When an item from the list is selected, the corresponding value is stored in the database.

Parent: `TDBListBox`

Usage: This component can be used like an ordinary `TDBListBox`. If the “Values” property is not set, it behaves in the same way. To enable the mapping functionality, just set the “Values” property to the corresponding database values for each item in the “Items” property.

Customization: is done by setting properties in the object inspector and by implementing event handlers.

Requires: a `TDataSource` for data storage.

Provides: an enhanced `TDBListBox` component that supports a mapping between the displayed values and the stored values.

Friends: -

5.8.8 Mapping db-aware combobox

Name: `TboDBMapComboBox`

Purpose: A database-aware combobox that maps the selection to a value (just like the `TDBRadioGroup`)

Level: Component

Description: Similar to the db-aware listbox (5.8.7), this component provides the same functionality in a combobox.

Parent: `TDBComboBox`

Usage: See 5.8.7

Customization: See 5.8.7

Requires: See 5.8.7

Provides: an enhanced `TDBComboBox` component that supports a mapping between the displayed values and the stored values.

Friends: -

5.8.9 Button Panel

Name: `TboButtonPanel`

Purpose: A panel that arranges its children (normally buttons) automatically according to a defined style.

Level: Component

Description: When creating dialogs (and especially when creating resizable dialogs), buttons normally should have a fixed position in relation to the dialog (most of the time on the bottom, flushed to the right). Achieving this order is possible with the standard button controls, but is a rather tedious task. This component provides an easy solution for that and adds some additional essential functionality.

When inherited forms are used, it is common to add (or sometimes even hide) buttons on a form. Then the layout of the buttons has to be changed manually.

This component does all that automatically, and arranges the buttons according to their tab order, so that it becomes easy to insert buttons in a derived form.

Parent: TWinControl

Usage: The component is used just like a panel. When it is placed on the form, it can be aligned using the standard properties. It also provides a special “floating” mode where it is aligned to some border of the form, but “floats” above the other controls of that form (very useful when using the notebook-like controls). Buttons (or, if desired, also other controls) can be added to the panel and are arranged automatically according to the settings of the panel.

Customization: To customize the layout of the panel and its child controls set the properties of the panel to the appropriate values. Also event handlers can be implemented for special functionality. For further customization, a subclass has to be used.

Requires: -

Provides: A panel that automatically arranges its child controls (somewhat similar to the layout managers known from other GUI libraries).

Friends: -

5.8.10 Standard button

Name: TboStdBtn

Purpose: A button that implements some commonly used functionality.

Level: Component

Description: When using buttons in a dialog, there are a few standard buttons that are used most often (such as “OK”, “Cancel”, “Close”, etc). Those buttons have a defined functionality and a predefined setting of some of their properties (e.g. the caption and modal result properties). This component implements those standard buttons so that they can be implemented by setting one property to the type of standard button to use.

Additionally it provides an event that is called when the button is clicked and before the designated action is executed. This event can be used to perform some action (e.g. validation of the dialog input) before a button functionality (e.g. “OK”) is executed.

Parent: TButton

Usage: Use like the normal button control. The additional property can be set to define the type of standard button that this control should implement.

Customization: Like a normal control this component is customized by setting its properties and implementing event handlers.

Requires: -

Provides: An enhanced button control.

Friends: Often used together with a button panel (5.8.9)

5.8.11 Progress dialog

Name: TboProgressDialog

Purpose: A component for displaying a progress dialog while performing a lengthy operation.

Level: Component

Description: When performing operations that take more than a moment, an application should display a progress dialog to provide visual feedback about the operation and its progress. This component is used for creating such dialogs very easily.

Parent: TComponent

Usage: First an instance of the component has to be created, either by placing it on a module at design time or by creating it at runtime. Then the properties of the component can be set to adjust the look of the progress dialog. When the operation starts, the application calls the “Show” method. While the operation is running, the “Position” property can be updated to show the actual progress. After everything is finished, the progress dialog is hidden using the “Hide” methods. If desired it can also display an “Abort” button, which can be used to abort the operation. The component provides different methods for telling the application that the user has clicked the abort button. The application can poll a flag in the component, or the component can raise an exception.

Customization: To customize the look of the progress dialog the component has various properties. For further customization a subclass has to be derived where some virtual methods can be overridden.

Requires: -

Provides: An easy way of creating a progress dialog.

Friends: -

5.8.12 Abstract property storage

Name: `TboAbstractPropStorage`

Purpose: An abstract component for implementing a persistent property storage.

Level: Component

Description: Nearly every application has to store some settings that have to be persistent between sessions. These are mostly application settings like window positions, dialog values, etc. that are not typical application data that is usually stored in a database but instead application settings. These properties can be stored at various places, such as the registry, preference-files², or even in the database together with the application data. Since each of those storage methods requires its own access methods, this component provides a uniform interface for different types of property storages.

The data that is stored in those places can have different data types. The storage places, however, often support only a single data type for all properties. So this component also implements the conversion of the different property types to string values for easier storage and maintenance.

Parent: `TComponent`

Usage: Since this is an abstract form, it cannot be used directly. Derive a subclass from this component instead and implement the abstract methods for the desired storage location.

Customization: In a derived class some virtual methods can be overridden to customize the functionality of the component.

Requires: -

Provides: An interface for property storage components.

Friends: -

5.8.13 Property storage for preference files

Name: `TboIniPropStorage`

Purpose: Implements the property storage interface (5.8.12) for storing property values in preference-files (INI-files).

Level: Component

Description: This component implements the interface defined in 5.8.12 and implements the functionality to store the data in preference files (the typical Windows INI-files).

²Also called INI files because of their extension.

Parent: Abstract property storage ([5.8.12](#))

Usage: Place the component on a module and define the properties for the location of the INI-file. Then the methods defined in the property storage interface ([5.8.12](#)) can be used to store and load properties.

Customization: The location of the preference-file is customized by setting the properties of the component. Some other functionality can be implemented by overriding event handlers.

Requires: -

Provides: An implementation of the property storage interface ([5.8.12](#))

Friends: -

5.8.14 Property storage for the registry

Name: TboRegPropStorage

Purpose: Implements the property storage interface ([5.8.12](#)) for storing property values in the windows registry.

Level: Component

Description: This component implements the interface defined in [5.8.12](#) and implements the functionality to store the data in the windows registry.

Parent: Abstract property storage ([5.8.12](#))

Usage: Place the component on a module and define the properties of the component for the location in the registry where to store the properties. Then the methods defined in the property storage interface ([5.8.12](#)) can be used to store and load properties.

Customization: The location in the registry is customized by setting the properties of the component. Some other functionality can be implemented by overriding event handlers.

Requires: -

Provides: An implementation of the property storage interface ([5.8.12](#))

Friends: -

5.8.15 Property storage for a database table

Name: TboDBPropStorage

Purpose: Implements the property storage interface (5.8.12) for storing property values in a database table.

Level: Component

Description: This component implements the interface defined in 5.8.12 and implements the functionality to store the data in a database table. The database table has to provide columns for the name of the property, the value, and a section name, if desired. All these fields have to be string fields.

Parent: Abstract property storage (5.8.12)

Usage: Place the component on a module, assign a dataset for storing the properties and (eventually) define the field names for the fields in the dataset. Then the methods defined in the property storage interface (5.8.12) can be used to store and load properties.

Customization: The storage location is customized by setting the properties of the component. Some other functionality can be implemented by implementing event handlers.

Requires: A dataset where the properties can be stored.

Provides: An implementation of the property storage interface (5.8.12)

Friends: -

5.8.16 Property storage in memory

Name: TboMemPropStorage

Purpose: Implements the property storage interface (5.8.12) for storing property values in memory.

Level: Component

Description: This component implements the interface defined in 5.8.12 and implements the functionality to store the data in memory. The whole memory block of properties can then be accessed at once for custom saving/loading.

Parent: Abstract property storage (5.8.12)

Usage: Place the component on a module and the methods defined in the property storage interface (5.8.12) can be used to store and load properties. However, the properties are only stored in memory so they are lost between different sessions if persistence is not implemented manually.

Customization: The functionality of the component can be changed by deriving a subclass and overriding virtual methods.

Requires: -

Provides: An implementation of the property storage interface (5.8.12)

Friends: -

5.8.17 Property storage proxy object

Name: TboProxyPropStorage

Purpose: Implements the property storage interface (5.8.12) for storing property values in another property storage object.

Level: Component

Description: This component implements the interface defined in 5.8.12. Instead of implementing a storage location by itself, this component let's another property storage object handle the actual storage of the properties. This is useful in an application where the real property storage object is defined only once. Other forms that need to access the properties can use a property storage proxy object for doing that. This way any changes to the storage location have to be done only once.

Parent: Abstract property storage (5.8.12)

Usage: Place the component on a module and connect it to a master property storage. Then the methods defined in the property storage interface (5.8.12) can be used to store and load properties and are handled by the master storage.

Customization: The functionality of the component can be changed by deriving a subclass and overriding virtual methods.

Requires: Another property storage that actually executes the storage operations.

Provides: An implementation of the property storage interface (5.8.12)

Friends: -

5.9 Application settings management

This part of the toolkit provides abstractions and components for handling user preferences and application settings.

5.9.1 Application settings manager

Purpose: Provides an abstraction for editing application settings.

Level: Design Pattern

Description: Creating an application settings dialog can be a tedious and time intensive task, because various different settings have to be implemented and stored. This pattern allows creation of application settings dialogs by just specifying the settings along with their types (and some additional information). Predefined forms and classes for each setting type then implement the edit actions required for setting and storing the preference data.

The pattern uses an application settings form for user interaction. For each type of application setting, a preference class that manages user interaction for that setting has to be implemented. This class utilizes a frame for providing the user interface. The class and the frame work together similar to the bridge pattern [Gam94]. This structure was chosen because the frames are needed only while the setting is edited and thus can be dynamically created, while the preference class itself is used constantly.

For some common types of application settings an implementation of the preference class and the preference frame is included in the toolkit. A class diagram showing the structure of this pattern is shown in Figure 5.7.

Requires: Implementations of the classes of the pattern are provided with the toolkit. If an application wants to use a setting type that is not included in the toolkit, a corresponding preference class and a preference frame have to be implemented.

Provides: a collaboration style for creating application settings dialogs.

Friends: The pattern consists of the preference management form (5.9.2) and the preference classes and frames (5.9.3).

5.9.2 Preference management form

Name: TboPrefMgrFrm

Purpose: A form for editing application settings.

Level: Form

Description: This form implements the application settings manager pattern (5.9.1). It provides the functionality for managing a set of application preferences, each represented by a preference object (5.9.3), and also includes the user interface for allowing the user to edit those preferences.

The user can select the application settings from a list, optionally divided into preference groups. After he has selected a setting, the preference object shows the preference frame for that setting and the user can modify the setting.

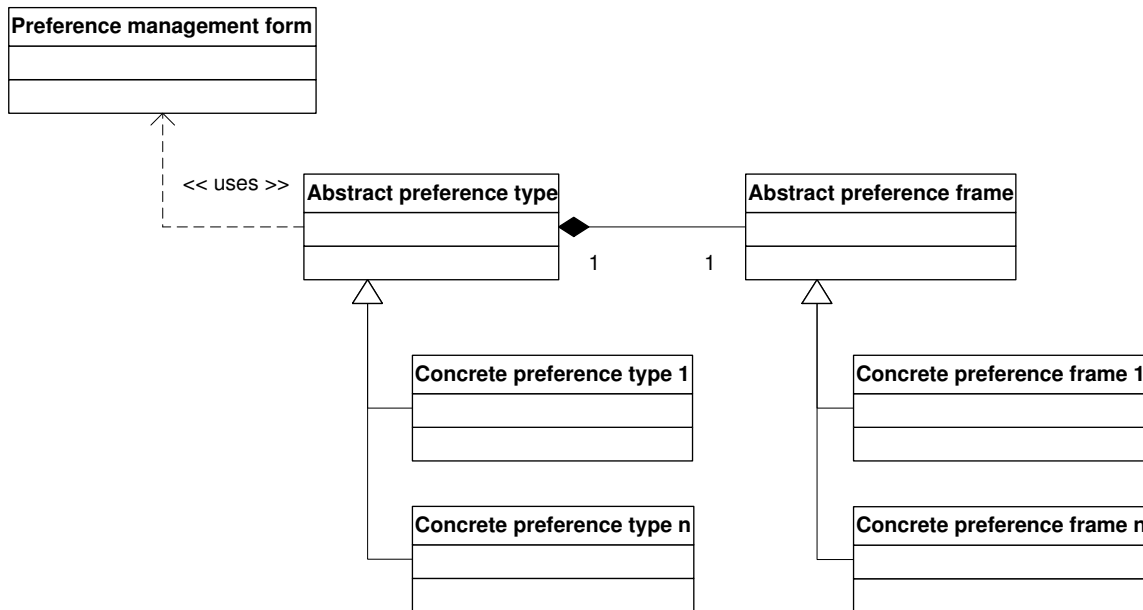


Figure 5.7: Class diagram for the application settings manager pattern

Parent: TForm

Usage: An application has to derive a subclass from this form and implement the abstract methods that define the set of application preferences and the property storage (5.8.12) where the settings should be stored. After that the form handles all other actions.

Customization: The application settings are set in a derived class by adding preference objects to the list. Further customizations can be implemented by overriding virtual methods.

Requires: a property storage (5.8.12) for storing the application settings.

Provides: the form for implementing the application settings manager pattern (5.9.1).

Friends: Uses a button panel (5.8.9) and a property storage (5.8.12).

5.9.3 Abstract preference class and frame

Name: TboPrefTypeBase, TboPrefFrameBase

Purpose: An abstract class for managing one type of application setting and a corresponding edit frame.

Level: Class/Frame

Description: This class provides the interface for managing one type of application settings. It also includes a frame, as described in the application settings manager pattern (5.9.1), that is used for implementing the user interface. The interface of the class includes methods for defining the properties of an application setting (such as name and category) and for handling the edit operations for that setting by creating a preference frame and loading and storing the setting value from the property storage (5.8.12). The frame just contains the user interface elements for that type of preference setting.

Parent: TObject/TFrame

Usage: Create a subclass derived from this class and implement the functionality for the preference type in the subclass. Also create a subclass derived from the frame and implement the user interface for the preference type. Then the class can be used together with the preference management form (5.9.2).

Customization: The behavior of the class is customized in the subclass by overriding virtual methods.

Requires: -

Provides: An interface for preference management classes and frames.

Friends: is used to implement the application settings manager pattern (5.9.1).

5.9.4 Preference class and frame for strings

Name: TboPrefTypeString, TboPrefFrameString

Purpose: A preference edit class and frame for editing single-line text settings.

Level: Class/Frame

Description: Implements the interface defined in 5.9.3 for string-type application settings. The user can enter the string value in an edit control.

Parent: TObject/TFrame

Usage: Add the preference class to a preference management form (5.9.2) and define the properties of the preference in the constructor.

Customization: The preference can be customized by specifying values to the constructor. For further customization, a subclass has to be derived.

Requires: -

Provides: An implementation of the preference class and frame interface as defined in 5.9.3.

Friends: Used in a preference management form (5.9.2).

5.9.5 Preference class and frame for numbers

Name: TboPrefTypeNumber, TboPrefFrameNumber

Purpose: A preference edit class and frame for editing numerical settings.

Level: Class/Frame

Description: Implements the interface defined in 5.9.3 for numeric application settings. Using the comfort numeric edit field (5.8.3) the allowed values and properties of the number can be specified. The preference class supports setting the minimal and maximal value allowed, the number of decimal digits and also supports storing floating point numbers as integers using the comma shift option of the comfort numeric edit field (5.8.3).

Parent: TObject/TFrame

Usage: Add the preference class to a preference management form (5.9.2) and define the properties of the preference in the constructor.

Customization: The preference can be customized by specifying values to the constructor. For further customization, a subclass has to be derived.

Requires: -

Provides: An implementation of the preference class and frame interface as defined in 5.9.3.

Friends: Used in a preference management form (5.9.2).

5.9.6 Preference class and frame for multi-line strings

Name: TboPrefTypeMemo, TboPrefFrameMemo

Purpose: A preference edit class and frame for editing multi-line text settings.

Level: Class/Frame

Description: Implements the interface defined in 5.9.3 for text application settings. The user can enter the value for the setting in a memo field.

Parent: TObject/TFrame

Usage: Add the preference class to a preference management form (5.9.2) and define the properties of the preference (name, category) in the constructor.

Customization: The preference can be customized by specifying values to the constructor. For further customization, a subclass has to be derived.

Requires: -

Provides: An implementation of the preference class and frame interface as defined in 5.9.3.

Friends: Used in a preference management form (5.9.2).

5.9.7 Preference class and frame for database values

Name: TboPrefTypeDBValue, TboPrefFrameDBValue

Purpose: A preference edit class and frame for application settings that consist of a record of a database table.

Level: Class/Frame

Description: Implements the interface defined in 5.9.3 for database value application settings. The user selects a record from a dataset, and one field of that record is stored in the application preference. Depending on the application it can also be allowed to store a NULL value, that means selecting no record.

Parent: TObject/TFrame

Usage: Add the preference class to a preference management form (5.9.2), provide a dataset where the user can select a record from and define the properties of the preference in the constructor.

Customization: The preference can be customized by specifying values to the constructor. For further customization, a subclass has to be derived.

Requires: A dataset with the records.

Provides: An implementation of the preference class and frame interface as defined in 5.9.3.

Friends: Used in a preference management form (5.9.2).

5.10 Other parts of the application

Other parts of the application include a main dialog for task-oriented applications and an application about dialog.

5.10.1 Application main dialog

Name: TboAppMainBtnFrm

Purpose: Provides a template for creating application main dialogs for task oriented applications.

Level: Form

Description: In contrast to the typical user interface of document oriented applications, task oriented applications need another style of user interface. While many approaches may be appropriate in such a situation, the one taken here is a dialog that contains a number of buttons, one for each task. The buttons can be grouped so that tasks that logically belong together are also physically in the same region.

This form provides a starting point for creating such interfaces. On top of the form it contains a section where the application name and probably some logo can be shown. At the bottom the form contains an “Info” button, which is used to display information about the application and an “Exit” button that quits the application. Between those two areas the developer can place the buttons for the different tasks. For bigger applications a page control can be used to group buttons on different pages.

Parent: TForm

Usage: Derive a subclass from this form and add buttons for the tasks to the button area. Other properties of the form can be customized in the object inspector. The standard implementation uses the application about dialog described in section [5.10.2](#) for displaying the application information. A virtual function can be overridden to provide some other info dialog.

Customization: Customization can be done by setting properties of the form in the object inspector and adding controls to the form.

Requires: -

Provides: A simple application main dialog.

Friends: The standard implementation uses the application about dialog ([5.10.2](#)) for displaying the application information.

5.10.2 Application about dialog

Name: TboAppAboutFrm

Purpose: A standard about dialog for an application.

Level: Form

Description: This is an implementation of a standard application about dialog. It shows the application name, its icon (both fetched from the global application object and thus set automatically), the application version (fetched from the version information resource of the application executable), a copyright information and some links to the creator of the application (mail address and homepage URL). It also includes an “OK” button that can be used to close the dialog.

Parent: TForm

Usage: The dialog can be used as is by creating an object.

Customization: To customize the dialog, a subclass has to be derived. In this class the properties can be modified and controls can be added to modify the look of the dialog.

Requires: For the dialog to work correctly the application executable has to include a version information resource.

Provides: A simple way of creating an application about dialog.

Friends: -

5.11 Utility components

The utility components are used by the rest of the toolkit and include classes for the storage of database primary keys and classes for the handling of SQL statements.

5.11.1 Database key value object

Name: `TboUtKeyObject`

Purpose: An abstract class that stores the value of a database table's primary key.

Level: Class

Description: In a relational database, each table should have one or more fields defined to be the key of each table. This key field(s) are used to provide a unique identifier for each record in the database relation.

An application often needs to store the value of such a database key, for example when passing it from one function to another. This object provides an abstract way to handle database keys of any structure. Since most often a single integer value will be the database key, the interface of the key value object provides a way to optimize access for that special case.

The object provides an interface to retrieve the key from the current record of a dataset, set it to a record, set it to the fields of a record where the field names are prefixed with a string (for handling foreign keys), locate the record with the current key in a dataset and create another key object of the same type (as described in the prototype pattern in [Gam94]).

Parent: `TObject`

Usage: Since this is an abstract class, it cannot be used directly. Instead a subclass has to be derived that implements the abstract methods. Those subclasses can be used for the purpose described before.

Customization: A subclass can override virtual methods to customize various operations.

Requires: -

Provides: an interface for handling a database key value object.

Friends: -

5.11.2 Database key object for integer keys

Name: `TboUtKeyInteger`

Purpose: An implementation of the database key value interface (5.11.1) for primary keys that consist of a single integer field.

Level: Class

Description: This class implements the database key value interface for primary keys that are defined as a single integer field, which is the most common case. It provides the operations defined in the interface for that case.

Parent: Database key value object (5.11.1)

Usage: See 5.11.1

Customization: The name of the database field that contains the primary key can be specified in the constructor.

Requires: -

Provides: An implementation of the database key value interface (5.11.1).

Friends: -

5.11.3 SQL dataset helper object

Name: `TboUtSqlDataSetHelper`

Purpose: A class for modification of the SQL select statement of a dataset.

Level: Class

Description: A database application normally uses the SQL query language to create the datasets required for processing. These queries are usually built by the application programmer. For efficient use of a relational database however, they need to be modified according to various selections by the user. For example, if the user wants to get a result dataset in some specific order, an application can sort the records after fetching them from the database — but it would be better to tell the database to return a sorted result set. A similar situation is the filtering of a dataset: it can be performed locally on the client, or directly on the database server, which is much more efficient. This has to be done by modifying the SQL statement.

Since SQL statements can get arbitrary complex and have fixed and variable parts, this component uses a string replacement approach. The SQL statement is created using some special strings that are later replaced by the actual filter or sort statements. For seamless operation these place holders should be valid SQL constructs so that they don't interfere with the dataset operation if they are not replaced — for whatever reason.

The class provides methods for constructing filter strings and sort order definitions. The expressions are created programatically which provides the flexibility to implement new classes for other SQL dialects or databases. Since the data access components of the VCL are different for each type of database, a specialized subclass of this component has to be derived for each database client.

Parent: TObject

Usage: Since this is an abstract class, it can not be directly used. Instead a subclass has to be derived for the specific dataset component. Then the filter and sort order expressions can be constructed by accessing the properties and calling the methods of the object. After that is finished, the resulting SQL statement can be applied to the dataset.

Customization: A subclass can change the functionality of this object by overriding virtual methods.

Requires: a dataset for applying the SQL statement.

Provides: a database independent way of modifying a SQL statement.

Friends: This interface is used by many classes throughout this toolkit.

5.11.4 SQL dataset helper for IBX

Name: TboUtIBXDataSetHelper

Purpose: An implementation of the SQL dataset helper (5.11.3) for InterBase Express.

Level: Class

Description: This class implements the functionality of the SQL dataset helper object for the InterBase Express (IBX) data access components.

Parent: SQL dataset helper object (5.11.3)

Usage: Use like the parent component, provide an IBX Dataset.

Customization: Can normally be used without customization, to change the functionality a subclass has to be derived.

Requires: An IBX dataset.

Provides: An implementation of the SQL dataset helper object for InterBase Express.

Friends: -

Chapter 6

Toolkit overview

This chapter provides an overview of the toolkit described in the last chapter by showing a classification of the components in the toolkit and a class diagram.

6.1 Characterization

The toolkit is embedded in the C++ Builder IDE. Since it is an extension of the VCL framework, its components are used like the standard VCL components. The standard components are included in the component palette and can be added to a form in the usual way. Most components in the toolkit, however, are forms or similar classes. These components have to be added to a C++ Builder project, and then custom classes can be derived from them. For easier differentiation the classes in the toolkit are prefixed with `bo` (e.g. `TboAppAboutFrm`). So all classes that belong to the toolkit are grouped together in the different views in the C++ Builder IDE.

By using this approach no additional tools are necessary to use the small business toolkit. The component package has to be installed, and then the whole toolkit can be used by just adding the required forms to the project. The version control system should be used to separate the actual source files of the toolkit used by different projects. This makes it very straight forward to create applications using the toolkit, but of course does not replace the need for examining the structure and usage of the toolkit before the components can be used.

6.2 Classification

The components in the toolkit have different abstraction levels. Some of them can be used directly and are specialized for some purpose, while others are abstract classes that have to be specialized when used. Sometimes a concept is described as a design pattern, and both the implementation of the abstract classes in the pattern as well as concrete implementations of those classes for common uses are provided.

Table 6.1 shows all abstract components that need to be specialized before use. They provide the interface for some concept, but no concrete implementation. Usually these classes are not directly used in an application, apart from using a pointer to such an object. Instead, the specialized classes are used. If in some application a class is derived directly from one of these abstract components, chances are that this should be a reusable component and it should be added to the toolkit.

Table 6.2 shows all components that are designed to be used directly in an application. There are also abstract classes among them, but they only require implementation of some minor features (for example adding the fields of the concrete record) and not implementation of the basic functionality of the class. These abstract classes need to be subclassed before they can be used. Other classes in the toolkit can be used directly by creating an instance of the class. Of course it is also possible to create a subclass of those classes to modify the behavior of the base class. The standard components are used by dropping them on a form or datamodule in the object inspector.

Finally, table 6.3 lists all design patterns of the toolkit. They have no concrete equivalent in the source code, but instead describe how components can be used together to obtain some desired functionality.

Component	Name	Reference
Abstract subitem class	TboDeDetSubBase	Sect. 5.3.5 p. 36
Record selection form	TboMdSelBaseFrm	Sect. 5.5.6 p. 43
Report properties form	TboRepPropFrm	Sect. 5.6.4 p. 49
Property value selector	TboRepPropFrameBase	Sect. 5.6.6 p. 50
Property value selector with enable/disable functionality	TboRepPropFrameEnabledBase	Sect. 5.6.7 p. 51
Abstract comfort edit field	TboComfortEdit	Sect. 5.8.2 p. 57
Abstract property storage	TboAbstractPropStorage	Sect. 5.8.12 p. 64
Abstract preference class	TboPrefTypeBase	Sect. 5.9.3 p. 69
Abstract preference frame	TboPrefFrameBase	Sect. 5.9.3 p. 69
Database key value object	TboUtKeyObject	Sect. 5.11.1 p. 74
SQL dataset helper object	TboUtSqlDataSetHelper	Sect. 5.11.3 p. 75

Table 6.1: Classification — abstract components

Component	Name	Usage	Reference
Simple record entry form	TboDeSimpleRecFrm	subclass	5.3.1 p. 33
Simple record entry form with entry log	TboDeRecLogFrm	subclass	5.3.2 p. 34
Compound record entry form	TboDeDetailRecFrm	subclass	5.3.4 p. 35
Subitem class with text representation	TboDeDetSubText	subclass	5.3.6 p. 37
Simple doc. printing	TboVoSimpleQrpt	subclass	5.4.1 p. 38
Compound doc. printing	TboVoCompoundQrpt	subclass	5.4.2 p. 38
Simple master data record set	TboMdSimpleFrm	instance	5.5.1 p. 39
Master data edit list form	TboMdListFrm	instance	5.5.3 p. 40
Master data edit form	TboMdEditFrm	subclass	5.5.4 p. 42
Auto-generated master data edit form	TboMdEditAutoFrm	instance	5.5.5 p. 42
Record selection with simple list	TboMdSelListFrm	instance	5.5.7 p. 44
Record selection with one filter	TboMdSelFilter1Frm	instance	5.5.8 p. 44
Record selection with two filters	TboMdSelFilter2Frm	instance	5.5.9 p. 45
Record selection with text search	TboMdSelSearchFrm	instance	5.5.10 p. 46
Record selection with incremental search	TboMdSelIncSearchFrm	instance	5.5.11 p. 46
Basic report template	TboRepBaseQrpt	subclass	5.6.1 p. 47
Auto-generated list report	TboRepAutoListQrpt	instance	5.6.2 p. 48
Report properties form for QuickReport	TboRepPropQuickrepFrm	instance	5.6.5 p. 50
Property value selector for a date range	TboRepPropFrameDateRange	instance	5.6.8 p. 51
Property value selector for a database value	TboRepPropFrameDBLookup	instance	5.6.9 p. 52
Property value selector for the sort order	TboRepPropFrameSortOrder	instance	5.6.10 p. 53
Data set iterator	TboIntDataSetIterator	instance	5.7.2 p. 54
Fixed record size data file writer	TboIntWriteFixedRecord	instance	5.7.3 p. 54
Data export to Word	TboIntExRecWord	instance	5.7.4 p. 55
Comfort keyboard handler	TboKeyHandler	(comp.)	5.8.1 p. 56

Table 6.2: Classification — concrete components

Component	Name	Usage	Reference
Comfort numeric edit field	TboNumEdit	(comp.)	5.8.3 p. 57
Comfort db-aware numeric edit field	TboDBNumEdit	(comp.)	5.8.4 p. 58
Comfort date edit field	TboDateEdit	(comp.)	5.8.5 p. 59
Comfort db-aware date edit field	TboDBDateEdit	(comp.)	5.8.6 p. 60
Mapping db-aware listbox	TboDBMapListBox	(comp.)	5.8.7 p. 60
Mapping db-aware combobox	TboDBMapComboBox	(comp.)	5.8.8 p. 61
Button Panel	TboButtonPanel	(comp.)	5.8.9 p. 61
Standard button	TboStdBtn	(comp.)	5.8.10 p. 62
Progress dialog	TboProgressDialog	(comp.)	5.8.11 p. 63
Property storage for preference files	TboIniPropStorage	(comp.)	5.8.13 p. 64
Property storage for the registry	TboRegPropStorage	(comp.)	5.8.14 p. 65
Property storage for a database table	TboDBPropStorage	(comp.)	5.8.15 p. 66
Property storage in memory	TboMemPropStorage	(comp.)	5.8.16 p. 66
Property storage proxy object	TboProxyPropStorage	(comp.)	5.8.17 p. 67
Preference management form	TboPrefMgrFrm	subclass	5.9.2 p. 68
Preference class for strings	TboPrefTypeString	instance	5.9.4 p. 70
Preference frame for strings	TboPrefFrameString	instance	5.9.4 p. 70
Preference class for numbers	TboPrefTypeNumber	instance	5.9.5 p. 71
Preference frame for numbers	TboPrefFrameNumber	instance	5.9.5 p. 71
Preference class for multi-line strings	TboPrefTypeMemo	instance	5.9.6 p. 71
Preference frame for multi-line strings	TboPrefFrameMemo	instance	5.9.6 p. 71
Preference class for database values	TboPrefTypeDBValue	instance	5.9.7 p. 72
Preference frame for database values	TboPrefFrameDBValue	instance	5.9.7 p. 72
Application main dialog	TboAppMainBtnFrm	subclass	5.10.1 p. 72
Application about dialog	TboAppAboutFrm	instance	5.10.2 p. 73
Database key object for integer keys	TboUtKeyInteger	instance	5.11.2 p. 75
SQL dataset helper for IBX	TboUtIBXDataSetHelper	instance	5.11.4 p. 76

Table 6.2: (continued)

Design pattern	Reference
Compound record entry	Section 5.3.3 on page 34
Master data edit	Section 5.5.2 on page 40
Report properties (sort, filter)	Section 5.6.3 on page 48
Data export pattern	Section 5.7.1 on page 53
Application settings manager	Section 5.9.1 on page 68

Table 6.3: Classification — design patterns

6.3 Class diagram

Figures [6.1](#), [6.2](#) and [6.3](#) show the class diagram for the toolkit. Since there are many different parts in the toolkit, the class hierarchy is relatively flat. However there are several areas of related classes for each part of an application. They are grouped using the UML¹ *package* notation.

¹<http://www.omg.org/uml/>

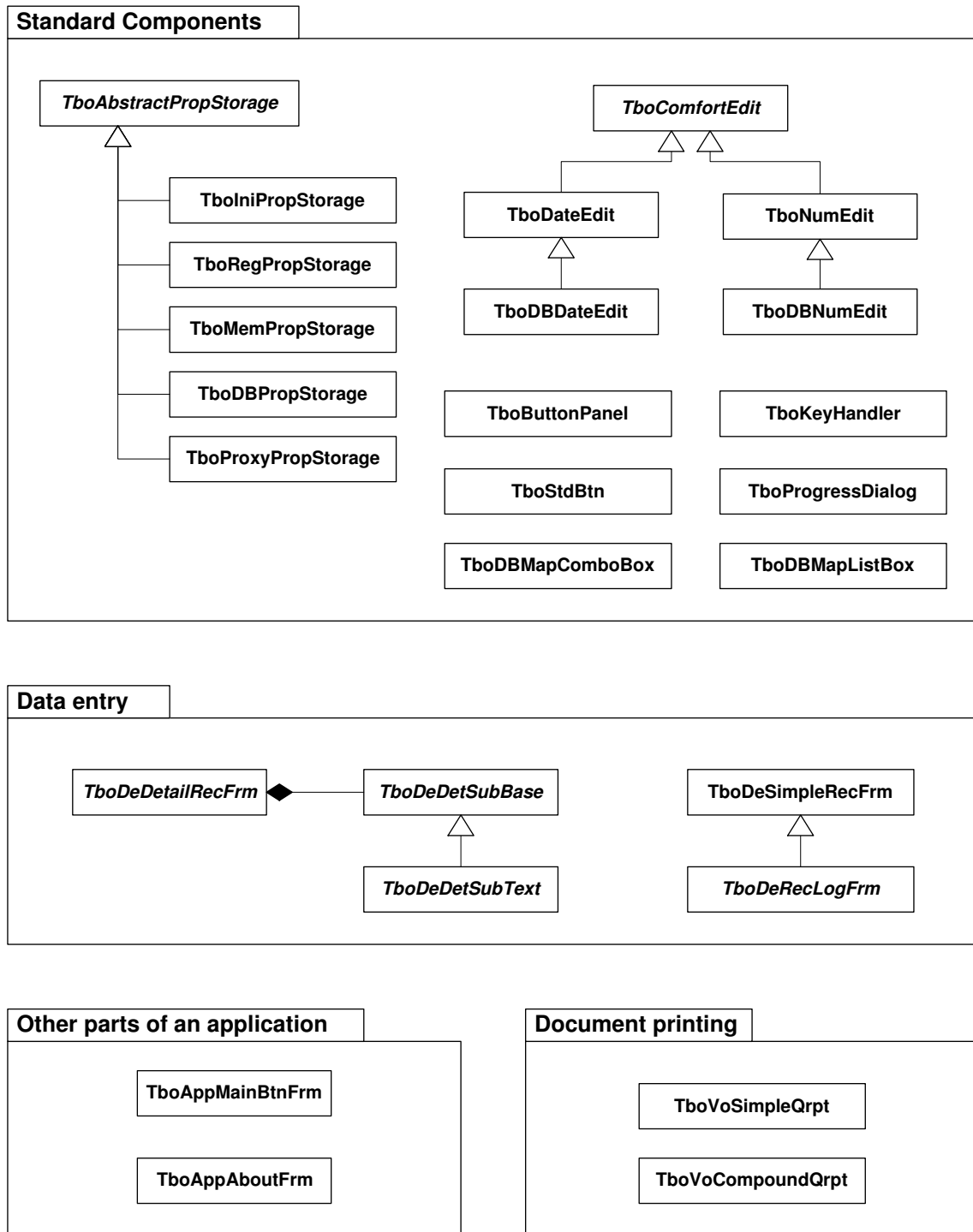


Figure 6.1: The class diagram of the toolkit, part 1

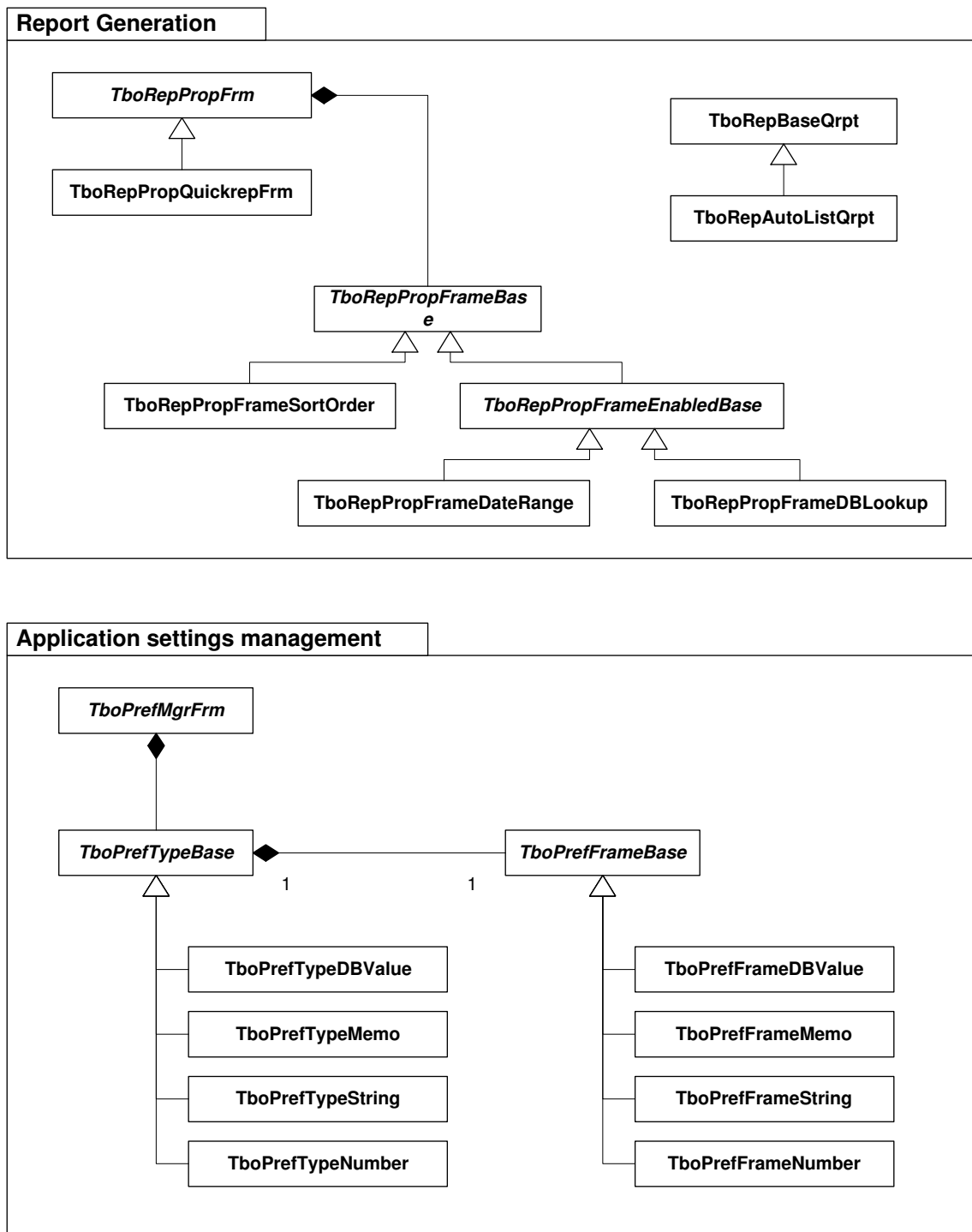


Figure 6.2: The class diagram of the toolkit, part 2

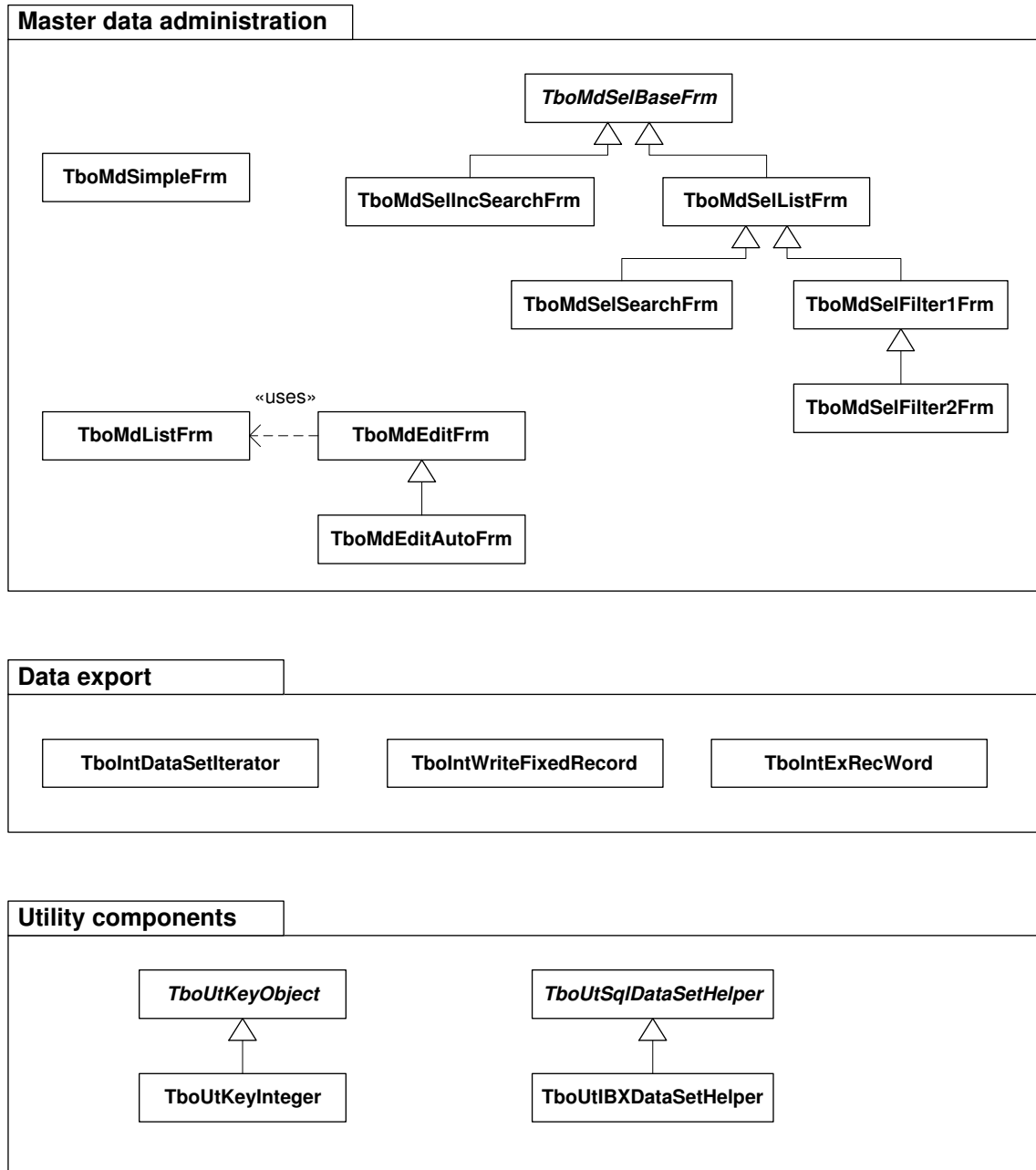


Figure 6.3: The class diagram of the toolkit, part 3

Chapter 7

Toolkit Evaluation

In this chapter usage scenarios for the toolkit are described. Two sample applications are developed using the toolkit, a case study shows how the toolkit can be used and the benefit from using the toolkit is analyzed.

7.1 Overview

To evaluate the reuse benefit gained when building applications with this toolkit, two of the sample applications described in section 2.4 were re-implemented using the toolkit. The original implementation, which was done some time ago, is then compared against the new implementation and the reuse benefit is investigated.

7.2 Applications

7.2.1 Invoice application for a production company

The invoice application described in section 2.4.1 uses nearly all parts of the toolkit. The main dialog is created using the *application main dialog form* (5.10.1) and uses an *application about dialog* (5.10.2) for displaying application information. The delivery notes and the invoices have a similar structure, so they are designed as a compound record and thus handled using the *compound record entry pattern* (5.3.3). The documents for the delivery notes and the invoices are printed with the *compound document printing report* (5.4.2).

The customers and articles are edited using the *master data edit pattern* (5.5.2). *Record selection forms* (5.5.6) are used to select records from this master data sets. The *application settings management* components (5.9) are used for managing some customizable application settings.

Finally there are also some reports created using the *report generation* components (5.6). However, reports are of lower priority for this application.

7.2.2 Production monitoring application

The second application implemented with the toolkit was the production monitoring application of section 2.4.2. It is a relative straightforward application that consists of a data entry part and a report part.

For data entry the *simple record entry form* (5.3.1) is used. The application also contains one master-data set that is managed using the *master data edit form* (5.5.4). Since the application is small and has a straight forward user interface, it uses an *application main dialog form* (5.10.1) that also contains some application functionality apart from providing the choice of functions. It also uses an *application about dialog* (5.10.2) for displaying application and version information.

Reports are generated using the *report generation* components (5.6). For one report, however, a custom solution had to be implemented because that report consists of a special graph that could not be handled by the report generation component.

7.3 Case study

To get an impression of how the toolkit can simplify some tasks in developing applications in the small business domain, a little case study is described in the following.

One part of the toolkit deals with managing master-data sets. Nearly every application in the small business domain includes similar functionality. A master-data set is managed by adding records, editing records and deleting records. For the developer, it is tedious to implement the same functionality for every master-data set in every application.

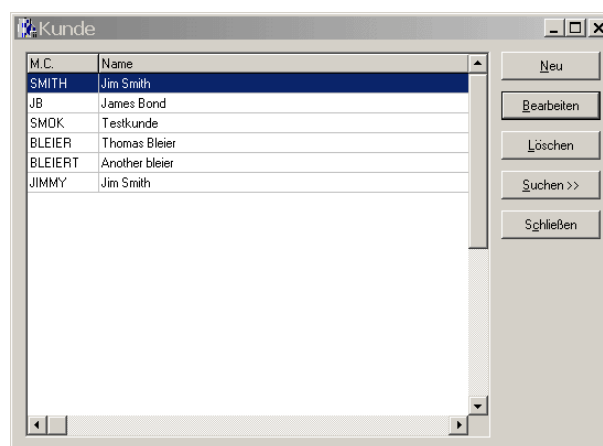


Figure 7.1: Case study — master data set list dialog

So, the toolkit provides a master data edit pattern, as described in section 5.5.2. A list form is used to let the user select the desired record, and a separate edit form is used

for modifying the data of the record. Figure 7.1 shows the list form for some example master data set.

An individual edit form has to be generated for each master data set. For prototyping or where a simple edit form is appropriate, the toolkit provides an automatically generated edit form. The form shows edit controls for each field in the dataset. Figure 7.2 shows such a form.

Figure 7.2: Case study — automatically generated edit form

When such an edit form is not sufficient, the toolkit provides a base form for the creation of customized edit forms. That base form provides all the necessary record handling functionality, the developer just needs to place the controls for the fields of the dataset onto the form. Figure 7.3 shows such a form, in that case the customer edit form of the invoice application described in section 7.2.1.

Figure 7.3: Case study — customized master data edit form

So, when the developer has to create the functionality for maintaining a master data set, all he has to do is to define the data set, eventually create a custom edit form (which is straightforward) and use the classes in his application. The usage of the classes is simply a matter of creating instances of the classes and executing a method. The code for doing this is shown in Figure 7.4. The additional objects in this code (`keyobject` and `datasethelper`) are the utility classes from the toolkit as described in section 5.11.

```
TboMdListFrm *frm = new TboMdListFrm(this);
TEditKundeFrm editfrm = new TEditKundeFrm(frm);
TboUtKeyObject keyobject = new TboUtKeyInteger;
try {
    editfrm->DataSetHelper = MainMod->KundeHelper;
    frm->Exec(editfrm,keyobject,MainMod->KundeListHelper);
} __finally {
    delete keyobject;
    delete frm;
}
```

Figure 7.4: Case study — example code for the master data edit functionality

7.4 Reuse benefit analysis

7.4.1 Introduction

Measuring software reuse is a complex research area and would be a topic for a master's thesis on its own. Since this not the main focus of this document only some practical applications of software metrics are discussed here. For further information see [[Kaf85](#)] for an overview of different approaches and [[Fra96](#)] for a more detailed description on this topic.

The goal of this analysis is to estimate the reuse benefit gained from building applications with this toolkit. The most accurate way would be to develop a specification for some application and implement this specification two times, once using the toolkit and a second time without it. Because of time and economical reasons this is not feasible and thus was not done here. Since there were two existing applications that needed to be re-implemented it was possible to take a similar approach with less expense of time.

What was done here was to re-implement the two applications by using this toolkit. The original implementation was done a few years ago. Because of this the requirements of the new applications changed slightly. This makes the comparison and the calculation of the reuse benefit inaccurate. On the other had it provided the unique chance of comparing two implementations for nearly the same specification.

7.4.2 Comparison

The comparison is done using some simple software metrics. It compares both the old and the new implementations of the applications described in section [7.2](#).

First, the *lines of code (LOC)* in each implementation are compared. Both (the old and the new) implementations are client/server database applications. Since the implementation language was Pascal in the old systems and C++ in the new systems, which are

very similar languages, the lines of code of both implementations are comparable for our purpose.

The second metric used in the comparison is the *number of classes (NOC)*. This number gives an overview of the complexity of the application. Again, since both implementations use a similar language, this comparison is sufficient.

Table 7.1 shows the results of the comparison for the different implementations. It shows the complete number of code-lines and the number of classes of the two applications. The figures show that the new implementations are much bigger than the original ones — but this is explained in the following.

Application	Lines of Code (LOC)	Number of Classes (NOC)
Invoice / Old	6306	27
Invoice / New	17947	61
Production monitoring / Old	4744	13
Production monitoring / New	7628	30

Table 7.1: Overall comparison of the implementations

For determining the reuse benefit it is interesting to know how much of the code was reused from the toolkit and how much was created specifically for the application. Table 7.2 splits the results of the previous analysis into these categories. It shows that a big part of the applications source codes is actually from the toolkit.

Application	New		Reused	
	NOC	LOC	NOC	LOC
Invoice / New	13 21%	5.357 30%	48 79%	12.590 70%
Production mon. / New	7 23%	1.962 26%	23 77%	5.666 74%

Table 7.2: Reuse benefit — new classes vs. reused classes

Another comparison can be done with the classes in the application without counting the classes in the toolkit. Table 7.3 shows those new classes that were generated from scratch¹ for each application compared to the classes that were derived from a class in the toolkit and thus only extend the functionality of the toolkit. It shows that in addition to the reused classes a considerable amount of the code is just an extension or refinement of the concepts in the toolkit and thus can be created with lower effort. All tables also list the lines of code in the corresponding classes.

When these results are combined for each application, a diagram can be drawn to show the distribution of the source code on the three kinds of components (reused component, adapted by deriving from a reused component, created from scratch). Figure 7.5 shows this diagram for the classes of the two examined applications, and Figure 7.6 shows the same diagram for the lines of code.

¹which also includes classes that were derived from some VCL base class.

Application	New / from scratch		New / derived	
	NOC	LOC	NOC	LOC
Invoice / New	1 8%	986 18%	12 92%	4.371 82%
Production mon. / New	4 57%	1.071 55%	3 43%	891 45%

Table 7.3: Reuse benefit — built from scratch vs. derived classes

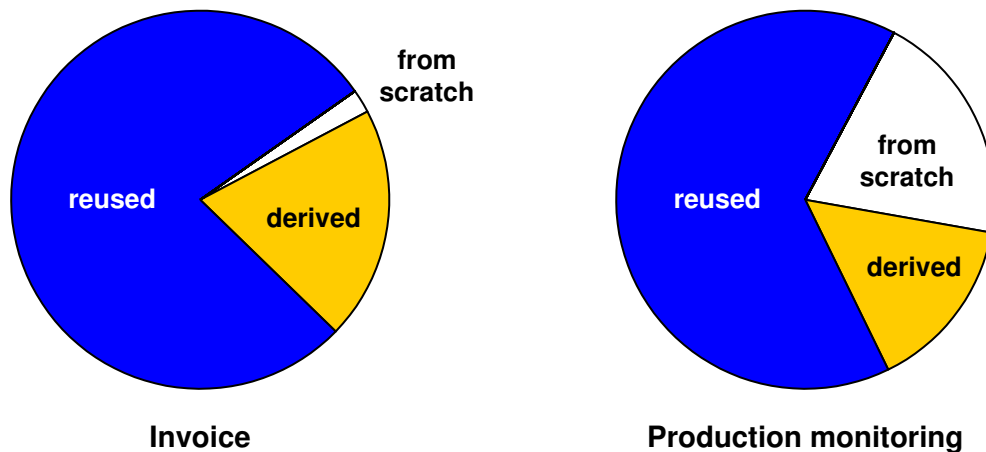


Figure 7.5: NOCs distribution in the applications

7.4.3 Evaluation

The first thing noticed from the examination in the previous section is the fact that the new applications are bigger than the old ones. This is only partly due to the fact that there were some changes to the requirements. For the most part this comes from the toolkit, where a reusable and thus more general design and implementation was used, which, of course, is larger than a specifically designed software. On the other hand, applications built using the toolkit also provide some functionality (mostly *comfort* functions, for example when editing) which otherwise would not have been implemented, but is part of the toolkit.

Examples for similar situations can also be found in the literature. Authors like [Bar91] and [Tra94] state that there is some overhead involved in creating reusable parts. This overhead also affects the final applications and requires some additional effort when creating the components, but on the other hand is spread among all applications that re-use the components. However, this additional effort must not be higher than the savings gained from building applications with reused components. For the small business toolkit this is the case (though this can only be a provisional statement since only two applications have been built yet).

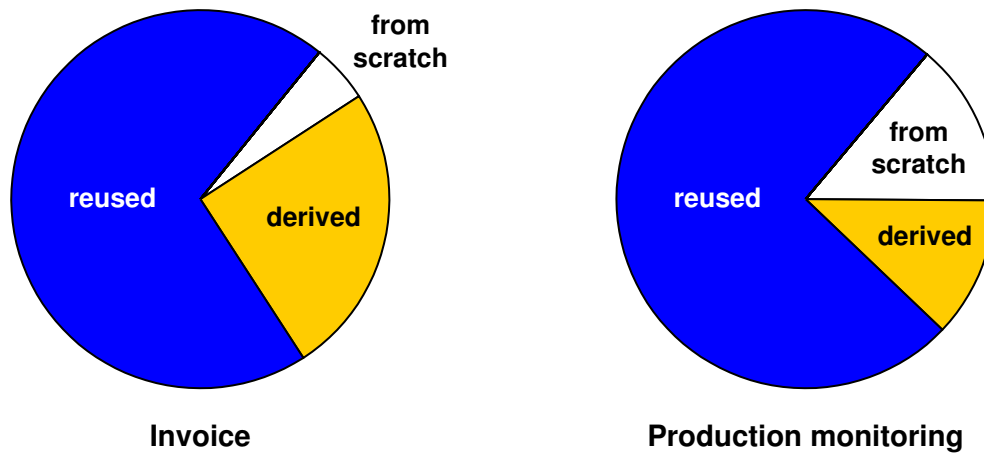


Figure 7.6: LOCs distribution in the applications

For the investigated applications, a high reuse percentage of about 75% was achieved because the toolkit has its focus on exactly that application domain. Experiences made by others (e.g. [Lim94], [Pri01] and [Fra96]) show very different reuse percentages, where the 75 percent achieved here are near the upper limit. This is due to the fact that for the investigated applications much of the functionality could be implemented using components from the toolkit.

The reuse benefit would drop if some required functionality can not be achieved with components in the toolkit and has to be built from scratch. In such a case it may be possible to create a reusable design for the new functionality and add it (or at least parts of it) to the toolkit, so that it is available later on.

In this evaluation the reuse benefit is calculated from the *lines of code* and *number of classes* in an application. These simple software metrics are easy to calculate and are often used, but there are many others.

One interesting metric in this context would be the development costs in terms of development time. While the exact development times were not measured while working on this toolkit, the figures described for example in [Lim94] seem reasonable when compared to the experiences made here. Lim states that developing components for reuse requires about twice the effort of developing the same functionality for a single application, but reusing such a component in an application only requires about 20 percent of the effort that would be required to create the functionality from scratch. This is also the estimation that can be made in this case after creating the small business toolkit.

To be honest it has to be said that the results of this analysis can not be seen as a reference for the general reuse benefit of this toolkit. The analysis of only two applications is much too less to derive general applicable results. The results should rather be seen as

some sort of clue to make an educated guess for future applications. On the other hand an investigation of the reuse benefit in a similar way as described here can be made in the future after developing more applications with this toolkit to verify or falsify the estimations made here.

Chapter 8

Summary and Conclusions

This chapter summarizes the work carried out for this master's thesis. It discusses conclusions drawn from building and using the toolkit and provides some directions for future work.

The work in this thesis started with an idea: to create an environment that allows fast development cycles of small business applications. Such an environment consists of many parts, and most of them are readily available such as database systems, development environments and version control systems. The part that was missing was a toolkit that supports this specific application domain. This master's thesis described the development of such a toolkit.

The thesis started with a domain analysis of the application domain, which should be the first step in every work that has to do with software reuse. After that the requirements for the toolkit were defined to get a view of what the final result should look like. Then the toolkit was designed and implemented. This was the main part of this thesis. Finally the reuse benefit when using the toolkit was evaluated.

At its current state, the small business toolkit contains 63 classes with 15.676 lines of code, five design patterns and additionally 6.511 lines of test code for the components. Among these classes are ready-to-use components and abstract classes that can be customized with little effort. The toolkit also includes more generic abstractions that provide assistance when developing applications in the small business domain. Beside that visible content, the toolkit also provides a basis for extension. Additional reusable components that are found during development of applications can be easily added to the toolkit and used later in other projects.

By using the toolkit, software development in the small business domain can be dramatically simplified. For the examined applications reuse benefits of about 80% in terms of development times were achieved. Though this cannot be taken as the definitive reuse benefit for every application in the domain, it gives a good clue of what reductions in the development effort can be reached if the software engineering process is strictly reuse-based. But, of course, this requires the willingness to accept compromises by favouring the usage of existing components.

The goal of this thesis as described in section 1.2 was reached, but the initial vision, however, is not completely fulfilled. A big step towards realizing this vision was achieved, but such a toolkit is constantly work in progress. There are two main directions for future enhancements:

Evaluation The evaluation carried out in chapter 7 can only be seen as a rough overview of the benefits achieved when using this toolkit to build applications. Many more applications and use cases have to be analyzed to get good and representative results. This, of course, can only be done after some time of production use of the toolkit.

Extension A reusable toolkit has to be constantly maintained. The work in this thesis provides a valuable basis that can be built upon, but as more and more applications are built with the toolkit, it (should) get better. This includes fixing bugs in the components of the toolkit, enhancing the functionality of the existing components and including new components that have a reuse potential.

Appendix A

Sample source documentation

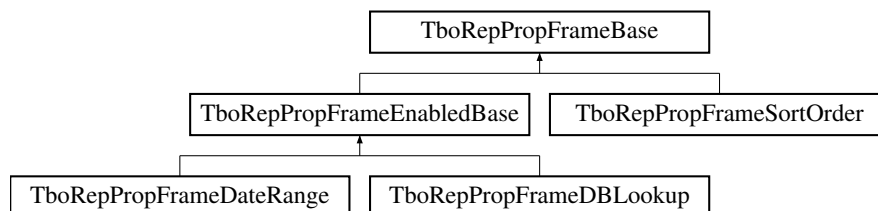
This section provides an example of the source documentation generated with doxygen. It was generated automatically from the comments embedded in the source files of the class.

A.1 TboRepPropFrameBase Class Reference

A frame for specifying some property of a report.

```
#include <TboRepPropFrameBase.h>
```

Inheritance diagram for TboRepPropFrameBase::



Public Methods

- `__fastcall TboRepPropFrameBase (TComponent *Owner)`
Constructor.
- `virtual void __fastcall Initialize ()`
Initialize the frame.
- `virtual void __fastcall Validate ()=0`
Validate the user input.

- virtual void __fastcall **Apply** (**TboUtSqlDataSetHelper** *datasethelper, AnsiString &description)=0

Apply the property settings to the dataset helper object.

- virtual void __fastcall **Finished** ()

Clean up the frame.

A.1.1 Detailed Description

A frame for specifying some property of a report.

A derived class has to provide the functionality to validate if the user settings are correct and to apply the settings to a dataset helper object.

A.1.2 Member Function Documentation

virtual void __fastcall **TboRepPropFrameBase::Apply** (**TboUtSqlDataSetHelper** **datasethelper*, *AnsiString & description*) [pure virtual]

Apply the property settings to the dataset helper object.

Parameters:

datasethelper The dataset helper object

description The method can return a human readable string describing the current settings for the property. All those descriptions are concatenated and can be printed on the report.

Reimplemented in **TboRepPropFrameDateRange** (p. 99), **TboRepPropFrameDBLookup** (p. 103), and **TboRepPropFrameSortOrder** (p. 105).

void __fastcall **TboRepPropFrameBase::Finished** () [virtual]

Clean up the frame.

Called when the properties dialog is hidden.

Reimplemented in **TboRepPropFrameDBLookup** (p. 103).

void __fastcall TboRepPropFrameBase::Initialize () [virtual]

Initialize the frame.

Called before the properties dialog is shown.

Reimplemented in **TboRepPropFrameDateRange** (p. 99), **TboRepPropFrameDBLookup** (p. 102), **TboRepPropFrameEnabledBase** (p. 98), and **TboRepPropFrameSortOrder** (p. 104).

virtual void __fastcall TboRepPropFrameBase::Validate () [pure virtual]

Validate the user input.

If some invalid data is found, the method should display an error message, set the focus to the controls with the invalid data and raise an EAbort exception.

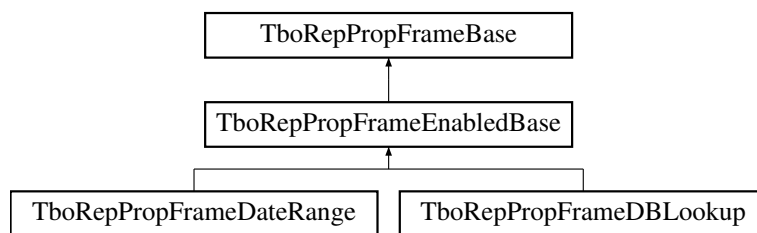
Reimplemented in **TboRepPropFrameDateRange** (p. 99), **TboRepPropFrameDBLookup** (p. 103), and **TboRepPropFrameSortOrder** (p. 105).

A.2 TboRepPropFrameEnabledBase Class Reference

A report property frame that can be enabled/disabled.

```
#include <TboRepPropFrameEnabledBase.h>
```

Inheritance diagram for TboRepPropFrameEnabledBase::



Public Methods

- **void __fastcall PEnabledClick** (TObject *Sender)
Called when the user clicks the enabled checkbox.
- **__fastcall TboRepPropFrameEnabledBase** (TComponent *Owner, AnsiString caption)

Constructor.

- virtual void __fastcall **Initialize** ()

Initialize the frame.

Public Attributes

- TCheckBox * **PEnabled**

The checkbox for enabling/disabling the property.

- __property bool **IsSelected**

Is the property selected/checked?

A.2.1 Detailed Description

A report property frame that can be enabled/disabled.

Provides a property where the user can select if he wants to include the property in the filter or not.

A.2.2 Constructor & Destructor Documentation

__fastcall TboRepPropFrameEnabledBase::TboRepPropFrameEnabledBase (TComponent * *Owner*, AnsiString *caption*)

Constructor.

Parameters:

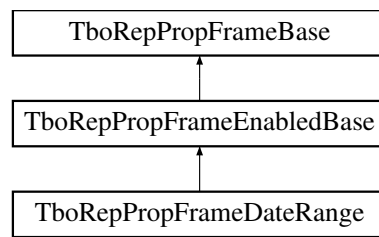
caption Text that is displayed beneath the checkbox that describes the property

A.3 TboRepPropFrameDateRange Class Reference

A report property for selecting a data range.

```
#include <TboRepPropFrameDateRange.h>
```

Inheritance diagram for TboRepPropFrameDateRange::



Public Methods

- void __fastcall **FrameResize** (TObject *Sender)
Called when the frame is resized.
- void __fastcall **PComboChange** (TObject *Sender)
Called when the user selected a value in the quick selection combo box.
- void __fastcall **PEnabledClick** (TObject *Sender)
Called when the user clicked the enabled checkbox.
- __fastcall **TboRepPropFrameDateRange** (TComponent *Owner, AnsiString caption, AnsiString field, int showmonths=14, int showyears=2, TDateTime startdate=TDateTime(0))
Constructor.
- virtual __fastcall ~**TboRepPropFrameDateRange** ()
Destructor.
- virtual void __fastcall **Initialize** ()
Initialize the frame.
- virtual void __fastcall **Validate** ()
Validate the frame.
- virtual void __fastcall **Apply** (TboUtSqlDataSetHelper *datasethelper, AnsiString &description)
Apply the settings to the dataset helper object.

Public Attributes

- **TboDateEdit * PDateVon**
The edit control for the start date.
- **TboDateEdit * PDateBis**
The edit control for the end date.
- **TLabel * PVonLbl**
The label for the start date.
- **TLabel * PBisLbl**
The label for the end date.
- **TLabel * PComboLbl**
The label for the quick selection combo box.
- **TComboBox * PCombo**
The quick selection combo box.

Protected Methods

- virtual void __fastcall **AddMonths** (TDateTime startdate, int count, int increment)
Add whole months (first of month until last of month) to the quick selection combo box.
- virtual void __fastcall **AddYears** (TDateTime startdate, int count, int increment)
Add whole years (1.1 to 31.12.).

Protected Attributes

- **AnsiString FField**
The field which is compared against the dates.

A.3.1 Detailed Description

A report property for selecting a data range.

This property provides the selection of a data range, with the addition of a quick-selection combo box.

A.3.2 Constructor & Destructor Documentation

```
__fastcall TboRepPropFrameDateRange::TboRepPropFrameDateRange (TComponent  
* Owner, AnsiString caption, AnsiString field, int showmonths = 14, int showyears = 2,  
TDateTime startdate = TDateTime(0))
```

Constructor.

Parameters:

caption The description of that report property.

field The field of the dataset that is compared against the dates.

showmonths The number of months that are added to the quick selection combo box

showyears The number of years that are added to the quick selection combo box

startdate The date where to start when filling the quick selection combo box. If this is TDateTime(0), the current date is used.

A.3.3 Member Function Documentation

```
void __fastcall TboRepPropFrameDateRange::AddMonths (TDateTime startdate, int  
count, int increment) [protected, virtual]
```

Add whole months (first of month until last of month) to the quick selection combo box.

Parameters:

startdate The date where to start. The first month added is the month that contains that date.

count The number of months to add

increment The increment factor. A value of 1 starts at startdate and adds each month after that date, a value of 2 adds each second month and a value of -1 adds the months before the start date.

```
void __fastcall TboRepPropFrameDateRange::AddYears (TDateTime startdate, int count,  
int increment) [protected, virtual]
```

Add whole years (1.1 to 31.12.).

The first year added is the year that contains that date.

Parameters:

count The number of years to add

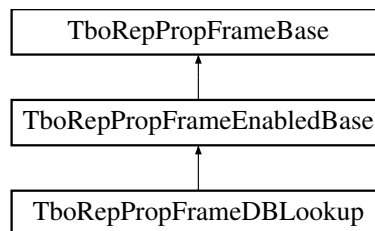
increment The increment factor. A value of 1 starts at *startdate* and adds each year after that date, a value of 2 adds each second year and a value of -1 adds the years before the start date.

A.4 TboRepPropFrameDBLookup Class Reference

A report property for selecting a database value.

```
#include <TboRepPropFrameDBLookup.h>
```

Inheritance diagram for TboRepPropFrameDBLookup::



Public Methods

- void __fastcall **PEnabledClick** (TObject *Sender)
Called when the user clicks the enabled checkbox.
- __fastcall **TboRepPropFrameDBLookup** (TComponent *Owner, AnsiString caption, TDataSet *lookupdataset, AnsiString listfield, AnsiString keyfield, AnsiString reportfield)
Constructor.
- virtual void __fastcall **Initialize** ()
Initialize the frame.

- virtual void __fastcall **Validate** ()

Validate the frame.

- virtual void __fastcall **Apply** (TboUtSqlDataSetHelper *datasethelper, AnsiString &description)

Apply the selection to the dataset helper object.

- virtual void __fastcall **Finished** ()

Clean up the frame.

Public Attributes

- TDataSource * **PDataSrc**

The datasource for the lookup data.

- TDBLookupComboBox * **PLookup**

The lookup combo box.

Protected Attributes

- AnsiString **FFilterField**

The field in the dataset that is compared against the selection.

- bool **FWasDatasetActive**

Was the dataset active when the frame was initialized?

A.4.1 Detailed Description

A report property for selecting a database value.

Allows the user to filter all records that have some value in some database field by selecting that value from a lookup dataset.

A.4.2 Constructor & Destructor Documentation

__fastcall TboRepPropFrameDBLookup::TboRepPropFrameDBLookup (TComponent * Owner, AnsiString caption, TDataSet * lookupdataset, AnsiString listfield, AnsiString keyfield, AnsiString reportfield)

Constructor.

Parameters:

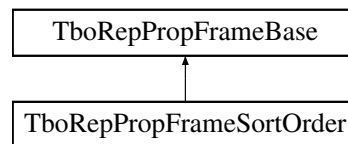
- caption** The description of the property
- lookupdataset** The dataset with the lookup information
- listfield** The field in the lookupdataset that should be shown
- keyfield** The field in the lookupdataset that contains the value that the records should match.
- reportfield** The field in the report dataset that should be compared

A.5 TboRepPropFrameSortOrder Class Reference

A report properties frame that defines the sort order for a report.

```
#include <TboRepPropFrameSortOrder.h>
```

Inheritance diagram for TboRepPropFrameSortOrder::



Public Methods

- **__fastcall TboRepPropFrameSortOrder** (TComponent *Owner, AnsiString fieldmappings, AnsiString caption="")
Constructor.
- virtual **__fastcall ~TboRepPropFrameSortOrder** ()
Destructor.
- virtual void **__fastcall Initialize** ()
Initialize the frame.

- virtual void __fastcall **Validate** ()
Validate the frame.
- virtual void __fastcall **Apply** (TboUtSqlDataSetHelper *datasethelper, AnsiString &description)
Apply the frame to the dataset helper object.

Public Attributes

- TLabel * **PSortLbl**
The label for the sort order.
- TComboBox * **PCombo**
The combo box for selecting the sort order.

A.5.1 Detailed Description

A report properties frame that defines the sort order for a report.

A.5.2 Constructor & Destructor Documentation

__fastcall TboRepPropFrameSortOrder::TboRepPropFrameSortOrder (TComponent * Owner, AnsiString fieldmappings, AnsiString caption = "")

Constructor.

Parameters:

fieldmappings Contains the sort expressions and their description. The different sort orders are separated by a newline () and have the form "Description=expression". The description is shown in the combobox and the expression is used for the datasethelper. The string is assigned to the "Text" property of a TStringList, and then processed with the Names and Values properties, so it has to be in the correct format for that. If the expression of an entry is equal to CboRepPropFrameSortOrderNoSort, no sort order is set.

caption Caption for the sort order. If empty, a default is used.

Appendix B

Glossary

Architecture The architecture of a software defines its overall structure. It does not focus on implementation details but instead documents general design decisions made.

Borland C++ Builder is a development environment manufactured by a company named Borland ¹ which includes a visual form designer, a code editor, a debugger and an application \rightarrow *framework*.

Borland Delphi is a development environment similar to \rightarrow C++ *Builder*, but its base language is Object Pascal instead of C++. In fact Delphi was the first of the two siblings and so, for example, the VCL, which is used in both products, is coded in Pascal.

Component In this thesis the term component refers to some part of an application that has a defined interface and can be reused in other applications. A special form of a component is a \rightarrow VCL component, where (part of the) the interface is defined by the requirements of the development environment.

Customized Applications or tailor-made applications are, as the name implies, built for some specific purpose, in contrast to standard applications that are built by some company and sold to the public.

Domain Analysis describes the process of gathering the requirements for not just a single application, but instead for a whole application domain, which is group of similar applications (e.g. various different invoicing applications).

Form A *form* is a special kind of \rightarrow *module* that contains a user-interface dialog.

Framework The word *framework* is used with different meanings. Here a framework is a \rightarrow *toolkit* that additionally provides an application structure that defines the control flow in an application and where the application uses the framework by extending it at predefined points. An example of a framework is the \rightarrow VCL class library.

¹<http://www.borland.com/>

Master Data Set Business applications often have to handle data sets for their operation the are not modified often, but used constantly by the application. An example of this would be the customer or article database in an accounting application. These data-sets are called master data sets.

Module A module is a part of an application that provides some specified functionality. In the development environment used in this thesis a module often consists of one (main) class (for example a \rightarrow form).

RDBMS A Relational Database Management System is an application that manages relational databases, most often using SQL (Structured Query Language) as its user interface. Most business-applications are based on a database system for data storage.

Rapid (Application) Development is a style for application development that supports visual creation of the user interface and provides support for managing the code associated to the user-interface. Popular rapid application development environments beside Borland Delphi and C++ Builder are Microsoft Visual Basic or Sybase PowerBuilder.

Reusable Entity is a piece of software that can be reused in another project. This piece of software can be as small as a single function or as large as a whole application module.

Reuse Level The \rightarrow reusable entities in a toolkit can be categorized into different levels. The level defines common characteristics of the components. Section 3.4 describes the reuse levels used in this thesis.

Small Business Applications (SBA) In the context of this thesis the term small business refers to businesses with few employees. This has special impact on the requirements for an application in terms of security (complex security and access models are not needed in this case), availability (24 times 7 availability or real-time processing is not required), scalability (the application is not scalable to more than a few dozen users) and other requirements.

Toolkit The word *toolkit* is often used with several meanings. In this thesis it is a collection of components that are in some way related and can be used together or alone to solve a problem. In contrast to a \rightarrow framework a toolkit does not define a structure or the control flow that an application has to follow.

VCL Visual Class Library. The \rightarrow framework provided by the development environments Delphi and C++ Builder from Borland. Provides visual components for user interface creation and non-visual components for program control structures.

Object Inspector The object inspector is a part of the \rightarrow C++ Builder IDE that is used to visually change the properties and events of the parts of an application.

Appendix C

Bibliography

- [Aga00] Ritu Agarwal, Jayesh Prasad, Mohan Tanniru, John Lynch, *Risks of Rapid Application Development*, ACM, 2000
- [Ara89] Guillermo Arango, *Domain Analysis: from art form to engineering discipline*, ACM SigSoft, 1989
- [Ara94a] Guillermo Arango, *A Brief Introduction to Domain Analysis*, ACM SigSoft, 1994
- [Ara94b] Guillermo Arango, Martin Griss, Will Tracz, Mansour Zand, *Software reuse — issues and perspectives*, Proceedings of the ACM symposium on applied computing, 1994
- [Bar91] Bruce Barnes, Terry Bollinger, *Making Reuse Cost-Effective*, IEEE Software, 1991
- [Bec99] Kent Beck, *Extreme programming explained*, Addison Wesley Longman, 1999
- [Bey98] P. Beynon-Davies, S. Holmes, *Integrating rapid application development and participatory design*, IEE Proceedings, 1998
- [Ble01] Thomas Bleier, *A CVS plugin for Borland C++ Builder and Delphi*, Vienna University of Technology, 2001
- [Boe99] Barry Boehm, *Making RAD Work for Your Project*, 1999
- [Boo99] Grady Booch, Jim Rumbaugh, Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley Longman, 1999
- [Bor00] *Borland C++ Builder 5 developer's manual*, Borland, 2000
- [Cur99] William Curry, Giancarlo Succi, Michael Smith, Eric Liu, Raymond Wong, *Empirical Analysis of the Correlation between Amount-of-Reuse Metrics in the C Programming Language*, ACM, 1999
- [Fow99] Martin Fowler, *Refactoring: improving the design of existing code*, Addison Wesley Longman, 1999

Bibliography

- [Fra95] Robert B. France, Thomas B. Horton, *Applying Domain Analysis and Modeling: An Industrial Experience*, ACM, 1995
- [Fra96] William Frakes, Carol Terry, *Software Reuse: Metrics and Models*, ACM, 1996
- [Frs95] Steven Fraser, Honna Segel, Jim Coplien, Judith White, *Application of Domain Analysis to Object-Oriented Systems*, ACM, 1995
- [Gal95] Harald Gall, Mehdi Jazayeri, Rene Klösch, *Research Directions in Software Reuse: Where to go from here?*, ACM, 1995
- [Gam94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, 1994
- [Hen97] Scott Henninger, *An Evolutionary Approach to constructing Effective Software Reuse Repositories*, ACM, 1997
- [Jac97] Ivar Jacobson, Martin Griss, Patrik Jonsson, *Software Reuse: Architecture Process and Organization for Business Success*, Addison-Wesley, 1997
- [Jun00] Hyo-Taeg Jung, Dong-Kwan Kim, Young-Jong Yang, Sang-Duck Lee, *A Design and Implementation of Object-oriented Framework-based RAD Tool (IN-TRAD)*, IEEE, 2000
- [Kaf85] Dennis Kafura, *A Survey of Software Metrics*, ACM, 1985
- [Kar95] Even-André Karlsson, *Software reuse: A holistic approach*, Wiley, 1995
- [Kon99] Ray Konopka, *Introduction to Component Building*, <http://community.borland.com/article/0,1410,20569,00.html>
- [Koz96] W. Kozaczynski, *Categorization of Business Systems Components*
- [Kru95] Philippe Kruchten, *Architectural Blueprints - The "4+1" View Modell of Software Architecture*, IEEE Software, 1995
- [Lim94] Wayne C. Lim, *Effects of Reuse on Quality, Productivity and Economics*, IEEE Software, 1994
- [Lub91] Mitchell D. Lubars, *Reusing Designs for Rapid Application Development*, IEEE, 1991
- [Lun93] Chung-Horng Lung, Joseph E. Urban, *Integration of Domain Analysis and Analogical Approach for Software Reuse*, ACM, 1993
- [Mci69] M. D. McIlroy, *Mass-produced software components*, Proceedings at the NATO Conference in Software Engineering, 1969
- [Mor01] Peter Morris, *Creating Custom Delphi Components*, <http://delphi.about.com/library/bluc/text/uc080701a.htm>
- [Mur93] Robert B. Murray, *C++ strategies and tactics*, Addison-Wesley, 1993

- [Oes98] Bernd Oesterreich, *Objektorientierte Softwareentwicklung: Analyse und Design mit der UML*, Oldenbourg, 1999
- [Pri01] Margaretha Price, Donald Needham, Steven Demurjian, *Producing Reusable Object-Oriented Components: A Domain-and-Organization-Specific Perspective*, ACM, 2001
- [Pri90] Rubén Prieto-Díaz, *Domain Analysis: an introduction*, ACM SigSoft Software Engineering Notes Vol. 15, No. 2, 1990
- [Ran01] Nadeesha Ranasinghe, *History of Component Based Development*, <http://infoeng.ee.ic.ac.uk/~malikz/surprise2001/nr99e/article1/>
- [Rin97] David Rine, *Success factors for software reuse that are applicable across domains and businesses*, ACM SigSoft, 1997
- [Rob96] Don Roberts, Ralph Johnson, *Evolving Frameworks*, <http://st-www.cs.uiuc.edu/~droberts/evolve.html>
- [Sam97] Johannes Sametinger, *Software Engineering With Reusable Components*, Springer, 1997
- [Sha95] Mary Shaw, *Architectural Issues in Software Reuse: It's not just the functionality, it's the Packaging*, ACM, 1995
- [Str97] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1997
- [Tra94] Will Tracz, *Software Reuse Myths Revisited*, IEEE, 1994
- [Zub00] John C. Zubeck, *Implementing Reuse with RAD Tools' Native Objects*, IEEE, 1997

Appendix D

Internet references

This chapter lists all references to web sites that were mentioned in this document, apart from the papers in the bibliography.

- BorCVS, a CVS plugin for C++ Builder and Delphi
<http://borcvs.sourceforge.net/>
- Borland, the makers of Delphi and C++ Builder
<http://www.borland.com/>
- CVS, the concurrent versions system
<http://www.cvshome.org/>
- DUnit, a port of JUnit to Delphi
<http://http://dunit.sourceforge.net//>
- Doxygen, a source documentation system
<http://www.doxygen.org/>
- EasySoft, the makers of Universe FreeLine, an accounting software
<http://www.easysoft.de/>
- Firebird, an open-source sister of InterBase
<http://firebird.sourceforge.net/>
- IBX or InterBase Express, client components for accessing InterBase
<http://codecentral.borland.com/codecentral/ccweb.exe/author?authorid=102>
- InterBase, a relational database engine from Borland
<http://www.interbase.com/>
- JUnit, an extreme programming testing framework
<http://www.junit.org/>
- QuSoft, the makers of QuickReport, a report builder for Delphi/C++ Builder
<http://www.qusoft.com/>
- Sage KGK, the makers of PC Kaufmann, an integrated commercial application
<http://www.sagekhk.de/>

- UML, the standardized modeling language
<http://www.omg.org/uml/>

Vienna University of Technology
Distributed Systems Group
Argentinierstraße 8 / 184-1, A-1040 Vienna, Austria
<http://www.infosys.tuwien.ac.at/>

Thomas Bleier
Hauptstraße 1, A-7372 Weingraben, Austria
thomas@bleier.at, <http://www.bleier.at>