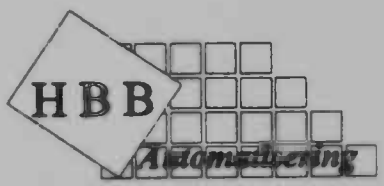
*University* of Groningen **Faculty of Mathematics and Physical Sciences**

**Department of Computing Science**






# Introduction of a Development *with* Reuse Model into the Development method of HBB Automatisering

Johan Bennink

begeleider: E. Dijkstra

augustus 1995



RuG

# Introduction of a Development *with* Reuse Model into the Development method of HBB Automatisering

**Johan Bennink**

**8 August 1995**

Distribution: J. Bennink
R.G. Heller
M.R.S. de Boer
Archive HBB Automatisering (2 copies)
Rijksuniversiteit Groningen (15 Copies)

Commissioned by: HBB Automatisering
Author: J. Bennink
Date: 8 August 1995

# Summary

In the research presented here the development method currently used by HBB Automatisering is extended with Reuse of code. The current development method is very ad hoc and does not produce any form of system documentation; design, implementation and testing are not seen as separate phases. To prevent problems with the maintainability of the applications, the development method will be modified using software engineering techniques. The introduction of reuse of at the implementation (code) level is the first step in a series of projects that are planned to improve the development method of HBB Automatisering.

The design phase is formalized by the introduction of design documents. These documents give a description of the components that are part of the application to be developed. Each document contains a full description of the component, a pseudo-code design and a test plan for the component. The implementation phase uses the design documents as the basis of the implementation. After the implementation is completed, the components are tested using the test plan specified in the design document. All components are integrated by an appointed (lead) developer. The lead developer makes sure that all requirements are met and that all functionality is present.

Reusable components are stored in a software repository along with a set of attributes that are used to identify, classify and describe the components. A tool is supplied to assist the developer during the design phase with locating components and accessing the component attributes.

An implementation of the model for the Visual FoxPro platform is described as well as an evaluation of this implementation. The results of this evaluation are discussed and conclusions about the usefulness of the model are drawn.

J. Bennink

Groningen, July 1995

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

The research described in this thesis has been carried out in fulfillment of the requirements for a Masters degree in Computer Science at the University of Groningen.

The goal of the research was the introduction of reuse techniques into a development method. The development method of HBB Automatisering (HBB), a software company which I founded with two fellow students, was used to investigate the possibilities of introducing reuse techniques.

This current development method used at HBB is not a textbook example of the use of software engineering techniques, but rather an accumulation of ad hoc techniques that form a surprisingly successful development method. Creating good — in the sense of working properly — applications is one thing, being able to provide long-term maintenance and support is another.

HBB Automatisering realizes that a long-term commitment to their applications is crucial for its clients. Therefore, several projects are foreseen — of which this is the first — to improve the development method using software engineering techniques. Reuse of code was chosen as the first project because the current development method is very much based on direct implementation.

## 1.2 Contents of the Thesis

In chapter 2 a brief discussion of the background of software reuse is given. The next two chapters, 3 and 4, describe the current and the revised development method respectively. The discussion of the current development method is very short because it uses very little software engineering techniques and is mainly based on ad hoc procedures. The implementation of the revised development method for the Visual FoxPro platform is described in chapter 5. An evaluation of the revised development method is given in chapter 6. For this evaluation a prototype application was built and a modification of the prototype was made. Finally, chapter 7 gives some concluding remarks and directions for future reseach.

# Chapter 2

# Background

Software reuse has attracted increasing attention over the past years and is now of major interest. The term *software reuse* is applied to many techniques, methods, and processes. It is the reapplication of all kinds of knowledge about one system to another similar system. The aim of software reuse is to (i) increase the quality of the created systems and (ii) to reduce the development and maintenance effort, both in time and cost. This chapter is intended to give the reader a short introduction into the field of software reuse.

## 2.1 Informal and Formal Reuse

Informal reuse has been used for as long as computers have been around. It occurs when a software component, not originally designed and implemented with reuse in mind, is reused. Generally, in such a case, some modifications are required to adapt the component to the new application. This form of reuse is also called *code-grabbing* or *software salvaging*.

Formal reuse is the (re)use of software components that have been specifically designed, built, archived and catalogued for reuse.

## 2.2 Reuse Challenges

The challenges associated with implementing software reuse can be divided into two categories: managerial and technical. What follows is a brief summary of some of the challenges; a detailed discussion on this subject is beyond the scope of this thesis. A discussion on reuse challenges and inhibiting factors can be found in Reed [16], Biggerstaff and Richter [4] and Tracz [20].

### 2.2.1 Managerial Challenges

The introduction of a reuse program into the development process can, in the long run, result in a substantial reduction of the overall development costs. Introducing such new techniques means a large investment of company resources. Developing a successful reuse program is a nontrivial investment that does not have an early payoff. The organizational structure of most companies precludes such capital investments regardless of the potential long-term payoff.

A reuse program should therefore be initiated at a time that the company is not involved in projects that are under deadline pressures. Furthermore, it should be a top-down effort. Higher management has the influence and drive to follow through on long term goals such as improving the development process.

It should also be understood that initiating a reuse program can mean different things. It could mean simply buying reusable component libraries trying to capitalize on high-quality libraries. It could also mean initiating a reuse program to motivate developers to build reusable components instead of starting from scratch with each new project. This would result in the creation of a central repository that can be utilized and maintained company-wide. The goal here would be to increase the productivity of the entire company in the long term. It is very important to set realistic goals for the reuse program. Is it possible to reuse across projects? How repeatable is the software process in general? These are just some of the questions that need to be answered before realistic goals can be set and investments made.

For reuse to be successful developers must be encouraged to find and reuse existing components rather than write new components from scratch. This attitude must be promoted by the higher management or it will not be successful. Measuring productivity will be different when employing reuse techniques. Developers should not only be credited for finishing a project on time, but also for how many components were reused. This could be achived by tightening the requirements for component size and maintainability of the component. Both criteria are greatly influenced by an increase in the amount of reuse. If *development for reuse* is not a separate discipline developers should also be credited for how many new — reusable — components they write. Developers could even be credited each time that their reusable components are reused.

### 2.2.2 Technical Challenges

Depending on the size of a software repository it may be impossible for a developer to know the details of all components that are available for reuse. To overcome this problem it should be possible to somehow locate reusable components. Furthermore, enough information on the components should be available to understand how the components work and how they are to be used. Preferably there should be a way to share information about the use of the components. Developers must be encouraged to find and reuse components, this will involve supplying several techniques and tools to aid the developer.

The use of reusable components will only be beneficial if the components are of a guaranteed quality. Quality is not limited to defect-free code, but also includes the existence of design information, testplans, documentation, checklists and examples of how to use the components. Standards should be defined to obtain a consistent layout of this information across the entire repository.

Change management is another important challenge. Modifying th structure of interface of a library after it has been released will in most cases cause great difficulties for the users of the library. Whereas exstensions of a library generally do not form a problem. Using a library is equivalent to using a programming language. Most programmers do not welcome changes in a programming language!

## 2.3 Using Reuseable Components

Development of reusable components does not involve any new techniques. The techniques and tools used for traditional application development can be maintained although standardization is very important. The techniques and tools for using reusable components however are much less known. Before a reusable component is available for reuse it will have to be archived in some form. Information about the component will have to be catalogued and there will have to be a tool that enables a developer to locate and understand the component. How reusable components are reused depends on the way that the components are stored.

The aspects of reusing a component discussed here depend heavily on the number of reusable components in the repository. For small repositories locating a component will in most cases be as simple as going through a list of all components, whereas for large repositories examining all components might be a time consuming task.

### 2.3.1 Archiving

Traditionally, archiving is performed on the basis of source files and libraries. Hierarchical directories add a mechanism for efficient access and navigation. Because one of the primary challenges associated with software reuse is change management, some archiving methods use a version control system to help maintain reusable components.

Software repositories add a higher level of support for accessing components and change management than the traditional archiving methods. The essence of software repositories is to provide a standard, integrated way of managing all types of software engineering data, whereas traditional archiving methods are mostly restricted to the source-code.

### 2.3.2 Cataloging

Cataloging involves creating a database of information about the reusable components in the repository. One way to catalog components is by classification based on a set of attributes. The attributes used in the classification should accurately describe the reusable component. Such techniques are only suitable for components that have been specifically developed for reuse.

### 2.3.3 Locating

Depending on the way the components are catalogued there are several ways of locating the components available to the developer. Information retrieval systems for example offer techniques for querying, such as boolean expressions and proximity matching. If a catalogue is used based on a classification mechanism then searching for reusable components involves providing values for one or more classification attributes which may match one or more reusable components.

### 2.3.4 Comprehending

To decide whether a reusable component provides the needed functionality, the developer should be aided by tools that provide information about the reusable components such as functionality, dependencies and any other information required to allow a reusable component to be selected.

An example of such a tools is the class browser. The class browser allows a developer to inspect the properties and methods of classes. Some class browsers can even give information about the procedures and methods called from a specific method, or a list of procedures calling the method.

Of course, these tools rely on the presence of some form of documentation. Without good documentation a developer will have a hard time comprehending the component. In fact, it can be argued that undocumented components are not suitable for reuse.

### 2.3.5 Retrieval

How a reusable component is retrieved depends heavily on how the components are archived and whether the relation between the repository and the reused component should be kept intact. If the component is stored as a source file it can be retrieved by simply copying, or referencing the source file. If a version control system is used retrieval may involve a 'check out.' Even more difficult means of retrieval may be necessary if the components are stored in a repository. A software repository however, often provides the benefit of tracking who has (re)used a given reusable component.

## 2.4 Concluding Remarks

The opening paragraph of this chapter mentioned that software reuse is *the reapplication of all kinds of knowledge about one system to another similar system.* The main topic of discussion in this chapter — apart from a short excursion into the subject of managerial challenges — has been the reuse of code. Biggerstaff and Richter [4] note that the payoff for code reuse quickly reaches a ceiling, and they are supported by several other authors such as Horowitz and Munson [8] and Neighbors [13].

So why is reuse of code the main topic of this chapter, and in fact of the entire thesis? In section 1.1 the most important reason was already given. The current development method used at HBB is very implementation oriented. Maintenance and support for the applications developed using this development method will become more and more difficult, and costly. HBB would therefore like to change the development method to be able to give guarantees to customers about support. To make the transition as smoothly as possible the techniques are introduced one at a time, starting with the *Design and Implementation* phase.

### 2.4.1 Reuse as we see it

The use of definitions for terms such as *component* differs somewhat from other literature. Therefore it is important to discuss some of the definitions used in this thesis.

**Development for Reuse** During application development the resources are directed to solving the problems at hand, ie. developing the application within the available time, and for the agreed amount. Whereas Development *for* Reuse depends on higher initial costs, both in time and money, that will be 'repayed' in future projects. This strategy is in conflict with the project goals of application development. Therefore Development *for* Reuse will be considered a separate discipline.

**Development with Reuse** This is what the research is about. Currently reuse is achieved in the form of *code grabbing*, ie. cutting existing code from previous projects and pasting the code into a piece of a new project. However, the purpose of this research is to increase the quality and maintainability of the applications, not the introduction of reuse techniques on its own.

**Component** The term *component* is used to denote both application components, made specifically for one application, and reusable components. Furthermore, a component usually denotes a procedure of function. In this thesis the term *component* is used to denote procedures, functions, modules, and even sub-systems. Especially when the term *component* is used to denote a reusable component this extended definition applies.

# Chapter 3

# Current Development Method

The current development method does not use explicit life cycle phases. In fact, application development is conducted by ad hoc design based on the functional specification followed by immediate implementation and testing of the software components. A full description of the current development method is presented in Bennink [2].

To help visualize the development method we will first introduce a software life cycle, shown in figure 3.1, that represents the current development method used at HBB.

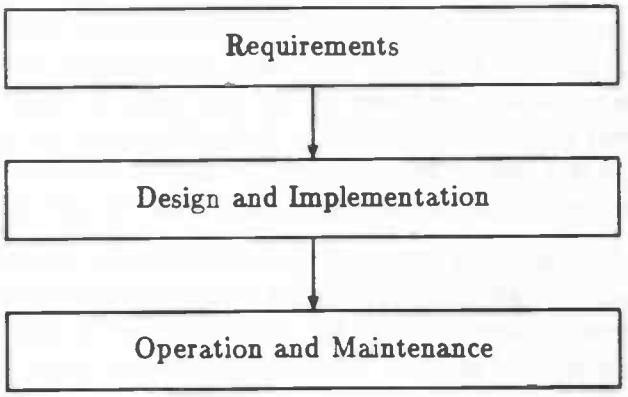



Figure 3.1: Software life cycle representation of the current development method.

Application development is carried out in the form of projects. Each project uses the life cycle model to divide the application development into phases. Each phase contains tasks which have to be performed and deliverables which have to be produced.

## 3.1 Requirements

The *Requirements* phase is used to obtain an understanding of the problem of the customer. This is achieved by conducting interviews with the management and the future users. These requirements are used to create a functional specification and a global decomposition. The deliverables for the *Requirements* phase are:

**Requirements Specification Document** The Requirements Specification (RS) document is the receptacle for all requirements — both functional and non-functional — gathered during the interviews with the customer and the future users. The requirements might be contradictory, redundant, disordered, ambiguous, possibly even incorrect and perhaps — but hopefully not — incomplete. The requirements can be stated in natural language or any other technique for representing the requirements, provided it is understandable for the customer.

**Functional Specification Document** The Functional Specification (FS) document is stated in natural language and contains an adequate statement of needs that are all demonstrable. The specification should be clear enough — and sufficiently detailed — to allow the customer to understand it, and allow the developers to proceed from its definitions. Moreover, the FS should be unambiguous and consistent.

**Global Decomposition** The functional specification is used to divide the specified subsystems — each containing several software components — among the developers. Each developer is allotted an equal share of the subsystems that have to be implemented. This division of labour is based on implementation effort, not on design decisions. Therefore the global decomposition is not seen as part of the design phase.

## 3.2 Design and Implementation

In this phase each developer is working on the software components that were allotted to him during the global decomposition. From the functional specification of the software component the developer starts implementing based on the first possible design solution, or partial solution, of the component that comes to mind[1]. Any unresolved parts of the solution are designed and implemented in a similar fashion. After completion of a software component the developer tests the validity of the implementation against the FS and RS documents.

For each project a *Lead developer* is appointed. The lead developer is responsible for collecting and compiling the finished software components to a complete application. As a consequence, a prototype of the application developed so far can be shown to the customer for testing very soon after implemenation has started. The implementation phase is concluded with the acceptance of the application by the customer.

Due to the implicit nature of the *Design and Implementation* phase it is not possible to describe this phase in more detail.

## 3.3 Operation and Maintenance

This phase is not really part of the application development process. It is a phase that takes place after the application has been developed and is officially delivered to the customer. The 'Maintenance' part involves correctional maintenance only, it does not involve adding functionality. Functional enhancements are added in a new — separate — project with it's

[1]Please note, this is how application development is currently conducted, not necessarily how it should be done.

own life cycle based on improvement suggestions. The deliverables for the Operation and Maintenance phase are:

**Improvement suggestions** Improvement suggestions resulting from the use of the application by the customer which lead to follow-up projects.

**User Error reports** Basically error reports are classified by HBB into two categories: 'do it for a future release' or 'do it now.' The 'do it now' changes get done right away, because delay is — for some reason — unacceptable. The modified application is sent to the customer right away. The less urgent ones are gathered first, and are fixed after some period of time.

# Chapter 4

# Revised Development Method

Reuse can be conducted across the entire software development process. To prevent the introduction of too many changes at once the current investigation is limited to reuse of source-code. The changes made in the development method will only effect the *Design and Implementation* phase since it is the only phase involving the creation of source-code.

## 4.1 Life Cycle Model Changes

To emphasize the changes made in the *Design and Implementation* phase, the life cycle will be modified. Design and Implementation will be viewed as two separate phases in the new situation and Testing will be added as a separate phase. The revised life cycle is shown in figure 4.1.

Figure 4.1 might give the impression that the Design, Implementation and Test phases are carried out one after the other. In fact these phases are carried out independently for each software component. After the component is tested it is integrated into the application developed so far, which is why this method is called *incremental integration*. The software components do not go through the phases simultaneously, but proceed at their own pace, ultimately — on completion and integration of the last component — resulting in the finished application.

Since the *Requirements* and *Operation and Maintenance* phases remain unchanged they will not be discussed here. Before discussing the phases that will be modified, the next section introduces a software repository.

## 4.2 The Software Repository

For reuse to take place during application development, there must be a collection of reusable components, and developers should be aware of their presence. Two kinds of reuse can be distinguished: sharing of newly-written code within a project and reuse of previously-written code on new projects. Similar guidelines apply to both kinds of reuse. The latter kind of reuse is accomplished by a global repository. To enable reuse of the first kind, a local repository — with the same structure as the global repository — is used to store the components made specifically for one project. By using the same hierarchy as the global repository it should be

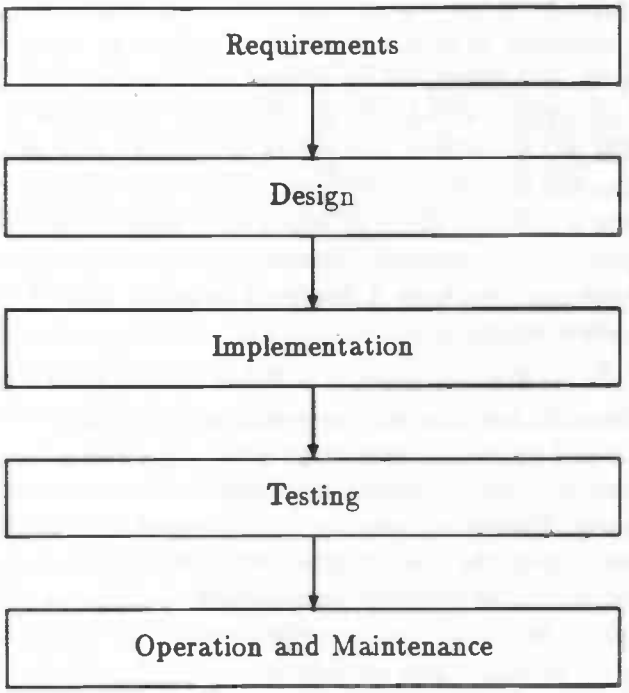



Figure 4.1: Software life cycle of the revised development method.

possible to classify these specific components for reuse, possibly incorporating them — after some generalization — into the global repository. In this model, Development *for* Reuse is considered a separate discipline from Development *with* Reuse. Evaluating and generalizing components is a resource intensive process. In many cases application development is under severe pressure of company resources. Development *for* Reuse should therefore be conducted at a time that the company is not involved in projects that are under deadline pressures, or by a part of the company that is not influenced by deadlines.

The current discussion is only concerned with the global repository; a local repository could however be maintained in much the same way as the global repository. In fact, it might be preferred to do so because this reduces the effort needed to turn application specific components into reusable components. Besides distinguishing between global and local repositories it might be a good idea to create repositories, or distinguish between components in one repository, for specific problem domains.

Reusable components can be stored in many different ways. A software repository does not only contain the reusable components themselves, but also stores information on the components such as keywords, documentation, testcases, checklists and examples of how to use the component. This information allows a developer to search for a component, and access information on the component in order to comprehend its functionality.

### 4.2.1 Repository Structure

The repository can be structured in several ways. Burton et al. [5] describe an integrated environment for software reuse based on their reusable software library (RSL), a relational database

approach. Arnold and Stepoway [1] present additional background information regarding the desirable properties of a repository for reusable software. The emphasis of their approach is on customization and the mapping of retrieval information into an information retrieval (IR) database. Another IR approach is described by Frakes and Nejmeh [10]. Their paper focuses on the reuse library built by using CATALOG, an IR system developed at AT&T. The paper concludes with some trends in IR research and development likely to improve IR systems as tools for reuse and includes the formats of the module and function prologs as well. Prieto-Diaz and Jones [15] propose a different strategy based on a faceted classification scheme. The Asset Management Program (AMP) consists of several groups of people that work together to create, maintain, and manage the Asset Library. The paper describes the organizational structure and the classification scheme.

The repository structure chosen for this method is based on the RSL approach from Burton et al. [5]. The first classification mechanism is a repository identification code to specify the category of the component and its hierarchical relationship to other components. To keep the tree structure shallow — and the identification code tractable for use in the design phase — the tree hierarchy is defined using three hierarchical levels. Although more levels are not prohibited it is strongly recommended to keep the tree structure shallow. The second mechanism permits descriptive keywords to be associated with the components. Unlike the RSL approach there is no limitation to the number of keywords. A controlled vocabulary as proposed by Prieto-Diaz and Freeman [14] to prevent duplicate and ambiguous keywords will be used.

The hierarchical levels are obtained from a distinction of application components. The repository will be limited to one implementation language, the hierarchical levels are based on that language. Each implementation language uses a separate repository. The tree structure with three levels is not required for all repository components. The three levels of the repository hierarchy are:

**Global Component Level** The global component level categorizes components based on the application components distinguished in the implementation language. An example of such a categorization for Visual FoxPro (VFP) would be the distinction between menus, forms, reports, labels, (auxiliary) procedures and form controls.

**Detailed Component Level** This level is a more detailed categorization of the global component level categories. For example, the global component category **controls** could be categorized into: descriptive labels, textboxes, buttons, listboxes, checkboxes, optionboxes, and grids. One could also distinguish between the type of variable that is returned, or the type of operation performed by the component. For this research the first categorization is chosen because it more closely resembles the current practice.

**Functional Component Level** At the functional component level the categorization is based on functionality. For example, textbox controls might be distinguish categories for: personal data, address data, product data, identification codes, etc. Every level below this functional component level is considered a functional component level as well.

### 4.2.2 Repository attributes

The component information stored in the repository is gathered during the component design from the design document (see section 4.3.3.) Depending on the component type not all information is used. The component information stored in the repository is shown in table 4.1. The components 1, 2 and 3 are used to identify a component. Attributes 4 to 8 are used to classify components, the other attributes are descriptive.

The reusable component code is stored in a format usable by the implementation language and is determined by the archiving method. A single file can be used to store the component code, or multiple components can be grouped into one library file. Depending on the language a special type of library file can be used, or the code could even be stored in a database.

Reusable components go through a life cycle of their own. The repository may contain several versions of the same component. Usually the new versions will be bug-fixes, or enhancements of the quality, efficiency or functionality. It is allowed to use older versions of components provided that the components are not marked 'Disabled'. Components marked 'Disabled' have bugs, or use unsupported features and their reuse is therefore prohibited. The next (maintenance) release of an application using the 'Disabled' component will be forced to use another component, i.e. will not compile without specifying a replacement component.

Components marked 'Obsolete' should no longer be used. They do not present any problems, but newer — better — components have been developed to replace these components. Better in the sense that the new components are of a better quality, are more efficient, or more generic. The (re)use of 'Obsolete' components for new projects should be strongly discouraged, if not prohibited. Reuse of these components in existing projects should not be prohibited, although some form of discouragement could incite developers to use a replacement component.

A status description is stored in the repository to aid the developer. It contains the reason for the component's status, and possibly a reference to a replacement component.

Although it is possible — and for large projects it may be essential — to use the repository for the (local) application repository as well, the model presented here does not use a repository for (local) application components. The design documents from the local components should provide adequate information to enable reuse within the project.

### 4.2.3 Repository Browser

The Repository Browser is an application that allows a developer to search for components in the repository using on-line querying. A repository with thousands of useful components of which a developer is not aware that they are at his/her disposal is useless. A developer should always have some global knowledge of the components found in the repository, however, as the number of components in the repository grows it may not be humanly possible to know of all components that are available. The repository browser can be used to search for reusable components or to access component information without knowledge of the underlying hierarchy of the repository. The repository browser could serve as a maintenance tool too, allowing component attributes to be changed, such as Component status.

Although the repository browser can assist a developer in locating a component it is not a replacement for a thorough knowlegde of the repository. Without prior knowledge of the available components using the repository will be a very time-consuming process. The repos-

1. Repository identification code. This is a code specifying the complete hierarchy path of the component.
2. The name of the software component.
3. A version number.
4. The base class.
5. A number of keywords that identify the component based on its functionality.
6. The author of the software component.
7. The component type.
   - Function. A function. This type is added mainly for backward compatibility with older, existing, libraries.
   - Class. An OO class definition.
   - Datastructure. A description of a datastructure. In most cases a database structure description.
   - Documentation. This can be a programming standard, component description, or usage description, or any other kind of relevant documentation.
   - File. Any other file. A complete description of the use, and structure is given in the description attribute.
   - Method/Event. This type is used to denote the method/event code of classes.
8. Component status. The status of a component is retained in this attribute. Possible status values are:
   - **Enabled**. This component can be reused without restrictions.
   - **Obsolete**. Reuse of this component should be discouraged.
   - **Disabled**. Reuse of this component is prohibited. Any application currently using the component should be modified, preferably with the next release.
9. Status description.
10. A description of the parameters.
11. A description of the — optional — return value.
12. The properties.
13. A list of methods/events.
14. A brief textual description of the functionality.
15. The component code, or a reference to its location.
16. References to documentation and checklists.
17. An example of the use of this component.

Table 4.1: Component attributes stored in the repository.

itory browser merely supplements this knowledge by displaying detailed information on the components.

## 4.3 Design Phase

The design phase is performed individually for each function stated in the Functional Specification (FS) document. Several solutions should be proposed, of which one is chosen for implementation in the next phase.

### 4.3.1 The Design Process

To sucessfully introduce reuse it is necessary to eliminate ad hoc design. The design process should be modified in such a way that the developers design solution is captured in some form or another

To formalize the design process a developer is required to use a *design document* (see section 4.3.3) for each software component. Each new — auxiliary — component that the developer uses during the software component design should be specified in a separate design document. The use of reusable components does not require a separate design document because a design and implementation of the reusable component is already present. The effect hoped for is that a developer will invest time in trying to find and specify a reusable component instead of creating new — ad hoc — components during the design process. However, it is not a requirement that the design contains at least a minimum number of reusable components. This would seriously interfere with the prime objective of the design phase, i.e. designing a solution to the functions stated in the Functional Specification (FS) document and the requirements from the Requirements Specification (RS) document.

The appointed *Lead Developer* should be given the authority to enforce the correct use of the design documents. When design documents are misused by developers the lead developer should be authorized to reject the design. Enforcing correct use of the design documents could be made part of the design validation.

### 4.3.2 Design Validation

Validation of the design is very important. Undetected errors that are carried forward to the implementation phase and remain undetected until the test phase can be very expensive to correct. They may require a complete redesign and reimplementation of parts of the application.

After the developer has designed the software component, the FS and RS documents are used to validate the design. Each functional requirement should be met by a part in the design. Moreover, for each part in the design there should be a motivation in the FS and/or RS document. This prevents the design solution from being overcomplete. Besides validating the design using the Functional Specification and Requirements Specification documents the lead developer inspects the design documents to see whether they are used correctly. Misuse of the design documents does not necessarily mean that the design is wrong, but it may lead to a degradation of the maintainability of the resulting application.

### 4.3.3 The Design Document

The design document is used to retain vital design information about the software component. This information is retained in free-form natural language format, which allows the developer to describe the functionality and choices of the component in detail. The design document is divided into two sections: Component Attributes and Component Design.

**Component Attributes** Each software component is described by several attributes, shown in table 4.2. A distinction is made between a function and a class definition. Although this distinction is introduced rather early in the design process it should not lead to any problems. Moreover, by forcing the developer to think about this distinction early in the design process, the resulting implementation will not comprise large sets of separate — but related — functions that should have been turned into a class.

| Attribute | F/C | Description |
|---|---|---|
| Name | F,C | The name of the software component. |
| Version | F,C | A version number. |
| Base class | C | The base class. |
| Keywords | F,C | Several keywords that identify the component based on its functionality. |
| Author | F,C | The author. |
| Parameters | F | A description of the parameters. |
| Properties | C | The properties. |
| Methods | C | A list of methods. |
| Events | C | A list of (re)defined events. |
| Return value | F | A description of the — optional — return value. |
| Description | F,C | A brief textual description of the functionality. |

Table 4.2: Software component attributes. F and C denote the use of the attribute for a Function or Class respectively.

If the component is turned into a reusable component using Development *for* Reuse, then the component attributes can be used to fill the corresponding repository attributes. As was mentioned earlier, Development *for* Reuse is considered a separate discipline because of the conflict of interests with regular application development.

**Component Design** There are several design methods and notations, some more suitable for specific purposes than others. The notations considered for use during the component design are listed in table 4.3, an extensive discussion of design notations is given by Macro [11], Charette [6] and Sommerville [17].

For the proposed development with reuse method the pseudo-code notation is chosen. Most developers are familiar with pseudo-code, and resistance against the use of pseudo-code is low because it allows the developer to think in terms of an implementation. However, this is alos one of the major drawbacks of this notation. Developers should be educated not to mistake the use of pseudo-code as a sign that implementation is imminent. Unlike Macro [11] this method does not restrict the use of pseudo-code to the level immediately 'above' (i.e. before) implementation.

- Jackson System Development (JSD)
- CCITT Structured Design Language (SDL)
- Petrinets
- Design Structure Diagrams (BSI 6224)
- Structured Analysis and Design Technique (SADT)
- Structured Data Flow charts
- Structure charts
- Pseudo-code

Table 4.3: Different design notations.

The pseudo-code language syntax should provide a higher degree of abstraction than an implementation language. The help with the acceptance of the model the major program control-flow structures should at least be present in the pseudo-code language. Currently, the developers are very implementation oriented, switching to a notation with a formal nature will probably meat much resistence amongst the developers.

Reusable components used in the design can be specified using the repository identification code, component name and version number of the component. Using this approach the reusable components specified in the design are clearly distinguishable from the other pseudo-code.

The component design also contains the pre- and postconditions of the component, and if any, the assumptions, simplifications and limitations in the design of the component. A test plan describing the tests and their required outcome to test the correctness of the implementation of the component is also included.

## 4.4 Implementation Phase

After the design phase is completed and one of the suggested solutions is chosen, the developer can proceed with the implementation.

### 4.4.1 The Implementation Process

The implementation of the software components is carried out on the basis of the design document. The pseudo-code designs from the design documents are implemented in the implementation language. Although this is mainly a creative process carried out by the developer the design document should serve as a blueprint during this process. Ie. no design alterations are allowed in this phase. Furthermore, the defined programming standards (see appendix A) should be followed to guarantee consistency with other application code.

The method used to integrate the reusable components specified in the design depends heavily on the implementation language, but almost all languages provide some mechanism to include additional code. Languages such as Pascal use include files, more modern languages such as Visual FoxPro use external (compiled) modules, or units. Languages like C use external declarations, and the reusable components are added during the linking process. Checklists supplied

with the reusable components should be used to check for correct use of the components.

However simple this may sound, the implementation of a design can be very difficult. We will not discuss the pitfalls of implementing designs here since it is beyond the scope of this investigation. A brief discussion of implementation pitfalls is given by Charette [6].

## 4.5 Test Phase

Component testing is concerned with inspecting or executing a component with the intent of finding errors. Testing and debugging are sometimes considered the same thing. This is not true, they are related but different. Testing is used to detect the presence of errors while the object of debugging is to locate the position of an error and correct it.

Testing can be conducted by the author of the component (author testing), or by another developer (adversary testing) using static or dynamic methods. To test the application to-be a different testing method is used, called integration testing.

**Static testing** Static testing is a disciplined review and analysis of the application design and code. It is useful for finding logic errors in an application, as well as questionable programming practices. Static testing includes manual code inspections, structured walk-throughs, static path analysis and other techniques that don't require the application to be executed.

**Dynamic testing** Dynamic testing is the execution of the code under controlled conditions to observe the results. A well-defined test plan is usually a necessity when using dynamic testing as the tests are derived from the specification, and are checking for particular results.

Both static and dynamic testing can be undertaken from either a white-box or black-box perspective. Black-box testing does not use any of the internal design knowledge of the component, but relies on what the external behaviour of the component should be. Inputs are specified, and outputs are observed. White-box testing is based on the internal logic of the component.

**Integration testing** Somewhere during the development of the application the modules will have to be assembled to form the complete application. One approach is to wait until all modules have been tested, and then combine them into the application and test the application in its entirety. Another strategy is to produce the application in increments of tested units. Modules are integrated and tested in small sets. Once the modules are correctly integrated new modules are added, and tested in combination. Testing is completed when the last module is integrated and tested.

The test phase presented here uses all three testing methods. Testing is conducted at two levels in the development process; at the software component level and at the application level. Application development proceeds in an incremental way by adding software components to a prepared framework. This framework is considered the application to-be until the last software component is integrated.

After all application components have been integrated one more set of tests is used, the acceptance or certification test. If the tests are passed, the application is officially complete.

Due to the use of actual data throughout the implementation and test phase the acceptance test is — in most cases — a formality. The acceptance test should show that the application meets all requirements stated in the RS and FS document. The specification for this test is prepared during the design or implementation phase.

### 4.5.1 Software Component Testing

Software component testing is carried out by the implementor of the component. First a static test is performed — in the form of code reading — to determine the correctness of the implementation using the RS and FS documents. If no errors are detected a dynamic test is performed using the tests specified in the design document. For large components, such as modules or sub-systems[1], code inspections will be used.

### 4.5.2 Application Testing

Application testing is conducted by the lead developer. This type of test is also called integration test. The lead developer is responsible for integration of the developed software components and for testing their validity. At this level the testing process is only involved with finding errors in the implementation, not correcting them. The *lead developer* conducts a black-box test of the application modules. Any errors found by the lead developer are reported to the implementor of the component containing the error. When the developer of the component is convinced of the correctness of his implementation a white-box test is performed. Although this could be considered debugging the aim of this test is not to fix the problem, but to locate it. Once the problem is located the approriate developer is instructed to fix it.

The lead developer is responsible for making sure that all functions specified in the FS document are implemented correctly and that the requirements stated in the RS document are all met.

[1] Please note, the definition of a software component used in this thesis differs from definitions found elsewere in the literature. A software component can be a procedure, a module and even a sub-system.

# Chapter 5

# Implementing the Revised Development Method for Visual FoxPro

During the implementation of the revised development model — from January to June 1995 — Visual FoxPro was only available as a Beta version. Visual FoxPro was chosen, instead of the previous version (FoxPro v2.6a), because of the new powerful features. Visual FoxPro has an active data-dictionary and is fully object oriented. The latter simplifies the introduction of the proposed reuse techniques.

The revised development method is described in the order of the life cycle phases. Since the Requirements, and Operation and Maintenance phases are not changed by this investigation they will not be described, see section 3.1 and section 3.3 for a discussion of these phases. The first section of this chapter describes the software repository.

## 5.1 The Software Repository

In this section the implementation of the software repository will be described. The structure of the repository hierarchy and the repository browser will be described and the techniques used to create complex functional components. The components supplied with this first implementation of the development method will be listed as well as the concepts used in their implementation.

### 5.1.1 Repository Structure

Although this discussion of the repository structure is restricted to the Visual FoxPro platform it should be possible to adopt it to other platforms. The repository hierarchy is constructed from the different language components. For Visual FoxPro this leads to a repository hierarchy with categories as shown in figure 5.1.

The repository hierarchy levels are implemented using MS-DOS directories. The implementation of the libraries depends on the library category, table 5.1 gives a list of library categories and the type of library stored in that category for the Visual FoxPro implementation.

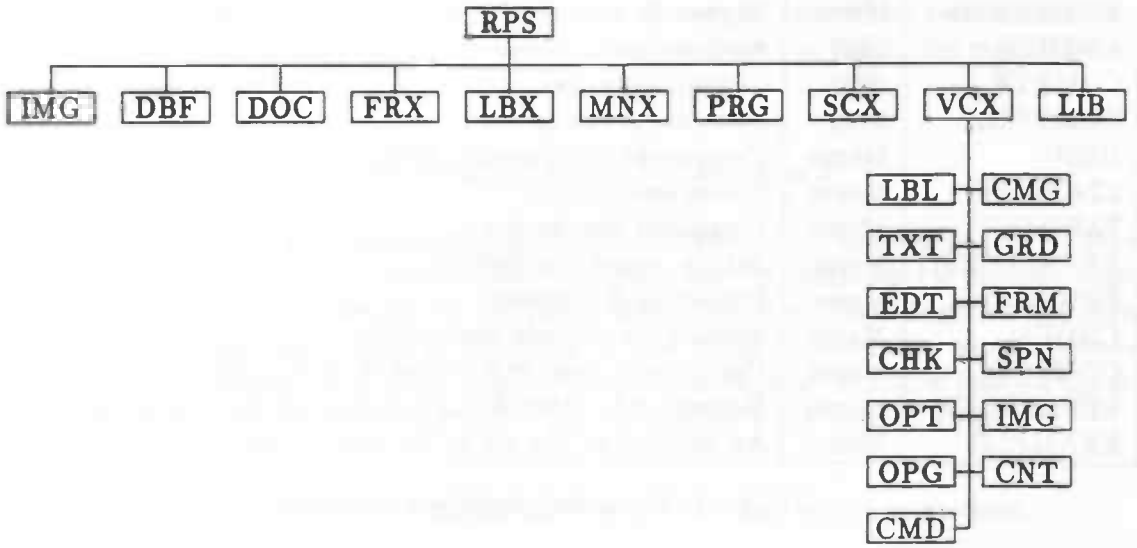



Figure 5.1: Base repository hierarchy.

| | | |
|---|---|---|
| IMG | Images | BMP/ICO/CUR images and pictures used for button and image controls. |
| DBF | Databases | DBC/DBF databases and tables. The tables stored here are used as templates. For example tables with lookup values are stored here. |
| DOC | Documentation | ASCII, Latex or WordPerfect documentation describing programming standards and other conventions. |
| FRX | Reports | FRX report files. |
| LBX | Labels | LBX label files. |
| MNX | Menus | MNX menu files. |
| PRG | Procedures | PRG procedure files. |
| SCX | Forms | SCX form files. This category is used to store standard dialogs only. Template forms should be stored in the VCX\FRM category. |
| VCX | Classes | VCX library files. Mainly form controls. |
| LIB | Libraries | FLL/DLL/OCX 3rd party libraries. |

Table 5.1: Visual FoxPro library categories and their contents.

| | | |
|---|---|---|
| RPSID | C(100) | Repository identification code. |
| COMPNM | C(20) | Component name. |
| VERSNR | N(6,3) | Version number. |
| BASECLASS | C(20) | Baseclass. |
| KEYWORDS | Memo | Keywords. |
| AUTHOR | C(8) | Author. |
| COMPTP | N(3) | Component type. |
| COMPSTAT | N(3) | Component status. |
| DESC | Memo | Component functionality description. |
| STATDESC | Memo | Status description. |
| PARDESC | Memo | Parameter description. |
| RETVALDESC | Memo | Return values description. |
| PROPDESC | Memo | Properties description. |
| CDDESC | Memo | Method/Event code description. |
| COMPCD | Memo | Component code, or a reference to the code. |
| REFERENCE | Memo | References to other documentation for the component. |
| EXAMPLE | Memo | An example of the use of the component. |

Table 5.2: Repository database attributes.

Visual FoxPro 3.0 does not support the use of classes in reports, labels and menu's. These categories are therefore used mainly for storing templates and should unfortunately be reused with *code-grabbing*.

### 5.1.2 Repository Attributes

This section describes syntax of the attributes used to identify, classify and describe the repository components. The current database structure of the repository is shown in table 5.2.

Memo fields are used extensively because they allow free-form input. The format used in the repository can be adjusted very quickly, without affecting the database structure. After the evaluation phase a more complex database scheme could be made to make database maintenance easier.

**Repository Identification Code** The repository identification code uses a similar syntax as MS-DOS directories. The code consists of the global, detailed and functional component levels. The first two are single levels, the functional component level can comprise several sub-levels. The hierarchy levels should use descriptive names, preferably created using the abbreviation rules (see section A.2.) The syntax for the repository identification code is:

$$< Global > \backslash < Detailed > \backslash < Functional >$$

**Component Name** Component names should be descriptive. Component names are created using the abbreviation rules. The methods and events of classes use the classname as part of the component name. The syntax of the component name is:

$$[< ClassName > .] < Name >$$

**Version Number** The version number consists of a major and minor version number, and a revision level. Both major and minor version numbers have two digits, the revision level has one digit. The syntax is:

$$< Major > . < Minor >< Revision >$$

New components start with version number 1.000, first release, no updates and no revisions or interim releases. The major version number changes when the component is changed dramatically, smaller changes are reflected by the minor version number. Small bug-fixes are reflected by the revision level code. After a maximum of nine revisions the minor version number is changed and the revision level is set to zero. After a maximum of 99 minor releases the major version number is increased and the minor version number is set to double zero.

**Baseclass** This field is used for classes to store the baseclass of the component.

**Keywords** Keywords should be listed as a comma separated list. The order in which the keywords are listed is not significant.

**Author** The name of the author. Author names are coded, the author codes should use the abbreviation rules also. Currently three author are defined:

*BER* Marc de Boer
*BNK* Johan Bennink
*HLR* Ron Heller

**Component Type** The component type is coded as shown below. Checklists, Design Documents, etc are referenced using the **Documentation References** field, and cannot be entered or accessed separately. The documentation component type is used to enable the inclusion of programming standards, and other conventions.

1 Function
2 Class
3 Data-structure
4 Documentation
5 File (general)
6 Method/Event

**Component Status** The component status is coded as:

1 Enabled
2 Obsolete
3 Disabled

**Component Functionality Description** A detailed description of the functionality of the component. It is not necessary to describe the parameters and return values for functions, or the properties and methods/events of classes.

**Status Description** A short description explaining the reason for the component status. Enabled components do not have a status description. To add a reference to a replacement component $REPLACEMENT =< RPSID > \backslash < COMPNM > \{< VERSNR >\}$ should be added as the first line of the description.

**Parameter Description** Parameters are listed as:

$$< Parameter >:< Description >$$

Parameter names should be created using the abbreviation rules. The description should be short and descriptive. Possible values for parameters should be listed as:

$$< Value > - < Value - description >$$

**Return-value Description** This field is used for return values of functions only. Changes made in reference variables should be described in the **Parameter Description** field. The first line of the description should be $TYPE =< Variable - type >$ to denote the variable type of the return value. Variable types are codes as shown in table A.2 in appendix A.

**Properties Description** Properties are listed as:

$$< Propertyname > [*][P] =< Description >$$

Properties added to the current class are marked using a star (*.) Protected properties are marked with the option letter **P**. New Property names should be created using the abbreviation rules. The property description should be short but descriptive. Possible values for properties should be listed as:

$$< Value > - < Value - description >$$

**Methods/Events Description** Methods and Events should be listed as:

$$< Method/Event > - < Description >$$

Methods/events that are defined within a single class should be described here. Methods/events that are defined in several classes should be described in a separate repository entry.

**Component Code** This field either holds the component code, or a reference to a file containing the code. Visual classes for instance hold a reference to a class library. A reference is entered as $FILE =< path + filename >$.

**Documentation References** This field contains a comma separated list of files containing additional documentation on the component. This documentation includes design documents, checklists and test reports. References to project documentation from other projects should be used carefully. The contents of such documentation is modified over time and the reference could become invalid. This field may also contain a `SEE ALSO:` directive on the first line followed by a comma delimited list of other, related, components in the repository.

**Example** This field holds an example of the use of the component, or a reference to an example file or program. References are entered as $FILE =< path + filename >$.

### 5.1.3 Repository Browser

The first implementation of the repository browser is restricted to the repository database — containing the component information — and a very simple browser form. During the evaluation of the development method the need for a production repository browser application will be investigated. If needed a repository browser application will be provided in the future.

The repository consists of only one database (see tableImpDBFStruc.) For the current implementation this is sufficient, a production repository browser could require a more elaborate database to allow complex search features. The reports supplied with the first implementation give listings of the repository contents sorted on specific repository attributes.

- Complete listing with all attributes sorted on RPSID+COMPNM+VERSNR.
- A summary listing with a description of the functionality and properties/methods or parameters/return value descriptions for classes and functions respectively.
- Listing of Keywords and the components using them.

### 5.1.4 Repository Components

The components developed for the initial repository are listed in appendix C. In this section some of the concepts used in the development of these components are discussed.

**Encapsulating the base-classes** The Visual FoxPro (VFP) base classes are encapsulated to enable default values for components to be modified very easily. New classes are derived from the VFP classes with **hbb** added as prefix to their name. Specific properties can be set in these classes to introduce company defaults such as colours or fonts used by HBB. This approach could also be used to create customer defaults, i.e. classes based on the HBB classes with specific alterations to the customer defaults. The **hbb** prefix would then be replaced with an appropriate three letter identification for the customer.

The encapsulating classes should be used to derive other new classes from instead of the VFP base classes. For customers with their own set of encapsulating classes, their classes should be used during the application development instead of the HBB classes.

**Functional Data Containers** Functional data containers (FDC) are visual container classes situated in the VCX\CNT category. An FDC is built using other reusable components and

can contain other FDC's. This container-in-container technique allows components of unlimited complexity to be built without losing the flexibility of using the components separately. The FDC's are *building blocks* of known quality that can be used to quickly build a form. In figure 5.2 an example of the use of FDC's is given. The containers form a layer over the smaller containers or objects used in their creation.

The number of layers shown in this example will — in most cases — be too high. In most cases the initial implementaion of a container component will be using normal components (non container components) only. After some time, when parts of the component are usable on their own they can be places in a separate container and the intial container component can be updated. Ultimately the FDC from figure 5.2 might evolve.

This example also shows an example of the encapsulation of Visual FoxPro base classes (Layer 1) which are used to add company defaults. Layer 0 and Layer 1 components are not considered functional data containers (FDC), only components based on a container class are considered FDC's.

## 5.2 Design Phase

In the design phase the design process is formalized by forcing the developer to use a design document during the software component design. Although most modern application development platforms support visual creation of menus, forms, reports and labels it is not recommended to use these tools in the design phase unless strict rules on how far the visual design is to proceed are set up. Otherwise, a developer might be tempted to start the implementation before a design using the design documents is completed. The current development method used at HBB does not use any rules to regulate the design process and the problems associated with ad hoc design (see section 1.1) were the main reason for starting this research.

### 5.2.1 The Design Document

The design document contents varies depending on the type of software component. Each of the design documents distinguished for the Visual FoxPro platform is described below.

The attributes used in the design documents are similar to the properties used in Visual FoxPro. The benefit of this similarity is that the solution is oriented towards an implementation, but without being one. This will be especially important during the initial introduction of the reuse model. The current developement method is very implementation oriented, the acceptance threshold for the reuse model will be lower if the developer is allowed to gradually adjust to the new approach. A risk of this type of design documents is that the developers stick to their implementation oriented development, but call it a design phase. It is therefore essential to evaluate the design documents after a trial period, and if necessary adjust the documents. The documents supplied should be considered draft versions anyway, the documents will probably need adjustments after the trial period to streamline their use. The syntax of the attributes is similar to the syntax used in the software repository (see section 5.1.2.) The design documents supplied with the initial version of the reuse model are listed in appendix B.

**Menu Design** In (Visual) FoxPro menus are mainly used to set up the menu, and perform application initialization. Although application initialization is performed by the menu, it is

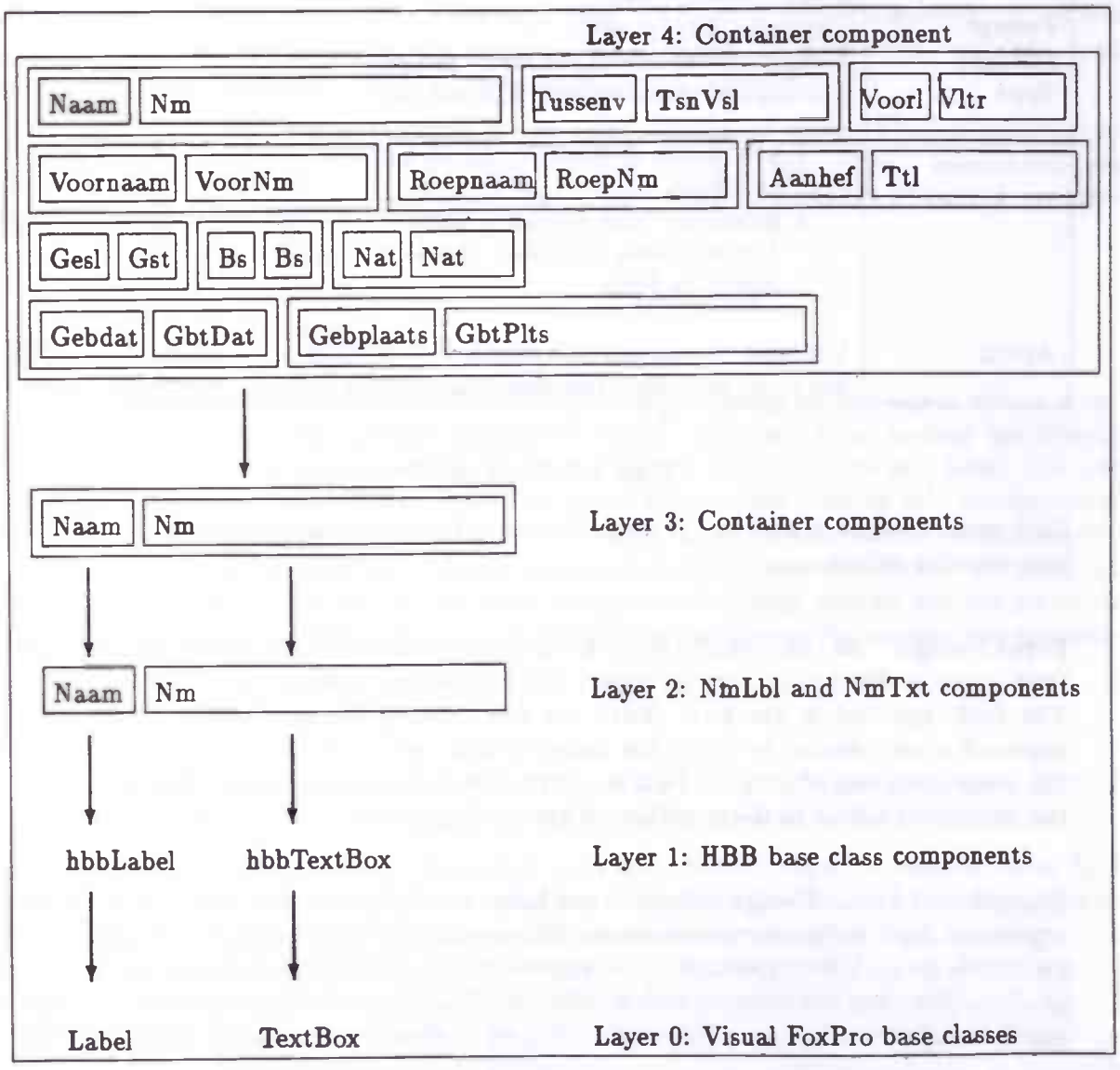



Figure 5.2: An example of the use of Functional data containers to create layers of containers.

preferred to gather all the initialization code in a separate procedure which is called from the menu. Menus are designed using a **menu design document**, where the developer is required to specify the attributes shown in table 5.3 for each (sub)menu.

| | |
|---|---|
| MenuName | A menu for this entry. All entries with the same **MenuName** appear in the same menu. At least one entry should have the name 'Menu Bar', which is the top-most menu. |
| Prompt | The menu prompt text. |
| Message | A short description of the menu purpose. |
| Type | The type of menu/option. This can be:<br>• Command. A function call, or a piece of code, specified in the **action** attribute.<br>• Submenu. The submenu name is specified in the **action** attribute. The submenu definition should be specified elsewhere in the menu definition form. |
| Action | For type 'Command' the function — or code — that should be executed. For type 'Submenu' the MenuName of the submenu is specified here. |

Table 5.3: Menu attributes.

Each menu consists of a menu bar with submenus. Each submenu contains zero or more menu bars or other submenus.

**Form Design** All (data-entry) forms are designed using the **form design document**. The form layout is sketched on squared paper. The form is described by the attributes of table 5.4. The fields specified in the form sketch are described by the attributes of table 5.5. This approach is very similar to the ad hoc design of forms as it is currently used. Instead of using the visual designers of (Visual) FoxPro a form sketch is made on paper. Hopefully this helps the developers adjust to the more formal approach smoothly.

**Report and Label Design** Reports and labels are designed in the same way as forms. The report and label design documents consist of two parts: a visual layout sketch, and a list with attributes for each field specified in the report or label. Both reports and labels use the same attribute lists, but the layout sketch is different. This sketch should also define the papersize, and for labels the label size. The attributes used in these documents are shown in table 5.6.

**Code Design** Besides the visual design aspects of the design phase the software components use pseudo-code design. The **code design document** used for the pseudo-code design varies with the type of software component. Methods and events of classes, database field validation and *trigger* code, and procedure/function code are specified using the attributes shown in table 5.7. The designs of classes are specified using the attributes from table 5.8, the code design of methods and events is specified using the attributes of table 5.7. For visual classes, such as container components, a layout sketch similar to the form sketch should be supplied as well.

| | |
|---|---|
| Name | A unique name to identify the form. |
| BaseClass | The form is based on a form class. |
| Tables | The databases/tables used in the form. |
| Type | Forms can be either:<br>• Modal. A modal form remains active until it is closed. Most dialogs are modal.<br>• Modeless. Modeless forms can coexist with other modeless forms, i.e. multiple modeless forms can be active at the same time. |
| Properties | Other property values. |
| Methods | Any methods and their functionality used in the form. The actual method code design should be specified using a code design document. |
| Description | A description of the functionality of this form. |
| Limitations | A list of assumptions, simplifications and limitations made during the design of the form. |
| Test cases | Test cases — and their outcome — required to test the implementation of the form. |

Table 5.4: Form attributes.

| | |
|---|---|
| Name | A unique name to identify the field. Different fields should at least be unique within the form. |
| BaseClass | The baseclass of this field. |
| ControlSource | The source of this field. This attribute denotes the table column that the field is bound to. |
| InputMask | An *input picture* used for entering into the field using a specific format. |
| Properties | Other properties. |
| Methods | Any methods and their functionality used by this field. The actual method code design should be specified using a code design document. A reference to the code design document should be given here also. |
| Description | A description of the use of this field. |
| Extra | Any other options for the field should be described in free-form notation here. |

Table 5.5: Form field attributes.

| | |
|---|---|
| ControlSource | The source of this field. This attribute denotes the table column that the field is bound to. It is also possible to specify a calculation to be used as the source of the field (a calculated field), i.e. the sum of all values of a table column for example. |
| FormatMask | A *format picture* used for formatting the data of the field. |
| Description | A description of the use of this field. |
| Extra | Any other options for the field — such as whether repeated values are printed — should be described in free-form notation here. If specific options are used regularly they could be included as separate attributes in the future. |

Table 5.6: Report and label field attributes.

| | |
|---|---|
| Name | The name of the software component. For class methods and events the name should be preceded by the classname as: $< ClassName > . < Name >$ |
| Parameters | A description of the parameters of the software component. Parameters of the predefined methods and events should also be specified explicitly to enhance readability. |
| Return value | A description of the — optional — return value of the software component. Again, return values of predefined methods and events should be specified too. |
| Description | A brief textual description of the functionality of the software component. |
| Pseudo-code | The pseudo-code implementation of the software component. |
| Limitations | A list of assumptions, simplifications and limitations made during the design of the component. |
| Test cases | Test cases — and their outcome — required to test the implementation of the components. |

Table 5.7: Attributes for software components.

| | |
|---|---|
| Name | The name of the class. |
| BaseClass | The baseclass for this class. |
| Properties | A list of properties of the class definition with a short textual description of their functionality, and/or possible values. |
| Methods/Events | A list of methods and events supplied with the class. The code design should be specified using the attributes shown in Table 5.7. |
| Description | A brief textual description of the functionality of the class. |
| Limitations | A list of assumptions, simplifications and limitations made during the design of the component. |
| Test cases | Test cases — and their outcome — required to test the implementation of the components. |

Table 5.8: Attributes for classes.

### 5.2.2 The Pseudo-code Language

The proposed pseudo-code language is an abstraction of the Visual FoxPro language. The main language characteristics such as structured programming commands, general variable manipulations and a small subset of the commands for manipulating databases and records are maintained. The pseudo-code language also supports the use of pre- and postconditions. The exact syntax of the pseudo-code language is kept fuzzy. Formalizing the language syntax will only be necessary if developers misuse the pseudo-code syntax, i.e. produce very implementation oriented pseudo-code designs. The appointed *lead developer* could be instructed to make sure no misuse occurs.

### 5.2.3 Specifying Reusable Components

Menus, reports and labels are treated different from forms. Forms support the use of object oriented classes. Reuse is achieved by specifying a reusable component in the **BaseClass** attribute of a form field. The properties and methods attributes in the form field design documents should be left blank. For Menus the reusable components are specified in the **MenuName** attribute, all other attributes should be left blank since they will be 'inherited' from the reusable component. Visual FoxPro does not support object orientation in menus, which means that reuse is achieved by *code-grabbing* for the time being. Hopefully the next release of Visual FoxPro will support object orientation in menus, and reports and labels as well. Reports and labels use the **ControlSource** attribute to specify the reusable component, again, all other attributes should be left blank. Reusable components are specified using their repository identification code, component name and version number. The syntax for the component reference is: `RPS\<repository ID code>\<Component name>{Version}` The component reference always begins with `RPS\` to differentiate between pathnames used to denote files, and component references. For example, assume we are reusing a Functional Data container (FDC) that will display personal data of clients in a client data-entry form. In the form layout sketch the location of the FDC is shown, the container component itself is specified in the field-list of the form design document (see figure 5.3.) The **BaseClass** attribute is used to specify the reusable component, i.e. `RPS\VCX\CNT\PERS\FullPers{1.000}`.

| **Field description** | |
|---|---|
| Name | cntFullPers |
| BaseClass | RPS\VCX\CNT\PERS\FullPers{1.000} |
| ControlSource | |
| InputMask | |
| Description | This container is a combination of three other containers, cntNm, cntVltr and cntTsnVsl. Together they form a container with a persons full name. |
| **Properties** | |
| Alias | PERS |
| **Methods/Events** | |
| | |

Figure 5.3: Example of a form field attribute entry.

Specifying reusable components in pseudo-code is achieved in a similar way. The component

reference is used as the name of the function and uses the same syntax as a normal function call, or variable reference. For example, assume we want to reuse a component that formats a date in a given format. The pseudo-code call to the component — in this case a non-visual function — might look like :

```
...
dFormattedDate = RPS\PRG\FMT\DAT\FormatDate{1.000}(Date(), "DD/MON/YEAR")
...
```

Although a developer should have an adequate knowledge of the components in the repository, it may not be possible for him to know all the components in full detail because of the size of the repository. Therefore the developer is assisted by a repository browser (see section 5.1.3.) Using the repository browser it is possible to call up detailed information about components using only global knowledge of the available components in the repository.

## 5.3 Implementation Phase

In the implementation phase the selected design solutions are implemented. The process of translating the designs into the implementation language is mainly a creative process. As was mentioned in section 4.4.1 a discussion of the implementation process is beyond the scope of this investigation, we limit the discussion about the implementation phase to the introduction of reuse techniques in this phase.

### 5.3.1 Using the Design Documents

The design documents are used as the basis of the actual implementation of the application components. The forms, menus, reports, labels and other parts of the application are described in detail by the design documents. The implementation of menus is very straightforward. The menu design document uses a similar layout as the Visual FoxPro (VFP) menu designer. For the visual implementation of reports and labels the layout sketches of the design documents are used. The fields in the report are implemented using the field attribute lists. During the implementation of a form, the form layout sketch is used to create the visual layout of the form. The form fields are created using the field attribute lists of the design document.

### 5.3.2 Retrieval of Reusable Components

Retrieval of reusable components used in menus, reports and labels is, unfortunately, only possible through *code grabbing*. Visual FoxPro does not support the use of classes in menus, reports and labels. Hopefully this support will be added in a future version of Visual FoxPro. Forms and non-visual application code does support the use of classes. The following discussion is mainly concerned with the retrieval of objects for the implementation of forms, although reuse of non-visual classes is possible for some other application components.

Depending on the type of field (the baseclass) specified in the form design document the class library containing the reusable component is opened, and the component is dragged (drag-and-drop) onto the form. Visual FoxPro takes care of instantiating the class and inserting the object into the form.

Reusable components specified in the pseudo-code design (i.e. non-visual components) are reused by opening and closing the class library manually in the initialization and finalization code of the form respectively. A class can be instantiated using the CREATEOBJECT function, i.e. $< Variable >= CREATEOBJECT(" < BaseClass > ")$, where $< BaseClass >$ denotes the reusable component to be instantiated and $< Variable >$ is the variable used to reference the instantiated object.

The reusable components are clearly marked in the pseudo-code design (see section 5.2.3.) Since the library path is part of the component identification finding the class library containing the specified reusable component is no problem. After the component is inserted, either using the visual designer, or by the developer in a code snippet, the checklists of the reusable components should be used to check for correct use of the component. In some cases a reused component relies on the presence of other library components. The checklist entries of the reusable components indicate which dependencies exist, and which other things have to be checked to be able to successfully (re)use the component.

## 5.4 Test Phase

The formal test strategy as described in section 4.5 is introduced into the development method. However, the test phase is very dependent on the test plans, and test cases stated in the design documents and on the checklists of the repository components. The current repository does not have enough test cases to adequately use and evaluate the test phase.

Software component testing is carried out by the implementor of the component. The *Lead Developer* is reponsible for integrating the components into the application to-be.

# Chapter 6

# Evaluation of the Revised Development Method

To test the reuse model presented in this thesis it will first be used to create a new prototype application, and second to modify an existing application. Since Visual FoxPro differs substantially from FoxPro v2.6a the modification will be of the prototype developed with the reuse model and not of an existing HBB application.

The initial repository consists of just over 100 components (see appendix C for a summary.) A full production repository would probably require more components in order to be of any long term use. However, there is no point in building reusable components when they are never used. Once the model is being used at HBB new components will be developed on demand. The size of the repository is sufficient to evaluate the usefulness of the model.

## 6.1 Building a New Application

The new application is a prototype Telemarketing application and is based on an existing module developed for the Relatiebeheer application. The Relatiebeheer application was developed for a direct-mail company to manage client/product data. The Telemarketing application is used to call prospects based on the information from the Relatiebeheer database. Prospects that have not become customer after a certain period of time will be called to ask for the reason they did not become customer (i.e. prospects are possible clients, who have not returned their offer/contract.)

After defining the requirements, and completing the functional specification, the development proceeded with the design phase. The Requirements, and Operation and Maintenance phases will not be discussed since they are not modified by this model.

### 6.1.1 Design phase

The current development method used at HBB does not use an explicit design phase. Therefore the revised development model does not only introduce reuse, it introduces an entirely new design phase. For this evaluation we will concentrate on the reuse introduced in this phase, and not on the design phase itself.

In the first part of the design process the design documents were used to describe the application components using descriptions of their functionality. Sketches were made to define the visual layout of forms, reports and labels. This part of the design, although new, was very intuitive to use. The next step in the design process was detailing the application components described in the design documents using pseudo-code. This was done using the code design documents.

**Locating Components** During the first part of the design process reusable components such as buttons, containers, etc. were specified. In the pseudo-code design non-visual reusable components were used such as support functions for conversion of data.

Locating reusable components was very easy because of the clear repository structure, and a thorough knowledge of the available components. Component details such as method parameters or allowed property values could be inspected using the repository browser. Although the design documents and the repository browser are implemented separately, it seems that they could be integrated. This would allow the design documents to be filled in using an automated entry form which uses the repository browser to access component details.

Although the size of the repository still allowed browsing through all components, a production repository — with many more components — will probably necessitate a filter mechanism to restrict the number of components to a reasonable number.

**Functional Data Containers** The use of Functional data containers (FDC) seems very promising. During the prototype design only a few FDC's were available. Nevertheless, specifying a complete data-entry container for personal data with a standardized user-interface (look-and-feel) makes rapid application development very easy. Although the experience from this prototype indicates an advantage of the use of FDC's, it will depend on the stability of the problem domain whether FDC's will work all the time. Currently HBB builds applications for a reasonably stable domain (processing client/product data) which would greatly benefit from Functional data containers.

Even though the Functional data containers (FDC) seem useful, providing containers such as show in figure 5.2 will be very inefficient. When the smaller containers of the layers 2 and 3 are never used there is no point in creating them. As with reusable components in general, Functional data containers will probably start out as a simple container with all objects included into layer 4. On demand this FDC could be split up to use components of layer 3 complexity, and so on.

**Ad hoc Design** The introduction of design documents greatly reduced ad hoc design. For instance, during the design of a class only the functionality of the methods and events is described; their implementation is described in a separate document. Using the design documents the design process proceeds in a layered fashion where every following layer adds more detail to the design until the point is reached where implementation can start.

The transition from ad hoc design to using the design documents was not very hard. It took some getting used to but after completing a few documents they became second nature. This easy transition was achieved because the notation used in the design documents is very similar to the notation used by Visual FoxPro. This allows the developer to hold on to his way of thinking without actually performing ad hoc design.

**Design Documents** It was already mentioned that the design documents supplied should be regarded as draft versions. During the use of the documents this became apparent. The design documents required a lot of repetitive work, such as entering project and author information and supplying the descriptions of reusable components. As was mentioned earlier this problem could be overcome by integrating the design documents with the repository browser somehow. Apart from this minor inconvenience using the design documents was very intuitive.

**Local Reuse** During the application development reuse withing the project seems very easy. Local reuse was not included in this research because it would mean that development *for* reuse would have to be incorporated as well. Developing reusable components during the development of an application leads to a conflict of interest because valuable resources have to be spend developing components that can be reused in subsequent projects. Even though local reuse was not incorporated in this research it might be worth investigating.

Designing components using the design documents was very intuitive. The extra work involved in using the documents did not prove to be a problem. Locating reusable components did not interfere with the actual design process and specifying reusable components using the component reference was simple.

### 6.1.2 Implementation phase

Not all repository components have been implemented. In some cases only an empty component was available. Testing the runtime behaviour of the application was therefore not always possible.

The sketch of forms, reports and labels included with the design documents was used to create the visual layout of the application component. Using the sketches did not cause any problems. Placing controls (buttons, checkboxes, etc) onto the form using the Visual FoxPro form designer was very easy. The library was located using a part of the component reference, the repository identification code. The repository identification code is a relative directory path which enables the developer to locate and open the library file containing the reusable component. After opening the library, reuse consists of a drag-and-drop of the required control onto the form. Visual FoxPro takes care of adding the required code to the form, and the project manager includes the library in the project automatically.

Reuse of other, non-visual, components involved including the library containing the component in the project manually. The library was opened in the initialization part of the form, menu or program file and a statement to close the library was added to the finalization part of the component. Calling the reusable component is no different than using other classes, or functions.

After an application component ws completed the checklists of the reusable components used in the application component were used to check for correct usage of the reusable component. Although the checklists were still rather short their use to prevent common mistakes proved very helpful.

The use of naming conventions (appendix A) made checking for correct variable types of return values very easy. Only untyped variables required a closer look, other variables could be checked instantly using the variable-type prefix.

### 6.1.3 Building Effort

The introduction of a design phase obviously affected the building effort. Although the use of the design documents was somewhat time-consuming, they prevented ad hoc design which was one of the reasons for starting this research. Resistance against using the design documents was low, except for the repetitive work involved. The solution to this small problem might be the integration of the design documents with the repository browser. This would enable the use of repository information in the design documents.

Locating components in both the design and implementation phase was very straightforward. A reasonable knowledge of the repository was enough to find the appropriate component in the design phase. In the implementation phase, the form designers of Visual FoxPro allowed an easy selection of the libraries, mainly because the repository identification code was identical to the directory structure. For reuse of non-visual components simple procedures were used to automate their reuse, and keep the application consistent.

Apart from using repository components and introducing a design phase, the use of the programming standards has resulted in better readable code. This increase in code readability, and consistency ultimately reduces the maintenance costs, and increases the possibility of being able to give long term support.

It is difficult to compare the development time and effort of the Telemarketing prototype with existing application developed using the current (ad hoc) development method. Due to the lack of documentation it is not possible to calculate the development effort of these existing applications. The development effort of the prototype is shown in table 6.1.

| Type of Component | Development time | Divided over | |
|---|---|---|---|
| | | Design | Implementation |
| RS Document | 2.00 | X | X |
| FS Document | 4.00 | X | X |
| Tables (5) | 1.00 | X | X |
| Menus (1) | 0.30 | 0.20 | 0.10 |
| Forms (5) | 25.00 | 15.00 | 10.00 |
| Reports (1) | 0.45 | 0.25 | 0.20 |
| Integration | 2.00 | X | 2.00 |
| Total | 35.15 hours | | |

Table 6.1: Development effort for the Telemarketing prototype in hours and minutes.

The ratio of reuse vs. new code was approximately 34:66. Although it must be said that this was due to the fact that one very complex component was (re)used in multiple forms. But even a ratio of 25:75 would not have been that bad.

## 6.2 Modifying an Existing Application

Besides using the revised development method for new applications, a modification of an existing application was performed to evaluate the method. For this modification the prototype

Telemarketing module was used. During the initial development of the Telemarketing prototype an existing (reusable) component was used that was developed for the Relatiebeheer application. The Relatiebeheer application is used by data-typists, the data-entry components were therefore oriented towards fast entry of data without having to look at the screen all the time. The Telemarketeers do not need fast entry of data. They have to ask the prospects some questions and give them the possible answers. Therefore they would like to see a data-entry form that shows them the possible answers which allows them to guide the prospect through the questions-and-answers of the form. The modification of the Telemarketing application consisted of a modification of those aspects of the data-entry forms of the prototype.

### 6.2.1 Modification Aspects

The presence of documentation, other than the source-code, is very important. It is often very hard to see what design decisions were made based on the source-code. The revised development method uses design documents to supply this documentation. The application design described in the design documents is used to make the modifications instead of using the source-code. The result of such modifications are new versions of the design documents with a modified design which can be reimplemented. Reuse of components during the design of these modifications is not different from reuse during the original design.

**Design phase** Reuse of components may not be any different, the redesign of existing components is. During the redesign of a component care should be taken not to change the pre- and postconditions of the component. Such changes would probably require changes in the components that use the modified component. However, these aspects are not due to reuse, but can be attributed to good design practices in general.

Modifying the existing design documents proved to be very straightforward. The level of abstraction in the design documents allowed the developer to make modifications without thinking about the consequences to the current implementation. The resulting design is in most cases a 'cleaner' solution because no attempt is made to recover existing (source-)code. The modification consisted mainly of replacing of some controls with other types of controls, preserving the other parts of the container component as much as possible. The result was a new component with, for the Telemarketeers, a more user-friendly interface.

We do not discuss changes in the reusable components since the model presented here considers development *for* reuse a separate discipline. Changes in pre- and postconditions would require changes in other parts of the application during redesign of components. Changes in reusable components might require changes in several applications. Again, many of the problems related to these changes can be prevented with good design practices (such as achieving low coupling [18]).

**Implementation phase** During the implementation phase the new design documents were used to implement the modified design. In some cases this may involve a complete reimplementation of a component. In this case much of the existing component could be salvaged. Furthermore, reuse during the modification of existing applications components was not different from reuse during the initial implementation of the component either.

Whether parts of the original implementation are reused, or a complete reimplementation of

the component is performed, it will be necessary to use the checklists of all the repository components used in order to detect possible mistakes in the use of reusable components.

### 6.2.2 Modification Effort

Redesign of components is more difficult than the initial design, this is however not due to the development model. Redesign must take into account the existing dependencies to other components. Only a loosely coupled design can help reduce the problems related to these dependencies. Reimplementation of components will also require more work than the original implementation. The amount of extra work needed depends on the level of coupling in the original design. A high level of coupling will require more code to be rewritten. Whereas a low level of coupling will require very little changes outside the redesigned component, i.e. changes remain local.

For the modification of the Telemarketing prototype changes were local to one data-entry form. Overall redesign was somewhat harder than the initial design. This can be attributed to the fact that modifications have to fit within the existing design, which inevitably takes some extra time. This is also true for the reimplementation. Especially when (large) parts of the existing implementation are reused, care should be taken not to introduce errors. A major benefit of the use of the design documents is the visibility of the modifications. By comparing the old design documents with the new documents the changes are visible without having to look ate the implementation. Comprehending the designs is, due to the higher abstraction level of the design documents, much easier than comprehending the implementation.

Again, as with the original implementation, it is difficult to compare the development time and effort between the current and proposed development method for modifications because no measurements are available of the current development method. The modification effort of the prototype is shown in table 6.2.

| Type of Component | Modification time | Divided over | |
|---|---|---|---|
| | | Design | Implementation |
| Forms (1) | 2.30 | 1.45 | 0.45 |
| Integration | 0.15 | X | 0.15 |
| Total | 2.45 hours | | |

Table 6.2: Modification effort for the Telemarketing prototype in hours and minutes.

For this modification one complex component was replaced with a new component. All other parts of the data-entry form remained unchanged. Therefore the ratio of reuse vs. new code looks very negative, **20:80**. If we consider the unchanged code as reused code, which is in fact true since we used *code grabbing*, the ratio looks much better **64:36**.

# Chapter 7

# Concluding Remarks

## 7.1 Conclusions

The models presented here introduced reuse techniques into the development method used at HBB Automatisering. An implementation was given for the Visual FoxPro language, and an evaluation of the model showed the model was usable. Furthermore, the development method was formalized in several places providing vital information for maintenance and future extension of the applications.

It is very difficult to compare the current method and the method proposed in this thesis because the current method does not allow any measurements of time and cost effectiveness. The proposed development method is, due to the introduction of a explicit design phase, more time consuming. Depending on the number of reusable components used, and their complexity some time saving is achieved because less new code is introduced. The major benefits of the proposed method are for modifications of existing applications. Due to the existance of the design documents modifications are much easier. With the current development method modifications are made directly in the source-code, which, especially when the application was developed some time ago, is not be that simple.

The model can also be used as a starting point to introduce other changes into the development method used at HBB Automatisering. I do not claim that the model is perfect, many areas of research are still left uncharted, but it does improve the quality of the applications developed and make them more maintainable.

## 7.2 Enhancements and Future Research

Some of the areas of further research that might improve the model are given in this last section. Research will be necessary to investigate the necessity of the features mentioned here. The enhancements and future research suggested in this section are listed in random order.

**Local Repositories** At the end of section 4.2.2 it was already mentioned that the repository could be used for local components too. The benefits of storing newly-written components in a local repository should be investigated. A local repository could increase the probability

of reuse, and make the process of classification for insertion into the global repository much simpler.

**Visual Designers** The benefits of the Visual designers supplied with most modern visual development platforms are not used. Instead of using a form layout sketch the Visual designer of Visual FoxPro could be used to create the layout. The benefits of this approach are that the form could be used to give the customer an idea of how the application will look like in the form of an empty prototype. Furthermore, forms do not have to be sketched first, but are created directly. However, some safeguard has to be introduced to prevent the developer from continuing in the Visual designer when he should use the design document. Eventhough the Visual designer is used, and some implementation is generated the development process is still in the design phase.

**Borland Delphi Implementation** HBB is currently using Borland Delphi as a development language as well. An implementation of the model for Borland Delphi would enable the use of knowledge obtained during this research for that platform as well. This would also be a major test of the adaptability of the model.

**Requirements Phase** Currently the Requirements phase is not modified. Further study will be necessary to find out whether a more formal approach in the Requirements phase could result in more reuse in the Design phase.

# Bibliography


[1] Susan P. Arnold and Stephen L. Stepoway. The reuse system: Cataloging and retrieval of reusable software. In *Software Reuse: Emerging Technology* [19], pages 138–141.

[2] J. Bennink. HBB automatisering; current development method. Internal project report, 1995.

[3] Ted J. Biggerstaff and Alan J. Perlis. *Software Reusability; Concepts and Models*, volume 1 of *Frontier Series*. ACM Press, 1989.

[4] Ted J. Biggerstaff and Charles Richter. *Reusability framework, assessment, and directions*, chapter 1. Volume 1 of *Frontier Series* [3], 1989.

[5] Bruce A. Burton et al. The reusable software library. In *Software Reuse: Emerging Technology* [19].

[6] Robert N. Charette. *Software Engineering Environments; Concepts and technology.* McGraw-Hill Book Company, 1986.

[7] Richard J. St. Dennis. Reusable ada (r) software guidelines. In *Software Reuse: Emerging Technology* [19], pages 257–264.

[8] Horowitz Ellis and B. Munson John. *An expansive view of reusable software*, chapter 2. Volume 1 of *Frontier Series* [3], 1989.

[9] Gerhard Fisher, Andreas C. Lemke, and Christian Rathke. From design to redesign. In *Software Reuse: Emerging Technology* [19], pages 282–289.

[10] W.B. Frakes and B.A. Nejmeh. An information system for software reuse. In *Software Reuse: Emerging Technology* [19], pages 142–151.

[11] Allen Macro. *Software Engineering; Concepts and Management.* Practical Software Engineering Series. Prentice Hall International (UK) Ltd, 1st edition, 1990.

[12] Microsoft. *Microsoft Visual FoxPro 3.0 Developers Guide (Beta 2)*, 1994.

[13] James M. Neighbors. *Draco: A method for engineering reusable software systems*, section 12.7.1. Volume 1 of *Frontier Series* [3], 1989.

[14] Ruben Prieto-Diaz and Peter Freeman. Classifying software for reusability. In *Software Reusablility*, pages 106–116. Computer Society Press of the IEEE, 1987.

[15] Ruben Prieto-Diaz and Gerald A. Jones. Breathing new life into old software. In *Software Reuse: Emerging Technology* [19], pages 152–160.

[16] David R. Reed. Tools for software reuse. *Object Magazine*, pages 63–67, February 1995.

[17] I. Sommerville. *Software Engineering.* International Computer Science Series. Addison-Wesley Publishing Company, 1982.

[18] I. Sommerville. *Software Engineering*, pages 53–54. In *International Computer Science Series* [17], 1982.

[19] Will Tracz. *Software Reuse: Emerging Technology.* Computer Society Press of the IEEE, 1988.

[20] Will Tracz. Software reuse: Motivators and inhibitors. In *Software Reuse: Emerging Technology* [19], pages 62–67.

# Appendix A

# Programming Standards

Most of the standards are adopted from the current standards used at HBB [2]. The standards are all concerned with the source-code. A consistent use of name conventions, abbreviations and source-code layout is a first step in creating readable and maintainable applications and components.

## A.1 Name Conventions

The name conventions used are adopted from the suggested coding convention by Microsoft[12] —known as the hungarian coding convention— with some minor modifications. Only conventions for variable and object names are covered in this section.

In the following we denote both database fields and memory variables as variables. Variable names are created using (combinations of) abbreviations and can be combined with a sequence number. Although variable names should be as descriptive as possible, their length should not be excessive, the preferable length is 3 to 15 letters[1] (although Visual FoxPro v3.0 supports variable names of up to 254 significant characters.) The use of single letter variable names should be avoided unless they do not present any risk of misinterpretation.

We will use type-coding for some variables using two prefix letters, the first letter denotes the variable scope (table A.1) and the second letter is used to denote the variable type (table A.2). The scope prefix is recommended but not required. In some cases explicit scoping does not apply. For example, in the main program of a standalone application, there is no difference in visibility for variables scoped as PUBLIC or PRIVATE. The type prefix is always relevant and is always required.

Arrays are allowed to use different variable types in their rows and columns. A separate variable type for array variables was added to prevent any confusion about the actual type of the data stored in an array. FoxPro allows variables to change their type at runtime. By using the variable type as part of the variable name this might lead to confusion. The use of this feature is therefore limited to variables that are type-coded using the added variable type 'Untyped'.

Database fieldnames are used in uppercase letters. Since the scope of a fieldname is not

[1] Please note that on the MS-DOS platform filenames can only be 8 characters long (excl. an extension.)

| | |
|---|---|
| l | Local |
| g | Public (global) |
| p | Private |
| t | Parameter |

Table A.1: Variable Scope.

| | | | |
|---|---|---|---|
| a | Array | l | Logical |
| b | Double | m | Memo |
| c | Character | n | Numeric |
| d | Date | o | Object |
| f | Float | t | DateTime |
| g | General | u | Untyped |
| | | y | Currency |

Table A.2: Variable Types.

bound to the program execution database fieldnames do not use a scope prefix. The type of database fields normally never changes, it is therefore not necessary to use a variable type in the fieldname.

FoxPro uses a special notation for memory variables that contain copies of database fields. These variables are referenced by preceding them by an **m.** code. Since these variables are copies of database fields these memory variables do not use scope- and type-coding either. All other memory variables use the coding conventions.

Additionally the following naming convention is used for classes. The name of the class is prefixed by the type of the baseclass of the new class definition. Table A.3 lists the baseclasses defined by Visual FoxPro.

| Prefix | BaseClass | Prefix | BaseClass |
|---|---|---|---|
| chk | CheckBox | img | Image |
| cbo | ComboBox | lbl | Label |
| cmd | CommandButton | lin | Line |
| cmg | CommandGroup | lst | ListBox |
| cnt | Container | ole | OLE |
| ctl | Control | opt | OptionButton |
| <User-defined> | Custom | opg | OptionGroup |
| edt | EditBox | pag | Page |
| frm | Form | pgf | PageFrame |
| fpg | FormPage | shp | Shape |
| frs | FormSet | spn | Spinner |
| grd | Grid | txt | TextBox |
| grc | Column | tmr | Timer |
| grh | Header | tbr | ToolBar |

Table A.3: Visual FoxPro BaseClasses and their prefix code.

## A.2 Abbreviations

The use of abbreviations enables the component and variable names to be more descriptive without turning into small pieces of prose.

### A.2.1 Abbreviation Rules

If the original word can be deduced from it's abbreviated form then the abbreviation is said to be 'understandable'. Sometimes it may not be possible to create an abbreviation this way, there should be an alternative for these cases. The length of abbreviations should not be too long. The rules presented here assume a length of 3 to 8 letters unless it is otherwise not possible to create an abbreviation. The rules for creating abbreviations are shown in table A.4. The rules should be tried in order until an acceptable abbreviation is created.

1. Is it possible to create an abbreviation for the word using existing abbreviation? If so, use that abbreviation. Abbreviations for compound words should always be created from the words that make up the compound word.
2. Try to create an abbreviation using the first letter of each syllable in the word to be abbreviated.
3. Try to create an abbreviation using several letters of the syllables in the word to be abbreviated.
4. Try to use complete syllables to create an abbreviation.
5. If it is not possible to create an abbreviation then the developer is free to create an abbreviation from other letters, syllables or words. It is still preferable to keep the abbreviation as logical as possible.

Table A.4: Steps for creating abbreviations.

### A.2.2 Creating New Abbreviations

New abbreviations should be created using the abbreviation rules. The newly created abbreviation should be *registered* with the abbreviation list owner (ALO). The ALO is responsible for maintaining and distributing the abbreviation list. He decides whether a new abbreviation is accepted or not. He is free to change the new abbreviation, or it's meaning, before it is registered.

The use of new, unregistered, abbreviations is limited. Developers are responsible for keeping track of where they use the new abbreviation. If the ALO decides to change the unregistered abbreviation, the developer will have to modify each instance used so far. There is no limitation on the use of registered abbreviations. Registered abbreviations never change. The meaning of abbreviations can only be extended as long as it does not lead to confusion.

## A.3 Source-code Layout

The source layout rules apply to all source-code from program files and the PowerTools. There are rules for indentation of the source-code, and for the use of comments.

```
FUNCTION <function-name>
[PARAMETERS <parameter-list>]
   [<statements>]
[RETURN [ <expr> ]]
```

```
IF <expL>
   <statements>
[ELSE
   <statements>]
ENDIF
```

```
FOR <mem-var>=<expN1> TO <expN2> [STEP <expN3>]
   <statements>
   [LOOP]
   [EXIT]
ENDFOR
```

```
DO WHILE <expL>
   <statements>
   [LOOP]
   [EXIT]
ENDDO
```

```
SELECT [ALL | DISTINCT]
       [<alias>.]<select item> [AS <column name>]
[,     ... ]
FROM   <table> [<local alias>]
[,     ... ]
INTO   <destination>
[WHERE <joincondition>
[AND   ... ]
[AND | OR <filtercondition> ... ]]
[GROUP  BY <group column>
[,         ... ]]
[HAVING <filterconfition> ]
[UNION [ALL] <SELECT <command> ]
[ORDER  BY <order_item> [ASC | DESC]
[,         ... ]]
```

```
DO CASE
   CASE <expL1>
      <statements>
   [CASE <expL2>
      <statements>
   ...]
   [OTHERWISE]
      <statements>]
ENDCASE
```

Table A.5: Indentation rules for language constructs.

### A.3.1 Indentation

The source-code is indented using three spaces. The rules for indentation of several program structures is shown in table A.5. Some of these rules are very dependent on the FoxPro language.

### A.3.2 Comments

The use of comments in the source-code is essential for any form of maintainability, although it is not the only technique. The rules for placing comments are:

1. Each program, function and class is documented using a comment header. (See section A.3.3)
2. Source-code within functions that performs a significant task should be commented with a description of the task.
3. Comments placed on the same line as source-code are not clearly distinguishable from the code and are therefore not used. This means that the comment option of FoxPro using a '&&' is not used.
4. FoxPro uses a '*' symbol to denote the start of comments on a separate line, ie. lines containing no code. To enhance readability all descriptive comments should use a '**' followed by one space as in:

```
** Calculate the expected profit on the invested capital.
```

```
lfProfit=pfCapital * pfInterest
```

5. To indicate that a comment describes a crucial (design) limitation, or an assumption a '*!' code is used as in:

```
*! This relies on the fact that the InputMask property is set to
*! 99/AAA/9999.
lcMonth=UPPER(SUBSTR(THIS.Value,4,3))
```

6. To add temporary comments — used mostly during development — or to denote temporary solutions or bug workarounds a '*?' code should be used.

```
*? The following code is skipped until the DE works properly.
*IF NOT USED('EMP')
*   USE EMP IN 0
*ENDIF
```

7. Application code that is commented out for some reason uses a '*' code as in the above example.
8. Comments added at a later date, as for bug-fixes, should include an author code and the date of insertion in the format as shown in:

```
** BNK-20/11/1994: The code below was added to fix a bug occuring when a user
** did not select any list-entry and chose the Exit button immediately.
IF lnSelected>0
   ...
ENDIF
```

### A.3.3 Comment Headers

Each program, function and class is documented using a comment header. The entries in the header depend on the type of the component. In table A.6 all comment header entries are shown in order of appearance. The first column denotes for which type of code the entry is included in the header.

```
A       **
A       ** Name           : <Function name>
C       ** Base class     : <Parent class>
F,C     ** From library   : <Library name>
A       ** Type           : <Program | Function | Class>
A       ** Version        : <Version number>
A       ** Based on       : <Function name>.<Version number>
A       ** Date           : <Creation date of this version>
A       ** Author         : <Name of the author>
F,P     ** Parameters     :
F,P     **    <Parameter1>     - <Parameter1 description>
F,P     **    ...
C       ** Properties     :
C       **    <Property name>  - <Property description>
C       **    ...
C       ** Methods        :
C       **    <Method name>    - <Method description>
C       **    Parameters       :
C       **       <Parameter1>  - <Parameter1 description>
C       **    ...
C       **    Return value     : <Return value type>
C       ** Events         :
C       **    <Event name>     - <Event description>
C       **    Parameters       :
C       **       <Parameter1>  - <Parameter1 description>
C       **    ...
C       **    Return value     : <Return value type>
C       **    ...
F,P     ** Return value   : <Return value type>
A       ** Description    :
A       **    <Description text>
A       ** Keywords       :
A       **    <Comma delimited keyword list>
A       ** Example        :
A       **    <Example text>
A       **
```

Table A.6:

| Comment header layout. | |
|---|---|
| A | Included for all types. |
| F | Included for functions. |
| C | Included for OO classes. |
| P | included for programs and modules. |

# Appendix B

# Design documents

This appendix shows the design documents that are used for the initial version of the reuse model. After a trial period these documents will be evaluated, and if necessary, modifications will be made.

| | | |
|---|---|---|
| **Project** | | Preliminary Software Component design document for classes. |
| **Author** | | |
| **Version** | | |
| **Based on version** | | |
| **Name** | | |
| **BaseClass** | | |
| **Description** | | |
| **Properties** | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| **Methods/Events** | | |
| | | |
| | | |
| | | |
| | | |
| | | |

**Limitations**:

**Test-cases**:

| | | | | |
|---|---|---|---|---|
| **Project** | | Preliminary form design document. | | |
| **Author** | | | | |
| **Version** | | | | |
| **Based on version** | | | | |
| **Name** | | | | |
| **BaseClass** | | | | |
| **Tables** | | | | |
| **Type** | | Modal | | Modeless |
| **Description** | | | | |
| **Methods/Events** | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Limitations:

Test-cases:

| Field description | |
|---|---|
| Name | |
| BaseClass | |
| ControlSource | |
| InputMask | |
| Description | |
| Properties | |
| | |
| | |
| | |
| Methods/Events | |
| | |
| | |
| | |

| Field description | |
|---|---|
| Name | |
| BaseClass | |
| ControlSource | |
| InputMask | |
| Description | |
| Properties | |
| | |
| | |
| | |
| Methods/Events | |
| | |
| | |
| | |

| | | |
|---|---|---|
| **Project** | | Preliminary Software Component design document for procedures and functions. |
| **Author** | | |
| **Version** | | |
| **Based on version** | | |
| **Name** | | |
| **Return value** | | |
| **Description** | | |
| **Parameters** | | |
| | | |
| | | |
| | | |
| | | |
| | | |

**Limitations**:

**Test-cases**:

Pseudo-code design:

| | | |
|---|---|---|
| **Project** | | Preliminary menu design document. |
| **Author** | | |
| **Version** | | |
| **Based on version** | | |
| **Name** | | |
| | | |
| **Description** | | |

Menu entries

| MenuName | Prompt | Message | Type | Action |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |

| | | |
|---|---|---|
| Project | | Preliminary report design document. |
| Author | | |
| Version | | |
| Based on version | | |
| Name | | |
| | | |
| Description | | |

| Field description | |
|---|---|
| ControlSource | |
| FormatMask | |
| Description | |
| Extra | |

| Field description | |
|---|---|
| ControlSource | |
| FormatMask | |
| Description | |
| Extra | |

| | | |
|---|---|---|
| Project | | Preliminary label design document. |
| Author | | |
| Version | | |
| Based on version | | |
| Name | | |
| | | |
| Description | | |

| Field description | |
|---|---|
| ControlSource | |
| FormatMask | |
| Description | |
| Extra | |

| Field description | |
|---|---|
| ControlSource | |
| FormatMask | |
| Description | |
| Extra | |

| | | |
|---|---|---|
| **Project** | | Preliminary Checklist document. |
| **Author** | | |
| **Version** | | |
| **Based on version** | | |
| **Name** | | |
| | | |
| **Description** | | |
| **Checklist** | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# Appendix C

# Repository Summary

This appendix lists a few components that are part of the initial repository developed for this project. Once the repository is used in real projects the number of components will steadily grow. The components supplied are just a basic set, necessary to test the concepts of the reuse model. Currently the components in the repository are distributed as follows:

| NrOf. | Type of component |
|---|---|
| 47 | Controls |
| 24 | Containers (Funtional data containers) |
| 25 | Encapsulating classes |
| 4 | (Programming) standards |
| 8 | Support functions |
| 4 | Dialog forms |

| Repository Identification Code | |
|---|---|
| Componentname | Component type |
| Versionnumber | Author |
| Description | |
| Parameter/Property description | Return value/Method code description |
| Keywords | References to (other) documentation |

| PRG\CLS_SUPP | |
|---|---|
| AddAlias | Function |
| Version 1.000 | Author BNK |
| This procedure adds an alias to the object's ControlSource and/or Alias property using the Alias property from its parent. Extensive checks are done to make sure that the object has a ControlSource, a parent, the parent has an Alias property etc. Aliases are only overwritten if the parent object wants to force alias overwriting. It is not added as a method to a class because it would require the method to be added to at least two classes. This way maintenance is simpler as well. | |
| oObject : The object used to add an alias to. | |
| Function, Alias, Container objects | |

| SCX\SPLASH | |
|---|---|
| About | |
| Version 1.000 | Author BNK |
| This is a special type of splash screen. This screen is used to display a dialog box with information about the application similar to the FullSplash screen (ie. it display the same information.) This dialog however can be called from a special "About..." menu option. | |
| | |
| Template form, About dialog, Application information, Application version, Copyright information, License information | SEE ALSO: FullSplash |

| DBF | |
|---|---|
| RESOURCE | Datastructure |
| Version 1.000 | Author BNK |
| This table is used by an application to store variable-constants and lookup data. It can be used for storing user authorization, printer information and any other application data that needs to be stored in a way that allows easy modification of and additions to the data. The application using a specific resource should know which fields to use, and to what type to convert the fields. All fields are stored as character strings.<br>STRUCTURE:<br>ResNm : C(15) The resource name.<br>ResItem1 : C(10) Resource value/key.<br>ResItem2 : C(10) Resource value/key.<br>ResItem3 : C(15) Resource value/key.<br>ResItem4 : C(15) Resource value/key.<br>ResItem5 : C(40) Resource value/key.<br>ResItem6 : C(80) Resource value/key.<br>ResItem7 : C(200) Resource value/key.<br>ResItem8 : M Resource value/key.<br>SysRes : L Denotes the type of resource. SysRec=.T. resources are not part of the application resources, but are system resources, and should not be modified.<br>ResDesc : M A description of the functionality of the resource. | |
| | |
| Table, Resources | |

| VCX\TXT\PERS | |
|---|---|
| txtNm | Baseclass hbbTextBox |
| Version 1.000 | Author BNK |
| This field is used to enter a last-/surname. The first letters of each name is converted to UPPERcase, the rest to LOWERcase, additional spaces are removed.<br>BNF syntax:<br>txtNm = [<Woord>[<Scheiding><Woord>]*]<br>Woord = <Hoofdletter>[<Letters>]<br>HoofdLetter = 'A'..'Z'<br>Letters = <Letter>[<Letters>]<br>Letter = 'a'..'z'<br>Scheiding = '/'\|' '\|'-' | |
| ControlSource = NM | FldFmt()* - Convert all characters from the field into LOWERcase, except for the first letters of words/names and strip extra spaces. |
| Datafield, Personal data, Naam | |

| VCX\CMD\PERS | |
|---|---|
| cmdGst | Baseclass hbbCommandButton |
| Version 1.000 | Author BNK |
| This is a three-state button. Depending on the value of the ControlSource (identical to txtGst) the button shows a picture of the gender. | |
| Picture = Used to hold the Gender bitmap. Initialisation is performed in the Init clause, changes are mdae using the Click event. | Init() - Initialize the Picture property. If the ControlSource field currently has a value other than 0,1,2,M,V,? the field is initialized to 2 or ? depending on the fieldtype, which should be either numerical, or character.<br>Click() - Depending on the current contents of the field the picture is updated in sequence. The component can use both numerical, and character fields. The field values are translated to bitmaps using the following scheme:<br>0,M = Male bitmap<br>1,V = Female bitmap<br>2,? = Unknown gender bitmap |
| CommandButton, Personal data, Geslacht | CHECKLIST:<br>* Make sure that the bitmaps are included into the project. The bitmaps are:<br>Male : PIC\GENERAL\MALE.BMP<br>Female : PIC\GENERAL\FEMALE.BMP<br>Unknown: PIC\GENERAL\GENDER.BMP |

| VCX\TXT\PERS | |
|---|---|
| txtGst | Baseclass hbbTextBox |
| Version 1.000 | Author BNK |
| This field is used to enter the gender of a person. The field is validated using (M) to denote male, and (V) to denote Female. If the gender is unknown (?) can be entered. Field values are converted to UPPERcase. | |
| ControlSource = GST | FldFmt()* - Return the field in UPPERcase.<br>Valid() - Check is the field contents is "M", "V" or "?" (see Description). |
| Datafield, Personal data, Geslacht | |

| VCX\CMD\TBL | |
|---|---|
| cmdTblNew | Baseclass hbbCommandButton |
| Version 1.000 | Author BNK |
| This button is used to set a form in the New state (fsNew.) The New state can only be set if the current state of the form is fsReadOnly or fsQuery. If the form is already in the fsNew state nothing happens, for the fsReadOnly and fsQuery state all fields are enabled and a new, empty, record is created. | |
| | Valid() - Check to see whether the form/container is in the fsReadOnly or fsQuery state and set fsNew state for the current active form/container. The New button is disabled. All fields are enabled, and a new, empty, record is created. If the form/container was in fsNew or fsEdit state nothing happens. If the valid method is overwritten a test for these states after calling the original valid can be used to process these respective states. |
| Button, Table, State, New | CHECKLIST:<br>* Make sure that the Form class used has a State property, and that the fs-constants are defined. |

| VCX\HBB | |
|---|---|
| hbbTextBox | Baseclass TextBox |
| Version 1.000 | Author BNK |
| HBB TextBox class. | |
| BackColor = Blue<br>ForeColor = Yellow<br>DisabledBackColor = Blue<br>DisabledForeColor = Gray<br>SelectedBackColor = Red<br>SelectedForeColor = Yellow<br>Name = txt...<br>Width = AutoSize to the ControlSource field (mostly from a table). If an InputMask is set the length of the InputMask has precedence.<br>InputMask = If blank, use the length of the ControlSource to create an appropriate InputMask.<br>MaxLength = Set to the length of the ControlSource. Do this first to save duplicate code, the Width and InputMask property use this length. | Valid() - THIS.Value=THIS.FldFmt()<br>FldFmt()* - Return the field value in a format appropriate for that field.<br>Proper()* - Return the field value in a Mixed case format. (PROPER)<br>Upper()* - Return the field value in UPPERcase. (UPPER)<br>Lower()* - Return the field value in LOWERcase. (LOWER)<br>StripSpaces()* - Return the field value with extra spaces removed. (ALLTRIM) |
| HBB baseclass, TextBox | CHECKLIST:<br>* Check to see if the Width, InputMask, and MaxLength properties are set correctly. |

| VCX\CMG\BUTTONS | |
|---|---|
| cmgTblEditBtn | Baseclass hbbCommandGroup |
| Version 1.000 | Author BNK |
| This component supplies four buttons, of which two have double functions depending on the state of the form. The first button is used to add new records, the second to Query the table and locate records. The third button is used to start editing during New and Edit state to Save the record data. The last button is used to exit the form, and to abort the data entry, ie. discard changes. | |
| oNew : cmdTblNew<br>oQuery : cmdTblQuery<br>oEditSave : cmdTblEditSave<br>oExitAbort : cmdTblExitAbort | SetButton()* - This function takes one parameter to denote<br>the button currently pressed (ie. "NEW", "QUERY", "EDIT/SAVE", "EXIT/ABORT"). From this information the other buttons are enabled/disabled.<br>The form state is also taken into account. The buttons are enabled/disabled using the following states scheme:<br><br>State New Query Edit Save Exit Abort<br>fsReadOnly + + + - + -<br>fsNew - - - + - +<br>fsEdit - - - + - +<br>fsQuery + - + - - +<br><br>oNew.Valid() - First call the original valid, then call cmgTblEditBtn.SetButtons("NEW")<br>oQuery.Valid() - First call the original valid, then call cmgTblEditBtn.SetButtons("QUERY")<br>oEditSave.Valid() - First call the original valid, then call cmgTblEditBtn.SetButtons("Edit/SAVE")<br>oExitAbort.Valid() - First call the original valid, then call cmgTblEditBtn.SetButtons("EXIT/ABORT") |
| CommandGroup, Table, Editing, Buttons, State | |

| VCX\TXT\ADS | |
|---|---|
| txtPstCd | Baseclass hbbTextBox |
| Version 1.000 | Author BNK |
| This field contains a ZIP code as '9999AA' (dutch format). All letters are converted to UPPERcase, on-screen a space should be shown between the '9999' and the 'AA'.<br>Perhaps validation using a PTT postcode table could be realized. | |
| ControlSource = PSTCD<br>InputMask = "9999 AA"<br>FormatMask = "R" | FldFmt()* - Convert letters to uppercase. Do not store the extra space, it is for input convenience only! Most of this should be taken care of by the InputMask property! |
| Datafield, Address data, Postcode | |

<table>
<tr><td colspan="2">VCX\HBB</td></tr>
<tr><td>hbbContainer</td><td>Baseclass Container</td></tr>
<tr><td>Version 1.000</td><td>Author BNK</td></tr>
<tr><td colspan="2">This component is used to create FDC's, it can contain other components. The component has added intelligence for handling aliasses in the components within the container.</td></tr>
<tr><td>Name = cnt...<br>Alias* = This property is used to store an alias that can be used by the AddAlias() function to add an alias to the ControlSource or Alias property of a control.<br>ForceAlias* = Should existing aliasses be overwritten.<br>State* = fsReadOnly. The state of the container, initially fsReadOnly. A container can be in an fsNew, fsEdit, fsReadOnly or fsQuery state. The fsNew state indicates that the data in the container is new, not saved in the table yet. The fsEdit state is used to indicate that the data in the container is currently being edited, or at least editing is activated. fsReadOnly indicates that the data is currently not being edited, but the data is present! in the table. The fsQuery state is used to indicate that the container is currently in Query mode. The data entered will be used to find (a) matching record(s).</td><td>Init() - (pseudocode follows)<br>** By default we call this procedure.<br>=AddAlias2All()</td></tr>
<tr><td>HBB baseclass, Container, Alias, State</td><td>CHECKLIST:<br>* Is a SET PROCEDURE TO ... ADDITIVE with the AddAlias function in it present somewhere in the project.<br>* Make sure that all data fields associated with the container's main table are set to READONLY (ie. the controls ReadOnly property=.T.)<br>* Make sure that the fsNew, fsEdit, fsReadOnly and fsQuery states are defined using #DEFINE statements.</td></tr>
</table>