Generative Programming of Graphical User Interfaces

Max Schlee

28 November 2002

University of Applied Sciences of Kaiserslautern
Department of Computer Science and Micro-System Engineering
„Applied Computer Science“

Advisor: Prof. Dr. Ulrich W. Eisenecker
Co-Advisor: Prof. Dr. Manfred Brill
Table of Contents

Part 1 GUI

1 Introduction .......................................................................................................................... 3
  1.1 Purpose ............................................................................................................................ 3
  1.2 How to read this thesis ..................................................................................................... 3
  1.3 Acknowledgements ......................................................................................................... 4

2 The GUI Background ......................................................................................................... 6
  2.1 Hard and Software Ergonomics ..................................................................................... 6
  2.1.1 Dialog ....................................................................................................................... 6
  2.1.1.1 The basic rules of dialog presentation (ISO 9241-10:1996) .................................... 7
  2.1.1.2 Dialog types ........................................................................................................ 8
  2.1.1.3 Dialog modes ....................................................................................................... 12

  2.1.2 GUI style guidelines ................................................................................................. 12
  2.1.2.1 OS specific guidelines ....................................................................................... 12
  2.1.2.2 Company- and project-specific guidelines ......................................................... 12

  2.1.3 Window anatomy ...................................................................................................... 13
  2.1.3.1 Parts of the application window ......................................................................... 13
  2.1.3.2 Dialog box .......................................................................................................... 14

  2.2 Overview of graphical user interface elements ............................................................. 16
  2.2.1 Static elements ......................................................................................................... 16
  2.2.2 Display elements ..................................................................................................... 16
  2.2.3 Buttons .................................................................................................................... 17
  2.2.4 Setting elements ..................................................................................................... 18
  2.2.5 Selection elements ................................................................................................. 18
  2.2.6 Input elements ........................................................................................................ 21

  2.3 QI ................................................................................................................................. 22
  2.3.1 Introduction .............................................................................................................. 22
  2.3.2 Basic concepts of programming with QI ................................................................. 22

Part 2 Generation

3 Generation .......................................................................................................................... 31
  3.1 GUI Generation .............................................................................................................. 31
  3.1.1 Introduction .............................................................................................................. 31
  3.1.2 GUI Builders .......................................................................................................... 32
  3.1.2.1 Qt designer ....................................................................................................... 33
  3.1.2.2 Glade ................................................................................................................ 34

  3.1.3 GUI-Generators (State of the Art) .......................................................................... 35
  3.1.3.1 Research Done On User Interface Generation .................................................. 36
  3.1.3.2 Application Domains of GUI-Generators .......................................................... 37
  3.1.3.3 Non-Model-Based GUI-Generators ................................................................. 38
  3.1.3.4 Model-Based GUI-Generators .......................................................................... 39

  3.2 Generative Programming ............................................................................................. 37
  3.2.1 Generative Domain Model ....................................................................................... 37
  3.2.2 Software System Families ....................................................................................... 38
  3.2.3 The Domain Engineering ....................................................................................... 39

  3.2.4 Feature Modeling ..................................................................................................... 40
  3.2.4.1 Feature Models .............................................................................................. 40
  3.2.4.2 Feature Diagrams ......................................................................................... 40
  3.2.4.3 Additional Information of Feature Models ....................................................... 41

  3.2.5 The Generator Principle ......................................................................................... 42

  3.2.6 Technology Projection ............................................................................................. 43

  3.3 Frame Technology ....................................................................................................... 44
### Part 3 Solution

#### 4 Realisation

<table>
<thead>
<tr>
<th>4.1 Prototype Description</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1 The Popup Menu «File»</td>
<td>70</td>
</tr>
<tr>
<td>4.1.2 The Popup Menu «Edit»</td>
<td>71</td>
</tr>
<tr>
<td>4.1.3 The Popup Menu «View»</td>
<td>71</td>
</tr>
<tr>
<td>4.1.4 The Popup Menu «Effects»</td>
<td>72</td>
</tr>
<tr>
<td>4.1.5 The Popup Menu «Options»</td>
<td>72</td>
</tr>
<tr>
<td>4.1.6 The Popup Menu «Windows»</td>
<td>73</td>
</tr>
<tr>
<td>4.1.7 The Popup Menu «Help»</td>
<td>73</td>
</tr>
<tr>
<td>4.2 The prototype feature model</td>
<td>74</td>
</tr>
<tr>
<td>4.2.1 The Equivalence Categories of the Implementation Components</td>
<td>77</td>
</tr>
<tr>
<td>4.2.2 The Dependencies Between the Implementation Components</td>
<td>79</td>
</tr>
<tr>
<td>4.2.3 The GenVoca Architecture</td>
<td>80</td>
</tr>
<tr>
<td>4.2.4 The GenVoca Grammar</td>
<td>82</td>
</tr>
<tr>
<td>4.3 Graphical DSL</td>
<td>84</td>
</tr>
<tr>
<td>4.4 Angie-Based gui-generAtor (ABA)</td>
<td>86</td>
</tr>
<tr>
<td>4.4.1 The distribution of generative domain model in ABA</td>
<td>87</td>
</tr>
<tr>
<td>4.4.2 The system requirements</td>
<td>87</td>
</tr>
<tr>
<td>4.4.3 The generation process in ABA</td>
<td>88</td>
</tr>
<tr>
<td>4.4.4 ABA-User Interface</td>
<td>88</td>
</tr>
<tr>
<td>4.4.5 The Process of Creating a XML-Specification</td>
<td>91</td>
</tr>
<tr>
<td>4.4.6 Frame Creation</td>
<td>92</td>
</tr>
<tr>
<td>4.4.7 The Assembly of a Component</td>
<td>93</td>
</tr>
<tr>
<td>4.4.8 Realization (ANGIE)</td>
<td>94</td>
</tr>
<tr>
<td>4.4.9 Projects Using ANGIE</td>
<td>98</td>
</tr>
</tbody>
</table>

#### 5 Conclusion and Future Research Directions

Index of Figures and Listings | 98 |
--- | ---
Figures | 98 |
Listings | 99 |
References | 106 |
1 Introduction

1.1 Purpose

The amount of time necessary for the development of an appealing Graphical User Interface (GUI) is still equal to, or even higher than the time necessary for the implementation of application functionality. However, the majority of applications need a GUI [EHSL2002]. This is why it became necessary to look for a solution allowing the reduction of development time as much as possible.

Generative Programming (GP) is a new paradigm that allows the automatic creation of an entire software family, using the configuration of elementary and reusable components. GP can be projected on different technologies, e.g. C++-templates, JavaBeans, Aspect-Oriented Programming (AOP), or Frame technology.

The purpose of this thesis is to introduce the GP paradigm in the area of GUI application generation. This is why it focuses on Frame Technology, which aids the possible implementation and completion of software components. In order to analyze GP, a tool named ANGIE was used (developed by the Delta Software Technology GmbH). Though this tool was used in the GP-WEB Project supported by the German Ministry of Education and Research (Bundesministerium für Bildung und Forschung-, abb. BMBF), only rudimentary user interfaces for a web-application were generated.

The evaluation of such a technical projection was done using an application as an example. For this purpose, a family of image processing program was chosen, as it provides enough variability, which makes it an excellent base for generative programming. We did not attempt to search for efficient algorithms and image processing solutions in order to use them in the program because it was not our purpose to create a faster and more efficient image processing software. Our purpose was to test the use of the GP paradigm in the field of GUI generation.

In this thesis, it is also demonstrated how automatically customized executable applications with GUI parts can be generated from an abstract specification.

This research was carried out as part of the PoLITe project. PoLITe stands for “The Development and Testing of the Manual for the Software Product Line Implementation Technologies”. This project was supported by the Foundation for Innovation in Rhineland-Palatinate, Germany. The development of the software lines is documented at the Fraunhofer Institute for the Experimental Software Engineering (IESE) and at the University of Applied Sciences in Kaiserslautern (its branch in Zweibrücken).

1.2 How to read this thesis

Part 1: GUI. This part is about GUI-related aspects such as the ergonomic software presentation, the Window anatomy, and the GUI elements. It also contains the technical information about Qt, a GUI framework used in the implementation work of this thesis.

Part 2: GUI Generation. The part is about the base of the generation of GUIs. “State of the Art” gives an overview of achievements in this area and presents the existing tools.
Afterwards, the basic concepts and technologies used in the next part ("The Creation Process") are introduced.

Part 3: The Creation Process. This is the most important part. It shows the use of concepts and technologies introduced in part 1 and part 2. It also describes the creation of designs and problems that came up in the process. The results are also presented in this part. Furthermore, it contains references and a glossary.

1.3 Acknowledgements

First of all, I would like to thank professor Ulrich Eisenecker for his excellent support and numerous suggestions for this thesis.

I would like to thank professor Manfred Brill for his interest in this research. This really meant a lot to me!

I thank my sister, Natalie Goes, for supporting the translation of this thesis into English. I am also very obliged to my brother-in-law, Travis A. Goes, for his proofreading of the English translation.

I would like to thank Mr. Alfons Brandl who allowed me to use his dissertation “Generierung interaktiver Informationssysteme und ihrer Benutzungsoberflächen für mehrere Benutzer”.

I thank my parents who supported me in my studies.

Last but not least, I would like to thank the members of the PoLITe project at the Fraunhofer Institute for Experimental Software Engineering (IESE) Dirk Muthig, Thomas Patzke, Michalis Anastapoulos, Roland Laqua, and at the University of Applied Sciences in Kaiserslautern (its branch in Zweibrücken) Marco Emrich and Andre Bossert. It was great working with you.
Part 1

GUI
2 The GUI Background

2.1 Hard and Software Ergonomics

The word “ergonomics” has its origin in the Greek word “ergon” (“work”) [Duda]. Ergonomics is a discipline dealing with the working people’s productive capacities and their limits as well as with the adjustment of work tools and working environment to the abilities and needs of human beings. In computer science, there are two types of ergonomics connecting the development of dialog-oriented systems: Hardware and software ergonomics.

The purpose of hardware ergonomics is to adjust working devices such as screen, keyboard etc. as well as the working environment including, for example, a table, a chair etc. to the physical and psychological characteristics of people [Dudb]. This type of ergonomics will not be regarded here.

The purpose of software ergonomics is to develop the criteria and methods of interactive program systems that satisfy the user’s needs concerning his mental, physical and social characteristics. This requires the cooperation of computer science and other disciplines, such as medicine, psychology and work sociology [Dudb].

In the 60s, scientists already tried to create ergonomic software [HS94]. However, an important part of the ergonomics theory was not developed until the 80s, when the systems that existed back then were analyzed. This part of the theory still influences the development of GUIs today [HS94]. The first GUIs offered the possibility to select a menu option or a control element directly using a mouse.

The focus of software ergonomics is not on the system but on the user. Only when the user’s requirements and expectations are taken into account is it possible to get his acceptance. While implementing a system, it is absolutely necessary to take into account mental abilities of the user such as memory, logical thinking, intelligence, etc. These abilities define the requirements for the dialog presentation, the development of menus, etc.

In the early 80s, Ben Schneiderman, one of the most famous software ergonomists, introduced the term “Direct Manipulation Interfaces” (DMI) that included all systems offering the users a more or less direct solutions to their problems. DMIs were not necessarily GUIs because the text mode applications such as the applications using Borland Turbo Vision could also meet this requirement.

2.1.1 Dialog

Dialogs serve as the basis for the communication between the user and the application. The ergonomics of a dialog-based user interface is measured by its ability to support the user’s work. The following is important:

- The dialog system must be adjusted to the information range required by the user [Hen99].
- The dialog system must offer the feedback about the dialog status to the user.
- The dialog system must have no components that the user considers unnecessary [Hen99].
- The components the user considers necessary must be organized taking into account the basic rules of human perception and action [Hen99].

2.1.1.1 The basic rules of dialog presentation (ISO 9241-10:1996)

In the early 80s, German researchers started working on a norm. It became obvious that an exact standard was impossible to develop. This is why they limited their work to the basic rules of the presentation and the general requirements an ergonomic dialog (between a person and a computer) must meet. The ISO 9241-10 describes the characteristics of an ergonomic system:

- Adequate tasks
- Self-descriptive system
- System that can be directed
- System meeting the user’s expectations
- Fault tolerance
- System that can be customized
- System that promotes learning

**Adequate tasks**

A dialog is adequate when it helps the user to carry out his task efficiently [ISO96].

The marker is automatically positioned on the first input element relevant for the task that is being carried out [Bal99].

**Self-descriptive system**

A dialog is self-descriptive when all of its steps are either accompanied by descriptive messages from the dialog system or they can be described to the user when requested [ISO96]. The dialog of a system must be able to let the user know at all times which level he is on, the system status (e.g. waiting for data, printing a file), which commands at that particular moment [Dudb]. This can be achieved with the help of an appropriate layout, labels and other appropriate interface elements, as they help the user to get acquainted with the system quickly and to use it intuitively [BS00].

**System that can be directed**

A dialog can be directed when the user can start it as well as influence its direction and speed until the goal is achieved [ISO96]. The user is excluded from the program direction when, for example, he cannot stop the dialog and has to continue it until a dialog session is completed which does not make sense. The user should also have the possibility to adjust the speed of the system to his liking: the scrolling speed, the repeating of the pressed key, animation speed etc. [BS00].
System meeting the user’s expectations

A dialog can be directed when it is consistent and is adjusted to the user’s abilities, e.g. to his skills in this field, to his education, his working experience as well as to the generally recognized rules [ISO96]. The reactions of the system have to be uniform and clear to the user. In equal situations, the commands must lead to equal results [Dudb], e.g. the dialog system messages always appear on the same spot. The user can always stop the dialog by pushing the same button [Bal99].

Error tolerance

A dialog is error-tolerant when the planned results can be achieved despite of the fact that they contain incorrect input, with the help of little or no corrections carried out by the user [ISO96]. For example, a formular-based dialog is not error-tolerant when it deletes the input area after the user made a single error, so that the user has to repeat all the input in order to correct the error [BS00].

System that can be customized

A dialog can be customized when it can be adjusted to the specific task requirements as well as to the user’s individual preferences [ISO96]. This is the flexibility of the system. The user must be able to use a program in different situations, depending on the task, work situation, his skills etc. [Dudb]. Some functions, such as file-open, can be accessed through toolbars, menus and shortcuts.Depending on the context, the user uses one way or another [BS00].

System promoting learning

A dialog promotes learning when it helps the user to learn about the system as well as aids him in this learning process [ISO96]. Learning by doing is promoted when the user is encouraged to experiment without running the risk of making fatal mistakes [Bal99].

2.1.1.2 Dialog types

There exist the following dialog types:

Natural language dialog

The researchers are still trying to make the communication with a system to resemble the communication with a person as much as possible [ZZ94]. Acoustic speech input, speech recognition and synthetic speech output, however, do not make a natural language dialog, as it flows in a person-to-person communication [ZZ94]. The first practical applications may have a very reduced vocabulary.

Question-answer dialog

This type of dialog is the easiest one to carry out and it is well supported by the procedural programming style. This type of dialog is also easy to use: the application asks a question that needs to be answered. The disadvantage of this dialog type is that the user only answers the questions asked by the application and it is impossible for him to take
initiative. This dialog type can be used when the user has little or no experience and when the input has to follow in a certain order [Gei90].

![Image](image1.png)

**Figure 1** Question-answer dialog

**Command dialog**

The user can take initiative when communicating with the computer [Zei94]. The user types in the data using a specific language, the command language defined by its specific grammar and vocabulary. It is assumed that the user knows the commands and the syntax he uses by heart, which requires a lot of learning and typing. The advantages of this command user interfaces are:

- Setting any parameters supports complex actions [BS00].
- Recent actions can be looked up in history [BS00].
- Customizing is easy thanks to script programming [BS00].

It is suitable for inexperienced users who can quickly deal with complex input [Gei90]. The command user interface is the oldest dialog type. The DOS Shell and Unix Shells (bash, csh) are the best-known command user interfaces.

![Image](image2.png)

**Figure 2** DOS Shell

**Menu dialog**

A menu dialog offers 1 of n selections; all the options are shown horizontally or vertically. The menu dialog is an important step towards showing the user the state of a program, the structure of its functions and files in the user-program dialog [Zei94]. The entire number of options can be divided into smaller groups that the user runs step-by-step as different menu levels. This way, the structure becomes clearer and the amount of information the user is required to read is reduced [Gei90]. Apart from the quick access to the important menu options, it has another advantage: The user is completely led through the system and it is impossible to make a selection of options that does not make any sense.

![Image](image3.png)

**Figure 3** Menu dialog
Direct manipulation dialog

The concept of direct manipulation came from Ben Shneiderman [She92]. Objects, their attributes and interrelations are presented graphically. The main idea of this concept is the graphic presentation of reality. This gives the user a feeling as if he were working with real objects. The changes become visible immediately, so that the user can quickly decide whether a particular change leads to the planned goal or further actions are necessary in order to achieve it [BS00]. A typical example of this dialog type is the trash can in Windows.

Figure 4 Direct manipulation dialog 1 (screen shot WindowsXP)

Figure 5 Direct manipulation dialog 2 (screen shot WindowsXP)

Form dialog

A form dialog is frequently used in typical business applications [Zei94]. When a mouse is used, the experienced users also become interested in this dialog type because they can access separate input areas.

The user’s part of inputting the data can be reduced if a form dialog is combined with a menu dialog [Gei90]. The user sees several information units at once; these units describe facts belonging together. The user can proofread and correct this description before he confirms the validity of its contents. The application programmers believe that the form dialogs have the advantage that almost all systems have the tools to easily carry out this type of dialog [Zei94].

Figure 6 Form dialog (screen shot Microsoft Paint Version 5.1)

Form dialogs must let the user easily recognize the logical structure of the information and the meaning of the information units [Zei94]. In the studies of human perception
there are presentation rules that show how the logical structures are recognized based on their visual presentation and order. These presentation rules are of considerable help when designing an appropriate layout.

In this paper, only several presentation laws are described.

**The Closeness Rule**
People perceive close objects as objects belonging together [Zei94]. This means that the pieces of information that belong together must also be placed close to each other.

![Figure 7 The Closeness Rule (based on [Zei94])](image)

**The Equality Perception Rule**
People perceive objects that look exactly alike as objects belonging together [Zei94]. This means that similar information pieces must also be presented in a similar way, e.g. using the same color, border and font.

![Figure 8 The Equality Perception Rule (based on [Zei94])](image)

**The Completeness Perception Rule**
People tend to perceive incomplete shapes as complete, e.g. to complete incomplete circles and rectangles. Bordered areas are perceived as units [Kli95]. This means that different pieces of information must be clearly separated from each other using blank spaces and separation lines.

![Figure 9 The Completeness Perception Rule (based on [Zei94])](image)

**The Boldness Perception Rule**
Clear and bold images are perceived as pleasant and are easily kept in memories [Kli95]. As a consequence, it is advisable to use few but clear and bold shapes.
2.1.1.3 Dialog modes

There are modal and non-modal dialogs.

- A modal dialog must be completed before another task of the application can be carried out. Modal dialogs can be used, for example, to open a window containing an important error message the user has to acknowledge it before he can get back to working with the program.
- A non-modal dialog enables the user to stop the dialog and carry out another action while the dialog window remains open. The goal is to be able to use as many non-modal dialogs as possible because this increases the flexibility of the program [Bal99].

2.1.2 GUI style guidelines

Uniform GUIs are required by the ISO 9241-10 in order to meet the user’s expectations and to promote learning. Specific considerations are necessary to provide a uniform look and feel of the GUI that is created by many people. The most important help are style guides and guidelines that provide GUI consistency. There are operating system specific rules as well as company- and project-specific rules.

2.1.2.1 OS specific guidelines

These are the rules set by the OS-developers who determine the basic look and feel of the dialog interfaces on a given operating system (Windows, OS/2, MacIntosh, Nextstep, OSF/Motif). These rules determine the use of the available elements, for example:

- The maximal number of colors, e.g. 5 colors (+- 2);
- The menu presentation e.g. 8 options in a menu lead to the smallest number of errors;
- Check boxes are used when the user is able to select several or no options. Starting from 5 elements, it is better to use list boxes (combo boxes).

2.1.2.2 Company – and project-specific guidelines

These style guides are an addition to OS-developer rules. They determine application-specific rules and have two purposes [BS00]:

- They regulate the use of the GUI surrounding, e.g. its font size, colors etc.
- They guarantee the uniformity of the user interface within the project: window layout (e.g. buttons are situated in the left or right bottom corner), the uniform presentation of frequently used data (e.g. address, bank account).
In order to make the programmer follow these rules, every dialog has to be proofread in accordance with the rules by the quality assurance [BS00]. Another way for ensuring conformity with style guides (which is better) is to use the GUI-builders and GUI generators when developing a GUI.

2.1.3 Window anatomy

A window is the basis for the presentation of any application information. A window is a rectangular area where the information exchange between the user and the application takes place. All the Widgets of the GUI are composed of sub-windows. In a way, every button in a dialog box is a sub-window of that dialog box. In The Windows Style Guide [Bal99] there are two types of windows, primary and secondary windows.

- Primary windows are windows where the user’s main activities take place. The most important primary windows are the application windows.
- Secondary windows are used for selecting options or when secondary activities (secondary dialogs) are carried out.

2.1.3.1 Parts of the application window

From this window all the other windows of the application can be opened. A typical application window contains a window frame, title bar, menu bar, toolbars, scroll bars, and a status bar.

![Window anatomy](image)

Every window has a border. The purposes of the application window title bar are to identify the window by displaying a title and the name of the current document (for a document-based application), and to display an icon for the application or the current document. The title bar can contain three window manipulation buttons: Minimize, Maximize and Close. The user can click and drag the title bar moving the window to a different location on the desktop. It is also possible to double-click on the title bar to quickly maximize and restore the window. The menu bar is situated directly below the title bar and contains a list of available commands. The tool bar appears directly below the menu bar and provides a great place for the most frequently used features. The buttons on the toolbar can contain graphics, text and tool tips [Cal96]. Using the scroll bars it is possible to move the document within the work space. The status bar provides a
place for the product to communicate to the user. It can display a progress indicator, date, time, etc. or verbal description of a selected menu item. Size grip is used to change the size of the application window.

There are two different modes of document-based interfaces: the *Single Document Window Interface* (SDI) and the *Multiple Document Window Interface* (MDI). In an SDI application, the workspace is at the same time the application window. This means it is impossible to open two windows in the application at the same time [MT99]. In an MDI application, the workspace is empty. From the workspace the user can open child windows. It is possible to work with several documents at once. A child window appears in the workspace and is limited by it.

### 2.1.3.2 Dialog box

Dialog boxes are used in secondary dialogs that are limited to the input about the interaction elements of the workspace [Bal99]. The dialog boxes can be modal or non-modal see 2.1.1.3. The modal dialog boxes are subdivided into application modal and system modal dialog boxes [ZZ94].

- Application modal dialog boxes: all the dialog objects of the application are blocked and no input can be entered. The input required by the application can only be delivered in an application modal dialog.
- System modal: all the dialog objects on the screen are blocked and input can not be entered with exception of the dialog object that is marked as system modal.

A dialog box must be easily comprehensible. Specific types of dialog boxes can be offered for specific dialog situations occurring in almost all applications.

![Message Dialog](image12.png)

**Figure 12 Message Dialog (screen shot MiniABA)**

A Message Dialog gives the user a message and he must acknowledge it.

![File Dialog](image13.png)

**Figure 13 File Dialog (screen shot Microsoft Windows XP)**

A File Dialog permits the user to select the directory and the name of a file.
A Color Dialog offers the user the option to select a color.

A Print Dialog shows all the available printers in a drop-down list box of which the user can select one. In addition, the redirection of the printing in to file, the number of copies, paper size and orientation can be set.
2.1.4 Overview of graphical user interface elements

The most important purpose of any program’s graphical user interface (GUI) is to simplify the program user’s work. This simplification is achieved primarily through the interplay of many distinguishable types of elements developed for this purpose.

2.1.4.1 Static elements

Static elements do not actively participate in users’ actions. Examples of these elements are: texts, pictures or graphic elements providing an explanation, headlines and groups of visual elements.

**Group Box**
The Group Box compiles different control functions into what is known as a “container”. A container is rectangular in appearance with an optional heading explaining the purpose of the group. Though the rectangular frame does not receive direct data input, users can quickly access the group box by pressing the key which is underlined in the group box title.

![](figure16.png)

**Figure 16 Group Box**

**Label**
Labels contain read-only text but the value of the labels can be changed while accessing the corresponding application. If necessary, the appropriation of access keys can also be defined, though in order to achieve this, it is necessary to switch to a corresponding control element, because the label itself can not receive data input.

![](figure17.png)

**Figure 17 Label**

2.1.4.2 Display elements

Display elements are used to inform the user about the status of work accomplished by the program.

**Progress Bar**
A Progress Bar fills from left to right showing the user the progress of work in percentages. When the bar is completely full, it signifies the completion of a programs operation. This is especially useful when following the progress of long operations, because the user can easily check the status.

![](figure18.png)

**Figure 18 Progress Bar**
2.1.4.3 Buttons

Buttons are elements the user can push when working with the program. In the majority of cases buttons are either “on” or “off”. In applications, there are three main types of buttons: Command Button, Radio Button and Check Button.

Command Button

Command Buttons are rectangular control elements. They are used to carry out certain operations that are started when the buttons are pushed. They can contain a description and/or a graphic element. Such well-known buttons as OK and Cancel are indispensable in the design of user dialogue boxes. All of the buttons with in a system need to have a standard format and follow the same basic rules, so that every user can identify them by appearance.

![Command Button](image)

Figure 19 Command Button

Radio Button (Option Button)

Radio Buttons give the user at least two options. Similar elements are grouped together and are not supposed to be used separately. The static element “Group Box” represents the group of Radio Buttons in the dialog. Each option label has to include the name of the access key in order for the user to be able to select one of the options quickly using the keyboard. If the application needs a large number of options to choose from, it is advisable to use a Single Select List Box.

![Radio Button](image)

Figure 20 Radio Button

Check Box (Flags)

A Check Box is used to turn settings on or off. Like a Radio Button, it can be selected or not. Unlike Radio Buttons, if one flag in the group is selected, it does not automatically deselect another flag in the same group. Flags that are similar or connected to each other need to be grouped together with the help of static elements such as a Group Box. Like Radio Buttons, Check Boxes need to have access keys for every flag. When a large number of options is needed or when the number of options is subject to change, a Multiple Select List Box should be used. A flag can also be neutral. This status is used when not all objects in the selection have the same attributes. An obvious example of this is the dialogue box “Properties” of WindowsExplorer, when selecting files with different attributes.

![Check Box](image)

Figure 21 Check Box (screen shot WindowsXP)
2.1.4.4 Setting elements

With the help of these elements users can set certain values within a specified range.

**Spin Box**
The Spin Box is rarely used. It consists of an input area and a little Scroll Bar with arrows (Spin). It gives the user access to a limited number of ordered values. A value may be typed in and then increased or decreased with the help of the spin. Any input value needs to be tested in case it exceeds certain limits. In order to skip the tests, this element can also be used in a cyclic mode, when the maximal value is followed by the minimal one and the other way around.

![Figure 22 Spin Box](image)

**Slider**
A Slider is used to set values that do not require precision, for example, the speed setting for the movements of the mouse arrow, the volume settings etc. The bar can be horizontal or vertical and can only use whole numbers.

![Figure 23 Slider](image)

**Scroll Bar**
A Scroll Bar is used to move the visible part of an object. It can also be used for horizontal or vertical navigation. The movements of the slider between the limits show the relative position of the visible part of the object. Based on this, the user can estimate the approximate size of the object.

![Figure 24 Scroll Bar](image)

2.1.4.5 Selection elements

Selection elements permit the user to select from a list of available options.

**List Box**
List Boxes contain a list of options to choose from. There are two types of List Boxes: Single Selection List Box and Multiple Selection List Box. The Single Selection List Box lets the user choose only one option at a time; the Multiple Selection List Box permits the user to choose several options at once.
Drop-Down List Box
This is a type of a List Box that utilizes the work space more economically. The user has to open the list before selecting one or more options. Afterwards, the list shows only the selected option.

Combo Box
The Combo Box is a combination of a Text Box and a List Box. The information can be either typed in or selected from a list. If the information is typed in, it can be either included in the list for further use, or not.

Drop-Down Combo Box
This is a combination of a Text Box and a Drop-Down List Box. The user can either select from a list of options or type the information in.
Date Picker
This element permits the user to select a certain date on the calendar. The user can also change months by pushing buttons with arrows.

Tree View Control
The Tree View Control is a hierarchy list of objects. The objects can be represented by appropriate symbols or by edges representing the hierarchy connections between the objects and showing buttons permitting the expansion and reduction of nodes in the hierarchy. Though these elements are optional the advantage of using them is obvious, as they permit the user to understand the tree-like structure.
List View Control
The List View Control is an extension of a List Box. It consists of several columns the width of which can be controlled by the user. The columns can act like Command Buttons, for example, they can sort the list. Every column can have a description and contain a graphical symbol with additional information about its contents. The rows represent the attributes of the object.

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Type</th>
<th>Date Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td></td>
<td>File Folder</td>
<td>03.10.2000 11:33</td>
</tr>
<tr>
<td>iEngine</td>
<td></td>
<td>File Folder</td>
<td>31.01.2002 23:41</td>
</tr>
<tr>
<td>Additions</td>
<td></td>
<td>File Folder</td>
<td>16.03.2002 19:06</td>
</tr>
<tr>
<td>ADOBEAPP</td>
<td></td>
<td>File Folder</td>
<td>02.02.2002 13:31</td>
</tr>
<tr>
<td>apache</td>
<td></td>
<td>File Folder</td>
<td>02.02.2002 16:23</td>
</tr>
<tr>
<td>DELTA</td>
<td></td>
<td>File Folder</td>
<td>11.03.2002 20:26</td>
</tr>
</tbody>
</table>

Tab Control (Property Sheet)
A tab Control consists of several pages, which can be viewed one at a time. When a user pushes the control element that controls one of the tabs, this tab becomes visible and the user can view and edit its properties. The Tab Title is any combination of text or icons.

2.1.4.6 Input elements
Input Elements permit the user to type in texts and numbers and to change them. These elements are fundamental for data input. The elements themselves do not have descriptions, thus Labels are required. It can also be necessary to define keys for quick access to input elements.

Line Edit (Text Box)
Line Edit is the simplest input element and has a rectangular shape. It is used for data input as well as output in one text line. It supports the Copy, Cut and Paste operations. In an application, the number of characters that can be entered may be limited.
Multi-line Edit (Multi-Line Text Box, Text Area)

Multi-Line Edit is somewhat similar to Line Edit. It is an extension that allows the input of multiple lines text. Horizontal and vertical Scroll Bars are used when viewing large texts in order to use work space more economically. When using this element, at least four lines must be visible.

```
Mother Nature's Son (The Beatles)
Born a poor young country boy - Mother 1Nature's son
All day long I'm sitting singing songs for everyone.
Sit beside a mounting stream - see her waters rise.
```

2.2 Qt

2.2.1 Introduction

For a long time, users of Linux had the impression that it was a “homemade” operating system because it did not have a uniform interface for all the applications. Every program had to be operated in a different way, which made the training for Linux significantly longer [Leb99]. The developers of KDE took this problem into account and made it their goal to create a uniform desktop based on an existing efficient tool kit. As the tool kit, they chose Qt developed by the Norwegian company Trolltech [Kra01]. Trolltech was founded in 1994 by several software developers, however, the development of Qt started already in 1992 [Leh99]. In Qt, the class concept of C++ is used; this is the reason why it can be easily expanded or adapted. This class library offers the user a solution for almost any problem, regardless of a specific platform, because Qt can be used not only with Linux but also with Windows for which it is also available. Among other things, Qt provides a framework of available system functionalities with limited portability, e.g. of data operations. This way, a high portability of Qt-based applications is achieved. In addition to the framework, the main task of the class library is to provide GUI elements in a simple way that works with different programs. Almost all the elements from the Windows as well as other elements are available [Kra01]. Besides, Qt offers an easy way to draw on the screen, anything from dots and simple lines to complicated figures and texts [Leh99]. Qt can be downloaded from the Trolltech home page: [www.troll.no/](http://www.troll.no/) The home page also contains the complete class description, a detailed tutorial and links to many sites for Qt-based projects.

2.2.2 Basic concepts of programming with Qt

QObject is the main class in the Qt hierarchy. It is the basis for all the GUI elements. Furthermore, a QObject contains the methods necessary for a productive message exchange between the instances of classes that are derived from QObject, the so-called
signal/slot concept. Besides, the QObject has methods that can capture the information about the hereditary structure of the classes derived from the QObject. The signal/slot concept as well as the information about the hereditary structures can not be carried out using the C++ concepts. This is why an additional pre-processor named Meta Object Compiler (MOC) was developed. It creates an additional source code file from a class definition of a class derived from the QObject. This file must also be compiled and linked to an executable program [Leh99].

2.2.2.1 The Hierarchy of QObject instances

QObject(QObject* parent = 0, const char* name = 0)

From the first parameter parent, a pointer can be transferred to another object of the QObject class or to a class derived from it. This way, the object is selected as the father node of the tree-like internal hierarchy. If the default parameter in its constructor is zero, the object is placed at the top of the hierarchy. The father node of the object can only be set via the constructor and can not be changed later. In the second parameter name, the programmer can give the object an additional name. If the default value “zero” is used in this parameter, the object receives no name. The names of the objects, unlike the first parameter, can be changed any time with the method QObject::setName.

QObject* pqFirst = new QObject(0, "I am without a Father");
QObject* pqSecond = new QObject(pqFirst, "My Father is pqFirst");

Listing 1 QObject constructor

In this Listing 1, a pointer pqFirst is created for an object with the name “I am without Father” that stands on top of the hierarchy. Then, another object with the name “My Father is pqFirst” is created and added to the pqFirst as its child.

If one of the created objects is deleted, this causes a chain reaction and all the child objects of the deleted object as well as their children are also deleted. In order to delete the objects delete is used, all objects are created using new [Leh99].

2.2.2.2 The Signal/Slot Concept

The GUI elements have to inform their surrounding that a user’s action has taken place. In most GUI libraries, this is achieved using the Callback functions. A function that is called is then recorded in the object. The Signal/Slot concept is an expansion of the Callback principle. In every class derived from the QObject new methods can be defined as signals or slots. The purpose of the signals is to send messages and the purpose of the slots is to receive them. The method QObject::connect connects the signals and slots at runtime.

The Signal/Slot concept is more flexible than the Callback principle, which is achieved by the following [Leh99]:
- Every class can define any number of additional signals and slots;
- The messages sent through the Signal/Slot mechanism can have any number of parameters of any type
- A signal can be connected to any number of slots
A slot can receive messages from several signals belonging to different objects. The user can create additional connections between the signals and slots or delete the existing connections at any time. If an object of the QObject class is deleted, all the existing connections to this object are also deleted in the object's destructor in order to prevent sending messages to the objects that no longer exist.

The signals and slots are declared in the class definition like normal methods. They must have a return type. In the declaration, the additional specification symbols “signals” and “slots” are introduced. The first line of a class definition must contain the Q_OBJECT Macro. This macro adds several internal structures to the class.

```cpp
class MyClass: public QObject {
    Q_OBJECT
    public signals:
        void mySignal(int);
    public slots:
        void mySlot();
};
```

Listing 2 Signal/Slot

In this class definition, a signal mySignal is defined by an int-parameter; the slot mySlot has no parameters. The slots act like normal methods and they can also be called like normal methods.

In order to connect a signal to a slot, the static method connect of the QObject class is used. It requires four parameters:

```cpp
QObject::connect(const QObject* sender, const char* signal,
                 const QObject* receiver, const char* slot)
```

“Sender” and “signal” specify the object sending the message and its signal method. “receiver” and “slot” specify the object that must receive the message. The method is transferred into a string with the help of 2 macros: SIGNAL and SLOT.
The first two lines are the directives for the pre-processor that include the header files `qapplication.h` and `qpushbutton.h`. These header files have the same names as the corresponding classes they contain. All Qt-classes start with the letter “Q”.

The first line of the function `main` creates an instance of the `QApplication` class. The purpose of this object is to take over several initializations and to interpret the parameters of the command lines. In the next line, an instance of the `QPushButton` class is created. This class represents the button labeled “Exit” that is not visible yet. The next command, “connect”, connects the button to the `QApplication` object. Clicking this button is then associated with quitting the program. When the `show` method of the `QPushButton` object is called, the button becomes visible. However, the button does not react to mouse clicks or a command typed on the keyboard because the `exec`-method of the `QApplication` method that receives such commands is executed in the last line of the `main` function. This method is constantly reading the most recent messages from the X-server. It interprets the messages and sends the interpretations to the objects involved in the events. It continues reading until the `quit` method of the `QApplication` object is called. In our case, this happens when the Exit button is pressed. This is the way the example appears on the screen.

![Figure 36 Screen shot of Qt-Application example](image)

### 2.2.2.3 QWidget Class

The objects are represented on the screen by rectangular areas of a certain size and with a certain position. This is carried out using the `QWidget` class derived from the `QObject` class with the purpose to copy the tree-like hierarchy. Every widget is drawn within its father widget. The widget objects without a father are called Top-level widgets. At first, a Top-level widget is not visible. In order to make it visible, the `show`
method must be called. Using the hide method, the user can hide it. The sub widgets are represented only within their father widgets. They automatically appear on the screen, the user does not need to do anything.

If something must appear on the screen, it is necessary to derive a class from the QWidget class and to overwrite the virtual `paintEvent` method.

**Dialogs**

Dialogs are top-level widgets. The QDialog class serves as the basis for almost all dialogs. It is derived from QWidget. There are two different dialog modes: non-modal and modal. A non-modal dialog is called like a normal QWidget window. In the modal mode, the dialog window is the only window receiving messages from the mouse and the keyboard, all the other windows of the program can not be used while the dialog window is being used.

Specific dialogs dealing with a particular problem are often needed. The development of such a dialog is not just a selection of necessary elements. A good program does not require extensive training and is easy to use [Leh99].
```cpp
#include <qapplication.h>
#include <qpushbutton.h>
#include <qmultilineedit.h>
#include <qdialog.h>
#include <qlayout.h>

void main(int argc, char** argv)
{
    QApplication qapp(argc, argv);
    QDialog* pqdMessageWindow = new QDialog();
    QMultiLineEdit* pqmleMessage = new QMultiLineEdit(pqdMessageWindow);
    QPushButton* pqpbClear = new QPushButton("Clear", pqdMessageWindow);
    QPushButton* pqpbExit = new QPushButton("Exit", pqdMessageWindow);
    QVBoxLayout* pqvblTopLayout = new QVBoxLayout(pqdMessageWindow);
    QHBoxLayout* pqhblButtonsLayout = new QHBoxLayout(pqvblTopLayout);

    pqdMessageWindow->setCaption("DialogWindow");
    pqdMessageWindow->setFixedSize(200, 100);
    pqhblButtonsLayout->addWidget(pqpbClear);
    pqhblButtonsLayout->addWidget(pqpbExit);
    pqvblTopLayout->addWidget(pqmleMessage);
    pqvblTopLayout->addLayout(pqhblButtonsLayout);

    QObject::connect(pqpbClear, SIGNAL(clicked()),
                     pqmleMessage, SLOT(clear()));
    QObject::connect(pqpbExit, SIGNAL(clicked()),
                     &qapp, SLOT(quit()));

    pqdMessageWindow->show();
    qapp.exec();
    delete pqdMessageWindow;
}
```

Listing 4 Dialog example

After the creation of the `pqd` message window top-level widget, three sub-widgets are created: `pqmleMessage`, `pqpbClear`, `pqpbExit`. Then, the layout objects such as `pqvblTopLayout` and `pqhblButtonsLayout` are created to control the position and the size of the sub-widgets. The layouts automatically organize the new widgets. The title of the dialog window is set using the `setCaption` method. The length of the dialog is adjusted using the `setFixedSize` methods. The “clicked”-signals of both buttons are connected to the corresponding slots.
Menus

A menu is a list of commands that can be activated by clicking on them or by using shortcuts. The typical options of any menu include File, Edit, View, Help. The QMainWindow is in charge of the menu bar. Apart from the menu list, there are also popup menus that can normally be used within the menu bar and that show a list of sub-commands for every command in the main menu. The QPopupMenu class is in charge of the popup menus. The typical sub-commands of the main command “File” include New, Open, Save, Save As, Print, Exit.

```cpp
#include <qapplication.h>
#include <qwidget.h>
#include <qpopupmenu.h>
#include <qmenubar.h>

void main(int argc, char** argv)
{
    QApplication qapp(argc, argv);
    QWidget* pqwMessageWindow = new QWidget();
    pqwMessageWindow->setFixedSize(50, 30);
    QPopupMenu* pqpmFile = new QPopupMenu();
    int nID = pqpmFile->insertItem("Exit", &qapp, SLOT(quit()));

    QMenuBar* pqmbMenu = new QMenuBar(pqwMessageWindow);
    pqmbMenu->insertItem("File", pqpmFile);

    pqwMessageWindow->show();
    qapp.exec();
    delete pqwMessageWindow;
}
```

Listing 5 Menu example

This is a typical example. If the user selects the command “Exit” from the File popup menu, the application ends. In the second line of the main function the instance of the QWidget class is created. The setFixedSize method adjusts its size. In the next line, the QPopupMenu object is created. The insertItem method provides the command’s
ID that can be saved for future comparisons in a variable. The first parameter is decisive for the command label. The second and third parameters determine the object and its slot that can be called. In the next statement, the QMenuBar object is created as a parent of the QWidget object. With the help of the insertItem method, a list of sub-commands of the menu option “File” is added.

Figure 38 Screen shot of menu example
Part 2

Generation
3 Generation

3.1 GUI Generation

3.1.1 Introduction

About twenty years ago computer users were satisfied with the character-based applications that carried out their work after the parameters were entered and without interaction with the users. However, the users were required to have certain knowledge of the operating system they were using.

Later, applications with Graphics User Interface (GUI) were created and they opened new horizons for users. In these applications, human imagination and perception were taken into account, this is why they could be used intuitively, which made work much easier. One of the advantages of these applications is the similarity of conditions in different operating systems, so that it is possible to feel comfortable even if working with different platforms containing such GUI applications. The user needs to learn how to handle the application only once and then he can use this technique with every GUI application.

These applications were easy to use, which does not mean they were easy to develop. Their development took a long time and required competent software developers which lead to high development costs.

Today, hardly anybody can imagine having software without a GUI. A GUI simply belongs together with the word “software”. No applications without a GUI are accepted on the software market, with an exception of such programs that do not require interaction with users.

Such Tools as, for example, the Use Interface Management Systems (UIMS), still make the development of GUls relatively complicated because every surface object must be created separately and must be written in a specific programming language at a relatively low abstraction level [Fäh96].

Tools that support paradigms of intertwined design and implementations that facilitate low-level design-, do not have an adequate balance between providing high-level design automation and giving designers extensive control over interface design [Bas91].

People have always searched for new self-development opportunities. Their mind helps them to develop different tools that liberate them from time-consuming and tiring activities. This way they create room for the development of new, innovative ideas [Otr02b].

The two categories of tools that support the development of user interfaces are builders and generators [Sch95a], [Pin95]
3.1.2 GUI-Builders

Builders start the task of building a user interface with the assembly of the static layout of the desired interface using What You See Is What You Get (WYSIWYG) editors. The focus of interface modeling is the issue of user-interface application semantics, interface presentation, interactive behavior, and dialog sequencing. Semantics deal with the description of objects of interest, operations for manipulating the objects, and the parameters required by the operations [Pin95]. A user interface includes a description of the presentation as well as an expression of dialog control. The user interface software designer uses a presentation specification tool to construct a static display from a library of interaction objects and to define links between elements of the display and the client program. Using a dialog control specification tool, the user interface implements or describes how interaction objects will be managed [Bas91]. The presentation shows how to display application objects and operations under various conditions [Pin95].

- Behavior defines the type of interactions supported by the interface, including the region where interactions are active, interactive events that trigger and terminate the interactors, and actions that interactors perform [Pin95].

- Sequencing can be modeled or can be derived from the semantic models at runtime [Pin95].

Whenever user interfaces have to support applications with the dynamic construction or manipulation of complex structure objects, keeping the user interface layout consistent with the application and dialog state is often poorly supported.

- While using WYSIWYG interface tools, style-guide compliance of user interfaces has to be checked “by hand”[Sch95a].

- Creating an alternative user interface is often as time-consuming as the design of the original user interface, because a lot of layout detail has to be reconsidered [Sch95a].

“In large organizations, there are usually different groups working on the design of the dialogs in an application and on their actual implementation. The designers know a lot about user interface design and usability issues but not necessarily enough about programming to write at least the skeletons of the dialogs themselves. Without a GUI builder, the designers usually draw the dialogs, either on a sheet of paper or with a drawing program, and the programmers then try to implement this design as close to the draft as possible. With a GUI builder, on the other hand, the designers can just design the dialogs in the GUI builder and click on the Generate button (or use whichever means of code generation the GUI builder provides), and the code is already written - the programmers can concentrate on more creative jobs. In fact, if you are a programmer, you do not even need to tell the designers about the Generate button-just ask them to save their design; then you can take the file, load it into the GUI builder, and click that Generate button yourself. The designers and your boss will think that it takes you half a day to implement a dialog design, while it actually takes you only five seconds, and you can play Quake the rest of the time” [Tro99].

Interface builders give designers extensive control over a certain task, such as defining the properties of a push button (shading, color, label, font, size etc.) with an interactive development paradigm. However, designers are supposed to handle too many details, e.g., dim the presentation of an operation when the operation is not executable, even if they are routine, distracting, or sometimes irrelevant to the current design focus. Fur-
thermore, these tools do not support high-level interface design activities; designers cannot use them to address conceptual design concerns, e.g., the semantics of an operation that affects presentation, behavior, and sequencing of interface objects [Bas91].

Examples of GUI builders are:

- Qt Designer – [http://www.trolltech.com](http://www.trolltech.com)

### 3.1.2.1 Qt designer

A GUI-Builder for the cross-platform application frameworks Qt 3.0 is developed by Trolltech. The Qt Designer starts out as a simple dialog editor and turns into a complete GUI builder allowing the interactive construction of application windows as well as of menus and tool bars. It also supports manually created widgets. Furthermore, it allows editing the code of C++ methods directly in the *Rapid Application Development* (RAD) surrounding [GL02].

![Figure 39 Window of Qt Designer](image)

With the Qt Designer the user can place and lay out widgets on his forms visually and preview them immediately. In addition to this Qt's SQL module allows the user to preview the data base.

Qt Designer can also be used to connect signals and slots in order to give the created forms the behavior the user wants to achieve. If the user wants to customize the behavior beyond that provided by the default signals and slots, he can write his own source code directly in Qt Designer. If strict separation of the user interface and the implementation is preferred, this can be easily achieved by subclassing the forms created using Qt Designer[Tro02a].

The created GUI and connected signal/slots are saved in XML format in a UI file. This file can be opened and processed with the Qt Designer. At the compilation of the project, the UI-Compiler starts automatically to create a binary code from the C++ code in the output file. Then this output will be also compiled with the project [Nol01].
3.1.2.2 Glade

Glade is a GUI Builder for one of the best known Linux APIs that is used for the development of GUI with the Gimp tool kit GTK+ and GNOME. It is released under the GNU General Public License (GPL). GTK+ creates a hierarchy of widgets that have different signals. In order to develop programs with the GTK+, the developer must be familiar with its widgets and methods.

Glade helps GTK-Programmers to use a RAD-approach and enables them to create GUI-code in different programming languages (C, C++, Ada95, Eiffel, Perl and Python) via external tools which process the XML interface description files output.

![Figure 40 The Main Window of Glade (taken from [GL])](image)

The main window contains a list of all the windows and dialogs of the project. Double-clicking on an item in the list shows the corresponding window or dialog. The menu and toolbar commands create new projects, load and save them, as well as build the source code [GL].

![Figure 41 Widget palette of Glade (taken from [GL])](image)

The second window is the widget palette. It shows icons representing all the available widgets. In order to add new windows and dialogs to the project, the user needs to select an icon in the palette [GL].
The property editor allows to alter the properties of widgets, such as their size or their label text. The signals page also allows designers to add signal handlers to widgets. This way it is possible to specify a function which is called if a corresponding button is pushed. It is also possible to add a menu bar to an application [GL].

3.1.3 GUI-Generators (State of the Art)

In contrast, automatic GUI-generation systems exclude designers from the details by automating all decision-making during the interface development, that is, designers have little control over the design [Bas91]. Generators try to solve many of the above mentioned problems. They generate an executable user interface automatically, using a declarative description (model) of the properties of an interactive application, that is application interface, user interaction task space, and presentation design rules. Generators hide interface design complexity by automating all design decisions that otherwise would have to be made by human designers. However, they give designers little control over design decisions. A user interface generator automatically produces a user interface from a specification.

Advantages of using generators are:

- User interface designers do not have to make presentation design commitments at early stages of the design process; the user interface can be centered around the concepts of users and tasks [Sch95a].
- Model-based tools support rapid prototyping of working interfaces, even if the specification is not elaborated out in all details [Sch95a].

However, capturing sufficient knowledge to automate all the decision-making processes is impossible and this is, why asking the automatic systems to generate high-quality interfaces for a wide range of application, is not practical yet [Bas91].

Examples of GUI generators are:

- AppWizard
- AppExpert
- JANUS
- BOSS
- EmuGen
- GENIUS
3.1.3.1 Research Done On User Interface Generation

There exists a number of studies about generating user interfaces from specification on a very abstract level. Jade [ZM90] and ITS [Wie90] generate static user interface from frame-like dialog descriptions. UofA [SG91] and Micky [Ols89] use the specification of application commands in order to generate GUIs and dialogs [Fäh96]. Each of these systems uses its own notation for the specification of user interfaces [Fäh96]. “DON does global layout of dialogue boxes, applying graphics design considerations such as balance, weight, and symmetry. Using a generate and test strategy, many alternative designs are created and evaluated. The best designs are presented to the UI designer for selection and possible modification with an interactive design tool” [Fol95].

One of the first systems that uses a data model is HIGGENS (Hudson and King, 1986). However, no rules for the derivation of user interfaces were used in this system. In the system developed by de Baar et al 1992, the dialog boxes and menus are generated from extended data models. There is no graphic dialog model, the course of dialog is specified by pre- and post-conditions [Fäh96].

In their approach, Petoud and Pigneur 1990 generated user interface based on entity-relationship models. This dynamics can be shown graphically but the graph can not be altered [Fäh96]. Besides, no explicit rules considering the ergonomic software development during the generation process are taken into account [Fäh96].

One thing that the existing systems used to generate user interfaces automatically have in common is that they are not using software development methods and that they can not adequately define the dynamics of a dialog course [Fäh96].

“Auto-DevGuide [BM92] is a rule-driven extension to Sun Microsystem’s DevGuide user interface design tool. The system accepts a data model and uses the rules to select appropriate dialogue box widgets, such as sliders, scrolling lists, and buttons. The rules are adapted from OpenLook, and hence automatically enforce the OpenLook style guide. This approach ensures a higher degree of UI design consistency than would otherwise be possible. The same approach can be used with other style guides, such as MOTIF. Gray increased the sophistication of the system by using sequential and hierarchical dependencies among interface objects to order them in the dialog box” [Fol95].

The Computer-Aided Design of User Interfaces (CADUI) is a particular area of Human-Computer Interaction (HCI). It provides the software support for any activity that is part of the development life cycle of an interactive application, such as task analysis, contextual inquiry, requirements definition, user-centered design, application modeling, conceptual design, prototyping, installation, test, evaluation, maintenance. In CADUI, the activity of designing a User Interface (UI) for an interactive application and the activity of re-engineering a UI to rebuild its underlying models are also considered. Generally, the CADUI tools support any activity not just for programming, especially design, specifications, modeling and evaluation of a UI [VP99].
At the moment, there is no research or tangible results regarding GUI generation in the context of software system families and product lines. This is why the development in research and technology regarding the GUI generation is presented without a direct connection to software system families.

3.1.3.2 Application Domains of GUI-Generators

The productivity of the software developers increases because the use of generators reduces the cost of the application development. The software developer can concentrate on the creative part.

The ergonomics of the GUI presentation increases. In order to create a presentation of user interfaces for a specific task and for a certain user, specific knowledge is required that is partially available in the form of directives and style guides. Only part of these rules, e.g. the one concerning the selection of interactive objects and the one concerning the layout, is so specific that their automatic support is possible. In this area, the development tools of today also do not provide enough functionality [Fäh96]. If the development is carried out manually, there is always a risk that the set of ergonomic rules may be ignored. This is why it is worth to leave this job to the generator because then it is impossible to ignore the ergonomic directives.

The generated program code is uniform and easily comprehensible. Programming standards as well as the code formatting and the denomination of variabilities can be taken into account.

The number of errors is reduced. It is impossible to do anything wrong. The mistakes made during the programming and the details that get lost due to lack of attention are impossible.

It is possible to generate high-quality software with low manning and at low cost. For instance, using the application generator JANUS/Enterprise for the development of an application, Artem GmbH generated 80-95% of the source code of equally high quality which significantly lowered cost and saved much time [Otr02a].

The communication between the client or the user respectively and the software developer is made easier. The results of the discussion can be presented right away. This way, errors in the requirements or in the model can be identified early and their correction is significantly cheaper. The aims can be defined more precisely because the client is already familiar with what he will get.

GUI generators enable the rapid prototype creation. The prototype-oriented approach requires the fast creation of early executable system versions at low cost. Prototypes serve as living specifications that look better than abstract specifications and that show the user the functionality of the application. It makes sense to use existing models from software engineering, for example, from structured analysis, for the creation of prototypes. This way, the connection between prototypes and specification is created (the tools of today do not have it), the development process is made easier, and inconsistencies between the specification and implementation are avoided [Fäh96]. Instead of discussing abstract models or performance specifications, it is possible to discuss the spe-
pecifics of an application very early in the development process. This way, the objectives for the software system can be better defined and new ideas can be tried out right away.

For the users of the generated applications the optimised dialogs can be of advantage. They save time while working with the application because there is less button pushing, less mouse movements, and fewer windows involved.

3.1.3.3 Non-Model-Based GUI-Generators

The GUI-Generators can be divided into two categories: non model based and model based generators.

This kind of GUI-generators is very simple to use. The creation of an application does not require knowledge of object-oriented design, object-oriented programming, or libraries, used by the created application. This type of generators is not bound by any model. It is easy to create an application using this type of generators, because all the specification process of the application is carried out in the form of a user-generator dialog. This type of generators is also the fastest one when it comes to the creation of application prototypes. Examples of such generators are AppWizard by Microsoft and AppExpert by Borland. These generators are very popular with programmers and are more wide-spread than others.

Another advantage of this type of generators is that the developer can start developing an application “from scratch” without any clear concept and having only little time. AppWizard or AppExpert can create all the necessary files, resources, and other elements often used in applications. Instead of re-inventing the wheel, the programmer can just add specific code into the generated skeleton.

It gives a beginning programmer the opportunity to learn more about the programming techniques with Microsoft Foundation Classes (MFC) or Object Windows Library (OWL). AppWizard and AppExpert generate object-oriented code. In order to study one of the MFC or OWL methods, the programmer can create samples of different kinds of applications, study the resulting code and find out how the generator creates one or another functional element.

Before creating an application, it is important to keep in mind the disadvantages of this type of generators:

- The generator does not create complete programs. After the creation of the “skeleton”, the programmer has to take over and write remaining program code. In this process, comments that can be found in the generated files are useful. They tell the programmer where to insert one or another type of code.

- The programmer may not like the denomination of variables and the code formatting style created by the generator. It takes a long time to adjust the generated application to the style the programmer is used to.

- Generators of this type can only be used for the creation of new applications. The changes in the generated code are not taken into account and the generator can not modify the created files, it simply overwrites them. That is, the programmers extensions which are directly inserted in the generated code get lost, when the code is re-generated, respectively decisions made during the code generation can not be revised later without re-generation the code.
It is advisable to postpone using such generators until the programmer has enough experience in using the MFC or OWL. The better the programmer understands the classes of these libraries, the better he can modify the generated code, and the better are the chances that he makes the right decision at the beginning of application code generation.

3.1.3.3.1 AppWizard

AppWizard is one of the aspects of Microsoft Visual Studio known by all programmers regardless of their qualification. It makes the creation of Visual C++ project files with source code for the Windows application or DLL easy. The programmer is only required to go through the sequence of dialog windows and select different options characterizing the application that will be created. Based on this, a set of files the project consists of, is created. After the creation of the structure scheme, the software developer can modify the source code.

One of the disadvantages of the AppWizard is the fact that this generation tool is used by many thousands of software developers around the world, this is why it only has a basic set of functions and options supported by default. This results in the necessity to set all the required options every time AppWizard is used. Starting from the fourth version of MS Visual Studio, it was possible to create a separate AppWizard that could be customized by the programmers. They can add necessary files and libraries and define the program-specific set of standard options etc. For more information see [LA98].

AppWizard Inside

The first thing the user sees when starting the AppWizard is the order of dialog windows. When selecting the options in these windows, the values are kept in a certain order. They can be used by AppWizard as a reference to which windows need to be displayed next. The user, while navigating the dialog boxes, creates a path for AppWizard.
The selected options are used as substitution macros in program texts created for the project. AppWizard has a basic set of files for every created path. These files are used as a basis for the creation of source files of the project. They are called *frames*.

![AppWizard Dialogs](image)

**AppWizard Dialogs**

![AppWizard Table](image)

**AppWizard Table**

**Example**

The frame in the English version of the dialog file of the application resource contains the following lines that define the About-window:

```plaintext
... 
IDD_ABOUTBOX DIALOGEX 0, 0, 160, 129
STYLE DS_MODALFRAME|WS_POPUP|WS_CAPTION|WS_SYSMENU
CAPTION "About $$ROOT$$"
...
```

**Listing 6 AppWizard (“about”)-window definition**

In this case, ```$$ROOT$$``` is a macro that is defined at the option setting in the dialog boxes of AppWizard. The variable ```$$ROOT$$``` represents the project name and during the application project generation it will be replaced by the project name. This way, the dialog box “About” always contains the project name [LA98].

AppWizard takes into account the order of the dialog boxes with different options that are defined by the user and that are later used for the specification of frames containing the order ```$$placeholder$$``` where placeholder is a macro known to the AppWizard.

![Frames specification](image)

**Frame**

```
$$Function$$
{
   ....
}
```

**Function**

```
myfunction()
{
   ...
}
```
3.1.3.3.2 AppExpert

Starting with the 4th version of Borland C++, there is a GUI-Generator integrated in the Integrated Development Environment (IDE). This generator is able to create different “skeletons” using the OWL. After about an hour of work it is possible to start a program sample with all its instrumental menus, windows, text editors, file operations etc.

![AppExpert dialog box](image)

Figure 46 AppExpert dialog box

Like AppWizard, it can generate MDI-, SDI-, or dialog-based applications. Unlike AppWizard, it can install a tool bar, status line, drag & drop, printing as well as setting the background color separately. This enables to work out the visual presentation and functional opportunities of the generated application in more detail. AppExpert has many additional means listed in topics, such as the information about the author, copyright, where the application was developed, etc. The main window list contains such information about the application as whether the application can be minimized or maximized, whether it has scroll bars, the information about the MDI and SDI class hierarchy. All the options can be set in any order. After the selection of options is completed and the generate button is pushed, the generation will be carried out.

AppExpert always generates the following files for any application:

- A data base file for the AppExpert source (APX)
- A basis header file (H)
- A basis source code file (CPP)
- A project file (IDE)
- A resource header file (RH)
- A resource script file (RC)

Depending on the selected options it can also generate the following files:
• Help source text files (RTF)
• A help project file (HPJ)
• Symbol file (ICO)
• Bitmap file (BMP)

3.1.3.4 Model-Based GUI-generators

“Models are intended to capture, to characterise and to abstract real-world information that will lead to future UI. As every model is built on some abstraction mechanism, it usually consists in a partial view of the real world with respect to the design problems. Consequently, one or several models may be required. Information is primarily stored into models according to an appropriate format allowing a computer-based system or an automata to so manipulate them, to access them, to retrieve them, to update them, to manage them” [VP99].

3.1.3.4.1 GENIUS

The computer supported tool GENIUS (GENerator for user Interfaces Using Software ergonomic rules) is part of the program “Work and Technology” that was, in turn, part of the project “Support tools for user-friendlier presentation of a user-computer interface”. It was developed as a practical application for an approach allowing the generation of software ergonomic user interfaces directly from a data model and a dialog description based on petri nets [Fäh96].
GENIUS is a tool for automatically generating user interfaces from data models. With this approach and its practical application with GENIUS, the data model that was developed in the analysis process and is presented as an entity-relationship diagram serves as the starting point for the generation of user interfaces [Fäh96].

**Definition of Views**

The data model includes the amount of information that the user wants to see and process. However, the data model does not show which data the user will need for carrying out a specific task. This is why the so-called views are defined in the concept described here. The views contain the information necessary for the processing of a certain task. Any entities, relations, and attributes can be put together to form a view. Views specify the logical system of information but not its visual presentation on the screen or in a window [Fäh96].
The definition of views is carried out interactively using the entity-relationship diagrams. In order to activate a simple and powerful view definition, several extensions of entity-relationship diagrams were introduced [Fäh96]. The logical grouping of attributes of one and the same entity is represented by complex attributes. Any elements of the data model that form a logical entity during a particular task can be grouped together [Fäh96]. The information about the formed groups is used for organization of the elements when generating a user interface. The graphic complexity of the data model can be reduced by creating hierarchies. This way, a group of data model elements can be substituted by a single symbol and then it can also be processed separately [Fäh96].

As not all the information can be presented graphically in an application, the definition of views is completed by additional information in form of text [Fäh96]. For every element of the graphic presentation, such as entity-relationship, attribute and view, there is a scheme that specifies the element characteristics that need to be described. Three types of information are specified in this scheme [Fäh96].

**Structural information**

The structural information shows separate relations between the single elements. In a view, it specifies which elements are grouped together and what their functions are. There are processing functions and navigational functions. Processing functions entail the data presented in the view and its processing. Navigational functions define the transitions between single views and describe the dialog courses [Fäh96]. This description in form of text is an addition to the graphic presentation and the definition of dialog course using dialog nets [Fäh96].

**Descriptive information**

The descriptive information contains a clear identification, a definition, and a short description of every element [Fäh96]. Attributes may have the following characteristics in the descriptive information: data type, length, format, and value range [Fäh96].
Information about the task characteristics

Part of the information that the scheme contains is derived from the task characteristics. Examples of such characteristics are frequency, priority, duration, and access. The characteristics are used when selecting or organizing the necessary interaction objects [Fäh96].

The accumulation of the schemes and knowledge basis is carried out separately. Most of the information used in the schemes is decided on during requirements analysis and, if a CASE tool was used, it is available in the data dictionary. In this way, single data and definitions are available for the application development as well as for the design of the user interface and they are consistently managed in a data base [Fäh96].

The knowledge base for generating user interfaces

A description for a user interface is generated from the defined views and the additional information contained in the scheme using the defined views and the additional information contained in the scheme with the help of the GENIUS rule-based system. A knowledge base containing the following types of knowledge was created for this generation process [Fäh96]:

- Interactive objects,
- Selection rules,
- Layout rules parameters.

In this process, the available interactive objects are not described with the help of specific elements of a tool kit or with the help of a certain user interface management system. They are described through their characteristics. The decision about their specific appearance and behavior (for example, about a certain presentation style) is not made before their implementation [Fäh96].

The definition of different abstract interactive objects for the knowledge base depends on the desired type of user interface. The interactive objects for mask-oriented user interfaces are menus, data fields, lists, groups, and mask frames. The interactive objects for graphic surfaces are windows, dialog boxes, menus, data fields, single selection list box, multiple selection list box, and scroll bars [Fäh96].

The selection of necessary interactive objects is carried out according to specific selection rules. The characteristics of the basic dialog task, for example, activation of a function, are decisive when choosing the selection criteria [Fäh96]. The selection rules were created using the existing sets of directives as well as taking into account national and international standards, e.g. ISO 1991 and DIN 1988. For example, most of the formatting rules as well as rules for the arrangement of data areas and field definers originated from the Smith and Mosier [SM86] directives. The directives for graphical user interface such as OSF/Motiv(OSF), Open Look(Sun), CUA(IBM) gave the information for the definition of the selection and arrangement rules for interaction objects [Fäh96].

The generation process

Rules are supplemented by the use of standard values. They set parameters of the interactive objects, such as the type and size of font. They are also used in more complex
settings, such as a window or the description of a menu bar that will be used in a standard way. The standard values make it possible to customize the generation process for a certain enterprise's style or for a certain application [Fäh96].

The generation of a user interface is carried out with the help of the rules in three steps. The steps of the generation process are [Fäh96]:

- **Step 1**: The interactive objects are matched with the views, data elements, and functions. For every view a window is generated. During the selection of interactive objects for the data elements of a view, the descriptive information is received from the schemes.

- **Step 2**: The settings of the parameter values for the interactive objects are carried out. There are three ways the parameter values can be set. First, they can be received directly from the data elements, (values, such as the length of a data field). Second, the predefined standard values can be set. Third, a parameter value results from an interplay with other interactive objects, e.g. at their positioning.

- **Step 3**: The last step is the arrangement of the interactive objects on the screen or in a window using the layout rules. As a result, the rule-based system gives a description of the static user interface that will be transformed into a specification for a User Interface Management System (UIMS). This transformation step makes it possible to flexibly use different UIMS for different surroundings.

GENIUS offers several options for managing the generation process. The easiest way is the alteration of the standard value for interactive objects. Then the interactive objects themselves and the rules can be altered. It is possible to have several knowledge bases for the generation of different user interfaces, e.g. for alphanumerical and graphical user interfaces [Fäh96].

The specification of dialog courses with dialog nets

The dialog nets in GENIUS are used for the setting of the dynamic user interface that needs to be specified [Fäh96]. This method is mainly used for rough descriptions of the dialog course in the windows or in the views. More detailed dialog courses in the objects, e.g. the alteration of the selectability of the surface objects, are normally not described graphically and are not integrated in the system described here yet [Fäh96]. However, they can be specified using the means of the applied UIMS [Fäh96].

Generation of executable dialogs from dialog nets.

The dialog nets with unspecified transitions are the main component for the specification of dialog courses in GENIUS. A generator creates a dialog net for a member of selected views, for every view one part. The transition in the dialog net matches the functions in the definition of views. The effect of transitions in the dialog course can be easily defined in a graphics editor for the dialog nets in GENIUS [Fäh96].

Rules for a UIMS are generated in two steps from the dialog net specification [Fäh96]:

- **Step 1**: The unspecified transformations are turned in completely specified ones; the information from the generation process is used for the static surface. This
information tells which surface object (button or menu options) needs to be selected for a certain function.

- Step 2: The completely specified transitions are turned into rules for UIMS.

3.1.3.4.2 JANUS

JANUS is a tool that automatically creates user interfaces for software systems in business and administration on the basis of an object-oriented model with additional information. Then JANUS can generate executable applications with all the necessary components, such as user interface, application classes, and data maintenance with the help of only a few programmers and at a low cost.

The object oriented analysis (OOA) model is created with the help of Rational Rose and serves as the basis for generation. The most important part of the object-oriented analysis is the class diagram of the application that models the skeleton for the application that will be fully developed later creating classes and their structures.

It is used in:

- Pilot applications of business and administration.
- Quality assurance. Apart from the application code of the user surface the help systems, connections to the data base as well as reports are created.
- Training.

Program requirements

- Rational Rose 98/2000
- JDK 1.2.2 or higher (optional)

Programming language

- C++ or Java code is generated.
- No programming skills are required.

Product properties

- Supports Microsoft Visual C++6.0 and Sun Java JDK1.2.2 or higher.
- Specification of UML models

Plattform

- Windows 95/98 oder NT/2000

Target group

- Consultants and people working in quality assurance, because the abstract Unified Modeling Language (UML) models can be visualized easily.
- System analysts, they do not need to create the GUI manually with the help of an editor.
People who have UML knowledge but do not have any experience in the development of object-oriented user interfaces.

Using JANUS, according to [Otr97] the following can be achieved:

- The application can be used right away. Its functions and changes of data can be tested immediately.
- Based on an OOA model, the code is generated in C++ or Java completely automatically. Such a transfer of OOA models into executable solutions cuts down the costs and saves time. Experiments show that the time needed for the development of applications can be reduced by up to 70%. The manual programming that usually takes a long time is also reduced.
- High-quality of the final application code can be achieved as well as uniform programming standards, e.g. code formatting, naming etc.

The JANUS Generator helps software developers with the following tasks [Otr97]:

- The development of new applications that are flexible and do not depend on a certain platform with an option to add new functions.
- The updating of existing applications taking into account possible future developments with the help of OOA and modeling as well as the automatic generation of object-oriented program code.

The high productivity of the JANUS Generator is achieved according to [Otr97] thanks to the following factors:

- The communication with different departments becomes easier because they do not have to discuss the structure of the program logic. The JANUS approach makes it easier for the customer (who will use the software) and the software developer to understand each other. This way, the aims of a projects can be defined more precisely and they can be worked on or tried out right away. Furthermore the mistakes concerning the requirements of a model can be identified early and their correction will be a lot cheaper.

- Unlike the traditional programming approaches, JANUS creates the rough draft of the project regardless of the user interface. This makes it possible to use different user interfaces without modifying the rough draft.

- The generator creates structures with a transparent program code that can be modified when necessary. It uses standard libraries, e.g. *Microsoft Foundation Classes* (MFC).

- Apart from the generation of pure application code, the user interface, help system, client/server or stand-alone-application, the connection to the data base as well as reports are generated.

The more detailed the elements contained in the description of the class diagram are, the richer is the functionality of the generated application.

The generation process requires the exact specification of the OOA model elements. The interaction elements of the user interface are determined according to the data types of the attributes. The additional information, e.g. the restriction of the value range, participate in the generation of the plausibility tests [Otr97].
The structures between the classes (inheritance, association, and aggregation) are taken into account during the generation, so that not only the registration forms for every class of rough drafts but also the navigation trails in the user interface are created after the evaluation of the associations [Otr97].

Development process

The process of developing an idea about the project to an executable application consists of the following steps:

1. Conceptual modeling
2. Detailed specification of an OOA model
3. Programming product logic

Conceptual modeling

The product is created in the form of an OOA-model. An existing system can be used as a starting point [Otr97].

Detailed specification of an OOA model

After the product is created as an OOA-model, the generation process can be directed with the help of different settings for the classes, attributes, and structures that need to be identified. The JANUS-specifier that is integrated into the Rational Rose makes this task easier [Otr97].

The JANUS-specifier helps the user to specify the class diagram created in Rational Rose to the point when the Janus-specifier has all the information it needs in order to generate the GUI of the application [Otr97].

Programming product logic

The JANUS-specifier helps to specify many logical interrelationships between the classes and their attributes in the area of business/administrative applications, e.g. the Euro and other currencies as well as their exchange rates and restrictions concerning the attributes (date A must be a bigger number than date B). Complex logical interdependencies can be implemented in the generated C++/Java source code. If the generation is repeated, the source code integrated in these sections will not be overwritten [Otr97].

In addition to the application generation, the creation of an ergonomic user interface in C++ using MFC is also supported by the generator [Otr97].

The application will not be perfect after the first try. However, it is possible to modify the original OOA model and create a new version of the application solution, taking into account the opinions of representatives of different programming departments and the potential users of the application. Protected areas in the generated source code allow developers to implement specific functions. If the code-generation is repeated, this specific program own code will be stored in the data base and then integrated again in the new source code [Otr97].
The connection to a data-base

The connection to data-bases depends on the type of the selected platform [Otr97].

- **OB**
  For the Prototype generation the data base *Object Base* (OB) which was developed by Otris was used until recently. It enables the fast creation of prototypes and their easy installation. It is not necessary to acquire a separate data base for the prototyping sector. OB is integrated in JANUS surrounding so that it is not necessary to purchase a separate data-base. However, the next version 2.6 of JANUS will no longer support this data base, because this task will be successfully carried out by MS Access.

- **MS Access**
  Using MS Access is meant for small and medium data volumes and for 1-25 users. The application is automatically connected to the data-base after the start. It is required that the system has an installed *Open Data Base Connectivity* (ODBC)-driver.

- **DB/2**
  This platform-independent data-base, it is developed by IBM, is also supported by JANUS. This support is provided especially for the DB/400 on the AS/400. In this case, the creation of the data base and the scheme transfer are also automatic.

The JANUS product family

The JANUS product family makes it possible to generate OO-applications automatically. The application developer will be supported with the creation of [Otr97]:

- **JANUS/Access**, **JANUS/Enterprise++**, **JANUS/Enterprise4000** - complete application up to platform-independent Client/Server applications.
- **JANUS/Prototype** - executable prototypes with a graphical user interface and data maintenance.
- **JANUS/GUIJ** - Java-interfaces based on the Swing Library (Sun).
- **JANUS/Web** – Servlets that make their generated applications ready for use through an Internet browser.
- **JANUS/COM** – COM-applications: it covers the complete functionality of the generated application in Visual Basic or VBA.

All the products from the JANUS product family enable the software developers to generate code in C++ or Java, based on a single OOA-model, e.g. one that is created with Rational Rose. If the OOA model changes, a new application is created and then it is possible to see which changes were caused by the change of the OOA model [Otr97].

3.1.3.4.3 **BOSS**

BOSS is a model based tool for setting all of the points of interactive applications, such as application interface, user interactions task space, and presentation design rules. The name BOSS is an abbreviation of the German “Bedienoberfächenspezifikationsystem”
which translates to “user interface specification system”. The following description of BOSS is entirely based on [Sch95a].

BOSS supports the development of user interfaces on very early stages of activities as well as the creation of standard user interfaces regardless of the task-level description. These user interfaces can be evaluated during the early stages of the development process.

BOSS has an Integrated Development Environment (IDE), which allows specification in a way similar to visual programming. The BOSS IDE supports a specification, generation, and evaluation cycle.

BOSS was not developed for specific application domains, such as business oriented systems. In general, it can manage interfaces with animated interactive graphics and other specific application objects.

Unlike other model-based systems, BOSS uses the encompassing specification model Hierarchic Interaction graph Templates (HIT) to cover all the points of the interactive application model. HIT supports the roles of the interactive application model as well as the roles of different persons taking part in the user interface development:

- The application analysts describe the structure of the task space given to the users by the interactive applications.
- Human factors experts work on the presentation design rules taking into account the ergonomic rules.

HIT is a compromise between a designer-oriented and an implementation-oriented specification model:

- Designers can express the ideas in a declarative manner.
- Application analysts describe the user interaction task space in a way that is closer to the psychologically motivated modeling approaches.
BOSS accumulates all the information necessary for the support of the user interface development process. The application analysts and the human factors experts take part in the creation of the interface model structure. In this process, the dialog designer is responsible for the specification refinement.

The model of the interactive application (HIT representation) consists of three main parts [Sch95a]:

- The description of the Application Interface (AI): the specification of the data structure and the necessary functions of the application itself. This is what an application analyst is in charge of.

- The User Interaction Task Space (UITs): A high-level description of the interaction tasks the user can perform within the application, not including the presentation aspects. This is also one of the application analysts tasks.
• The *Presentation Design* (PD): It describes the logical structure of the interactive application to an appropriate presentation (according to the software ergonomic rules).

3.1.3.4.4 EmuGen

This generator creates the *Multiple-User Interface* (MUI) applications. EmuGen is an acronym for "the Expandable models of Multiple-User interfaces GENerator". EmuGen generates executable MUI prototypes by copying even partly described MUI models. This gives the designers and the actual users the opportunity to discuss the details of a test of the actual MUI prototype. Based on the results of the discussion, the designer can improve the MUI models. The improved MUI models serve as the input of the next generation step, and another test and discussion may follow [Braa].

The input of EmuGen consists of a data model, a task model, and a user model. Using these models, EmuGen automatically generates an executable MUI application, which is based on Java-Swing and Java2D. Apart from this, EmuGen provides a Java-API for implementing of the additional aspects of MUI projects [Braa].

![Figure 50 Generation process in EmuGen](image)

**The Data Model**

The data model of EmuGen is an extended form of *Backus Naur Form* (BNF) for the specification of data types.
Its main component is the data type AHW which consists of the name, phases, task, and users of the workflow. The extensions of BNF used by EmuGen include references such as PrePhase. The example (Listing 7) shows the following language elements:

- “::=” means “defined as ...”.
- “[|]” means that only one alternative can be selected.
- The lists are defined with E* (E is the definer of an element type) [Brac].
- The references “->” always refer to a tuple type T (referred type, in the example Phase) with a selector T (reference selector, in the example Name) [Brac].

One Java class is generated for each data type of the data model. At runtime, the hierarchically organized functional code of a MUI application is an instance of the first data type defined in the data model [Brab].

The Task Model

The task model consists of a set of action and workflow definitions. In this model, the designer optionally specifies a set of actions for each type of the data model.

The definitions of actions consist of an identifier, a precondition, a list of parameters (empty in both examples), a body (Java code), and a list of post conditions [Brab].
The User Model

With this model, the designer specifies the subset of data types which represent the users of the MUI application. In addition to this, it is possible to restrict the access to the instances of the various data types [Brab].

```
users: WorkflowUser (Name)
access on TaskInOperation {
  (_, WorkflowUser):
  read Name, write Dokument, exec ok;
}
```

Listing 9 User model example (adopted from [Brab])

At runtime, each instance of `Workflow User` represents an `Actor` identified by `Name`. In addition to this, there is always one implicit user who started the MUI application. If no access rule for one data type is defined, all users have free access to the instances of this data type [Brab].

A set of phases and a set of transitions connecting the phases define the workflow. An EmuGen workflow transition consists of an action and its post condition. The EmuGen workflow transition is activated when the user executes the action and the post condition of this action becomes valid after the execution of the action body [Brab].

The components of the EmuGen user model are a set of user types and a set of role dependent access rules for the data and the action objects [Brab].

Example

“The users of a practical course are students and a supervisor. At the beginning of the course students with pre-degree may become participants by registering for the course. The supervisor can stop the registration phase and start the regular execution of the course. During the regular execution the supervisor may assign exercises to the participants who perform the actual exercise and submit their solutions. The supervisor can rate solutions of the students” [Braa].

The administrator is a person starting a generated MUI application. This person is regarded as an implicit user. Other users have to be initialized by the administrator before they can work with the MUI application [Braa].
Next, the following four steps of a possible execution of the generated MUI application for this example will be shown [Braa]:

- **Step 1** (upper left side of Figure 13): The administrator and the supervisor view the list of registered participants. Then another student decides to register for the practical course and clicks the button “Register”.

- **Step 2**: The administrator and the supervisor immediately see the changes in the list of participants. The registered student can logout and close the session, because the registration is already completed. Now, the supervisor decides to start the regular execution phase.

- **Step 3**: The supervisor has already given an exercise. The change of the actual phase causes a change in the user interface of the participants. The new student started the submission of the exercise.

- **Step 4**: The parameter of the action “Submit” is already given by the student.
3.2 Generative Programming

Today, computer developers face a very difficult task of developing software that processes very complex information, is very efficient and, at the same time, deliver top-quality results. Unfortunately, all of these software-developing requirements cannot be completely taken care of using currently available methods [GPW02].

Generative Programming is a young paradigm in software engineering. It contains a number of new concepts, which primarily provide a higher level of automation in software development as well as software reusability. The purpose of software development automation is not only to speed up the development process and reduce development costs, but also to improve software quality and error resistance. Besides, it helps reduce the cost of maintenance and similar necessities [Sch01a].

Generative Programming is a software engineering paradigm based on modeling software system families. When given a particular requirement specification, it can use configuration knowledge to automatically manufacture highly customized and optimized intermediate and end products from elementary reusable implementation components [CE00].

Generative Programming is considered to be a new software development paradigm. It does not compete with the existing paradigms, but supplements them. For example, the object-oriented programming has a high level of abstraction but barely any automation [Sch01a]. GP supports reusability and adjustability much better than object-oriented programming, frameworks and design patterns [EC99b].

3.2.1 Generative Domain Model

Generative Programming represents an approach permitting the creation of whole product families. The Generative Domain Model serves as the basis.

"[A generative domain model is a] model of a system family that allows the automatic generation of family members from abstract specifications. It consists of a means of specifying family members, the implementation components from which each member can be assembled, and the configuration knowledge" [CE00].

Figure 52 Generative domain model
It consists of three elements:

1. The left oval represents the methods used for the family member specification. It is made for users and computer experts. They use a specific language that has specific features and terms. This language is implemented as a domain-specific language (DSL). The purpose of a DSL is to give the user the opportunity to describe a particular system in a most suitable way. This helps to “order” a particular system. In order to do this, a text, a form dialog, or a graphical-interactive environment can be used [ES02].

2. The arrow represents the configuration generator. The configuration generator automates the product assembly. It accepts a DSL specification and analyses it. Then, if necessary, it carries out a buildability check and assembles a software product from the implementation components [EHLS02].

3. The right oval describes the world of the software developer. It contains developed elementary components the system can be assembled from. They must have maximum combinability with minimum redundancy. The use of a feature diagram which graphically presents the elementary components in the form of a tree-like structure is helpful.

It is advisable to separate the problem area from the solution area because then they can be altered independently from each other. New implementation components can be added or the ones already available can be improved without changing the code of the available applications. This is possible because the problem specification is carried out within the problem area and it does not directly involve the implementation components [Bli01].

With the configuration of elementary and reusable components it is possible to automatically generate a customized system that is also highly optimized.

### 3.2.2 Software System Families

Nowadays, most of industrial production is automated, unlike software development that has been falling behind for decades. The methods of software technology normally deal with the development of separate systems. The software industry produces either customized software for a specific problem field or software for different tasks. However, in this case the software cannot find optimal solutions for all the problems [GPW02]. In this context, the development of software families is promising.

Apart from the automation of software development, one of the main concerns of Generative Programming is the creation of system families. The GP principles assume that the members of a system family can be generated based on the common model of this system family, the generative domain model [Sch01a]. A system family does not need to be limited to code creation. It can include user interfaces, help information, system checks etc. as well [ES02].

Generative Programming includes methods and technologies for modeling of software system families. These methods and technologies are adjusted to each other, so that a highly customized and optimized software product can be automatically created according to the requirement specification [EHLS02].

The modeling of software system families allows the production of a large number of system variants based on specific requirements. Most of this process can be automated,
which significantly reduces the development time and cost as well as improves the software quality [EHLS02].

Instead of using a single product model, a generative domain model allows the creation of software system families on a common basis. This means that it is possible to create a whole range of products for a particular application field [GPW02].

It is important to distinguish between the terms “product family” and “product line”. A product family is a number of products that can be developed on a common basis. A product line consists of a number of products the characteristics of which are adjusted to the needs of certain users or a certain market [EC99b].

3.2.3 The Domain Engineering

Generative Programming uses methods of domain engineering. Domain engineering is a systematic approach that identifies important elements and requirements in a domain. This information is essential for the creation of an efficient program generator meeting the customer’s needs [Cle01]. In domain engineering, existing and possible systems are analyzed and modeled. Besides, the implementation components and the necessary reusability infrastructure are developed [EC99a] in order to provide a reusable solution for the whole system family.

“Domain engineers consider how single applications may change over time, but they also look at a whole range of applications in the domain to determine differences among those applications. We call these differences variabilities. Variabilities are important because they are the main ingredient in building program generators” [Cle01].

The domain engineering process generally consists of three phases:

- **Domain analysis** is a process of problem understanding. It does not offer a solution. It is not only a technical but also a social process [Cle01]. The most important activity of domain analysis is domain scoping. During domain scoping it is determined what lies within the domain and what not. On the one hand, it is very important to find suitable information sources (and financial sources). On the other hand, only those who actively participate in the project can become stakeholders [EC99a]. During this phase relevant information about the domain is gathered; terminology, boundaries, commonalities as well as variabilities of the domain are determined [Cle01]. Feature models serve as documentation help. Feature models consist of feature diagrams showing common and variable feature in a concise and comprising way and additional information such as dependencies, exemplar systems, etc. which can be organized in a textual or tabular format. This presentation is also implementation-neutral.

- **Domain Design**. This process serves to develop a common system architecture. During this phase the implementation components are determined, a dependency analysis is carried out, and a common architecture for all members of a system family is established. Dependency analysis identifies the dependencies between the implementation components. It is then used as the basis when determining a common architecture of all the systems of a family.

- **Domain implementation** involves the implementation of the architecture, the components as well as of the production plan. During this phase, the tools neces-
sary for the efficient creation of a customized application in the domain are created. This process may include developing program generators [Cle01].

It is obvious that during the development process some errors are made that become detected later. In order to achieve an incremental improvement of the domain model, the architecture and the implementation, the three phases of the development process can be run several times.

![Figure 53 Development process (based on Neu98)](image)

### 3.2.4 Feature Modeling

Feature modeling is central activity of domain engineering. It was introduced by the Feature-Oriented Domain Analysis (FODA). Feature modeling is the process of analyzing and modeling of common and variable features of concepts and their interdependencies, as well as describing their arrangement in a coherent model, the so-called feature model.

#### 3.2.4.1 Feature Models

With feature models, common and variable features within a system family can be modeled. The main component of the feature models are features. *Features are attributes of a system that directly affect end-users* [CHK90]. Apart from the name, a large amount of additional information can belong to a feature. This additional information comes mostly in the form of tables, lists, or free text. It can also be documented in diagrams or with the help of a suitable tool, for example, the feature model editor AmiEddi. Feature diagrams with this additional information make up a feature model.

The interdependencies between the variable features will be shown in the feature model. Originally, there were three feature types available in FODA: *mandatory*, *optional*, and *alternative*. Later, *or* features were added by Czarnecki [Cza98].

#### 3.2.4.2 Feature Diagrams

A feature diagram shows features and their relations to each other. This type of diagram has an advantage over UML-diagrams, as it does not require specific decisions about a specific implementation for representing variability [EC99a]. This diagram looks like a tree the branches of which are features. Features are represented by their names which appear in rectangles. The root of the tree represents concept node. This node is always part of a feature diagram. All the other nodes of the tree represent features. Each one of
these features can be described more fully with the help of sub features. This is how the hierarchy shows up on the diagram. The features are connected to each other with lines. The circle at the end of each line characterizes the type of the feature. A filled circle denotes a mandatory feature and an empty circle represents an optional feature. Two or more features connected with a filled arc form an inclusive or-group. Features connected with an empty arc are an exclusive or-group.

![Feature Diagram Example](image)

**Figure 54 Example of a feature diagram**

An example of a feature diagram is shown in Figure 54. It describes a car. Its root represents the car concept. The other nodes of the features are:

- **“Mandatory features: Every car has a body, transmission, and engine”**.
- **“Optional feature: A car may pull a trailer or not”**.
- **“Alternative features: A car may have either an automatic or manual transmission”**.
- **“Or-features: A car may have an electric engine, a gasoline engine, or both”** [CBUE02].

“Feature diagrams allow better modularity in large configuration spaces than flat parameter tables. Thanks to the hierarchical organization of feature diagrams, parameter sets that are pertinent to just one class of variants can be hidden in one branch of a feature diagram and need not be considered when configuring a variant outside this class. In this context, it is important to note that a feature diagram is not just a part of hierarchy of the software modules, but a hierarchical decomposition of the configuration space” [CBUE02].

### 3.2.4.3 Additional Information of Feature Models

Every feature model can also contain additional information [CE00]. In general, this information can be of great variety and of different importance in specific domains. Some examples are:

- **Source**: Sources of information for a feature could include books, standards, existing applications, or domain experts.
- **Stakeholder**: List of parties interested in the feature.
• **Synonyms:** Synonyms are several alternative names for the feature. If desired, it is possible to put down the origin of the names next to the names themselves.

• **Exemplar Systems:** The exemplar systems annotate the features with references to existing systems in which they occur.

• **Openness:** Variation points are open if new features are expected.

• **Semantic description:** A semantic description is a short description of the feature semantics.

• **Constraints:** It is possible that some features depend on other features. This way, the selection of one feature has to be followed by the selection of other features in order to preserve the integrity of a system. This dependency information is also needed in later phases, for example for an completing a configuration.

• **Priorities:** Different priorities may be assigned to the optional features and feature groups. This way it is possible to set the order of implementation or the relevance of the feature for the Stakeholders.

• **Premature:** It shows that the feature is under construction.

• **Binding Mode:** It describes how a feature is bound; there are three types of binding modes:
  - **Static Binding:** The static binding is carried out at compile time and cannot be undone.
  - **Changeable Binding:** The feature is bound early; however, the binding can be undone, if needed.
  - **Dynamic Binding:** The feature is bound when needed; the binding can be removed afterwards.

• **Binding Site:** It describes when, where, and who binds an feature.

### 3.2.5 The Generator Principle

The manual configuration and composition of the implementation components is complicated and may introduce errors in the software. It is very hard to carry out complex optimizations manually. A generator automates these processes and solves the problems previously mentioned.

The main purpose of a generator is to translate a specification of a higher abstraction level (mostly source code for a compiler) into a lower level format. Many generators have specific tasks, e.g. GUI, DBMS. They are often very efficient in their specific fields [Sch01a].

“Program generators are tools for creating application programs from specification” [Cle01].
A generator consists of the following components:
- A parser for the requirement specification (DSL)
- An optional buildability check
- A configuration optimization
- A component assembly

Figure 55 Generator stages (based on [EC99b])

A generator takes over a requirement specification. Then it analyses the requirement specification, carries out a buildability check (if necessary) and assembles a system or a system component from the implementation components. In this process, invalid specifications will be rejected.

3.2.6 Technology Projection

Generative programming is a paradigm and not a technology. This is why many technologies can be used to apply generative programming. These technologies need to be used in a particular way. This means that GP can be projected on different technologies. The technical projection describes how the generative domain model is adjusted to a particular platform or a particular programming language [EC99b].

The following technology projections for generative programming are available [Emr02]:
- Aspect-oriented programming with Java and AspectJ [Mac01]
- Frame technology (ANGIE) [ES02]
- Template meta-programming in C++ [CE00]
- Java and JavaBeans [EHLS02]

Figure 56 Technology projection of ANGIE
The projection of the generative domain model to the frame processor ANGIE is carried out in the following way:

- The DSL can be situated within ANGIE or outside of it. The DSL situated within ANGIE can only be text-based, that is, created with the help of the script functions, frames or slots. The DSL situated outside of ANGIE can be either text-based, dialog-based or graphical-interactive.
- The generator is projected on ANGIE frames and ANGIE functions. In this case, there can be fewer frames than script functions.
- The Solution Space is also projected on the frames and ANGIE script functions. In this case, there can be fewer functions than frames.

3.3 Frame Technology

Frame technology deals with the concept of frames and slots. In 1974, Marvin Minsky’s article “A Framework for Knowledge Presentation” [Min74] was published in the book “The Psychology of Computer Vision” (ed. Patrick Winston) [ES02]. The frame/slot approach originated in Artificial Intelligence (AI) and was then introduced to the area of picture identification. Later, it turned out that it is also possible to use this approach in the analysis and synthesis of languages [Sch01a]. A frame defines set values; the so-called slots. The slots of a frame can be filled with frame instances. In this way, complex hierarchies can be created. The purpose of the frames is to classify the scene descriptions or texts based on their patterns [ES02]. Suppose we have different frames for the scenes like soccer, table tennis, swimming etc. Now, one of these scenes is assigned to the best-matching frame, based on its pattern. In this example, it can be done based on the space where the event takes place: a soccer game takes place at a stadium; a game of table tennis can take place in a gym; a swimming race takes place in a swimming pool, a big pool filled with water [ES02].

Frame technology most not be viewed as a paradigm of its own, however, it can be used while working with the generative paradigm [ES02]. This technology proved effective and showed decent results in industrial use [Bas97].

3.3.1 ANGIE

About ten years ago, the Delta Software Technology GmbH developed the ROOT tool for the program analysis and transformation. It was, for example, used at the reset in 2000 and the introduction of the Euro. ANGIE was created as further development of ROOT, and is used for generating purposes. The name ANGIE stands for “A New Generator Engine” [ES02]. ANGIE is a frame processor. The purpose of a frame processor is to process the text components and to create source texts out of them. The ANGIE script language is a full-fledged programming language that is especially created for the development, extension and configuration of generators. Its syntax is similar to that of Visual Basic; frames added to it serve as object-oriented constructions. All lines containing the statements for the frame processor start with a dot. It is very important that ANGIE does not depend on the target languages. This means that ANGIE frames can contain not only C++, Java and code on other programming languages but also any non-code artifacts in non-binary formats such as the help and resource files, documentation etc. As a weakness of ANGIE can be consider that it is currently only available for the Windows platform.
The important elements of this language are [Sch01a]:
- Modularization with declaration and code modules;
- Script functions;
- Container for variables;
- Frame declarations;
- File declarations: global and local, static and dynamic;
- Arrays;
- Collections (with indexes);
- The necessary statements, assignments and function callings;
- Intrinsic functions, e.g. for the strings, arrays, collections etc.

“Intrinsic” means that these functions are an inseparable part of the frame processor.

3.3.1.1 Frames

Frames are patterns of code lines in the target languages that can be used multiple times. The characteristics of the frames are similar to those of the classes and objects in object-oriented languages. Listing 10 shows an example frame in ANGIE.

```plaintext
.Frame MyFrame (myParameter) , Column Off
   .Dim MyVariable = "World"
Hello <!MyVariable!> !
.End Frame
```

Listing 10 Frame

An ANGIE-frame consists of:
- A data type that is determined by the frame name. In the example above, it is MyFrame.
- Control over the formatting process: frame attributes. In the example above, it is ColumnOff.
- Frame variables. In the example above, it is MyVariable.
- Code block in the target language with an embedded slot reference. In the example, it is Hello <!MyVariable!> !

3.3.1.2 The creation of a frame instance

Any number of instances can be created. Later in the generation process, they can be modified and stored in a repository.
```
.frMyFrame = createFrame("MyFrame", "parameter")
```

The first parameter is the frame name, followed by as many parameters as declared in the frame constructor.

3.3.1.3 Slots

Slots in frames are always named. They are assigned variables as well as other subordinate frames. This way, the tree-like structure appears. It is similar to a syntax tree and can have any depth or complexity. It can be created, evaluated and processed with the generator scripts during the embedding process.
To embed a slot under a defined special character sequences are applied:
<!mySlot!>
<!!!>  - These flags can be defined by the user.

3.3.1.4 Optional Code Blocks:

An optional code block is only distinctive when all its slot variables have values other than EMPTY.
<? CodeBlock <!mySlot!><?
<??>  - These flags can be defined by the user.

If the slot <!mySlot!> has the value EMPTY, the whole text (the code block between <??>, in our example) is faded out.

3.3.1.5 The Definition Of Variables

.Dim myVariable As Type

The definition of a variable is carried out using the key word Dim followed by the name of the variable. As an option, its type can also be given: variant, integer, string, a reference to a frame instance, a reference to a collection instance or EMPTY. If no type is given, the variable automatically belongs to the variant type. Exactly like in Visual Basic, this type represents all the types available in this language.

3.3.1.6 Collections

A collection is an organized row of elements. The collection elements are variables of all types. The collection elements can be referenced like variables. “Read-only” as well as “Write” access is possible. It is also possible to access the collection elements using their running number or an alphanumeric index. Let’s have a closer look at these access types [DST02]:

- Access through a running number: every collection element is automatically assigned a running number. The collection elements can be addressed by these numbers, similar to array elements. This address is unequivocal, however, not irreversible. It can be changed when an element is removed from the collection and the indexes are changed.
- Access through an alphanumeric index: the intrinsic Add function offers the option to give an alphanumeric index to the collection elements. The user can access the elements using these indexes. Unlike in the case of the addressing by a running number, the indexes cannot be changed.

The collection in ANGIE has the following functions:
- CreateCollection creates a new collection.
- Count: returns the number of the collection elements.
- Add adds any number of elements to a particular collection instance.
- Remove removes an element from the collection.
- RemoveAll removes all the elements from the collection.
3.3.1.7 Script Functions
The main purpose of script functions is frame creation and manipulation. The external functions are implemented by the user.
This is what a function looks like:
.Function myFunction
... 
.End Function

3.3.1.8 Control Structures
Furthermore, ANGIE offers such control structures as loops and branching.

Conditional
For this purpose, there is an If-statement in ANGIE (Listing 11). It is important to notice that the condition must be followed by the key word Then in the same line.

```
.If  a = b Then
  .print("a=b")
.Else
  .print("a<>b")
.End If
```

Listing 11 If-Statement

Iterative
In ANGIE, there are three statements for the creation of loops: While-Wend, For-Next (Listing 12), ForEach-Next. The ExitFor-statement serves to quit a For-loop.

```
.For i = 0 To 2
   int n<!i!> = 0;
.Next
```

Listing 12 For-Loop

If this code fragment were carried out, the result would be:

int n0 = 0; 
int n1 = 0; 
int n2 = 0;

3.3.1.9 The Export of a Frame Instance in a File
The readily distinctive frames can be exported as source code (generation). The frame contents are at the same time code fragments. Only after the final export process the abstract repository contents turn into a specific source code.

```
.export(frMyFrame, „FileName”, „FileType”)```
This function is an inseparable part of the frame processor (an intrinsic function). Its purpose is to save the contents of a frame instance to a file. The second parameter, “FileName”, determines the file name. The third parameter, “FileType” defined in a XML configuration file, determines the expansion and the type of the file.

3.3.1.10 Example

This example shows the use of some of the constructions mentioned above.

```
.FRAME Hello(strName, nAge), Column Off
  .DIM strSalutation = (nAge > 16) ? "Hello " : "Hi 
  .DIM strAddition
  <!strSalutation!> <!strName!> !
  <? I would also like to say <!strAddition!>?>
.END FRAME
```

Listing 13 Frame example

The constructor of the “Hello” frame takes over two parameters: strName and nAge. The value of the frame variable “strSalutations” depends on nAge. If nAge is bigger than 16, then strSalutations receives “Hello”, otherwise “Hi”. As default, the frame variable strAddition has the EMPTY value. Then, the embedding of the slots is carried out.

```
.'The creation and export of the slots.
.DIM test1 = createFrame(“Hello”, “Mike”, 14)
.export(test1, “test”, “TXT”)
```

Listing 14 Frame creation example 1

After the code shown in the listing is carried out, the contents of the file test.txt look like this: Hi Mike!

```
.DIM test2 = createFrame(“Hello”, “Aloena”, 25)
.test.strAddition = “I love you!!!”
.export(test1, “test”, “TXT”)
```

Listing 15 Frame creation example 2

After the code shown in the listing is carried out, the contents of the file test.txt look like this:
Hello Aloena!
I would also like to say I love you!!!

3.3.1.11 Projects Using ANGIE

ANGIE served as an internal tool as well as a back end of other Delta products, such as SCORE Integration Suite and Pattern by Example (PBE) that support a pattern-based generative development process [DST02].
Part 3
Solution
4 Realisation

This chapter is about the design creation and problems that may appear in the creation process.

4.1 Prototype Description

As the domains examined in this example served the Corel Photo Paint 7.0, Imagine Version 1.6, ISO 9241-10-1996 and Windows guidelines. The result of this examination was a prototype described below.

![Prototype Appearance](image)

Figure 57 The prototype appearance

The prototype contains a title, menu and a tool bar as well as a status bar. Apart from this, it is possible to obtain information about most of the prototype elements. In order to do this, the user needs to click on “What’s this button” in the tool bar (on the very right) and then on the element he wants to find out about.

4.1.1 The Popup Menu “File”

![Popup Menu “File”](image)

Figure 58 The popup menu “File”
The popup menu “File” consists of a list of subcommands, such as “New”, “Open”, “Save”, “Save As…”, “Print”, “Close” and “Quit”.
- “New” creates a new MDI window in the workspace.
- “Open” opens a dialog window in order to let the user select a file that needs to be opened.
- “Save” saves an image. If the image has not been saved under any name, “Save As...” dialog opens.
- “Save As...” opens a dialog that helps the user determine the name and the destination of the file that is being saved.
- “Print” opens the dialog where the user can select the printing options and then print out a file.
- “Close” closes an activated MDI window;
- “Quit” helps the user to quit the application.

4.1.2 The Popup Menu “Edit”

The popup menu “Edit” contains the commands to undo the actions applied to the image (“Undo”) or to repeat them (“Redo”). Further commands include “Cut” (cuts from the Image and saves a copy to the clipboard), “Copy” (puts a copy on the clipboard), “Paste” (pastes from the clipboard) and “Delete” (deletes the image).

4.1.3 The Popup Menu “View”

The popup menu “View” contains the following commands: “Zoom”, “Toolbar” and “StatusBar”.
- “Zoom” opens a dialog window from which the user can determine the size of the image on the screen.
- “Toolbar” shows or hides the toolbar.
- “StatusBar” shows or hides the status bar.
4.1.4 The Popup Menu “Effects”

The popup menu “Effects” contains a list of image processing functions, such as “Brightness/Contrast”, “Rotate”, “Invert”, and “Binarize” that can be applied to the image. The separate parameters of these functions can be determined in the corresponding dialog windows, except for the “Invert” command that does not have such a window.

4.1.5 The Popup Menu “Options”

With the help of the popup menu “Options”, the user can determine the look and feel of the application as well as open the preference dialog where the user can select from a range of options.

- The group box “Effects” contains two Radio Buttons that determine whether a function from the popup menu “Effects” needs to be applied to all the images or only to one particular image of the workspace.
- The group box “Zooming” option determines image enlargement. When the check box “Zoom images to window size” is activated, the size of the image is
automatically adjusted to that of the MDI window, even after it is resized. The combo box *Zoom when open* determines whether the images that are being opened need to appear enlarged or smaller in size. The group box “Program Start” contains only one combo box where the user can determine what happens after the application is started. For example, after the application is started, an Open Dialog can be opened.

### 4.1.6 The Popup Menu “Windows”

![Figure 63 The popup menu “Windows”](image)

The popup menu “Windows” contains a list of MDI windows for the workspaces. Besides, there are two options concerning the way of arranging these windows in the workspace: “Cascade” and “Title”.

### 4.1.7 The Popup Menu “Help”

![Figure 64 The popup menu “Help”](image)

The popup menu “Help” contains the information about the program. The menu option “Content” is particularly important. It contains a description of the program functionality.
4.2 The prototype feature model

The prototype described above was represented by a feature diagram. Afterwards, a decision about the variability points was made. In this process, the ISO 9241-10:1996 and the Windows guidelines were consulted. For example, it was not sensible to put the menu “File” down as a variable because in this case a generated application could have no “File” entry on the menu bar. Such an application would not be practical, as no images could be opened or created.
Figure 65 Feature diagram Part 1
4.2.1 The Equivalence Categories of the Implementation Components

The necessary condition for the dependency analysis is the arrangement of the implementation components in the equivalence categories. The feature diagram gives important information about the necessary implementation components. The equivalence categories are [EV02]:

- Specific alternative features or sub features
- A single specific feature or a sub feature
- Specific Or-features of sub features
If you look closer at figure 67, you will notice that it does not coincide with the definition concerning the feature diagram mentioned above. This is because some alterations of the feature diagram took place when the equivalence categories were being determined. The alterations were carried out in the parts where the optional features "Icon" and "What's this info" appear in order to provide an easier presentation of the dependencies.
4.2.2 The Dependencies Between the Implementation Components

As the next step, the analysis of the dependencies between the implementation components is carried out. The arrows of the dependency diagram point away from the dependent category. Their significance is determined by the same reasons that led to the presentation of a particular dependency [CE99].

After the dependency analysis, the dependencies can be arranged. For this purpose, a strictly linear hierarchic order is used.

There is a rule [CE99] according to which the user has to select one of the equal arrangements, to his liking.
4.2.3 The GenVoca Architecture

Based on the arrangement that was just carried out, the architecture of all the systems of the family is determined with the help of the GenVoca Architecture. The lowest layer is called Config. This so-called Configuration Repository contains information that can be requested by all the components of the layer above it [CE99]. The components bordered by dots are optional layers that can be skipped. According to the principle of the GenVoca Architecture, a deeper layer parameterizes a higher one. Though this step is unnecessary in the case of the technology projection for generative programming with ANGIE, it is shown anyway.
<table>
<thead>
<tr>
<th>Rotate</th>
<th>Rotate</th>
<th>RotateWithDirection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brightness/Contrast</td>
<td>Brightness</td>
<td>BrightnessAndContrast</td>
</tr>
<tr>
<td>Effects</td>
<td>Invert, Binarize</td>
<td></td>
</tr>
<tr>
<td>WhatIsThisInformation</td>
<td>WhatIsThisInformation</td>
<td></td>
</tr>
<tr>
<td>Toolbar</td>
<td>Toolbar</td>
<td></td>
</tr>
<tr>
<td>StatusBar</td>
<td>StatusBar</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td>Effects, ZoomingCheckbox, ProgramStart</td>
<td></td>
</tr>
<tr>
<td>DefaultLookFeel</td>
<td>DefaultWindows, DefaultPlatinum, DefaultMotiv, DefaultSGI, DefaultMotivPlus, DefaultCDE</td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>Platinum</td>
<td>Platinum</td>
<td></td>
</tr>
<tr>
<td>Motiv</td>
<td>Motiv</td>
<td></td>
</tr>
<tr>
<td>SGI</td>
<td>SGI</td>
<td></td>
</tr>
<tr>
<td>MotivPlus</td>
<td>MotivPlus</td>
<td></td>
</tr>
<tr>
<td>CDE</td>
<td>CDE</td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>Close</td>
<td></td>
</tr>
<tr>
<td>Print</td>
<td>Print</td>
<td>PrintWithWhatIsThisInfo</td>
</tr>
<tr>
<td>GUI-Prototype</td>
<td>GUI-Prototype</td>
<td></td>
</tr>
<tr>
<td>Workspace</td>
<td>Workspace</td>
<td></td>
</tr>
<tr>
<td>FileNew</td>
<td>New</td>
<td>NewWithWhatIsThisInfo</td>
</tr>
<tr>
<td>FileOpen</td>
<td>Open</td>
<td>OpenWithWhatIsThisInfo</td>
</tr>
<tr>
<td>FileSave</td>
<td>Save</td>
<td>SaveWithWhatIsThisInfo</td>
</tr>
<tr>
<td>EditUndo</td>
<td>Undo</td>
<td>UndoWithWhatIsThisInfo</td>
</tr>
<tr>
<td>EditRedo</td>
<td>Redo</td>
<td>RedoWithWhatIsThisInfo</td>
</tr>
<tr>
<td>EditCopy</td>
<td>Copy</td>
<td>CopyWithWhatIsThisInfo</td>
</tr>
<tr>
<td>EditCut</td>
<td>Cut</td>
<td>CutWithWhatIsThisInfo</td>
</tr>
<tr>
<td>EditPaste</td>
<td>Paste</td>
<td>PasteWithWhatIsThisInfo</td>
</tr>
<tr>
<td>Windows</td>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>Help</td>
<td>Help</td>
<td></td>
</tr>
<tr>
<td>Contents</td>
<td>Contents</td>
<td>ContentsWithWhatIsThisInfo</td>
</tr>
<tr>
<td>Help</td>
<td>Help</td>
<td></td>
</tr>
<tr>
<td>DebugHelp</td>
<td>DebugHelp</td>
<td></td>
</tr>
</tbody>
</table>

Figure 71 The GenVoca architecture
4.2.4 The GenVoca Grammar

The definition of the GenVoca grammar is the last step of the design phase. The GenVoca grammar has a notation similar to the EBNF and can be easily projected on C++ Templates [CE99]. The parameterisation of the a layer by its preceding layer is represented by a parameter in the square brackets “[ ]”.
<table>
<thead>
<tr>
<th>Rotate</th>
<th>Rotate[Brightness/Contrast]</th>
<th>RotateWithDirection[Brightness/Contrast]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brightness/Contrast</td>
<td>Brightness[Effects]</td>
<td>Brightness[Contrast[Effects]]</td>
</tr>
<tr>
<td>Effects</td>
<td>Invert[Icons], Binarize[Icons]</td>
<td></td>
</tr>
<tr>
<td>Icons</td>
<td>Icons[WhatIsThisInformation]</td>
<td></td>
</tr>
<tr>
<td>WhatIsThisInformation</td>
<td>WhatIsThisInformation[ToolBar]</td>
<td></td>
</tr>
<tr>
<td>ToolBar</td>
<td>ToolBar[StatusBar]</td>
<td></td>
</tr>
<tr>
<td>StatusBar</td>
<td>StatusBar[Options]</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td>Effects[DefaultLookFeel], ProgramStart[DefaultLookFeel], ZomingCheckbox[DefaultLookFeel],</td>
<td></td>
</tr>
<tr>
<td>Windows_</td>
<td>Windows_[Platinum]</td>
<td></td>
</tr>
<tr>
<td>Platinum</td>
<td>Platinum[Motiv]</td>
<td></td>
</tr>
<tr>
<td>Motiv</td>
<td>Motiv[SGI]</td>
<td></td>
</tr>
<tr>
<td>SGI</td>
<td>SGI[MotivPlus]</td>
<td></td>
</tr>
<tr>
<td>MotivPlus</td>
<td>MotivPlus[CDE]</td>
<td></td>
</tr>
<tr>
<td>CDE</td>
<td>CDE[Close]</td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>Close[Print]</td>
<td></td>
</tr>
<tr>
<td>Print</td>
<td>Print[GUI-Prototype]</td>
<td>PrintWithWhatIsThisInfo[GUI-Prototype]</td>
</tr>
<tr>
<td>GUI-Prototype</td>
<td>GUI-Prototype[Workspace]</td>
<td></td>
</tr>
<tr>
<td>Workspace</td>
<td>Workspace[FileNew]</td>
<td></td>
</tr>
<tr>
<td>FileNew</td>
<td>New[FileOpen]</td>
<td>NewWithWhatIsThisInfo[FileOpen]</td>
</tr>
<tr>
<td>FileOpen</td>
<td>Open[FileSave]</td>
<td>OpenWithWhatIsThisInfo[FileSave]</td>
</tr>
<tr>
<td>FileSave</td>
<td>Save[EditUndo]</td>
<td>SaveWithWhatIsThisInfo[EditUndo]</td>
</tr>
<tr>
<td>Windows</td>
<td>Windows[Help]</td>
<td></td>
</tr>
<tr>
<td>Help</td>
<td>Help[Contents]</td>
<td></td>
</tr>
<tr>
<td>GettingStarted</td>
<td>GettingStarted[UsingHelp]</td>
<td></td>
</tr>
<tr>
<td>UsingHelp</td>
<td>UsingHelp[Config]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 72 The GenVoca grammar
4.3 Graphical DSL

Using a graphical DSL permits the user to automate the whole specification development process. It is no longer necessary to see the logic or the declarations in the specifications. The important part is that the user needs no knowledge of the language used in the specification (e.g. XML). GUI elements are easier for the human perception than text specifications. When editing a text specification, it is highly possible to make mistakes, such as typing errors. Besides, the semantic rules can be violated because the user has to create and run the logic of the system that is being created in his head, which is absolutely impossible if he has to create complex systems. The introduction of a graphical DSL makes it easier for the user and takes over this task. It is in charge of both dependencies appearing in the system and invalid input. The generator is becoming more and more user-friendly, the user can easily learn how to use it by trying out the available options. The user can not do anything wrong, as the system accompanies him at every step assuring an error-free creation of a specification that does not need to be verified by the generator. The generator can be used intuitively and is easy to understand. The whole process of the specification creation runs in the background, visible for the user.

When creating a graphical DSL, it is necessary to transform the feature model into GUI elements. This transformation has the advantage that the user does not need an advanced knowledge of the feature models used in the DSL. The transformation of the DSL must meet the following criteria:

The mandatory features do not appear in the dialog because they are available anyway. If such a feature does appear in the dialog, then it appears only as a group boxes title.

![Optional feature transformation](image)

Figure 73 Optional feature transformation

The logic of the optional features is completely covered by the checkboxes. An additional logic verification is unnecessary.

![Alternate group transformation](image)

Figure 74 Alternate group transformation

The Radio buttons are suitable for the group of alternative features. In this case, an additional logic verification is also unnecessary because the logic coincides with the widgets (RadioButtons).
A group of *or*-features can be represented with the help of check box. In this case, an additional verification is necessary because the user must make sure that at least one feature is selected. Or, if the superior feature is an optional feature, the user is not allowed to deactivate it, e.g. if the user does not select one of its sub-features that are gathered in an *or*-group).

Dependency relations can be expressed in two ways. The best way is to use enable/disable. It is impossible to select anything that is not supposed to be selected. The relations become clear when the elements are situated in one layout. The user can see how the alteration of some elements influences the others.

Unfortunately, the enable/disable relation does not work for the elements that are situated in different layouts. The user cannot see what options in different layouts are influenced by the changes of an option in one layout. For this purpose, message boxes can be used. When the user selects an option, they can inform him about the consequences of his selection and offer alternatives.
Example of a feature diagram transformation

Figure 78 Example of a feature diagram transformation

Figure 78 shows an example of a feature diagram transformation. The mandatory features are not shown in the GUI layout because it is not allowed to manipulate them. “View” is an optional feature. This is why it is represented by a check box in the layout. It is important to keep in mind that when this feature is not selected, it is impossible to select its sub-features. This is why if the “View” option is not selected, all other options of this layout become disabled. Another important thing is that the three features Tool Bar Check, Status Bar and Zoom are grouped together as an *or*-group. This means that at least one of them must be selected. This problem could be solved this way: if no feature from the group is selected then it signifies that these components are not available in the system. The conclusion is that View is not needed in the system. If Tool Bar Check, Status Bar and Zoom features are not selected, it is possible to deactivate the father node “View” which automatically disables sub-features. However, this state would be irreversible. The “View” component has no functional code and consists only of GUI parts, namely the menu entry where further popup menu entries for the specific components are created. If there are no such components, it means that no menu entries must be made. This is why an additional XML tag `<viewmenu>` is introduced in the specification. It is in charge of control over this menu entry. The optional features 25%, 75%, 100% and 150% of the “Zoom” feature are represented by check buttons. If “Zoom” feature is deactivated, these buttons are disabled.

4.4 Angie-Based gui-generator (ABA)

Unlike the well-known GUI generators such as JANUS, the ANGIE-Based GUI generator (ABA) is domain-specific. This means that the GUI part of the system that needs to be generated also undergoes a feature modeling process. This permits the creation of the systems containing only those GUI parts and functions that were specified by the user, both in their generated source code and in the executable binary file. Furthermore, it is possible to use the components created by the user as well as those created by a software designer. Though ABA is a (feature) model-based GUI generator, it does not require an extensive knowledge of the base model because it is completely projected on the GUI elements and the specificator is connected to the generator. This is the reason why the generator can be used intuitively and be easily understood [EHLS02].
4.4.1 The distribution of generative domain model in ABA

In ABA, the generative domain model is divided so that the problem area is projected on the GUI of the specificator that is connected to the GUI generator. Most of the configuration knowledge such as the default settings, dependent features, illegal feature combinations as well as optimizations is taken over by the specificator. The construction rules are carried out with the help of ANGIE script functions. The solution area is projected on ANGIE frames. The frame contents include such files as C++, make files, XPM and Developer Studio workspace. Make files have two variants; one for Windows and the other for Linux. The Microsoft Developer Studio Workspace Files (.DSW and .DSP) are created to make further work on the prototype easier.

4.4.2 The system requirements

For smooth and productive work with ABA, the following software support is required:
- Windows 9.x, NT, 2k, XP
- ANGIE R2.1
- Qt 2.3.0
- Visual C++ 6.0

The generator is platform-dependent and works only with Windows platforms. A Linux support is not possible because the generator does not work without ANGIE, and ANGIE is only available for Windows platforms. Unlike the generator, the generated GUI prototypes are completely based on Qt and can be compiled both in Windows and in Linux. This ability of Qt to function with different platforms was the only reason why this framework was chosen for this project. Qt is only available for C++, this is why this language was also chosen. Theoretically, any library and programming language could be used instead of Qt and C++. Visual C++ is used in the generator for the creation of executable Windows files. In this project, only the Microsoft Visual C++ 6.0 was used because the only version of Qt available free of charge on the Windows platform was made for this C++ compiler.
4.4.3 The generation process in ABA

The whole development process runs in the background and is visible for the user. This is the reason why the generator can be used intuitively and is easily understandable. The base of the process is the ABA user interface, it controls the whole generation process.

![Image of the generation process in ABA](image)

One of the most important tasks of the ABA user interface is the creation of the XML specification that is then transferred to the ANGIE-part of the generator. There, the system creates the source code according to the selected specification. If the user wishes, the “Make” process is also carried out. When it is completed, the ABA user interface can start the created application.

4.4.4 ABA-User Interface

After the general description of the generation process, the details of the ABA user interfaces can be explained.
When the user starts ABA, he has a possibility to customize the generator options if necessary. ANGIE Path shows a path leading to the ANGIE compiler that can be customized either manually or by clicking the “Browse” button in a dialog. Working Directory gives the information about the path where the program is situated. Target Generation Directory shows the path where all the generated sources are saved. The whole generation process is started when the “Generate” button is clicked. In the “Generation” section, the “Source+Exe” and “Only Source” options can be set and it can be determined whether a “Make” process needs to be carried out after the creation of the source code, in order to create an executable file that can be started when the “Run” button is clicked. When the user clicks the “Exit” button, he exits the Generator.
The specificator creates the specification for the system that is being generated. Not any component combination is useful or can be created. The specificator carries out a dependency and a buildability check in the Real Time. This means that all invalid combinations are identified right after they are entered and the user is informed about them. The decision about what happens next lies with the user.
This window shows the XML specification created by the specificator that is transferred to the ANGIE part of the generator after the “Generate” button is clicked. The user can see them but can not change them directly in the window without running the risk of making errors. The changes can only be made by the generator.

4.4.5 The Process of Creating a XML-Specification

The main purpose of the specificator is to create a specification of a system. In order to accomplish this, it is necessary to remove all the variable parts from the feature diagram. This process is called specialization. The first part of the figure shows an example of a feature model that undergoes a specialization process with the help of the ABA specificator. A tree-like structure appears. Its “leaves” are “weighed down” by zeros and ones. If a feature is annotated with 0, then the generator gets the message that the component associated with this feature can not be in the system. Alternately, the features marked with 1 are required by the system.

This tree-like structure is described with XML. Based on the XML presentation, the user can see the whole structure of the system that is being generated and can imagine the components the system consists of. Based on this presentation, the user sees the non-existing components and the corresponding parts of the system where these components will not be situated. In order to create an XML-specification, the following was agreed upon:

- Every tag has one initialization value.
- Every initialization value must directly follow the XML tag.
- No entries are allowed after the initialization value.
- All tag names need to be unambiguous. This means that tags with the same name are not allowed to appear in the specification. The unambiguity require-
ment is very important because the tag names of the specification are used in ANGIE as alphanumeric indexes for the Collection.

4.4.6 Frame Creation

The user can see the tree-like structure created by the specificator as a frame hierarchy tree. The logic of the component construction rules is divided over the whole frame hierarchy. Every frame is a world of its own. First, the concept frame is created. It contains the knowledge of the frames that only have to be “called to life”. With the creation of the corresponding frame the purpose of the specificator is fulfilled. The created frames create further subordinate frames.

Figure 85 Frame creation
4.4.7 The Assembly of a Component

Parts of a component appear in different parts of the code. This is why every component includes parts from its father node, as they contain the information about the correct positioning of the component parts. Figure shows this process. First, the concept frame is created. From the XML specifier it receives the information about what components are necessary and creates them. The created components receive further information about the components they can contain from the XML-specification. If required by the specification, these components are created.
4.4.8 Realization (ANGIE)

It was decided to use the ANGIE frame processor for the creation process because at that time XVCL was being developed. This is how the ANGIE part of ABA works.

```
Function main() As Variant
    Dim colSpec   = createCollection()
    Dim xmlhandle = xmlLoad("spec", "SpecXML")
    Dim i = 0
    For i = 0 To xmlCount(xmlhandle)
        add(colSpec, xmlGetElementText(xmlhandle, i), xmlGetElementName(xmlhandle, i))
    Next i
    xmlClose(xmlhandle)
    createApplicationH(colSpec)
    ...
    End Function
```

Listing 16 Main function

First, the `main()` function is started. In the first line of this function, the collection is created. In the next line, a handler variable for the XML file is created. In the `For`-loop the values from the XML specification are transferred to the collection, the tag names are used as alphanumeric indexes. Afterwards, the XML file is closed. Then, the concept frames are started when the functions associated with them are carried out.

```
.Frame ApplicationH(colSpec As Collection), Column Off
    Dim nTempValue1  = CInt(colSpec("effects"))
    Dim frEffectsH1  = (nTempValue1 = 1) ? createFrame("EffectsH1", colSpec) : EMPTY
    ...
    ...
</frEffectsH1>
    ...
    .End Frame
```

Listing 17 Frame creation and embedding
The frame that was “called to life” takes over a reference to the collection in the constructor and decides about the destiny of other subordinate frames. For example, in the listing a “question” is asked to the collection whether the system needs to contain the “Effects” component. Using the alphanumeric index “Effects”, nTempValue1 receives the corresponding value that is then used in the ?:-operator in order to decide whether to give the frame variable a frame instance or the EMPTY value. Then, the embedding of a slot dependent from this variable is carried out in the code area of the frame.

```
.Frame ApplicationH(colSpec As Collection), Column Off
  .Dim nTempValue1 = CInt(colSpec("effects"))
  .Dim strEffects = (nTempValue1 = 1) _
      ? ""              _
      : EMPTY

  .Dim nTempValue2 = CInt(colSpec("invert"))
  .Dim strEffectsInvert = (nTempValue2 = 1) _
      ? ""              _
      : EMPTY
...
...
<?
Here is the place for Effects code
<?
  Here is the place for Invert code
<?strEffectsInvert!>
/>
<?strEffects!>
?>
...
...
.End Frame
```

Listing 18 Hierarchy imitation with Optional blocks

As an alternative, optional code blocks can be used to imitate a hierarchic structure because they support interlacing structures. This can be carried out by frame variables receiving only two values- an empty string or EMPTY. If a frame receives an empty string, the frame is going to contain the contents of the optional block, otherwise it will not. If this process is carried out, the time that needs to be invested in the implementation is significantly reduced. It becomes obvious that the structure becomes too complicated starting from the 3rd level of the hierarchic structure imitation.

In the creation process both of the variants mentioned above are used.
5 Conclusion and Future Research Directions

The goal of this research was to apply generative programming in the area of GUI application generation and try it out using an image processing program as an application example. This goal was mostly achieved.

A generator was developed fulfilling the requirements mentioned above. There is also a specificator with a dialog-based DSL connected to the generator. The specificator permits the user to create a specification in the form of a dialog in an easily comprehensible way. The prototypes automatically created from an abstract specification could be used directly after they were created.

The generator permits the creation of a large number of prototype variants. This number can be determined formulas given in figure 87:

\[
V(x) = \begin{cases} 
V' & \text{if } X \text{ mandatory} \\
V' + 1 & \text{if } X \text{ optional}
\end{cases}
\]

\[
V' = 1 \times (u_1^{\text{man}} \times u_2^{\text{man}} \times \ldots \times u_k^{\text{man}}) \times (u_1^{\text{opt}} \times u_2^{\text{opt}} \times \ldots \times u_l^{\text{opt}}) \times (g_1^{\text{alt}} \times g_2^{\text{alt}} \times \ldots \times g_m^{\text{alt}}) 
\]

\[
g^{\text{alt}} = u_1^{\text{alt}} \times u_2^{\text{alt}} \times \ldots \times u_r^{\text{alt}}
\]

\[
g^{\text{or}} = (u_1^{\text{or}} + 1) \times (u_2^{\text{or}} + 1) \times \ldots \times (u_p^{\text{or}} + 1) - 1
\]

Symbols and abbreviations:

- \(V(X)\) - variants number of a random node \(X\)
- \(V'\) - intermediate results
- \(u\) - variant number of an \(X\) sub feature
- \(g\) - variant number of the group \(X\) sub feature
- \(\text{man}\) - mandatory
- \(\text{opt}\) - optional
- \(\text{alt}\) - alternative
- \(\text{or}\) - or

If there are no \(u\) s or \(g\) s, there are skipped in 2), this means that \(V'\) of a leaf node equals 1.

Figure 87 The feature diagram variant number calculation formula. Based on [ESBC01]
According to feature diagram we calculate:
\[ g^\text{opt} = 19845 \]
\[ g^\text{all} = 6 \]
\[ t^\text{opt} = 2^{42} \]
\[ V = 5 \times 10^{17} \]

In comparison with that, using the Microsoft AppWizard \( V \approx 6 \times 10^7 \) prototype variants can be created.

Unfortunately, there was not enough time to create the complete functionality of the generated image processing program. The generated prototypes do not have the functionalities connected to the properties, zoom dialogs and menu Edit. However, it is possible to add these functionalities any time, as the generator presentation permits the creation of extension for further development. This characteristic was used in the development process.

It is important that only the functions and GUI parts that were requested by the user were generated. This means that the functionalities that were not requested can not be activated in the created prototypes, because their code is not available. The prototypes are highly personalized and they fulfill the requests of the user who “ordered” certain functionalities. The requests of others are partially or fully ignored.

One might criticize that the examination of the GUI application generation with the resources was not carried out. Though the resources were not supported in all the parts (e.g. Java), it would, nevertheless, be interesting to carry out such an examination.

A useful expansion of the ABA would be import and export of the XML specification with automatic adjustment of the specifier GUI elements to the feature model the specifier contains. Since the new AmiEddi2 offers the possibility to project the features on the GUI elements, it would be practical to create a GUI generator as a plug-in for this project. Unfortunately, it could not be done in the present research because AmiEddi2 was not fully developed then.
Index of Figures and Listings

Figures

Figure 1 Question-answer dialog ........................................................................................................... 9
Figure 2 DOS Shell ...................................................................................................................................... 9
Figure 3 Menu dialog ................................................................................................................................. 9
Figure 4 Direct manipulation dialog 1 (screen shot WindowsXP) ............................................................. 10
Figure 5 Direct manipulation dialog 2 (screen shot WindowsXP) ............................................................. 10
Figure 6 Form dialog (screen shot Microsoft Paint Version 5.1) ............................................................. 10
Figure 7 The Closeness Rule (based on [Ze94]) ....................................................................................... 11
Figure 8 The Equality Perception Rule (based on [Ze94]) ..................................................................... 11
Figure 9 The Completeness Perception Rule (based on [Ze94]) ............................................................. 11
Figure 10 The Boldness Perception Rule (based on [Kl95]) .................................................................. 12
Figure 11 Window anatomy ...................................................................................................................... 13
Figure 12 Message Dialog (screen shot MiniABA) ................................................................................ 14
Figure 13 File Dialog (screen shot Microsoft Windows XP) .................................................................... 14
Figure 14 Color Dialog ............................................................................................................................ 15
Figure 15 Print Dialog ............................................................................................................................. 15
Figure 16 Group Box ............................................................................................................................... 16
Figure 17 Label ....................................................................................................................................... 16
Figure 18 Progress Bar ............................................................................................................................ 16
Figure 19 Command Button .................................................................................................................... 17
Figure 20 Radio Button ........................................................................................................................... 17
Figure 21 Check Box (screen shot WindowsXP) .................................................................................... 17
Figure 22 Spin Box ................................................................................................................................... 18
Figure 23 Slider ....................................................................................................................................... 18
Figure 24 Scroll Bar .................................................................................................................................. 18
Figure 25 List Box (screen shot VisualStudio 5.0) .................................................................................. 19
Figure 26 Drop-Down List Box (screen shot WindowsXP) ................................................................. 19
Figure 27 Combo Box (screen shot Visual SourceSafe 6.0) ................................................................. 19
Figure 28 Drop-Down Combo Box (screen shot VisualStudio 5.0) .......................................................... 20
Figure 29 Date Picker ............................................................................................................................ 20
Figure 30 Tree View Control .................................................................................................................. 20
Figure 31 Tree View Control (screen shot Windows98) ........................................................................ 21
Figure 32 List View Control (screen shot WindowsXP) .......................................................................... 21
Figure 33 Tab Control ............................................................................................................................. 21
Figure 34 Line Edit .................................................................................................................................. 22
Figure 35 Multi-line Edit .......................................................................................................................... 22
Figure 36 Screen shot of Qt-Application example .................................................................................. 25
Figure 37 Screen shot of dialog example ............................................................................................... 28
Figure 38 Screen shot of menu example ............................................................................................... 29
Figure 39 Window of Qt Designer .......................................................................................................... 33
Figure 40 The Main Window of Glade (taken from [GL]) ..................................................................... 34
Figure 41 Widget palette of Glade (taken from [GL]) ............................................................................ 34
Figure 42 The Property Editor of Glade (taken from [GL]) ................................................................... 35
Figure 43 Dialog window of AppWizard ............................................................................................... 39
Figure 44 The application specification process in the form of a dialog ............................................... 40
Figure 45 Frames specification ................................................................................................................ 40
Figure 46 AppExpert dialog box ............................................................................................................ 41
Figure 47 GENIUS generation process ................................................................................................. 43
Figure 48 GENIUS window ..................................................................................................................... 44
Figure 49 Generation process in BOSS ................................................................................................ 52
Figure 50 Generation process in EmuGen ............................................................................................... 53
Figure 51 Steps of a possible execution of the generated MUI application (taken from [Braa]) ......... 56
Figure 52 Generative domain model ..................................................................................................... 57
Figure 53 Development process (based on [Neu98]) .......................................................................... 60
Figure 54 Example of a feature diagram ............................................................................................... 61
## Listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 1</td>
<td>QObject constructor</td>
<td>23</td>
</tr>
<tr>
<td>Listing 2</td>
<td>Signal/Slot</td>
<td>24</td>
</tr>
<tr>
<td>Listing 3</td>
<td>Qt-Application example</td>
<td>25</td>
</tr>
<tr>
<td>Listing 4</td>
<td>Dialog example</td>
<td>27</td>
</tr>
<tr>
<td>Listing 5</td>
<td>Menu example</td>
<td>28</td>
</tr>
<tr>
<td>Listing 6</td>
<td>AppWizard (“about”-window definition)</td>
<td>40</td>
</tr>
<tr>
<td>Listing 7</td>
<td>Data model example (adopted from [Brab])</td>
<td>54</td>
</tr>
<tr>
<td>Listing 8</td>
<td>Task model example (adopted from [Brab])</td>
<td>54</td>
</tr>
<tr>
<td>Listing 9</td>
<td>User model example (adopted from [Brab])</td>
<td>55</td>
</tr>
<tr>
<td>Listing 10</td>
<td>Frame</td>
<td>65</td>
</tr>
<tr>
<td>Listing 11</td>
<td>If-Statement</td>
<td>67</td>
</tr>
<tr>
<td>Listing 12</td>
<td>For-Loop</td>
<td>67</td>
</tr>
<tr>
<td>Listing 13</td>
<td>Frame example</td>
<td>68</td>
</tr>
<tr>
<td>Listing 14</td>
<td>Frame creation example 1</td>
<td>68</td>
</tr>
<tr>
<td>Listing 15</td>
<td>Frame creation example 2</td>
<td>68</td>
</tr>
<tr>
<td>Listing 16</td>
<td>Main function</td>
<td>94</td>
</tr>
<tr>
<td>Listing 17</td>
<td>Frame creation and embedding</td>
<td>94</td>
</tr>
<tr>
<td>Listing 18</td>
<td>Hierarchy imitation with Optional blocks</td>
<td>95</td>
</tr>
</tbody>
</table>
Appendix

The CD Contents
The accompanying CD contains the complete source and an executable version of ABA, an electronic copy of this document in the PDF format, the Acrobat Reader and references to the used tools.
Glossary

A

AppExpert: This is one of the expert tools in Borland C++. It automatically creates programs that run on the Windows platform.

*Application Programming Interface* (API): This is a functionality complex that allows the interaction with an operation system, e.g. with the functions of the Windows API, or with any dynamic link library that contains a set of functions that can be called.

C

Character-based interface: A non graphical interface, being only capable of displaying ASCII characters.

*Computer Aided Software Engineering-Tools* (CASE-Tools): This is a general term for the software tools that automate or support the creation of a software draft as well as programming the software.

*Common User Access* (CUA): This is part of the *System Application Architecture* (SAA) developed by the IBM that deals with the user interface standards for the application and system software.

Configuration Knowledge: “The knowledge necessary to automate the assembly of system family members out of implementation components. It specifies illegal combinations of system features, default settings, default dependencies, optimizations and construction rules stating which configurations of components satisfy which configurations of features. Configuration knowledge can be implemented using generators.” [CE00]

D

*Data Base Management System* (DBMS): The purpose of a DBMS is the management and control of data.

*Data Base System*: It consists of one or more data bases, a →*Data Dictionary* and a →DBMS. All the data is accumulated in a data base.

*Data Dictionary* (DD): It contains the data in the data base.

Dialog: This is the interaction between a user and a dialog system. It has a specific purpose. The user is a person working with the dialog system. The primary dialog consists of steps that contribute to the direct accomplishment of a task. Secondary dialogs are used for additional information that depends on the specific situation.
Dialog box: A window where other windows (dialog box controls) appear. A dialog box is usually used for the accumulation of the user's additional information. This information is required for the completion of a command or a task.

*Dynamic Link Library (DLL)*: This library contains functions, classes, and resources. A program loads this library only when access to it needs to be obtained. The DLL elements can be simultaneously used by several applications and take up memory space only when they are being used.

**F**

*Frame*: “A component in any programming language that can be reused, not only as-is, but as adapted by other frames to fit a new application. A frame contains (1) program commands and variables and (2) frame commands and variables. The frame commands can add to or subtract from other frames' capabilities as the application requires.” [Bas97]

**G**

*Graphics Interchange Format (GIF)*: It was developed by CompuServe Inc. It is used for the maintenance of pixel images in one file and for the exchange of pixel data between platforms and systems. This is the most popular format for multi-byte graphics and images, especially in the Internet and the World Wide Web.

*Graphical User Interface (GUI)*: This is a general term for graphical user interfaces regardless of whether they are implemented with the system's own functionality or with graphical standards.

*Generative Domain Model*: “A model of a system family that allows the automatic generation of family members from abstract specifications. It consists of a means for specifying family members, the implementation components from which each member can be assembled, and the configuration knowledge” [CE00].

*Generative Programming*: “Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge. The generated products may also contain non software artifacts, such as test plans, manuals, tutorials, maintenance and troubleshooting guidelines, and so on” [CE00].

*Generator*: “A generator is a program that takes a higher-level specification of a piece of software and produces its implementation. The piece of software could be a large software system, a component, a class, a procedure, and so on” [CE00].

*GenVoca*: Don Batory’s GenVoca is a general model for the generation of software. The standardized description of the basic abstractions of a domain and its interfaces
are the basis for the generation of among one another compatible and replaceable components. The set of GenVoca-components implementing an abstraction or an interface are called a ‘Realm’. Components are parameterised using other components belonging to other realms and generate an implementation for the abstractions of a realm. New realms can be defined from already existing realms. The composition of components takes place through the intention of their parameters. Design rules define restrictions for the composition of components and their ability to instance. The language for GenVoca is P++ - an extension of C++. GenVoca has been used to implement a number of generators (e.g. a generator for communication protocols)“ [Goe99].

I

Integrated Development Environment (IDE): This is an environment that enables developers to edit, compile, debug, and view programs.

International Standards Organization (ISO): This is the main organization that develops and maintains international technical standards.

Interface Builder An interactive development system for user interfaces (UIDS). Some of the interface builders only set the dialog layout, others also connect it to the application program and create the communication between parts of the dialog base.

J

Joined Photographic Experts Group (JPEG): A group in the ISO in charge of the creation and specification of the JPEG standard for the compression of still images. The JPEG format is very popular for Internet and World Wide Web use.

M

Multiple Document Interface (MDI): This is a style of an application. Many data windows can appear in the program window.

Macro (in the context of C/C++ preprocessor): This is a name assigned to a set of commands defined by the user. When this name is mentioned in the program, it is replaced by the corresponding set of commands. Unlike functions, macros extend the program using text, “cutting and pasting”. Macros allow the substitution of parameters.

Microsoft Foundation Classes (MFC): A library with a set of classes which case the development of graphical user interfaces for MacOS or Windows. It shields the developer from directly using the primitive functions of the APIs of the respective windows systems.
Object-oriented Analysis (OOA): Modeling the software system requirements by using object-oriented concepts and notations.

Object-oriented Design (OOD): Modeling the design and the architecture of a software system using object-oriented concepts and notations.

Object Windows Library (OWL): This is a GUI class library for programming in C++ and Pascal. developed by Borland Inc.

Open Database Connectivity (ODBC): This is a standardized universal interface for accessing various relational data base systems. Originally it was specified by Microsoft.

Palette or tool window: This is a type of secondary windows. It displays a group of different tools or choices.

Product Line: “A product line is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market” [CE00].

Product Family: “A product family is a group of products that can be built from a common set of assets” [CE00].

Rapid Prototyping: This is a process of early development of executable applications. This approach is used mainly on the dialog part of the software project because it is difficult to specify the graphic presentation of the dialogs in the application. The prototype serves as a basis for discussing and improving the functionality of the user interface.

Single Document Interface (SDI): This is a style of an application. The program window contains only one window. An example of an SDI application is the Notepad program on Windows systems.

Software Ergonomics: It deals with the user-friendly presentation and interaction of software systems. Its aim is to adjust the software to the user's needs and perceptions.

Structured Query Language (SQL): This is one of the data base programming languages developed by IBM. It allows a relatively comfortable communication with a (relational) data base.
Swing: Class framework which allows to develop interactive GUI-based Java-applications.

**U**

User Interface Management System (UIMS): It supports the separation of software in an application part and a part implementing the user interface. It provides tools for the description of syntax and semantics of dialog components. The use of these specific tools is justified by different aims of the application and the dialog parts. These tools themselves can become interactive or can be defined in form of specification languages (such as UIL).

Unified Modeling Language (UML): The language for the visualization, specification, construction, and documentation of the artifacts of a software-intensive system. It was developed by Booch, Rumbaugh, and Jacobson for the Rational Software Corporation. In 1997 it was accepted as a standard by the Object Management Group (OMG).
References

[Bal99] Heide Balzert; *Lehrbuch der Objektmeshellung: Analyse und Entwurf*; Spektrum, Akademischer Verlag; 1999


[Bas91] Len Bass; *Developing software for the user interface*, Addison-Wesley 1991


[Braa] Alfons Brandl; *EmuGen: A Generator for Multiple-User Interfaces*; Technische Universität München; Technical Report 4/9/2001 [http://www2.in.tum.de/persons/brandla/emugenSpeech](http://www2.in.tum.de/persons/brandla/emugenSpeech)

[Brab] Alfons Brandl; *generating User Interfaces for Ad-hoc Workflows*; Technische Universität München; Technical Report 12/15/2000 [http://www2.in.tum.de/persons/brandla/emugenSpeech](http://www2.in.tum.de/persons/brandla/emugenSpeech)


[BS00] Peter Brössler, Johannes Siederleben; *Softwaretechnik: Praxiswissen für Software-Ingenieure*; München Wien Hanser, 2000

[Cal96] Alex Calvo; *The Craft of Windows95 Interface Design*; Springer-Verlag; 1996


[Cle01] J. Craig Cleaveland; Program Generators with XML and JAVA, Prentice Hall, 2001


[Dar00] Rik Darnell; *JavaScript Sprawochnik,* Piter 2000

[Dia93] Dialektika; *ObjectWindows dlja C++,* Kiew Dialektika, 1993

[DST02] Delta Software Technology GmbH; *Angie Online Help,* provided with ANGIE, version 2.1. 2002


[Eck96] Frank Eckgold; *Windows Anwendungs- und Systemprogrammierung,* Vieweg, 1996

[ECV02] Ulrich W. Eisenecker, Krzysztof Czarnecki, Markus Voelter; *Generative Programmierung für die Entwicklung von Softwaresystemfamilien,* Slides, OOP2002

[Eis02] Ulrich W. Eisenecker; *Generative Programming - From Theory to Practice,* PoLITE-Workshop, 9/11/2002


[Emr02b] Marco Emrich; *Frame Technology,* PoLITE-Survey, 2002

[Emr02c] Marco Emrich; *Introduction to Generative Programming,* In: PoLITE-Survey, 2002

[EHLS02] Ulrich W. Eisenecker, Mathias Henss, Markus Lang, Max Schlee; *Generative Programmierung,* Slides, Net.ObjectDays 2002


[EV02] Ulrich W. Eisenecker, Markus Voelter: Generative Programmierung für die Entwicklung von Softwaresystemfamilien, Slides, OOP2002

[Fäh96] Klaus-Peter Fähnich: Werkzeuge zur Entwicklung graphischer Benutzungsschnittstellen: Grundlagen und Beispiele, Oldenbourg, 1996


[Gei90] Georg Geiser; Mensch-Maschine-Kommunikation, München; Wien Oldenbourg, 1990


[GPW02] Dzims Rilko, Catherine Bohsung; Poster-Presentation of the GP-WEB project. GP-WEB-Workshop, University of Applied Sciences Kaiserslautern, Zweibrücken, 2002


[Joh00] Jeff Johnson; GUI Bloopers, Morgan Kaufmann Publishers 2000


[Kli95] Arnold Klingert; *Einführung in Graphische Fenstersysteme*, Springer-Verlag, 1995


[LA98] Richard Leinecker; Tom Archer; *Windows98 Programming Bible*; IDG Books Worldwide, Inc. 1998

[Leh99] Burkhard Lehner; *KDE- und Qt-Programmierung. GUI-Entwicklung für Linux*, Addison Wesley Longman, 1999


[NY90] Peter Norton, Paul Yao; *Windows3 Peter Norton's Programmiertechniken für Profis*; Borland GmbH; 1990


[Otr02a] Otris Werbeprospekt; *Janus Anwendungsbericht*, 2002

[Otr02b] Otris Werbeprospekt; *Janus Mehr Zeit für neue Ideen*, 2002

[Otr02c] Otris Werbeprospekt; *Pilotanwendungen auf Knopfdruck Vom UML-Modell zur perfekten Lösung*, 2002

[Otr02d] Otris, Die JANUS-Generatorfamilie “JANUS/Prototype”, 2002


[Pew97] Gerhard Pews; *Programmieren in Smalltalk-80*,
http://www.arg.informatik.uni-kl.de/~pews


[Hen99] Peter A. Hennig; *Vorlesung Multimedia*, Slides, Fachhochschule Karlsruhe, 1999


[Hym95] Michael I. Hyman; *Borland C++ for dummies*; IDG Books Worldwide, 1995

[Sch01a] Rüdiger Schilling; Generative Programming – Von der Theorie zur Praxis, Delta Software Technology GmbH, 2001, see www.delta-software-technology.com

[Sch01b] Siegfried Schnieders; *gtk+ und Qt - Ein Vergleich*, Linux Enterprise Magazine 1/2001, S. 49-52


[Sel00] Mario Selbig; *A Feature Diagram-Editor – Analysis, Design and Implementation of its Core Functionality*, Diploma thesis, University of Applied Sciences Kaiserslautern, Zweibrücken 2001


[She92] Ben Shneiderman; *Designing the User Interface, Strategies for Effective Human-Computer Interaction*, Addison-Wesley, 1992, Second Edition

[Tro02a] Home Page of Troll Tech Inc. (Trolltech).
http://www.trolltech.com/products/qt/tools.html

[Tro99] Trolltech Qt Designer Hilfe, Copyright Trolltech AS Version 1.1

[VP00] Victor Porev, Komputernaja graphika; Korneitschuk Kiew, 2000


[Wes98] Ivo Wessel; GUI-Design, Hanser Verlag, 1998
Hiermit versichere ich, dass ich meine Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.


Max Schlee