**UNIVERSITETET I OSLO**
**Institutt for informatikk**

# Evolution by Resemblance in Component-based Visual Application Development

Hovedoppgave

Balder Mørk

**12. mai 2004**

# Abstract

As end-user development (EUD) receives more and more focus in research, using prefabricated software components and visual application builders has been presented by several researchers as a useful aid in the process. One of the challenges of this approach is the emergence of situations when the available components aren't entirely suitable for the task to be solved. This thesis presents a possible solution to this problem, in the form of *cloneable components*. By allowing the end-user to perform changes to a clone of a familiar prototype, a new component with the desired properties can be created without risking damage to the original component. This form of *evolution by resemblance* lets end-users with little or no programming experience perform evolutionary software development based on existing software components.

To demonstrate these concepts in practice, the thesis presents the *SimpleBuilder*, a modification of Sun Microsystems' BeanBuilder application. The SimpleBuilder contains new functionality for working with cloneable software components. Example software components are provided, that also provide a uniform interface for inter-component communication, allowing a user of the SimpleBuilder to easily connect software components to each other without writing program code.

# Foreword

The thesis you are about to read marks the conclusion of one year's worth of research for the Norwegian Candidatus Scientiarum degree. The thesis work has included development of a component-based development toolkit that introduces the concept of cloning components as a way of aiding evolutionary development of reusable components. The thesis was written under the guidance of associate professor Anders Mørch at InterMedia, University of Oslo. InterMedia is a centre for interdisciplinary studies in new media and e-learning, and was established in 1998.

A note on the use of pronouns in this thesis: The reader may notice that I consistently use the forms "he" and "his", e.g. when describing different scenarios. The reason for this is precisely that; I wish to be consistent throughout the text, and I just happen to be male. By no means do I intend to exclude females from the area of software engineering.

## Acknowledgements

I would like to thank the following persons for support and help:

- **Anders Mørch** for guidance and help in writing, researching, and finding interesting approaches to an interesting problem area: end user development. As my guide and advisor on this thesis, he was an invaluable asset in my research.
- **Tone Bratteteig** for valuable comments on the thesis in one of its last incarnations, allowing me to do some much-needed, last minute polishing of the document you are now reading.
- **Heidi Fløtberget** for being my loving partner and faithful supporter during the work on this thesis, and for pretending to be interested in my enthusiastic elaborations on the subjects discussed herein.
- **Heidi Tovdal** for never doubting that I was able to pull this off in the normalized time frame.

- **Kent Gøran Carlsson** for nagging, pushing and poking me enough to actually make me work on the thesis although fun, sun and alcoholic beverages were available elsewhere.
- Every single lecturer and fellow student I have encountered during my years at the Department of Informatics. I have had the pleasure of meeting a whole host of interesting and knowledgeable people over the course of my studies, adding more value to my studies than any textbook or collection of lecture notes could ever have done. **Marit**, you're doing great work teaching our fresh fellow students, and I am confident that one day you'll get out of this place with a diploma.

# Table of Contents


**1 Introduction** **1**
- 1.1 Facilitating EUD through component based toolkits . . . . . 1
- 1.2 The cloning approach . . . . . 2
- 1.3 Problem description . . . . . 4
  - 1.3.1 Delimitations . . . . . 4
- 1.4 Thesis structure . . . . . 4

**2 Theory** **6**
- 2.1 Component based development . . . . . 6
  - 2.1.1 History . . . . . 6
  - 2.1.2 Status . . . . . 7
- 2.2 Evolutionary software development . . . . . 9
  - 2.2.1 History . . . . . 9
  - 2.2.2 Status . . . . . 12
- 2.3 Cloning and prototypes . . . . . 15
  - 2.3.1 History . . . . . 15
  - 2.3.2 Status . . . . . 17
- 2.4 Fitting the pieces together . . . . . 19

**3 Tools and methods** **23**
- 3.1 Developer tools . . . . . 23
  - 3.1.1 The Java programming language . . . . . 24
  - 3.1.2 Operating system and developer environment . . . . 24
- 3.2 Component technology . . . . . 25
  - 3.2.1 Component-based vs. traditional programming . . . 26
  - 3.2.2 JavaBeans . . . . . 27
- 3.3 The BeanBuilder . . . . . 28
  - 3.3.1 Features . . . . . 30
  - 3.3.2 Required extensions . . . . . 34
- 3.4 Development strategy . . . . . 37

**4 My contribution** **40**
- 4.1 A scenario for prototype-based EUD . . . . . 40

4.2 Working with cloneable components . . . . . 42
4.2.1 A process model for clone-based component development . . . . . 44
4.3 The SimpleBuilder . . . . . 47
4.3.1 Overview . . . . . 47
4.3.2 The palette and control panel . . . . . 47
4.3.3 Customization of components . . . . . 49
4.3.4 Extension of components . . . . . 51
4.3.5 Connecting two components . . . . . 52
4.4 The SimpleBeans . . . . . 52

**5 Discussion** **56**
5.1 The road taken . . . . . 56
5.2 Alternative routes . . . . . 58

**6 Further research** **61**
6.1 Remaining work . . . . . 61
6.2 Putting the SimpleBuilder to use . . . . . 62
6.3 Improving the SimpleBuilder for prototype-based end user development . . . . . 63

**7 Conclusion** **65**

**A Code listings** **73**
A.1 An example of an application stored as XML . . . . . 73
A.2 The SimpleBean and SimpleBeanObject classes . . . . . 79
A.3 Sample SimpleBean and corresponding BeanInfo . . . . . 81

# List of Figures


2.1 The STEPS model of evolutionary software development . . 11
2.2 The spiral model of software development . . . . . 13
2.3 Seeding, Evolutionary Growth and Reseeding . . . . . 15
2.4 Graphical representation of objects in SELF . . . . . 17
2.5 Cloning a text field in HyperCard . . . . . 19
2.6 The HyperCard container hierarchy. . . . . 19
2.7 The results of listing 2.1 on page 21. . . . . 22

3.1 Example of proxy objects for non-visual components . . . . 29
3.2 A screenshot of the Sun BeanBuilder . . . . . 31
3.3 A running application designed in the BeanBuilder . . . . . 33
3.4 Component wiring in the BeanBuilder . . . . . 34
3.5 The life cycle model of the SimpleBuilder development process 39

4.1 An imagined tool for client data retrieval. . . . . 41
4.2 The imagined tool after functionality extension . . . . . 42
4.3 The use distance and design distance in customization and extension . . . . . 45
4.4 The process model underlying the SimpleBuilder. . . . . 45
4.5 Screenshot of the SimpleBuilder directly after startup . . . . 48
4.6 A selected and an unselected component in the SimpleBuilder 50
4.7 The "Clones" tab of the builder palette . . . . . 50
4.8 Three steps of cloning and customization . . . . . 51
4.9 Just a few of the available methods when connecting two JFC components. . . . . 53

5.1 My initial process model for developing with cloneable components. . . . . 58
5.2 Inheritance scheme for clones made in the SimpleBuilder. . . 59

# Chapter 1

# Introduction

> Each problem that I solved became a rule which served afterwards to solve other problems
>
> — *René Descartes*

In this chapter, I will look at the premises for the work done as part of the thesis. I will discuss the background for my work, and present a summary of what my application should provide.

## 1.1 Facilitating EUD through component based toolkits

This thesis is part of an ongoing work at InterMedia, focused on the subject of end-user development (EUD)[1]. The idea of user empowerment at the workplace has long traditions in Scandinavia (Nygaard 1979, Bansler 1989, Ehn 1993), so it seems natural that we continue research in this area. A common problem with the use of software in today's workplaces is that work practice will often change faster that developers are able to update software. The workers therefore need the possibility to customize, or even create, software that is meant to support their day-to-day work (Mørch & Mehandjiev 2000).

In this thesis, I will explore a new approach to component-oriented end-user computing. Several attempts to support end-user development have

[1] For more information on end-user development, see the website of the "Network of Excellence on End-User Development", EUD-NET at `http://giove.cnuce.cnr.it/eud-net.htm`. EUD-NET is sponsored by the European Commission, with the goal of preparing a research agenda in the field of end-user development.

been based on toolkits that employ a visual interface for composing an application from ready-made components (Stiemerling, Hinken & Cremers 1999, Costabile, Fogli, Fresta, Mussio & Piccinno 2003). The end-user developer usually chooses components from a palette, and places these on a "canvas" or "root window". He then performs customization of the components by setting the values of the component's parameters. The level of customization allowed by a component is decided by the developer that originally supplied it, through the developer's choice of which parameters are publicly accessible. If the components available to developers in component assembly toolkits become too limited, the end-user developer has no choice but to either work around the limitations, or ask an accomplished programmer to perform the necessary changes to the components in question.

## 1.2 The cloning approach

My goal is to add a new aspect to the component based development process, in the form of *cloneable components*. In this context, I will define a cloneable component as "A software component that may be copied accurately, leaving a clone with all its properties set to the values of the original at the time of cloning". We need to account for the fact that users are likely to encounter situations where the choice of components presented to them is too limited to allow them to achieve their goal in a suitable way. I will therefore attempt to present the user with a way of expanding the palette of available components, and having these new components available for later use.

Human creativity is often based on taking the familiar and changing it to fit the present requirements (Ghiselin 1985). Based on this ideology, we want to supply a mechanism for this type of *evolution by resemblance*. The user should be able to select a familiar component, modify and extend it, and have the resulting component available for use in the current - and for later - software projects. Naturally, we also need to provide a basic set of components that may be used and modified as described here. A key point in the process is that the application stays "alive". By allowing the user to edit the application at any time, the traditional distinction between "application environment" (actual use of an application) and "design environment" (developer tools, component assemblers etc.) disappears. To account for factors such as changing work practices and the increasing skills of end-user developers, the project should be open for further modification by other end-user developers. An important concept in the Scandinavian tradition of software engineering is the idea of applications being designed with its intended use in mind, a view known as the *application perspective* (Bjerknes

& Bratteteig 1988). By letting the user immediately extend and improve an application based on experience from the use situation, this view is taken a step further.

As argued in (Petroski 1992), a driving force for evolution of artifacts is the presence of annoyances and mediocrity in existing solutions. Therefore, applications created with our framework must be modifiable by both the designer and other, future users. This flexibility is needed, not only on the application level, but on the component level as well. Components must be open for individual redesign and improvement, and they must be available for use in other applications.

Although we see an increase in computer literacy in today's workforce, few end-users are familiar with the formality of programming languages. It is not to be expected that average end-users have the necessary skills to perform programming in a traditional sense. Workers usually relate to the graphical interfaces presented to them, and often feel at home in this environment (Preece, Rogers, Sharp, Benyon, Holland & Carey 1994). Most people that work with computers today are also familiar with the concept of connecting pieces of electronic equipment. They know that to get visual feedback from their computer, they need to connect a monitor to the VGA port of the computer. Likewise, for the computer to receive input from the user, a connection between the keyboard and computer is required. I hope to lower the threshold of component assembly to a sufficient degree that the prospective end-user developer is presented with the simple task of choosing appropriate components needed for the application, and connecting them in such an intuitive process that it feels no more complicated that hooking up a modern Hi-Fi sound system. To achieve this, the number of options presented to the user when establishing connections between components needs to be reduced to an absolute minimum. Still, each component needs to be flexible enough to accommodate a wide range of uses.

Most component assembly tools today present the user with a plethora of options for configuration of components. This is okay for professional developers and very advanced users, but we need to reduce such complexity in order to avoid overwhelming and confusing an end-user developer. Assembly should be simple and straightforward. The user should not be required to learn how to work with a new, unfamiliar interface. When the user is satisfied with the state of the application, he should be able to save the results to disk, and run the application like any other program on his computer at a future point.

## 1.3 Problem description

Following on these premises, my challenge is this: To produce a component-based developer environment with a level of complexity suitable for domain experts with little or no programming experience. The tool should provide the opportunity to store specific configurations of a component as a clone. The state of these cloned components should be stored as default values for later use in future projects. The tool should also provide mechanics for extending a cloned component, i.e. adding features not present in the original. Finally, the tool should simplify the process of "wiring up" components to a level comparable to that of connecting common electronic equipment found in modern homes. This tool is not to be regarded as a product for deployment in companies and institutions. Rather, I hope to provide a proof of concept; an artifact that can serve as basis for further discussion within the field of end-user development, with a focus on component assembly with clone-enabled components.

### 1.3.1 Delimitations

The development of a system like this is likely to take time, both in terms of planning and implementation. As with any new development within an area, a tool like this requires evaluation and field experiments to assess its usefulness. Unfortunately, time does not allow for both development of a product that is mature enough for actual testing of the application to a degree that would provide useful feedback for further research. Therefore, this thesis will focus on the actual development of a prototype toolkit for clone-enabled component assembly. This means that I cannot present any empiric evidence of the benefits of such an application; only my own theories and personal experiences on how this may be used to aid end-user development in the workplace.

Delivering a complete system for clone-based end user development while at the same time writing a comprehensive report is beyond the scope of two semesters' worth of research. My hope is that the work done on the SimpleBuilder will inspire and excite future students, and perhaps one day lead to completion. It is also my sincere hope that I may return to this work one day, or perhaps do research on other areas in the field.

## 1.4 Thesis structure

The thesis is divided into seven chapters:

- Chapter 1: Introduction, which you are reading now. The chapter attempts to give a description of the premises for the thesis and the problem which inspired my work on the SimpleBuilder.
- Chapter 2: Theory. This chapter presents the theoretical foundation for this thesis, and discusses the literature on relevant issues.
- Chapter 3: Tools and Methods, in which I describe the process of my work, and the tools I used.
- Chapter 4: My Contribution. Here, I try to explain what knowledge is to be gained from this thesis, and what advances in the field I feel I have done.
- Chapter 5: Discussion, which contains a general discussion on the process of developing the SimpleBuilder, and what could have been done differently.
- Chapter 6: Further Research. In this chapter I present ideas for further exploration of the field, and how my work may be continued by other researchers.
- Chapter 7: Conclusion. In the final chapter I summarize the results of my research, and see in what respect I have addressed the problem description from chapter 2.

# Chapter 2

# Theory

> The difficulty lies, not in the new ideas, but in escaping the old ones, which ramify, for those brought up as most of us have been, into every corner of our minds.
>
> — *John Maynard Keynes*

This chapter presents the scientific background for this thesis. I look at the history of three points of interest (components, evolution and cloning), as well as the literature available on the subjects. I will also try to tie these subjects together, and see how research in these fields may impact on end-user computing.

## 2.1 Component based development

### 2.1.1 History

The roots of component-based assembly can be traced to the early research into object-oriented programming at the Norwegian Computing Centre (NCC) in the mid-60s (Dahl, Myhrhaug & Nygaard 1968, Nygaard & Dahl 1978). The introduction of objects, semi-independent software modules, led to the logical next step of ready-made components available for use in later software projects. This type of code reuse was mentioned as early as 1968, at the NATO Conference in Software Engineering[1] (McIlroy 1968). A pressing issue at the time was the so called "software crisis" (David & Fraser 1968). As software projects grew increasingly more complex, it was

[1] This is probably one of the earliest occurrences of the term "Software Engineering". The expression was chosen to emphasize the need to focus on a structured approach to programming, similar to that of engineers' approach to construction.

evident that products sometimes arrived late, and were unable to fulfill expectations. There was "(...) a rather large gap, between what was hoped for from a complex software system, and what was typically achieved". Douglas McIlroy[2], a visionary programmer of that time, argued that software engineering was using backward techniques compared to, for example, the electronics industry. He suggested a radical new strategy for building new software, based on composing software from a large array of available components; chosen from a catalogue. Hopefully this would ensure quality software, built from tried and tested components. A few years later, McIlroy admitted that his idea of components hasn't really caught on (McIlroy 1972). There were no examples of companies offering components as a retailed product, although software houses had employed component architecture in-house to shorten delivery times and reduce complexity,

In the mid-80s the concept of "software integrated circuits" was introduced by Brad Cox (Cox 1986), thus adopting the analogy to the electronics engineering industry, used almost twenty years earlier by McIlroy. New, object-oriented languages such as Smalltalk and C++, as well as the proliferation of powerful computers, had shifted the programmers' focus from tight, hand-optimized code to code that improved modularity, extensibility and maintainability. Cox envisioned these components as an aid in the evolutionary development of an application, in which parts of the application could be easily swapped for other, newer parts that may add functionality, speed and robustness. To create an infrastructure for these components he invented a new programming language based on C; *Objective C*. The main description of Objective C, and how to use it as a tool for creating software components, is discussed in (Cox 1986). This object-oriented language has lately been overshadowed by the success of C++, invented by Bjarne Stroustrup, but is still in use in the Mac OS X[3] and GNUStep[4] projects.

### 2.1.2 Status

Until recently, the arguments for component based development were for the most part concerned with economis: developers could spend less time reinventing the wheel, and more time on developing new code. Additionally, components were presented as a commercially viable product, that could be sold to other developers and companies. Several websites ex-

[2]McIlroy later went on to invent the pipes and filters of UNIX. Some people regard this as a form of component-based technology, as it allows the "gluing together" of separate programs.

[3]`http://www.apple.com/macosx/`

[4]`http://www.gnustep.org`

ist today with the main purpose of selling software components to potential developers, e.g. Chilkat Software[5], a vendor of .NET components, and ComponentSource[6], a website where vendors offer components to a large audience.

Lately, a new aspect of component-based software development has become evident: given an interface that is simple enough, ready-made components may enable a person with little or no programming experience to assemble his own application without writing any code. This possibility has been suggested by e.g. (Stiemerling et al. 1999) and (Costabile et al. 2003). One of many challenges to developers today is the high level of domain knowledge required to write an application that fulfills the client's needs. Developers tend to look at the application to be developed from their own viewpoint, that of a software developer. Software designers with an adequate understanding of the application domain are hard to find, and a scarce resource (Curtis, Krasner & Iscoe 1988). In the traditional Scandinavian school of software development, it has been attempted to solve this problem through close cooperation between developers and the eventual users of the system in question (Nygaard 1979, Bansler 1989, Ehn 1993). Obviously this requires a substantial amount of effort and time invested by the developers. This raises the costs of development, and in effect limits this methodology to large, mission-critical software projects.

With component-based software development, however, we could be facing a future where proficient users with sufficient technical knowledge could assemble simple applications to further productivity. Such "super users" are usually the first to take up new technologies and convey their understanding of it to their colleagues. The emergence of these expert users and their role in the workplace is discussed in (Kaasbøll & Øgrim 1994) and (Åsand, Mørch & Ludvigsen 2004). As the "end-user developer" has extensive domain knowledge, he is likely to be well equipped to understand the requirements of a new application. With components that are sufficiently generic, it should be possible to construct a wide range of applications for most application areas, using simple and intuitive visual building toolkits. Such a scenario is just one of several presented by researchers today, but it is this hypothesis that will be further explored in this thesis.

Several technologies for component development exist today. However, most of which are concerned with inter-component communication over network connections. The Object Management Group (OMG)[7] is a consortium whose members include most large software houses (and several smaller) active today. OMG was formed with the aim of providing a set

[5] `http://www.chilkatsoft.com`

[6] `http://www.componentsource.com`

[7] `http://www.omg.org`

of specifications for standardized component software. One of the chief results of OMG's work is the Common Object Request Broker Architecture (CORBA) (The Object Management Group 2003) specification for inter-component communication. Competing technologies include Microsoft's Common Object Model (COM) (Microsoft Corporation 2004) and Sun Microsystems' Enterprise JavaBeans (EJB) (Sun Microsystems 2003*a*) and Java Remote Method Invocation (RMI) (Sun Microsystems 2004). These technologies are not necessarily suitable in an end-user development context, though.

In the case of Microsoft products, the fact that they are tied to the Windows platform limits their use. There are several other platforms available to end-users today, and supporting only one operating system should be unnecessary, considering the alternatives. CORBA and EJB/RMI, on the other hand, are based on application servers that provide services to client software over a network connection. These servers can run a variety of operating systems, due to the open specifications of CORBA and the multiplatform nature of of Java. These technologies are fine for large, scalable applications like web shops, internet bank services and the like, but unnecessarily complex for developing small stand-alone applications for day-to-day use by end-users. Sun also provides component technology for stand-alone application development; JavaBeans. These are designed to run on a single computer, with the software components residing on local storage media. JavaBeans are discussed in more detail in chapter 3 on page 23, **Tools and Methods**.

## 2.2 Evolutionary software development

### 2.2.1 History

The stagewise model (Benington 1987) that was introduced in 1956, later refined into the waterfall model (Royce 1987), is generally considered to be the first attempt to describe a life cycle model for software engineering projects. The process was divided into sequential steps, each new step requiring the completion of the previous. Although the waterfall model included a feedback loop between stages in the development, the model proved to be less than optimal for systems requiring interaction with end-users. The reliance on early, elaborate documents describing the application to be developed led to cumbersome and poorly understood user interfaces, and consequentially large amounts of unusable code. When writing software of any size, there is always the possibility - or even probability - that the software will turn out to be something different than what was

planned. Indeed, it is often the case that the customer has only a vague idea of what he really needs, making the software specifications incorrect from the very start. Although Benington, in his 1956 document, stressed that an experimental prototype was required before the software development proper began, and that the system should be incremented step by step, this seems to have been ignored or forgotten by later users of the stagewise life cycle models.

As a solution to the problems of a stagewise process, the notion of an evolutionary approach to software development has been presented by several researchers (McCracken & Jackson 1982, Floyd, Reisin & Schmidt 1989, Dahlbom & Mathiassen 1993). The evolutionary approach to software development is based on a process where the application undergoes several stages of incremental expansion, where the direction of this evolution is determined by experience from use (Fischer 2002). A model for this type of evolutionary development with a high level of user participation is proposed in (Floyd et al. 1989). This model, known as the STEPS[8] model, has been further expanded to accommodate for the introduction of end-user tailoring of software as an additional force in the evolution of a software system (Wulf & Rohde 1995).

The idea of incremental improvements to a program corresponded well with Scandinavian traditions of constant dialogue and cooperation with the intended users of a system. The developer could make an early version of the program, and let the users provide him with feedback and suggestions. New features, and changes to existing ones, could then be incorporated in a later version of the software; this new version could be tested on users again, and so on. This process is sometimes referred to as *cooperative prototyping* (Bødker & Grønbæk 1991). Only through actual use of an artifact is it possible to discover deficiencies and errors, and this in turn leads to a refinement of the artifact through an evolutionary process (Petroski 1992).

The UTOPIA project (Ehn 1993) employed a design method called the *tool perspective*, in which the designers strived to develop computer tools similar to the non-computerized tools already in use by graphic workers in the newspaper industry. This could only be done through close cooperation with the intended users, and extensive use of prototypes was employed to reach the goal of simple, yet effective applications for end-users. The developers used mock-ups of the system they were working on, and let the intended users provide feedback that could be used to refine the design. This approach drew on the philosophical theories of Ludwig Wittgenstein, who in his book *Philosophical Investigations* formulates his idea of the "language game":

[8]STEPS is an acronym for "Software Technology for Evolutionary Participative Systems development.

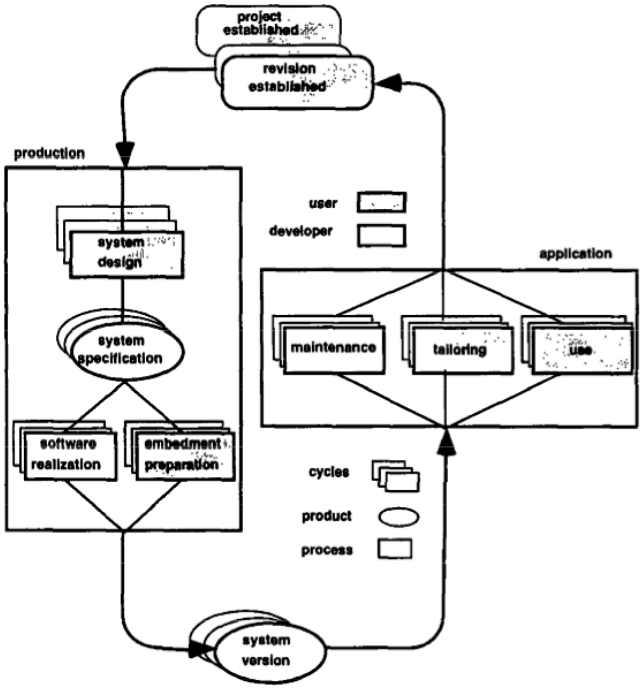


Figure 2.1: The STEPS model, extended to include end-user tailoring, from (Wulf & Rohde 1995).

> Let us imagine a language (...) The language is meant to serve for communication between a builder A and an assistant B. A is building with building-stones; there are blocks, pillars, slabs and beams. B has to pass the stones, and that in the order in which A needs them. For this purpose they use a language consisting of the words 'block', 'pillar', 'slab', 'beam'. A calls them out; — B brings the stone which he has learnt to bring at such-and-such a call. — Conceive of this as a complete primitive language.
>
> (Wittgenstein 1953, Aphorism no. 2)

One of the points Wittgenstein is trying to bring across is that a common understanding of what is being discussed can only be gained through communication. Correspondingly, the idea in the UTOPIA project was that a complete understanding of software requirements could only be gained through an ongoing dialogue between developers and users. A system's life cycle does not end at the time of deployment, though. System enhancement after deployment is needed, as weaknesses in the system become apparent, or the system's context of use changes. Again, the changing environment surrounding a system, and the dynamic nature of its users must be taken into account (Bjerknes, Bratteteig & Espeseth 1991). This extends the participatory nature of Scandinavian design approach into the maintenance and enhancement phase as well.

### 2.2.2 Status

Evolutionary software development in traditional programming has received criticism for its tendency to lead to unstructured and unmaintainable code (Kaasbøll 1997). Constant revisions to the original code base had a tendency to introduce bugs and "spaghetti code"[9]. To support an evolution-oriented way of developing software, a software development model known as the *spiral model* (see figure 2.2 on the facing page) was presented by Barry Boehm in the mid-80s (Boehm 1988). This model employed both specifying and prototyping approaches, in progressive cycles eventually leading to the finished product. Each cycle starts with planning the outcome of this cycle. It then proceeds to evaluation and prototyping, with an emphasis on risk assessment. The next step in development is then performed, with each cycle ending in the planning of the next cycle. This model has been very influential, but its complexity and scope is aimed at internal development project in large corporations. In the context

[9] Spaghetti code is defined as "Code with a complex and tangled control structure, esp. one using many GOTOs, exceptions, or other 'unstructured' branching constructs." (*The Jargon File 4.7.7* 2004)

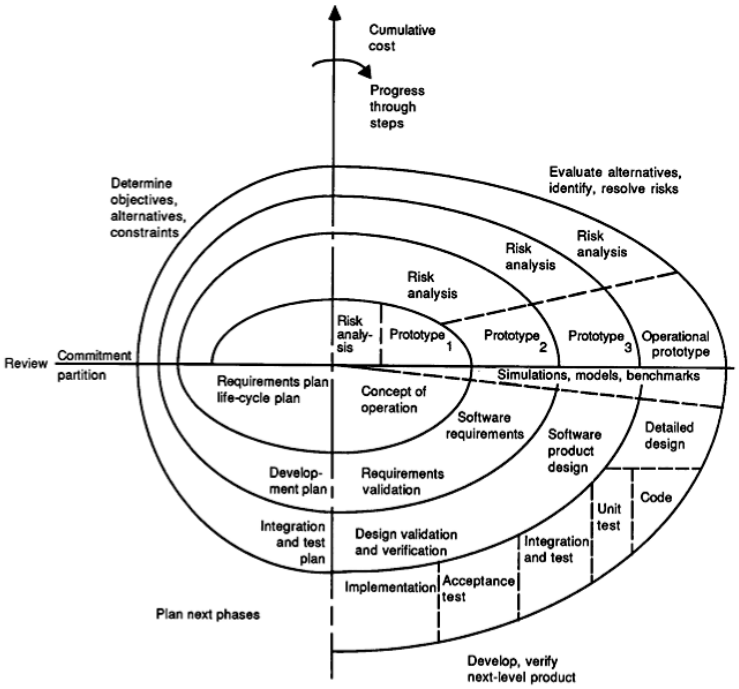


Figure 2.2: The spiral model of software development, from (Boehm 1988).

of smaller applications, and end-user development in particular, its emphasis on planning, design and risk analysis is too complex; requiring a substantial amount of formal training and experience from software engineering.

The use of mock-ups and prototypes as an aid in the development process is discussed in (Budde, Kautz & Kuhlenkamp 1992). Here, we are presented with different approaches to the use of prototyping. While prototypes are often made in the process of application development, there is often a lack of awareness about this fact. Consciously using prototypes to support evolutionary software development can help in several ways, and different kinds of prototypes are presented:

- A *prototype proper* is used to illustrate particular aspects of the user interface or parts of the functionality where uncertainty and ambiguity arises. This kind of prototyping manifests itself as several, smaller prototypes that are meant to highlight a specific portion of the application

- *Breadboards* are built to clarify purely technical issues, and to help the development team experiment with different construction-oriented solutions to the functionality of the application. The use of bread-

board prototypes is frequently seen in traditional software development, but it is seldom recognized as a prototyping strategy.

- Finally, *pilot systems* are prototypes of the application meant to be deployed and tested with users. This kind of prototyping requires a higher degree of completeness than the two previous types. As it is to be tested in the application area, by real users, it also requires at least rudimentary documentation, and some level of robustness. Pilot systems are frequently the basis for the final product, the the line between prototypes and early alpha versions of the product becomes blurred.

Several experiments have been performed to establish the advantages and disadvantages of evolutionary software development models. One of the best known is Barry Boehm's 1982 UCLA experiments, which was repeated at the University of Aalborg in 1990 (Mathiassen, Seewaldt & Stage 1995). The Aalborg experiment suggests that the spiral model is a useful framework for evolutionary software development, but the authors also stress that a mix of prototyping and formal specification is needed, as each of these approaches uncovers new problems that is best solved by the other. The techniques employed in these experiments were performed by IS students with formal training in software engineering methods. As long as this skill level is a requirement for the spiral process, it is beyond the capabilities of end-users or amateur developers. Again, it should be noted that the spiral model is intended for professional use in large software engineering projects.

Acknowledging the complexity of the spiral model, other, simpler models have been proposed - e.g. the Seeding, Evolutionary Growth, Reseeding model (SER) proposed by Gerhard Fischer (Fischer 2002). In this model, we see an iterative process that includes the users as part of the design team. After the seeding of a system, the developers take a back seat as the users focus on suggesting solutions to problems found in the system. Feedback from users is incorporated in the next version of the application, and this new version is reseeded for a new stage of feedback and evolutionary growth. A graphical representation can be seen in figure 2.3 on the next page.

The importance of evolutionary growth is also stressed in (Mørch 2003). Here analogies are drawn to evolutionary mechanisms in other domains, e.g. biology and architecture. This text also discusses techniques for involving end-users as active participants in the evolutionary process of software development, and argues that end-users need will need access to representations that are less formal than implementation code to fully understand a system. Mørch also presents the idea of *resemblance* as a useful way of describing relations between artifacts with a common evolutionary history.

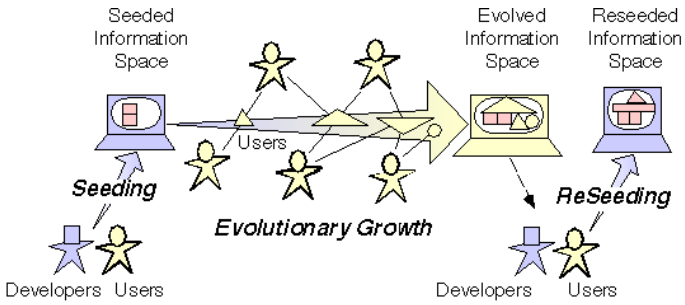


Figure 2.3: Seeding, Evolutionary Growth and Reseeding, from (Fischer 2002).

Making use of resemblance as a development technique could improve user participation, a concept that we will see a prototype implementation of in chapter 4.

## 2.3 Cloning and prototypes

### 2.3.1 History

Traditional object-oriented programming (OOP) is based on the concept of *inheritance*. This view of the object-world is based on a fundamental distinction between the terms *class* and *instance*. A class is a formalized description of all the properties we might find in the item we are trying to describe, and it has the power to generate instances. Instances are the individual member of the set described by the class, and they will need to contain values for the variables defined in the class description. A class that inherits properties from another is called a *subclass*, e.g. a `car` class inheriting properties from its *superclass* `vehicle`. The programming language Simula (Nygaard & Dahl 1978) is considered the first object-oriented language, and as such was the first to introduce this object-view.

Throughout the history of object-oriented programming, another approach has been championed. The idea of *prototypes*[10] implemented by *delegation* is part of the inspiration for the cloning concept presented in this thesis. While traditional OOP is based on the properties defined in the class construct, prototype-based OOP recognizes the observation that it is often difficult in advance to say which characteristics are needed to describe a concept. Ludwig Wittgenstein presents the following example of the problems

[10]Note that the term "prototype" is used in a different sense than in the previous section, and that the two uses must not be confused with eachother

encountered when one tries to describe a concept to a person without any background or earlier experience of it.

> When one shews someone the king in chess and says: "This is the king", this does not tell him the use of this piece-unless he already knows the rules of the game up to this last point: the shape of the king. You could imagine his having learnt the rules of the game without ever having been strewn an actual piece. The shape of the chessman corresponds here to the sound or shape of a word.
>
> (Wittgenstein 1953, Aphorism no. 31).

As we see, it is difficult to say in advance which characteristics are needed to describe a concept. It is easier, then, to deal with a specific example first, and describe later experiences in terms of similarities and differences to this. As new examples are encountered, people are able to make analogies to previous concepts, and, instead on focusing on the "defaults" from this previous concept, they can instead describe in what way a new example differs from earlier experiences. An illustrative example is given in (Lieberman 1986):

Suppose we wish to describe the properties of elephants, based on our encounter with the particular elephant Clyde. Using traditional object orientation, we will probably define the class *elephant*, with properties such as color, number of legs, weight etc. When we later meet the elephant Fred, we can fill in these fields, thus having Clyde and Fred both relate to the variables of the elephant class. Prototype-based object orientation holds a different view. When a person is asked to describe an elephant, he is likely to think of a specific elephant he has seen, either in real life or on television. Thus, we store the description of Clyde as our prototype for an elephant. If we are asked later about the characteristics of Fred, we can assume that the answer will be the same as for Clyde. If it turns out that Fred is a white elephant, we can describe him as "just like Clyde, except that the color is white".

Object creation based on this view of object orientation is based on creating new objects from earlier objects without involving classes, through a *cloning process*. The new object will have all the properties of its prototype, and only discrepancies need to be explicitly stated. This paradigm for object-orientation has been implemented in some experimental programming languages, such as SELF. As a language that wishes to implement this sort of object orientation needs to be dynamically typed (to allow for the fact that the structure of objects is unknown at run time), they are written as scripting languages or interpreted programming languages that run on a virtual machine. This usually implies inferior performance in terms

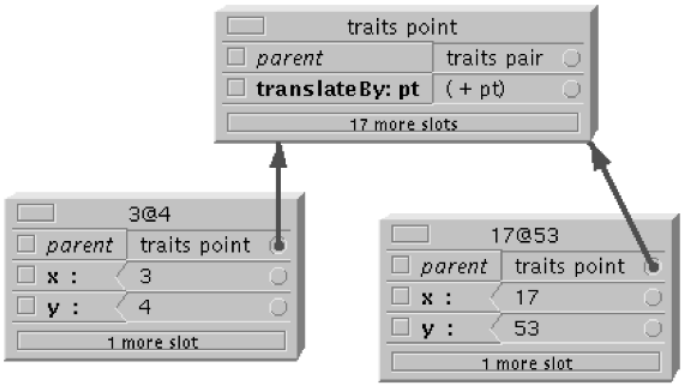


Figure 2.4: Graphical representation of objects in SELF, from (Smith & Ungar 1995).

of execution speed and memory footprint compared to conventional, compiled programs. Prototype-based object orientation has mostly appeared in academic and research languages, and the performance issues may very well be one of the reasons for this.

### 2.3.2 Status

Perhaps the best known example of classless object-oriented languages is SELF (Ungar & Smith 1987, Smith & Ungar 1995), conceived by David Ungar and Randall Smith during their time at the Xerox Palo Alto Research Center (PARC). SELF was further developed by Ungar at Stanford University, and a public release took place in 1990. Ungar was hired by Sun Microsystems Laboratories, where he led the SELF group from 1991 to 1995. SELF had features that provided implementation challenges, like the need to perform background compilation of new objects, and the need for an efficient virtual machine on which the programs could run. Most of the "HotSpot performance engine" used in the Java Virtual Machine is based on the techniques employed by Ungar for SELF (Smith & Ungar 2001). SELF is no longer an official project at Sun Microsystems Laboratories, but it has been developed on and off over the last years anyway. The latest version is 4.2.1, released for Macintosh and SPARC Solaris systems on April 16th, 2004.

SELF also employs a graphical interface for interaction with the objects, called *Seity* (figure 2.4). The "traits" term, seen in the name of the parent object in figure 2.4, is used to describe an object that is intended to be

cloned from, but not be put to use by itself. This may seem similar to a traditional class description, but traits objects to not contain implementation or instantiation details for its inheritors. As such, it can be seen as an equivalent of the *abstract classes* of traditional object-orientation. In SELF, the user may freely add "slots" to objects by direct manipulation of the data objects, where each slot contains another object. This other object may also be a method or a variable (in reality, every entity in SELF is an object!). To allow for this extreme mutability of the programs, SELF does not produce executable files as a result of compilation. Rather, the running system is stored as a snapshot, to be reloaded into the virtual machine the next time one wishes to start the program. The syntax of SELF is based on that of Smalltalk, but is simplified and adapted to the prototypical nature of the object orientation mechanisms of the language.

There is still a vocal minority of programming language developers, mostly based in the California area (e.g. at Sun Microsystems and Apple Computers), that continue to develop the prototype-delegation paradigm. Languages inspired by SELF include Cecil, Cel, Merlin and several other, unknown languages[11]. Additionally, there are other languages that are based on other syntaxes, but most owe the majority of their design to Smalltalk or SELF. Perhaps most notable of these is NewtonScript (Apple Computer, Inc. 1993), the scripting language used by Apple for their Newton PDA - the first PDA to be mass produced - in the early 90s. Drawing on experiences from the HyperCard application[12] - the flexibility of HyperCard also mentioned in (Mørch 1997) - the Newton team decided that a prototype-based object model was simpler and more intuitive for creating graphical interfaces with *container inheritance*. Container inheritance describes a relationship between GUI elements where unhandled messages sent to an object is passed further up the hierarchy. In HyperCard, for example, if a message is sent to a button, the system will look for a handler in the button, then in its card, then in the card's background and finally in the stack (see figure 2.6 on the next page). HyperCard also has features for cloning GUI elements. This is simply done by holding down a specific key, while dragging a clone from an element using the mouse. An example is shown in figure 2.5 on the facing page.

A fundamental concept in prototyping programming languages is the way

[11] For a non-exhaustive list of prototype-based programming languages, see `http://www.dekorte.com/Proto/Chart.html`.

[12] HyperCard was a Macintosh program for easy creation of databases, graphical user interfaces (GUIs) and prototypes for other applications. It was based on a concept of a "stack of cards" in which the developer could place GUI elements like text boxes, buttons etc. HyperCard included hypertext facilities from the first version, and has inspired several graphical development environments - including reimplementations for other platforms, such as SuperCard (`http://www.supercard.us`).

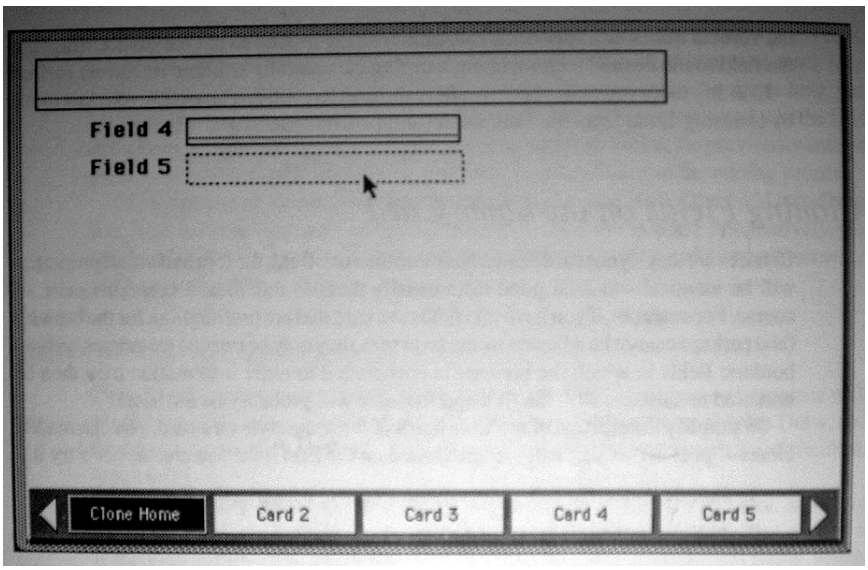


Figure 2.5: Cloning a text field in HyperCard.

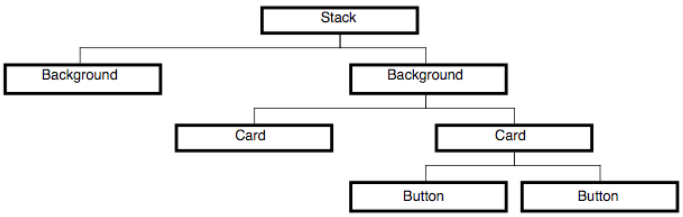


Figure 2.6: The HyperCard container hierarchy.

an object passes unhandled messages to its parent, or prototype object. NewtonScript's strengths in the area of GUI development has inspired prototype-based languages like Dialect[13]. Other projects inspired by NewtonScript include the Io programming language and Lua - an extension language for C.

## 2.4 Fitting the pieces together

Programming tools utilizing visual components seem to be the best way to approach end-user development (Chang, Ungar & Smith 1995, Stiemerling et al. 1999, Costabile et al. 2003). Direct manipulation of visual objects on

[13] `http://dialect.sourceforge.net/`

the screen is likely to simplify object-oriented programming and give the user a feeling of direct interaction with the pieces that make up an application. The open question is: What should be an object on the screen? Although in reality an on-screen component may consist of several objects on a source code level, it is argued in (Chang et al. 1995) that the programmer should be "tricked" into thinking that the representations on the computer screen are the actual objects he is working with. Domain modeling, the process of mapping programming language classes to real-world objects and concepts (Fowler & Scott 2000), is an acknowledged technique when developing information systems. In the context of end-users designing graphical interfaces and simple applications, however, it becomes awkward to use. Analyzing a GUI on the class level can quickly become counter-productive, as components usually regarded as a single entity are likely to consist of multiple classes on the API level (e.g. scrollable lists, that will at least consist of a scrollable view, the actual list, and a data type containing the actual values). Relating to GUI components is likely to be a more appropriate level of granularity for end-user developers.

Adding cloning functionality similar to SELF, but for GUI components, will further increase the flexibility of the builder application. A textual representation of GUI elements such as buttons and text boxes adds a level of abstraction to the development process, and users should be spared the burden of learning the structured and unfamiliar syntax of computer languages required to create a visual computer application. For example, consider the contents of listing 2.1 on the next page, required to produce the window in figure 2.7 on page 22. If I wanted to add any kind of functionality to the button in the example, several lines of code would have to be written, including the use of additional classes to handle events from the mouse and keyboard of the user. Furthermore, a large majority of computer systems in use today are based on the classic windows-icons-menus-pointing device (WIMP) paradigm for graphical interfaces. Since this is the setting in which end-users perform their daily work, this must surely be the most natural environment for end-users to customize their applications (Costabile et al. 2003). A hypothesis we address in this thesis is therefore the following: Visual components that are easily recognizable from previous experience with other applications will lower the threshold for learning end-user development.

Based on Petroski's theory that evolution of artifacts is done by correcting the faults of existing artifacts (Petroski 1992), such a toolkit should include the option to modify the available components to suit the needs of end-user developers. This way, the application is not only open for further development, but the components that make up the environment can be modified and improved as well. This is particularly important in situations where the set of available components do not support the functionality that the

```
import javax.swing.*;
import java.awt.*;

public class HelloWorldSwing {
  private static void createAndShowGUI() {
    JFrame.setDefaultLookAndFeelDecorated(true);

    JFrame frame = new JFrame("HelloWorldSwing");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.getContentPane().setLayout(new GridLayout(3,1));

    JLabel label = new JLabel("Hello␣World");
    frame.getContentPane().add(label);

    JButton button = new JButton("Useless␣button");
    frame.getContentPane().add(button);

    JTextField field = new JTextField("Enter␣anything");
    frame.getContentPane().add(field);

    frame.pack();
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createAndShowGUI();
      }
    });
  }
}
```

Listing 2.1: Example code for creating a window with a simple GUI with Java.

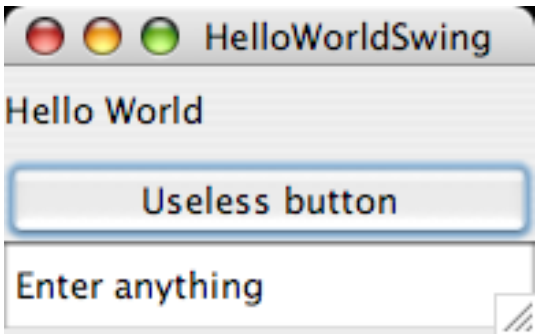


Figure 2.7: The results of listing 2.1 on the page before.

end-user wants to achieve. Modifiable components, presented in this text as a solution to the problem of limiting components, is not the only way to deal with this problem. Another solution could be to simply offer such a wide array of components that the end-user never will experience the problem of inadequacy in the available set of components. This will of course introduce another problem, that of efficient cataloguing of components, a concern that was presented along with the idea of software components were introduced (McIlroy 1968).

The notion of evolution and change within the context of a single component is also consistent with Brad Cox's view that software components should be easily replaceable (Cox 1986). The user is likely to make an early prototype of his application at a very early stage. By never actually "freezing" the application, he should be able to continually add functionality and refine existing features at later stages, following the ideas of evolutionary software development.

It is claimed in (Rieman 1996) that users often learn how to use new tools by exploration and experimentations, trial and error. However, this presents the risk of users involuntarily causing errors that could be hard to recover from. Providing malleable components from which the end-user developer can make clones is a goal with my component-based toolkit. *This way, the developer can safely experiment with a cloned component without damaging the original, as long as the original can at any point resume operation* if the clone turns out to be malfunctioning. When a new component has the desired properties, it will be available for later use from the palette of components that component assemblage tools usually provide. This visual equivalent to "programming by example" lets the user examine a relevant, but not quite appropriate component, and experiment with different parameters to fine tune it. After seeing the effects of changes made to the component, the user can proceed to make more drastic changes, thus learning more about the technical details of the toolkit in small increments.

# Chapter 3

# Tools and methods

> I can't understand a word you say. And you're poorly dressed. You must be some sort of technology expert. Or a rodeo clown.
>
> — *Dilbert's Pointy Haired Boss (from the comic strip "Dilbert" by Scott Adams)*

This chapter describes the tools used for the thesis, as well as the rationale for choosing them. I will give a walkthrough of the technologies employed in the SimpleBuilder project, including BeanBuilder; the basis for SimpleBuilder. A discussion on the work needed to change the BeanBuilder into the tool we were looking for is provided in 3.3.2 on page 34. I will also describe how I performed the development and research needed for the thesis.

## 3.1 Developer tools

Although I set out to develop a framework for application development, I naturally needed to choose an environment for developing the application framework itself. I also had to choose a programming language for implementing the toolkit. Due to the large number of source code files that would make up the final application, I decided that an Integrated Development Environment (IDE) was needed. An IDE combines management of source code with an editor, usually supporting standard features as symbol completion, syntax highlighting and coloration of the code. It should also include a compiler and a debugger, essentially simplifying the programming process to editing of source code, and pressing a button to perform compilation and packaging of the program.

### 3.1.1 The Java programming language

I had already decided to use the Java programming language, version 1.4, as the implementation language for this project. The two main reasons for this were its multiplatform nature, and the fact that it is the programming language I am most familiar with. Java is an object-oriented programming language developed by James Gosling and his fellow engineers at Sun Microsystems. Work on Java started in 1991, and the first public version was announced in 1995. It was designed to be a platform-independent language with syntax similar to C++ (Gosling, Steele & Joy 2000).

As the inventors wanted a simpler and more consistent language than C++, they decided not to include some of the more advanced features of C++, such as templates or pointer arithmetics. another C++ feature deliberately left out of Java is that of multiple inheritance. Multiple inheritance lets a class inherit from two or more superclasses. This was left out of Java because it can lead to problems, for example when the compiler has to decide which of two conflicting method implementations to choose from. Java's solution to this problem is the use of *interfaces*. An interface is similar to a class definition, except that it does not allow any implementation code for included functions. This means that it's entirely up to the implementing class how to actually implement a function. The result of this is that when a programmer writes code utilizing these implementor classes, he can safely assume that a function is supported, while ignoring the details of its implementation.

Java also draws upon other OO languages like Smalltalk, and one of Java's chief features is the built-in garbage collection. To provide platform independence, Java also contains the Java Virtual Machine (JVM). This means that programs are compiled into bytecode, which in return runs on the JVM. Any computer platform that has a working JVM can run Java applications. Java is also supplied with a large library of standard classes. Many of these are concerned with providing a unified interface for a supported platform's GUI features, essentially making Java programs platform independent. Other classes implement text processing tools such as regular expressions, or abstract data types such as lists and hash tables.

### 3.1.2 Operating system and developer environment

For most of my daily work, my preferred computer platform is the Apple Macintosh Operating System, version 10.3 (Mac OS X). Mac OS X is based on the operating system FreeBSD, which in turn is based on UNIX. This provides a stable, standardized platform and a host of open source development aids. As Mac OS X comes with a powerful IDE and a high level of

Java integration with the operating environment, this platform seemed like the obvious choice for my development work. Apple's free development environment *XCode* provides easy organization of source code files, as well as an editor, source code management and version control[1], one-click compilation and packaging, and an online help system and API specification for Java[2]. With the development environment ready, I needed to search for a suitable starting point for my project. Writing a component-based development environment from scratch was well beyond the scope of a 1-year research project. I started looking for such an environment, which needed to have available source code, and be both component-based and written in Java. Luckily, Sun has developed their own technology for component-based development with Java; JavaBeans.

## 3.2 Component technology

For a piece of software to qualify as a *software component*, it has to be both a *unit of deployment* and a *unit of versioning and replacement* (Szyperski 1998). This means that a software component must be delivered in compiled form, runnable on a real or virtual machine. Deploying a software component should not require neither the installation of programming tools, nor the presence of a professional developer. The "versioning and replacement" part means that the software components should be easily replaceable, and be packaged as a bundle that can be swapped for another version without affecting other parts of a system. This also means that software component must be considered at the level of packages or modules, i.e. a set of classes rather than an object. A software component can consist of several classes, or just one. It will, however, usually be supplied with a descriptor of some kind, usually an auxiliary class which defines the component's interface. When the internal structure of the component is hidden in this way, exposing only the component's connection points, it is usually referred to as a *black box* component (Szyperski 1998). Alternative ways of component encapsulations are *white box*, where all the internals of the component are visible and modifiable, and *glass box*, where the code is visible, but not open for modification.

[1] Version control is offered through integration with the Concurrent Versions System, CVS. CVS allows storage of source code in a central repository, and has features for versioning and branching of source code trees. This also means that reverting to earlier versions in the case of newly introduced bugs is possible. It also has mechanisms for resolving conflicting changes to source code performed by multiple developers, but this was not needed for the work described here.

[2] For more information on XCode, see `http://www.apple.com/macosx/features/xcode/`.

### 3.2.1 Component-based vs. traditional programming

By letting the developer see the objects right in front of him while creating an application, visual software components such as JavaBeans let us bypass the extra level of indirection that the source code of conventional programming represents. With component assembly, there is no real distinction between the designed object and the runtime object. While working in a component assembly tool, the designed application is displayed in its own window, and what you see in this window represents the final application. When switching from the "design runtime" (the component assembly tool) and the "user runtime" (the running application) the components will stay where they were placed, and look the same as at they did in the design view of the application.

When a programmer is designing the individual components, significant care and planning is required. To ensure that the components are practically reusable, they need to be - among other things - robust, properly documented, and thoroughly tested. However, once a component is complete and working it should significantly simplify the development process for those that put it to use.

While object-oriented programming can be considered prerequisite for software components, object-orientation and component-orientation are not the same. Object-orientation focuses on programming according to a model of real or imagined entities in a system. Component-orientation focuses on the assumption that programs should be built by gluing together prefabricated components, and for the person doing component assembly the traditional distinction between class and object is no longer of interest: the component is the unit he relates to. In the special case of components made for visual builder tools, the intention is to reduce the amount of code written by a developer to an absolute minimum. In contrast, writing a program using conventional programming may include typing in thousands and thousands of lines of text in a structured, but often hard to grasp, language. Finished software components will still be developed in this traditional way, to furnish component libraries. As a means to simplify and encourage end-user programming, component-based development seems to be an excellent tool. While end-users are typically domain experts, they can not be expected to be familiar with conventional programming languages. They will, however, usually have experience in filling out preference forms and other simple customization of programs (Mørch & Mehandjiev 2000).

### 3.2.2 JavaBeans

JavaBeans were introduced with the 1.1 version of Sun's Java programming language. JavaBeans is not a specific product; rather, it is a set of specifications that a Java class needs to satisfy to be treated as a Bean (Sun Microsystems 2003*b*). JavaBeans are Sun's solution to the problem of making "software integrated circuits" (Cox 1986, Szyperski 1998) - software components with a clearly defined interface, which can later be assembled into complete applications. JavaBeans were introduced as a platform-independent solution to allow extensive reuse of self-contained software components (Sun Microsystems 2003*b*) with clearly defined public interfaces. JavaBeans require that all components are available in compiled form on the same host, and as such are suitable for stand-alone or client-side applications. They are specifically designed to be used with builder applications[3]. These application normally use the concept of a *palette* for holding the components, in the same way that an art painter's palette contains the colors used to create a painting. Most JavaBeans will be supplied with an icon to be used for representing the components in the palette of such a tool. JavaBeans will usually also include a BeanInfo class that can tell builder applications about the properties of the JavaBean, such as accessor methods, variables, a short name for use in menus etc, as well as information about the icon to be used for the palette.

Inspired by Sun's work with the Object Management Group and their Common Object Request Broker Architecture (CORBA, see 2.1.2 on page 8), Enterprise JavaBeans (EJB) were presented as the solution for server-side applications (Sun Microsystems 2003*a*). Enterprise JavaBeans are components normally running on an application server to provide services offered by underlying middleware systems; such as transactions, persistence and security. Enterprise JavaBeans are also interoperable with CORBA components, although more application server implementations for EJB exist than for CORBA.

A challenge presented by the component-based architecture of JavaBeans is that of the components' *persistence*. Persistence is a term used for the way components are saved to disk when the application is not running or being edited. After having done the necessary customization of the objects, the developer will want to store these changes. As objects and their state are normally associated with the run-time state of an application, JavaBeans has previously utilized the concept of serializable objects; that the objects in memory can be written to disk for later use and/or further modification

[3]The term "builder application" is used to describe a tool for assembling an application from pre-built components. It is not to be confused with the conventional developer tools used for creating the components in the first place.

(Gosling et al. 2000). Earlier, this was performed by simply writing the object data in memory to a file with the .ser extension. A problem with this was that the information about the components' relations to each other was difficult to preserve.

With version 1.4 of Java, a new interface for storing objects along with the graphs of interconnected JavaBeans, is provided by a system of XML-based Long Term Persistence (LTP) (Java Community Process 2003). LTP also stores the customizations done to the components that make up the application being developed. The mechanism essentially works by storing object names along with the method calls and corresponding parameters required to restore the object to its state at the time of serialization. Storing and restoring graphs of JavaBeans is done with only a few lines of code (although the resulting XML can be rather verbose). The Java code of listing 3.1 on page 30 demonstrates the basic principle of this mechanism. The results of storing a text field as XML can be seen in listing 3.2 on page 32. If the file had described multiple components, connections to other components would have been expressed as a sequence of set/get methods (see the appendix for an example). When such a file is reconstructed, it will result in a set of components and event listener object with the same relations they had at the time of saving to disk.

JavaBeans can be assembled into applications by using one of several available visual builder tools. JavaBeans can both be visual (buttons, listboxes etc.) and non-visual (e.g. components for manipulating and/or storing data). Non-visual components are normally represented by a visual proxy object to allow for manipulation and connection to other components, e.g. the DefaultListModel in figure 3.1 on the facing page. Of course, these proxy objects are invisible when running the assembled application. The developer must also connect the components to each other to specify the interaction between them. Wizards that query the object for available methods aid this work, and the developer can actually build an entire application without writing a single line of Java code.

## 3.3 The BeanBuilder

The intention with my thesis is to provide a functional toolkit that allows end users to develop their own applications without prior programming experience, but by reusing, customizing and integrating existing applications and application components. We hope to provide a toolkit that enables the user to employ concepts he is already familiar with from everyday work activities, so that end-users may develop an application that takes advantage of their domain expertise. To achieve this, there are a number of

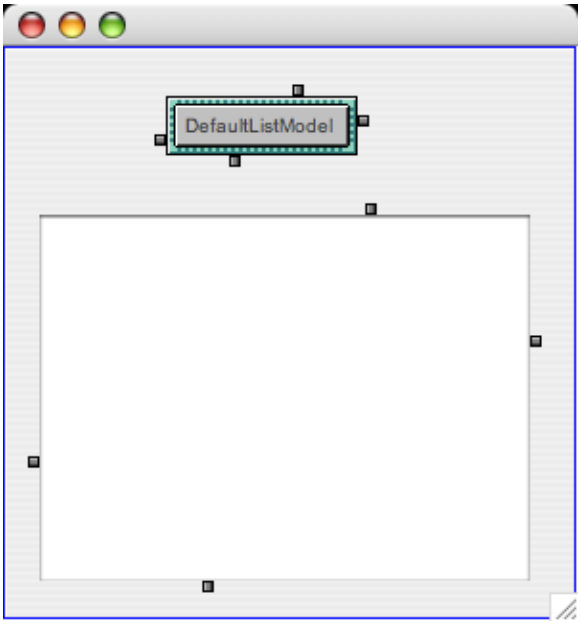


Figure 3.1: The DefaultListModel is a wrapper for the Vector class, i.e. a general storage class with functionality for event listener notification. This allows the user to add interaction to the list model in the same way as for visual components, e.g. for putting values typed in a text field into the vector represented by the proxy object.

```
/**
* @param fileName       the name of the XML file
* @param objectToWrite root container of the application
*/
void WriteFile(String fileName, Object objectToWrite){
  XMLEncoder e = new XMLEncoder(
    new BufferedOutputStream(
      new FileOutputStream(fileName)));
  e.writeObject(objectToWrite);
  e.close();
}

/**
* @param fileName the name of the XML file
*/
void ReadFile(String fileName) throws IOException {
  XMLDecoder d = new XMLDecoder(
    new BufferedInputStream(
      new FileInputStream(fileName)));
  System.out.println("Read object: " + d.readObject());
  d.close();
}
```

Listing 3.1: Example code for storing and restoring a JavaBean application from XML

requirements that need to be met by the toolkit.

### 3.3.1 Features

The BeanBuilder was built by Mark Davidson, software engineer at Sun Microsystems, as a simple demonstration of the new features of JavaBeans as of Java version 1.4. As such, it makes no assumptions to be a production-quality tool for application development. Still, it has a number of features that are interesting to us:

- **Long Time Persistence**
  The BeanBuilder utilizes the LTP scheme of Java 1.4 to store designed applications as references to components and their connections to each other (often referred to as *wiring*, an analogy to electronics) in an XML file. This also means that the BeanBuilder can be run in "interpret mode". This means that an XML file can be given as a command-line argument to the BeanBuilder, to run an assembled application without starting the builder environment. In this mode, the Builder simply reads the XML file, reconstructs the application, and

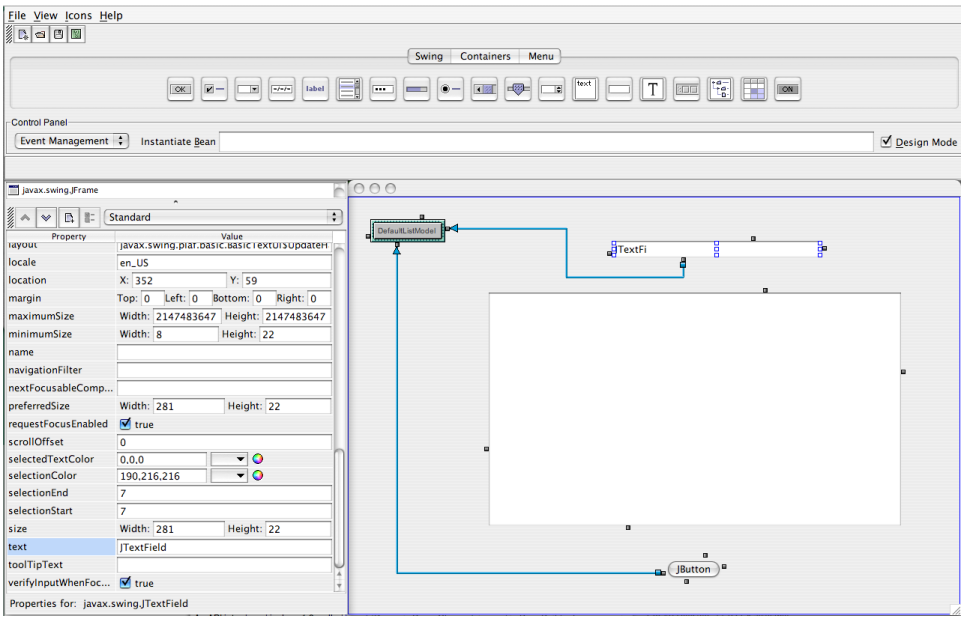


Figure 3.2: A screenshot of the Sun BeanBuilder in use. The palette with icons for different components can be seen in the top section of the shot, with the property sheet and the designed application in the lower left and right part, respectively. A larger picture of the similar SimpleBuilder can be found in figure 4.5 on page 48.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_03" class="java.beans.XMLDecoder">
 <object class="javax.swing.JTextField">
  <void property="preferredSize">
   <object class="java.awt.Dimension">
    <int>168</int>
    <int>63</int>
   </object>
  </void>
  <void property="bounds">
   <object class="java.awt.Rectangle">
    <int>62</int>
    <int>93</int>
    <int>168</int>
    <int>63</int>
   </object>
  </void>
  <void property="text">
   <string>XML example</string>
  </void>
 </object>
</java>
```

Listing 3.2: A JTextField component stored as an XML file, with its text set to "XML example". The integer values describe the component's size in pixels.

hands over control to it.

- **Easy wiring of components**
  An API introduced in Java 1.3, called *Dynamic Proxy Classes*, is used in BeanBuilder for connecting components. JavaBeans communicate through events. The BeanBuilder generates proxy objects for event listening at design time; objects that implement listener interfaces specified when connecting components. If a proxy object later receives such an event, it can trigger a user-specified method in a JavaBean. In practice, this means that no "glue code" is necessary to connect components, as objects that would have required specific event listeners before now can communicate through dynamically generated proxy objects. To create these proxy objects, the user draws a line between two components, and the BeanBuilder will present a wizard for connecting the components. When the user has selected a trigger event for inter-component communication, and the methods to be invoked at the source and target object, the proxy object is generated based on this information. This object will be stored along with all the other components when the user saves the application to

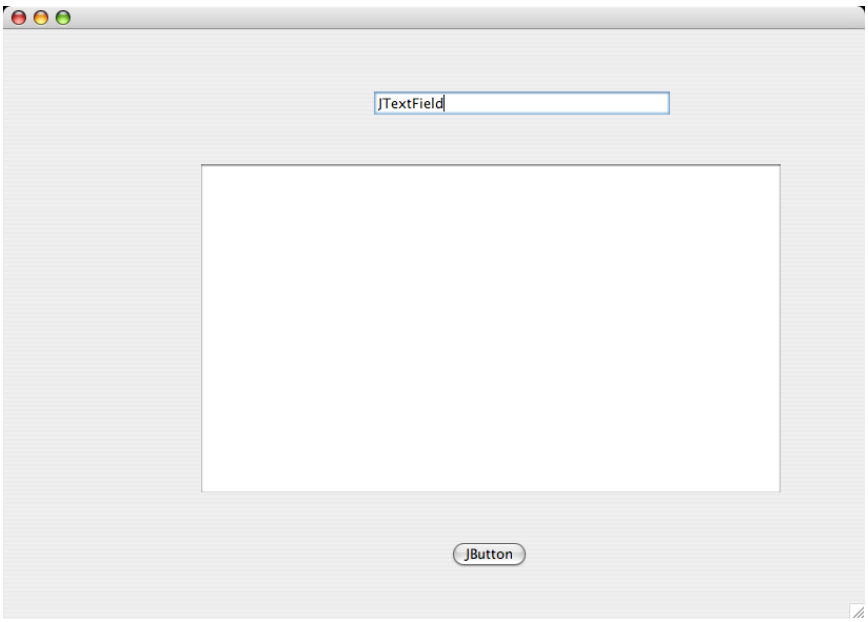


Figure 3.3: The application from figure 3.2 while running. Note that the proxy object, as well as the resizing handles and event arcs are no longer visible

disk. The advantages of avoiding any programming is that the user never leaves the purely graphical environment of component assembly, and that he does not have to learn a programming language to create simple applications.

- **Flexible layout management**
  The BeanBuilder also uses the new `SpringLayoutManager` from Java 1.4. This is a layout manager specifically designed for use in visual builder tools, to allow for flexible behavior when resizing windows of assembled applications. The layout manager works by "attaching" the edges of components to each other, so that for example a text field whose edges are attached to that of the containing window will be resized accordingly when the window changes shape. Traditionally, layout management in Java has required programming skills, and the ability to imagine the graphical results of a textual representation, in the form of program code.

- **Simple and small enough**
  The source code of the BeanBuilder consists of about 12,000 lines of Java code. While this may sound imposing, this is a small enough number for one developer to get an adequate grasp of the logic and structure of the application. The code is distributed in a logical direc-

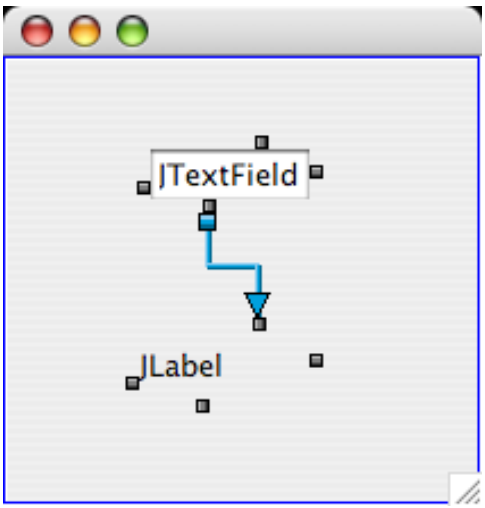


Figure 3.4: Component wiring in the BeanBuilder. Text entered into the JTextField component will show up in the JLabel component. This relation is represented by an arrow. When text is entered, an ActionPerformed event will be fired, and the proxy event listener will invoke the JLabel's setText() method with the JTextField's getText() as the argument.

tory structure based on the functionality of defined classes, making it simpler to find the appropriate file for a given functionality. In addition, the code is for the most part well commented, including tags for the JavaDoc tool[4].

- **Available source code**
  Perhaps the most important issue for us; the full source code of BeanBuilder is available from Sun's website. It is released under a BSD-style[5] licence, which essentially means that developers may do whatever they wish with the code, except claim that they wrote it.

### 3.3.2 Required extensions

Following the features mentioned in the problem description ( 1.3 on page 4), it is highly desirable from an end-user development point of view that we

[4]JavaDoc is a tool for extracting documentation from source code. Every class and method can be commented with tags that when run through the JavaDoc tool will produce meaningful documentation in HTML format. See listing 3.1 on page 30 for an example of the @params keyword in use.

[5]The term "BSD-style" comes from the license accompanying the Berkeley Software Distribution of the UNIX operating system. This was the first use of this particular wording in an open source license. Similar license are used for a host of software projects today, many of which have no relation to the BSD operating systems.

end up with a toolkit that is able to save the state of an application, including both the objects used and the relations between them. The user must also be able to (re)start an application under development with a simple double-click at any time after finishing a stage in the development process. However, it should also be possible to open the application with an editor to perform additional work, potential bug fixes, or changes needed to meet new requirements in the problem domain. In other words, the application must be extensible without requiring the user to wander too far from the use situation.

While developing the application, the user should be able to easily switch between layout editing of an application (analogous to placing the components of a Hi-Fi set, adjusting the position of speakers etc.), and the event-management part of the development (connecting the wires between tape decks and amplifiers, turning knobs and connecting different outputs and inputs). This needs to be a simple and intuitive process, and the user should be aware of which mode he is in at any time.

The user should also be able to easily switch between building mode and use mode of the application at any time. This is useful while developing the application, as the user is likely to want to quickly see the changes done to the application, and how it works out in practice. This can be seen as analogous to the test/debug/test cycle of conventional programming. By making this switch fast and cognitively simple, the developer may also request comments and suggestions from his co-workers, and even let them perform rearrangement and alterations to the GUI!

### Required changes in functionality

The most important change required in the BeanBuilder toolkit is the addition of *cloneable components*. This needs to be done in such a way that the original component is preserved, and the new component is the object of further modification and extension. This level of malleability in components means that we must provide features to save the state of the user's carefully configured components. This should also imply that we present the user with functionality for editing and extending components. It is likely that such a feature will be used mainly by end user developers with some experience and confidence in working with the toolkit (hopefully most users will reach this level eventually), and at this stage it should be safe to introduce the developer to simple Java code. In fact, changing the internals of a component without writing code is likely to be far beyond the scope of this project. A possible solution is adding an "edit" option to the components, which presents the user with a simple view of the Java code for the core functionality of the component. After the changes have been

committed, the new Java class could be compiled in the background. This is essentially what goes on in the background when a SELF object is changed (see 2.3.2 on page 17 for more on SELF).

It is necessary to separate the editing of cloned components into to categories: *Customization* is the act of setting default values like labels, colors, and other properties publicly available in the component. *Extension*, on the other hand, is the act of changing the nature of the component; adding functionality and behavior, offering new properties for editing and other changes that require a source code level redesign of the component (Mørch 1997). I consider both to be equally important to provide true user-editable components, but extension requires an easy interface for changing the Java code of components. More on this in chapter 4 on page 40.

While the BeanBuilder certainly makes life easier for programmers with Java experience by providing wizards for establishing relations between components, this wizard is much too confusing for novice developers. The wizard will present the user with all methods exposed by the source and target components, most of which will be irrelevant when connecting components. Knowing which methods to choose essentially requires knowledge of the Java API specification, which is clearly beyond what should be expected from end-users. One potential solution to this problem is supplying only components that have a small enough choice of public methods that it will be apparent how to connect components using the wizard. For this initial version of the SimpleBuilder this is the chosen approach. The example components supplied with the SimpleBuilder (called "SimpleBeans") only offer one pair of set/get methods: `setValueSBO()` and `getValueSBO()`[6]. Another approach is bypassing the entire system of using wizards for establishing connections, and instead providing components with a set of different types of "connectors", similar to the different types (RCA Audio, jack, antenna and SCART, for example) of connectors found on Hi-Fi equipment. This may potentially limit the versatility of components, but with sufficient planning I believe this to be a viable solution. It will then be evident how to connect the components, as a getString() "connector" of one component will only fit a setString() "connector" on another. This is the approach chosen in the EVOLVE tailoring platform (Stiemerling et al. 1999).

[6]The `SBO` part of these method names is short for "SimpleBeanObject", a simple wrapper object that can contain an object of any class, along with a field describing its type. More on this in 4.4 on page 54.

#### Changes in GUI

While working with component assembly, it is essential that the user is able to modify certain properties of the components. These could be attributes like position, size, the text on buttons, etc. However, the huge property sheet presented in the BeanBuilder is likely to be confusing and difficult to get an overview of. Even in "simple mode" the number of options is quite daunting to a novice user. Based on experience, presenting a user with too many options may even tempt the user to spend too much time being "creative" with fonts and coloration, and too little time on making an efficient and pleasant-to-use application. One possible solution to this is providing an "even simpler mode", which provides only the simplest of options to the user.

Another part of BeanBuilder that could benefit from redesign for the intended audience of our project is the switching between layout mode and event management mode. Today, this is done by choosing the mode from a listbox, and the current mode is indicated by the coloration of the connections between components. (When in layout mode, connections between components are used to "anchor" them to other elements of the GUI, to better support resizing of the application window. See the point on `SpringLayoutManager` in 3.3.1 on page 33) If we choose to provide each component with several connectors whose appearance depend upon the type of data passed through the connector, a possible solution could be to only display these connectors when in event management mode. In essence, the event management mode presents the user with the "wiring" of the application. This could be seen as analogous to peeking behind a Hi-Fi set to connect wires between the connectors at the back of components, while only relating to the buttons and dials in front of the components the rest of the time.

## 3.4 Development strategy

As this project was based around an existing application, I was faced with the problem of understanding a fairly large amount of source code written by a person that I could not get personal assistance from. Because of this, a large amount of time went into trying to grasp the structure and logic of the source tree accompanying the BeanBuilder distribution. Understanding code written by others is proven to be a time-consuming task (Paul, Prakash, Buss & Henshaw 1991), and because of the limited time available to me I had to start development at an early stage. As there were large uncertainties around what was technically feasible, and indeed about my

own skills as a programmer, I opted for a development model similar to the spiral model of Barry Boehm (Boehm 1988). As a functioning application was the starting point for my development, an evolutionary process with incrementally added functionality seemed the only option.

My first goal was to get the application to compile and run in the developer environment I had chosen. I then proceeded to setting up a tentative list of features I wanted to add, along with personal notes on how I hoped to be able to implement them. Every new feature I decided to add corresponded to a cycle in the spiral model (figure 2.2 on page 13). I started each stage with studying the appropriate part of the BeanBuilder source code, and searching the Java API documentation for functionality that might aid me. I then wrote a short, informal list of keywords that outlined the plan for implementing the relevant piece of functionality. I then proceeded to program the new functionality, followed by testing. This seemed to be the only viable way, as I was dealing with unfamiliar technology and code written by another person. The process model for my development can be seen in figure 3.5 on the facing page.

Every time I felt that I had added a useful and satisfactory feature, I committed the changes to CVS (see 3.1.2 on page 25). This way, if it turned out that I had done irreparable damage to the application, I could easily revert to an earlier version and discard the changes. As I learnt more about the features offered by the Java 1.4 class library and the mechanics of the BeanBuilder, I was able to more confidently assess every new piece of functionality. In effect I was learning about the features of the JavaBeans API and Java's mechanisms for reflection on a just-in-time basis. I often had to stop programming to scan the API documentation for information relevant to what I wanted to implement, as shown by the inner loop in figure 3.5. Luckily I had the Java API documentation at hand all the time, so that I could easily look up any class description or method I was unfamiliar with.

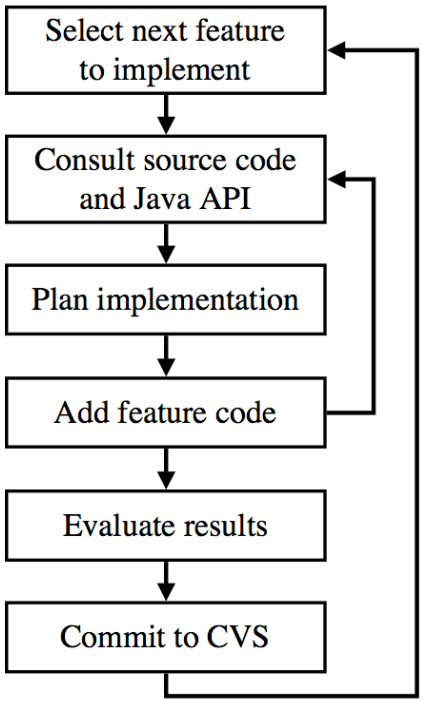


Figure 3.5: The life cycle model of the SimpleBuilder development process. This process was preceded by careful consideration of which features that were required, based on my theories on what the end result should look like.

# Chapter 4

# My contribution

> If I have seen farther than others, it is because I was standing on the shoulders of giants.
>
> — *Sir Isaac Newton*

In this chapter, I will describe my experiences from working with JavaBeans and the BeanBuilder, and what I feel I have achieved in the process. Finally, I will describe the SimpleBuilder, a modified version of Sun's BeanBuilder, and the SimpleBean framework for JavaBean components. But first of all, to help illustrate how the techniques described in this thesis could be put to practical use, I will start off by presenting an example scenario for end-user development.

## 4.1 A scenario for prototype-based EUD

To a certain degree, the discussions of this thesis have focused on the components at a microscopic level. To put the concept of cloning in component-based application builders in perspective, I will now present a future workplace scenario for this sort of end-user development.

Imagine Tom, a secretary of a small company. One of his responsibilities is trying to keep track of the business relations of the company. Tom is reasonably computer literate, and fascinated with the possibilities of end-user development. Using a tool similar to the SimpleBuilder, he has assembled a small application that connects to the company database of customers (figure 4.1 on the facing page). When building the application, Tom wanted a uniform size for the text fields used for displaying customer information. After he got the first text field exactly the right size, he could then clone it. By having a cloned configuration of the text field available in the builder

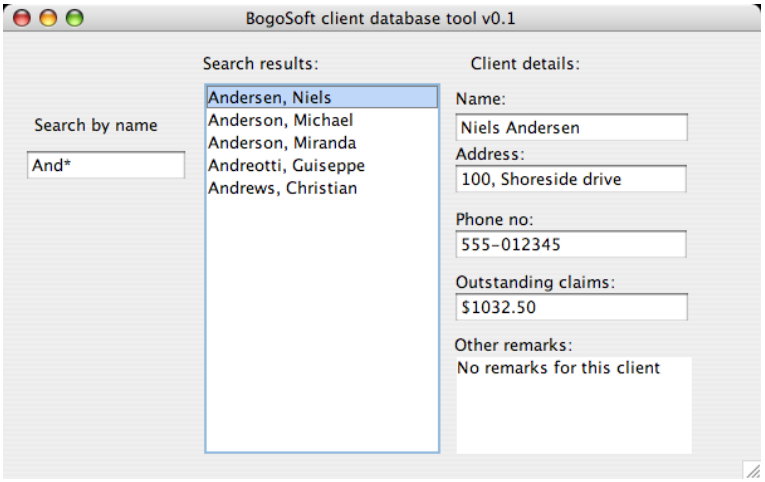


Figure 4.1: An imagined tool for client data retrieval.

palette, he could easily place identical-sized text fields for the other information items: address, telephone number and outstanding debt.

The application allows easy access to customer information, but as the client database grows, Tom finds it frustrating to have to browse the information for every single customer to identify the ones with outstanding debts. He wants his application to display this information at a glance. Sadly, the list of matching search results is already sorted alphabetically, and Tom actually prefers this way. Wishing to add some sort of text highlighting to the list component, he decides to try extending the component with this new functionality. To avoid damaging the already functional list component, he decides to clone it, and experiment with the code of the clone instead. Browsing through the source code, he finally finds the piece of code where the text color is set:

```
setForeground(Color.black);
```

Tom finds another piece of the code where the *if-else* construct is used, and he tries his hand at changing the Java code of his list component:

```
if(customer.getDebt() > 0.0){
  setForeground(Color.red);
}
```

When he tries to use the compiled result of this code, he realizes that every name appearing after the first customer with debt to the company also appears in red. He then realizes that the color must be reset for each customer, and writes:

```
if(customer.getDebt() > 0.0){
```

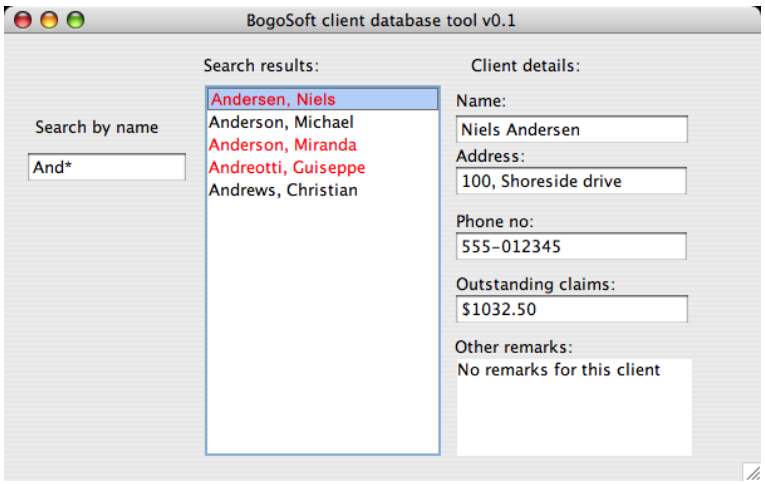


Figure 4.2: The same tool after the list component is extended with new functionality.

```
    setForeground(Color.red);
}
else{
    setForeground(Color.black);
}
```

The final result is a list that will display all customers who owe the company money in red text, saving Tom the time spent browsing through a long list to find debtors (figure 4.2). This simple example shows how the use of cloning techniques when customizing and extending visual components can help the end-user help himself; changes that would likely have warranted the involvement of a professional developer can now be done by an end-user developer. As the hiring of professional developers invariably implies extra expenses for a company, this kind of user empowerment can also help to cut development costs for small companies. Also, it can hopefully encourage users to acquire development skills, as it becomes easier for users to influence their day-to-day interaction with computer tools by designing the tools themselves.

## 4.2 Working with cloneable components

The act of building applications with cloneable components has not seen much exploration in literature on the Human-Computer Interaction (HCI) and software engineering fields of research. To be able to discuss this mechanism, we must agree upon a language for describing key concepts in the

process. Consequently, there is a need to establish a clear set of terms to be used. In (Mørch 1997), we are presented with three levels of end-user tailorability in applications: customization, integration and extension. Of these levels, customization and extension are of particular interest to this thesis. I will now give an explanation of some key subjects in the context of the SimpleBuilder project.

- The first step of this process is the *selection* of the component that most closely matches the needs for this particular part of the application. To aid the developer in this task, the builder tool can provide both graphic and textual aids, e.g. descriptive palette icons for the component, or short, descriptive tooltip texts. The user chooses the desired component, and places it on the design canvas. Careful thought must be given to how we can best help the end-user developer to find the component that is best suited to the task at hand, and a lot of resources are put into research on the most effective presentation of palette items by the producers of software that uses icons to offer functionality (e.g. office applications like Microsoft Word and Excel).

- *Customization* of components is defined in Mørch's paper as "Modfying the appearance of presentation objects, or editing their attribute values by selecting among a set of predefined configuration options." In classic component builder tools, the user is free to choose from a large palette of components. He may then place the component in his design, and begin the work of customizing it to fit the needs of this particular application. This is usually done through the use of a property sheet, where the user may modify the component within the limits set by the component's designer. This process is similar to the sort of customization done by most end-users in the preference panels of productivity suites and the like (Mørch & Mehandjiev 2000). As this activity must be performed within the configuration options given by the component designer, the level of customization can not be extended beyond what is offered by such a property sheet.

- *Extension* of components is then the logical next step in the modification of components needed to make up the final application. When extending a component, the user actually modifies implementation code, allowing for almost unlimited flexibility in adding desired features and functionality in an object. When taking the step from customization to extension, there is a significant increase in the *design distance* (see figure 4.3 on page 45)between the components with which the user interacts, and the underlying implementation code. The design distance describes the mental leap from understanding how the on-screen component works, and figuring out the logic underlying the implementation code. This also introduces a greater danger of

inadvertently damaging the object being extended. One solution to this is the traditional object-oriented technique of subclassing. In the SimpleBuilder, an alternative approach is chosen; cloning.

- *Cloning* components is presented in this thesis as a powerful technique to aid in end-user tailoring and development of software components. The advantage of having cloning functionality in a builder tool is (at least) two-fold. When doing customizations to a component, the user might want to store a particular configuration of the component for use later - either for the application he is working with, or another application. When using a builder tool with cloning functionality the user can make his desired customizations, producing a useful prototype. He may then clone this, and store this configuration in the palette.

  Additionally, the process of extending a component with new functionality requires some low-level editing on a source code level. Trying to perform these changes on a component directly can be potentially disastrous, as the typical expertise of an end-user developer does not include high programming skills. By cloning the original component - also known as the *prototype* (Lieberman 1986) - and experimenting with the clone instead, the user may safely experiment with code changes and added functionality without fearing that he might disturb the operation of the original component.

### 4.2.1 A process model for clone-based component development

Now that we have a useful vocabulary for describing component tailoring, we need to put the terms in a meaningful context. I will continue by presenting a process model of the steps included in component building with clone functionality (figure 4.4 on the next page). This model is not intended to describe the entire process of building an application from components; rather, it is focused on how the user may act in relation to a particular component. Although it is just a piece of the big picture, it is precisely this area that I wish to focus on in this thesis.

The first step of this process is the selection of a component. After choosing the desired component from the palette and placing it in the design pane, the user can the proceed to perform the desired customizations to the component. The user's interaction with the component has traditionally stopped here, but with the described process model the option to perform a cloning operation on the component is introduced.

The user may now decide on the next step in the process. He could decide that the customization done on the component is of potential use in later

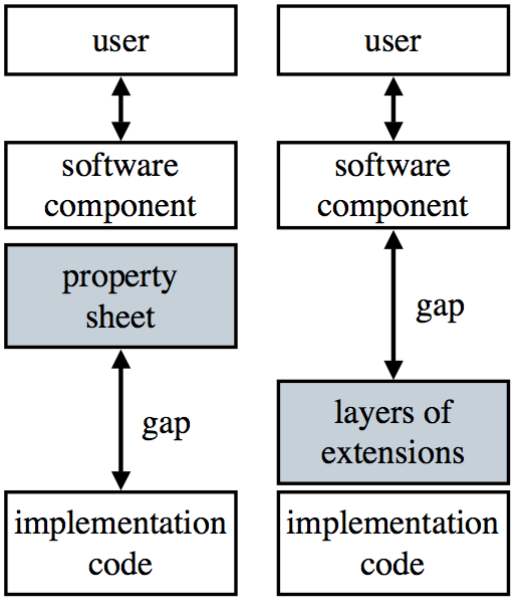


Figure 4.3: The use distance (user-component) and design distance (component-code) in customization and extension contexts. Shaded boxes represent the means used to perform end-user development. Adapted from (Mørch 1997).

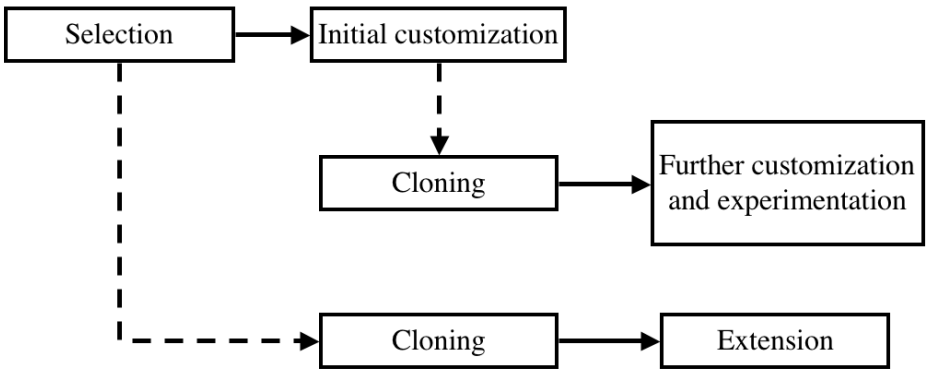


Figure 4.4: The process model underlying the SimpleBuilder.

applications, or even in the application currently being designed. If so, he can choose to clone the component (dashed line), leaving a copy of its current state in the palette. The user can now proceed to experimenting with other settings in the property sheet. The changes will be saved when exiting the builder tool, and the exact configuration of the component will be available again for later use. If the user wishes to experiment further without loosing a "good" configuration, he may of course make a clone of a clone, and so on.

If, on the other hand, he decides that the functionality or customizability offered by the original component is too limiting, he may decide to create an entirely new class, by creating an extended class definition based on the class of the prototype. Unless the end-user developer is an experienced Java programmer, this part of the cloning warrants a mechanism for easy and reasonably safe extension of a class definition with new functionality. Hopefully, the fact that the implementation code of the prototype is available in the newly-cloned class definition can help the user understand the correlation between Java code and component behaviour. The fact that he is working on a copy of the prototype also means that there is no risk of damaging the original, functioning component. After having performed the necessary extensions, components of this new type will be available for later use. Naturally, customized configurations of this new component will be available along with the cloned configurations of "standard" components.

Based on this, the SimpleBuilder presents us with two levels of cloning. We have *cloning from application*, in which a useful configuration of a deployed component is saved as a clone, that can itself be a basis for other clones. The other form of cloning is *cloning from palette*, in which the formal definition of the prototype is copied, in the form of Java code. This mechanism is intended for users that wish to extend a component with new or changed functionality. Because of the distinction between object and class in the Java programming language, both of these forms of cloning must be available if we want to aid the actions of customization and extension by adding the cloning element to the process. As mentioned in 2.4 on page 19 the component, rather than the object, is the level of granularity one is working with. Thus, in a builder tool context, the deployed components can be seen as the equivalent of the objects of traditional OO, while the palette items clicked to produce components can be compared to traditional classes. In the SimpleBuilder, I wanted to provide the benefits of cloning functionality on both of these levels.

## 4.3 The SimpleBuilder

### 4.3.1 Overview

The SimpleBuilder is a visual builder tool for assembling components based on the JavaBean technology. It is based on the Sun BeanBuilder application, a tool written to demonstrate some of the new features for component assembly in later versions of Java. The BeanBuilder has been extended in several areas; some of them are readily apparent, others are not. In this section, I will give a presentation of the tool in its current state, and describe the features it has to offer.

### 4.3.2 The palette and control panel

The palette (marked as 1 in figure 4.5 on the next page) is one of the most important parts of the SimpleBuilder interface[1]. It is divided into tabs according to the category of the components. Each tab holds a collection of components for easy selection and deployment on the design canvas (4). All icons have tooltip texts, which means that the name of the component represented by the icon will be shown when the mouse pointer is hovering over the icon. In addition, a short description of the component will be shown in the status bar under the control panel (2) After clicking on the desired icon, a subsequent click in the design canvas will instantiate a component. Newly instantiated components that display text, like labels and text fields, will have this text set to the short name of the component, e.g. “JTextField”.

When a component is cloned, either for extension or customization purposes, it will be placed in the “Clones” tab of the palette. If no such tab exists, one will be created. The SimpleBuilder also supports loading of new components, e.g. from Java Archive (JAR) files. These will be loaded in the "User" tab of the palette. To ensure that these components are available for later SimpleBuilder projects, the palette is stored as an XML file upon exit, and restored the next time SimpleBuilder is run. This XML file contains the class name of components to appear in the palette or, in the case of customized clones, the name of the component’s XML file. An example file is shown in listing 4.1 on page 49.

[1] In the screenshot, one will notice that the three components in the “Simple” category all have the same icon. The simple reason for this is my lack of skills with digital artwork software. I chose a generic icon - the image of a computer - for all my components, but of course this is simply a matter of swapping the Graphics Interchange Format (GIF) file for a more descriptive image.

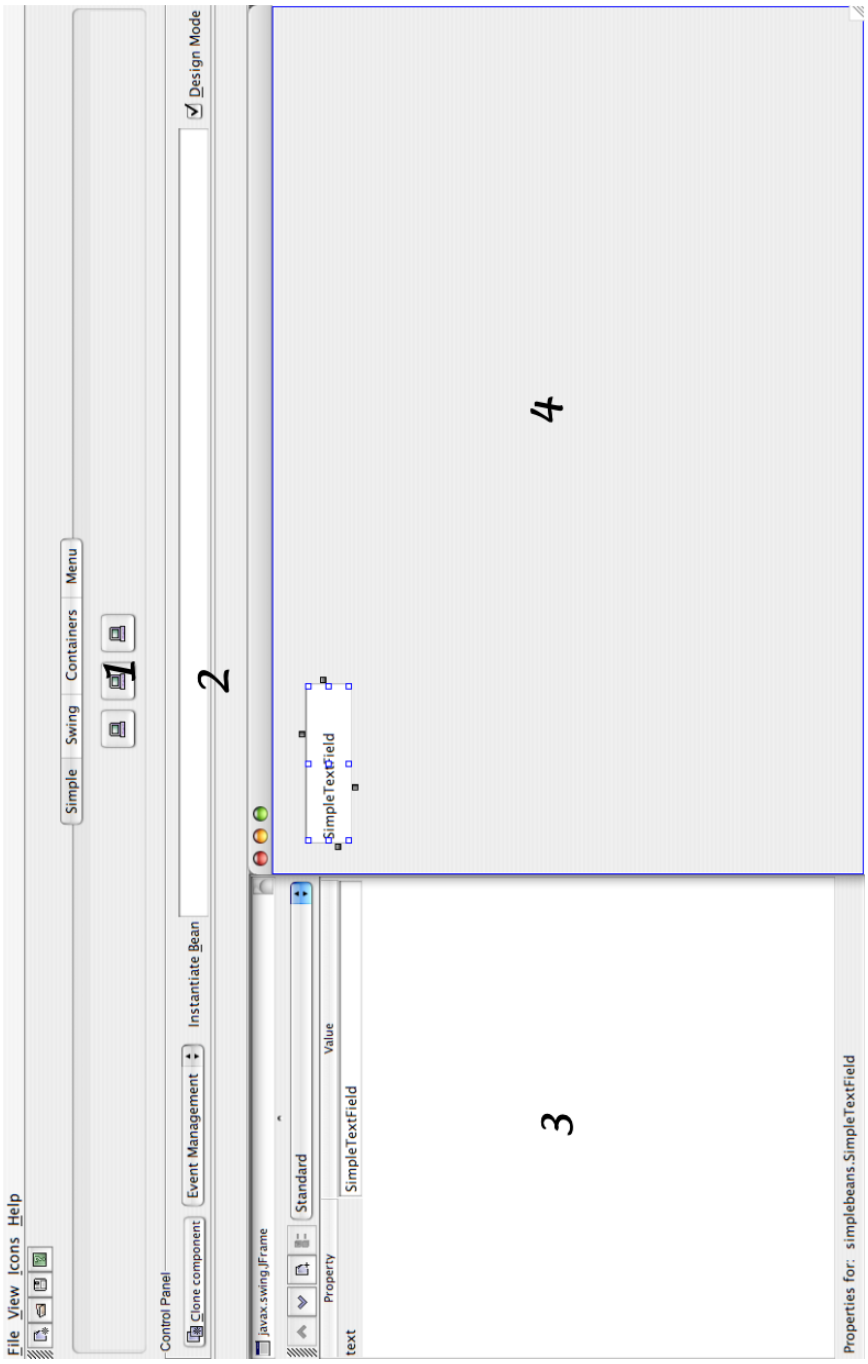


Figure 4.5: Screenshot of the SimpleBuilder directly after startup. In the property sheet (3) the selected component, a SimpleTextField, exposes a single user-customizable property - the text it contains.

```
<?xml version="1.0"?>
<!-- Palette configuration file autogenerated by
    SimpleBuilder. -->
<!DOCTYPE palette [
        <!ELEMENT palette (tab+)>
        <!ATTLIST tab name CDATA #REQUIRED>
        <!ELEMENT tab (item+)>
        <!ELEMENT item (#PCDATA)>
]>
<palette>
<tab name="Simple">
        <item>simplebeans.SimpleCanvas</item>
        <item>simplebeans.SimpleTextField</item>
        <item>simplebeans.SimpleScrollList</item>
</tab>
<tab name="Swing">
        <item>javax.swing.JButton</item>
        <item>javax.swing.JCheckBox</item>
        <item>javax.swing.JComboBox</item>
        <item>javax.swing.JTextField</item>
</tab>
<tab name="Clones">
        <item>javax.swing.JTextField15547391.xml</item>
</tab>
</palette>
```

Listing 4.1: Example XML file for the palette

### 4.3.3 Customization of components

The component currently selected (no. 1 in figure 4.6 on the next page) will be marked with white handles for resizing and moving the component. The grey handles seen in figure 4.6 are for event hookups between components - more on this in 4.3.5 on page 52. Selecting a component also displays its exposed properties in the property sheet, (area no. 3 in figure 4.5. Note that if there exists a corresponding `BeanInfo` class, only the properties that are listed there will be displayed in the sheet. Using the BeanInfo interface, the component designer can also specify different levels of detail for the property list, to allow users of assembly tools to ignore properties that are irrelevant for the task at hand. In this screenshot, this level of detail is set to “standard” (upper part of the property sheet). The component designer may choose which methods should be displayed for the settings “standard”, “expert” and “preferred” by specifying the corresponding level in the method descriptors of the BeanInfo class.

If the user wishes to clone a particular configuration of a component, this

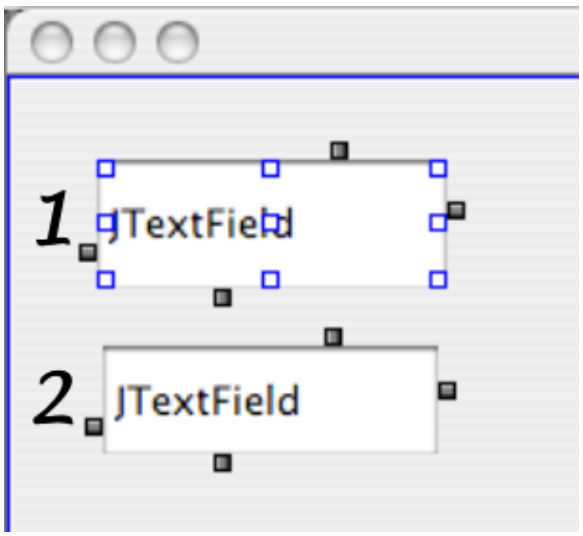


Figure 4.6: A selected (1) and an unselected component (2).

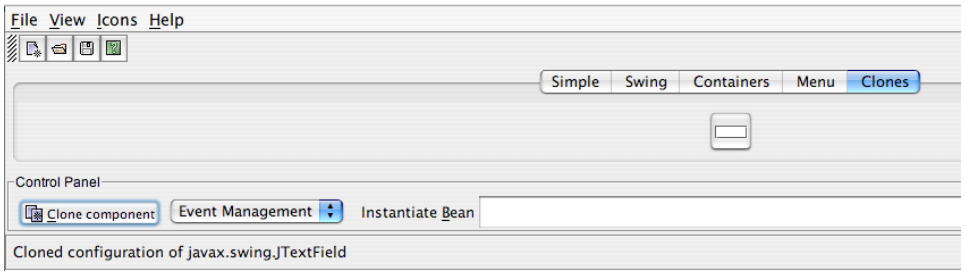


Figure 4.7: If, for example, the user wishes to clone the selected component from figure 4.6, a new item appears in the "Clones" tab of the palette. The message in the status bar reflects the fact that the icon represents a clone.

is done by clicking the "clone component" button (leftmost in the control panel). The SimpleBuilder will then store the specific configuration as a clone, and write the configuration to disk using the LTP mechanisms of Java 1.4 (see 3.2.2 on page 27). The file name of the stored clone is constructed from the class name of the component and its hash value. The resulting XML files are stored in the "clonedir" directory. After cloning, the clone will be available in the "Clones" tab of the palette (see figure 4.7). If the user selects this icon and deploys it in the design canvas, it will appear identical to its prototype's configuration at the time of cloning. All components added to the "Clones" tab will be available for later projects, and the cloned components behave just like any other component, except for the different starting values when deployed.

An illustration of an example process of cloning and customization is given

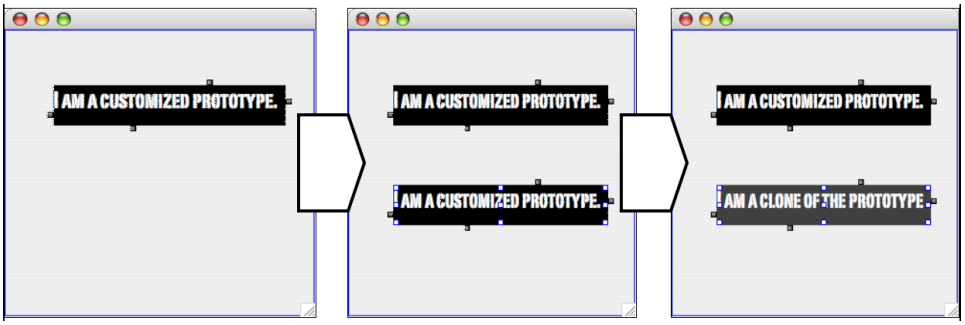


Figure 4.8: Three steps of cloning and customization. From left to right: Customizing the prototype component (in this case a JTextField) to be cloned, deploying an identical clone of the prototype, continuing customization of the clone.

in figure 4.8. Let us continue with the `JTextField` component from figures 4.6 and 4.7. The user performs the desired customizations (e.g. font, size and coloration), and decides that this is a configuration that can be useful in later projects. He therefore decides to add a clone of the component to the "Clones" tab of the palette. This is done by selecting the component in the application, and clicking the "Clone component" button. In the second step, he deploys the clone of the component, which will be identical to the prototype. In the third step, the users continues experimentation with the clone, changing text and color. The cloned configuration will be unaffected, remaining in the palette for further use.

### 4.3.4 Extension of components

If the user feels that the components available to him do not fully satisfy his needs, he has the option to clone component with the intension of modifications by extending its implentation code. In the SimpleBuilder, this is done by selecting an icon in the palette and clicking the "Clone component" button, thereby performing the cloning from palette described earlier. The application will then ask the user to supply a new name for the cloned class, to avoid overwriting the prototype class. The user can then proceed to experiment with the code of the cloned component. This mechanism can be used to add new functionality to a component, or change the existing implementation. Although this requires knowledge of Java programming, the process is made easier by supplying the full code of the prototype. This way, the user can study the logic of the existing code, and use this as a basis for his own, new functionality. Doing these changes will be perfectly safe, as the prototype class will not be affected in any way; the worst-case scenario is that the clone class is rendered useless. At the time being, there is

no tool to help users with no programming experience to do this particular task, but I have hopes that such a tool can be added to the SimpleBuilder package in the future.

### 4.3.5 Connecting two components

Simply placing the required components on the design canvas will of course not create a useful application. To provide any kind of functionality, the components will need to be connected in some way. The basic procedure for communication between JavaBean components is as follows: When the state of a component changes, such as text being entered in a text field, or a button being pressed, the component generates an `Event` object that is passed to every registered event listener. These listeners will be dynamically generated proxy objects. Proxy objects are initialized with information on which component triggered the event, as well as information on a target component in which a certain method will be invoked.

In the SimpleBuilder, such an event hookup is done by clicking and dragging an arc from the grey event handles of the source component to an event handle on the target component. When using standard JFC components, this action will cause an event hookup wizard to appear (for an example, see 4.9 on the facing page). The wizard queries the source and target object for available methods, and lets the user choose the method to be invoked in the target object and, if this method requires a parameter, a method in the source object to provide this value. This step is skipped if both components are SimpleBeans, however. More on this in 4.4 on page 54.

## 4.4 The SimpleBeans

The standard software components of the Java class library are not entirely suitable for clone-based end-user development. First of all, most of these components expose a large array of options for user customization, many of which can seem confusing or incomprehensible for a user with no programming experience. These components are also intended to be used in conventional object-oriented programming, and are designed to be very versatile. Secondly, the standard components are normally not accompanied by their source code. This effectively eliminates the possibility to perform extension of the components in the way I have described here. The problem of lowering the threshold for extending components sufficiently for end-user development led me to write a few simplified versions of commonly used GUI components. These components expose a far less complex choice of customizable properties, as can be seen from the property sheet

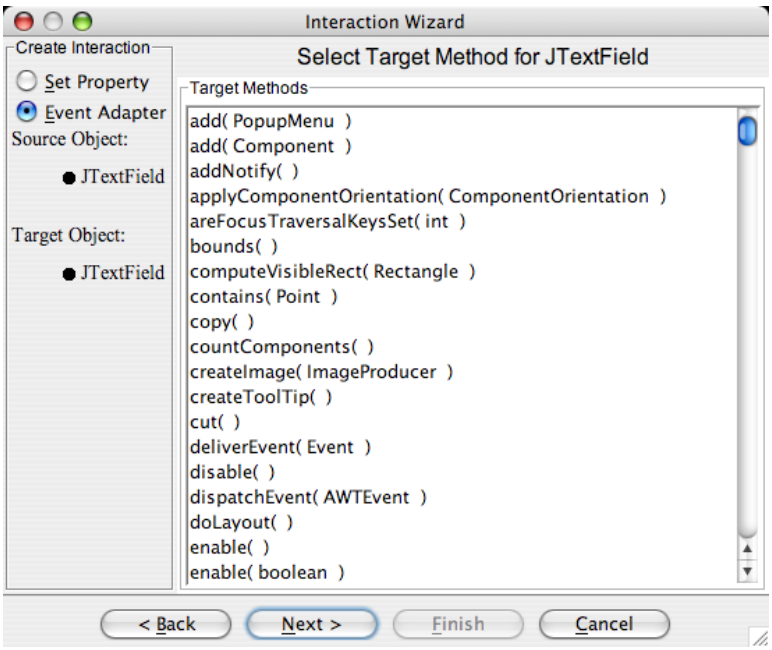


Figure 4.9: Just a few of the available methods when connecting two JFC components.

in figure 4.5. They are also supplied with source code, which resides in the "srcdir" directory directly under the SimpleBuilder's working directory. This means that while they offer fewer customization options than their JFC counterparts, they can be easily extended with new functionality.

Another problem with the general nature of the standard JFC components is that they expose a long list of available methods when they are to be connected to other components. When the user connects two components to each other in design mode, the SimpleBuilder will query the component's `BeanInfo` class for a list of available methods. As an example, some of the available methods presented when connecting to `JTextField` components are shown in figure 4.9. To make the process of connecting components easier, the example components all implement the same interface, which only offers one pair of set/get methods. This puts the responsibility of handling different data types on the component itself. This could prove a challenge for component designers, but should make life easier on the end-user developer for which the SimpleBuilder is intended.

### The SimpleBean framework

In order to be as simple and straightforward to use as possible, the example components accompanying the SimpleBuilder are implementing an interface called `SimpleBean`. At the present stage of development, this interface defines only two methods: `setValueSBO()` and `getValueSBO()`. These methods are used for simplifying the process of connecting two components on the design canvas of the builder. The fact that every component implementing the SimpleBean interface must support these two methods means that component hookup can be performed without querying the user for information about which methods on the source and target components should be connected.

The SBO part of the method names refers to the class `SimpleBeanObject`, the other part of the SimpleBean framework. These objects are used as wrapper classes for message passing between components. They can contain one instance of any Java object, such as strings, integers and the like. In addition to an object reference, the class also defines a field for describing the type of its contents. The currently used fields are `TYPE_STRING`, `TYPE_DOUBLE` and `TYPE_INT`. Additionally, a boolean variable describes whether or not the contained object is an array of the specified type, or a single value. This way, a SimpleBean component receiving a SimpleBeanObject can check its type, and behave accordingly. This provides a unified interface for sending text, number values or any other kind of data between components; the responsibility for properly handing data types is placed on the component implementation.

### SimpleCanvas

The SimpleCanvas component was the first component I wrote, mainly for testing the functionality of the SimpleBuilder. It consists of a simple panel with a user-customizable color. The user can resize and move this panel, and use it as a background for other components. This makes it useful as a means to add color to an application, perhaps to signify different parts of a GUI, e.g. one part for textual input and one part for feedback from the application.

### SimpleTextField

The SimpleTextField class is a subclass of the generic *JTextField*, the JFC component for text entry in graphical Java applications. While the JTextField provides a dazzling array of customization options, the SimpleTextField

simply lets you define its placement, its size and the text initially contained in the field. When the carriage return key is pressed, the component will fire an event object to instruct listeners that the text property has changed.

### SimpleScrollList

My last example component was an implementation of a scrollable list. Normally, a scrolling list consists of at least three parts: The scrolling pane itself, the list to be scrolled, and some sort of data collection to hold the data of the list. The SimpleScrollList is basically a scrolling pane that creates its own list component and data vector when instantiated. By hiding these implementation details from the user, the component will appear to the end-user developer as a self-contained component for displaying a list of strings. This is an example of how software components can contain objects of several different classes, yet appear as a single entity to the end-user developer.

# Chapter 5

# Discussion

> It is better to debate a question without settling it than to settle a question without debating it
>
> — *Joseph Joubert*

This chapter contains a general discussion on the process of making the SimpleBuilder, and the research done for this thesis. I will also describe some of the problems I encountered during my work, and alternative solutions that I rejected.

## 5.1 The road taken

The SimpleBuilder project seemed a daunting task to me at first, and I certainly saw my share of false starts and dead ends. I spent considerable time trying to find a suitable basis for my builder tool before I found the BeanBuilder. I scanned through the projects at SourceForge[1], but nothing seemed entirely suitable. I needed a tool with available source code, and a license that allowed me to use it as a basis for my own project. In addition, I wanted it to be mulitplatform, and I needed it to be small enough that it was possible for me to get a reasonable grasp of its logic and structure within the time allotted to a Cand. Scient. thesis - less than a year. Sadly, the projects I found while browsing the Internet were either too immature, or far too complex for a one-man, one-year project. I find it quite ironic that the (hopefully) best candidate, the BeanBuilder, is a product from Sun Microsystems, the company behind the Java programming language and the

[1] A website offering free web space and CVS access for open source developers, `http://www.sf.net`.

JavaBeans architecture. In retrospect, Sun's website was perhaps the first place I should have looked.

Although I have a good grasp of programming in general, and Java programming in particular, programming with graphical interfaces was an unexplored territory for me. I have taught beginner classes in object-oriented programming with Java for four semesters, but my only experience with GUI programming in Java was a few simple assignments using the Abstract Windowing Toolkit (AWT, the GUI class libraries used prior to Java 1.2) when I took the beginner course in Java at the University of Oslo. So, I had to read up on creating GUIs with the Java Foundation Classes (JFC, also known as "Swing"), the toolkit used in current versions of Java. An example follows: At one point I needed to perform the simple task of asking the user to input a string value, to be used as the name for a new, cloned class definition. I spent much time in finding out how to produce a modal window with the desired layout and feedback options, and returning the value entered by the user to the calling method. A few days after creating an ugly, but functional requester, I was made aware of the functionality in JFC for performing this exact task: the `JDialog` class of requester windows. Consequently, I had to go back to my code and change it to utilize this much cleaner solution for getting a string value from the user.

At the outset of my thesis work, I had only a faint notion of the technical details behind JavaBeans. Although I had come to read a lot of literature on the subject of component technology, most of this literature discussed the technicalities of different software component implementation in a very superficial matter. When writing the example JavaBeans for my application, called SimpleBeans, I spent a considerable amount of time pondering on the access modifiers of methods. Which methods should be public, which should be private, and so on. The SimpleBuilder uses reflection to inspect classes and examine which public interfaces the offer to the world outside, so I was sure that these access modifiers were an important point. As it turned out, the JavaBeans technology offers special classes for making this part of the process easier, namely the BeanInfo interface. By accompanying each component with its own BeanInfo implementation, I could easily control which properties should be visible to the SimpleBuilder, and which should remain hidden.

To have a good picture of the theoretical basis for the application, I searched books and archives for relevant (and some not-so-relevant) articles on the subjects of user participation, software components, evolutionary development and object cloning. I found the papers on prototyping and cloning particularly interesting, as they presented a view on object-orientation I had never encountered before. Although I had encountered some of the other subjects in courses taken at the University of Oslo, I think I can safely

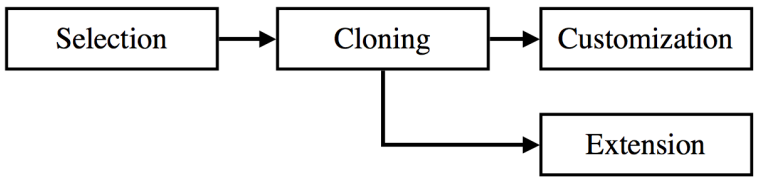


Figure 5.1: My initial process model for developing with cloneable components.

say that working with the idea of cloneable components has been a tremendous learning experience for me. Not only have I had to learn the principles of programming for a graphical environment, I have also had the pleasure to read pages upon pages of interesting article; each of them giving me valuable new perspectives on how component-orientated development and user empowerment in the workplace can be combined.

## 5.2 Alternative routes

One of the first things I needed to find out was what the process of cloning and modifying components should look like. I decided to focus on the two mechanisms of extension and customization, taken from (Mørch 1997). At first, I imagined that the process would look like the one in figure 5.1. First the user selects the component, performs a cloning operation, and then proceeds to modify this component either through customization or through extension. As I started programming, in became increasingly clear to me that the class/object duality of the Java programming language would make this impractical. As customization is done on individual component, while extension is performed on classes, the two cloning operations would need to be implemented differently.

As I was trying to make a descriptive illustration in Microsoft PowerPoint, I became aware of the pattern of my own work: When making text boxes, I started out by customizing one box; choosing color, font and line width. I then made copies of this, only changing the text and placement. This was a simple form of cloning, and I realized that the "cloning from application" concept could be done in a similar way. The user would want to produce a useful configuration of a component, and then cloning this to have the configuration available from the palette. Further customization can then be done. These ideas evolved into the process model described in 4.2.1 on page 44.

I also spent much time and energy on the problem of simple communica-

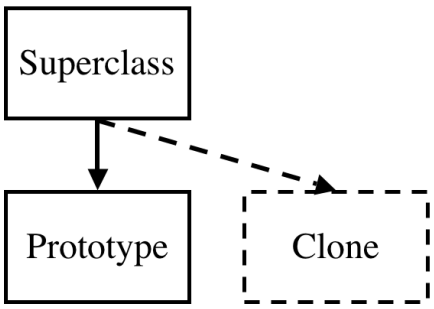


Figure 5.2: Inheritance scheme for clones made in the SimpleBuilder.

tion between components. JavaBeans communicate by sending subclasses of the `AWTEvent` class to eachother. These event objects can be fired when a change occurs in a component, such as a field receiving a new value, or a button receiving a mouse click. Before I had an adequate understanding of the concept of dynamic proxy classes, used to create on-the-fly classes for event listening, I tried to make the SimpleBean components themselves implement the `EventListener` interface. As my SimpleBeans initially were subclasses of the `SimpleBean` class, I cursed the lack of multiple inheritance in Java (see 3.1.1 on page 24). As my subclasses needed to subclass GUI components like textfields and the like, they could not at the same time implement the event listening code I wanted to include in the SimpleBean definition. As it turned out, there were good reasons for this. When i finally understood how JavaBeans were *meant* to be connected, through dynamic event listeners, I rewrote the SimpleBean definition to simply being an interface specification.

I was not quite sure how to write the code for extending classes, i.e. letting the user write new classes based on existing ones. I considered making new classes subclasses of the original, keeping with the mindset of traditional object orientation. However, I decided to abandon this idea in favor of another approach: making the clone a sibling class[2] instead (figure 5.2). I saw this as the best way to ensure that the cloned class mimicked the original in as many ways as possible. Thus, the mechanisms for extending classes in the SimpleBuilder are as simple as they are effective: a complete copy of the original source file is made, along with its BeanInfo class (if it exists). The copy is given a new name by the users, and compiled into a new class. As mentioned elsewhere in this paper, functionality to actually change the cloned class definition is lacking; the only option is traditional programming in a text editor.

[2]In class inheritance, the term "sibling class" is often used to describe a class which has the same superclass (the same parent, hence the term) as another class.

When I was writing the code for extending classes, I stumbled upon yet another dilemma. As recompilation was required to create extended class definitions, I needed a place to store the source code for these new classes - and also the source code and compiled bytecode of the prototype components that were to be cloned. Java classes are normally organized in *packages*. Package names consist of words separated by periods. If desired, the first word will usually represent the organization which created the package. The rest of the package name reflects the function of the class, and the directory structure in which it resides. Because of this, I wanted a directory structure that reflected the package containing the class to be cloned. After some thinking, I initially settled on a directory structure scheme like this:

*<working directory>/sourcedir/<package name>/<classname>/<classname>.java*

As an example, the path for the class *javax.swing.JTextField* following this sceme would be *srcdir/ javax/swing/JTextField/JTextField.java*. It then became clear to me that this would require considerable massaging of path name and package qualifiers in the code that dealt with retrieving, copying and compiling source code. As it turned out, I could exploit the fact that the Java Virtual Machine will search a directory structure corresponding to the package qualifiers when looking for a new class. Therefore, I decided that source files of classes belonging to the same package would reside in the same directory, e.g. *srcdir/javax/swing/JTextField.java*. This way, if a clone operation is performed, the resulting bytecode will be found as *srcdir/-javax/swing/JTextFieldClone.class*. This is exactly where the Virtual Machine will look if "srcdir" is in the search path, and the class *javax. swing.JTextField-Clone* is requested.

# Chapter 6

# Further research

> Genius begins great works; labor alone finishes them.
>
> — *Joseph Joubert*

This chapter will look at the path ahead for clone-based end user development and visual programming with the SimpleBuilder. I will suggest a few directions for further research, as well as some specific tasks for future researchers in the field. Hopefully, this chapter can inspire others to further work; perhaps as a new master's thesis.

## 6.1 Remaining work

As a truly useful tool for research on component cloning, the SimpleBuilder is far from finished. Before attempting to put it to serious use, it needs to be thoroughly tested for stability issues and lingering bugs. Missing from the toolkit is also the option to overwrite cloned components, i.e. saving work-in-progress without adding another clone to the palette.

Also, the SimpleBean framework is very rudimentary and not really thoroughly tested. The sample components are mostly for testing and demonstration purposes. Without a doubt, several more components are needed to provide the end-user developer with a palette of flexible and useful components for application development.

The SimpleBuilder should also be packaged for easy, double-click startup on all Java-enabled platforms. At the time being, environment variables containing information about the placement of auxiliary Java classes needs to be set via the command line; preferably through a shell script. This is fine for UNIX-based platforms, but could be a problem for users that are

accustomed to the point-and-click simplicity of windowing environments. Once these pieces of the puzzle are in place, the time-consuming work of evaluating the benefits of these new features must be done. The concepts must be tested on a real audience, and only then can the potential merits of the methods described herein be measured.

## 6.2 Putting the SimpleBuilder to use

Although considerable work has been put into understanding and developing Sun's BeanBuilder into the SimpleBuilder, this is clearly only the beginning of the story. No tool is useful unless it is put to work, and the SimpleBuilder is no exception. The SimpleBuilder was never intended to be a professional tool for "real world" usage, however. It is intended as a tool for research into a specific area within end user development; that of evolutionary development by cloning and following examples. While still lacking several of the features I would have liked to see in the product, the SimpleBuilder can be used as a starting point for simple exploration of this field of study. I envision several forms of research possible, based on the state of SimpleBuilder today:

- Research is needed to see whether my theories about evolution by resemblance are sound. To do this, it seems practical to gather a group of non-experts and ask them to perform specific simple tasks. I believe that at this stage this needs to be done as qualitative research, rather than quantitative. It should be interesting to receive feedback from people from the intended target group, providing the data needed to study whether the approach taken with SimpleBuilder seems to have a positive effect on productivity and creativity in end user development.
- The SimpleBuilder is based on a tool made by programmers, for programmers. As such it is likely that the placement of buttons is suboptimal, and that the interface is unintuitive, cluttered, confusing or otherwise inefficient. Valuable information could be gained by performing usability testing with end users. Usability testing, for example in the form of *heuristic evaluation* (Nielsen 2004) or "thinking-aloud" evaluation (Lewis & Rieman 1994), is an important step in determining problems with user interfaces. By letting several users perform such tests, one could learn a lot about how to make interfaces that simplify the development process for end user programmers. There is doubtless a lot that can be done to make the SimpleBuilder easier to use, and less confusing to newcomers.
- As prototype-based component development is such a fresh field of

research, there is no literature on suitable application domains for the use of such tools. This could be an interesting topic of research. Although the goal of end-user development research is to enable end-user empowerment in a large array of application domains, I suspect that there are some fields that lend themselves more easily to this activity than others. This in turn could provide basis for choosing suitable arenas for field testing of end-user development tools such as the SimpleBuilder.

## 6.3 Improving the SimpleBuilder for prototype-based end user development

Clearly, the SimpleBuilder is not a finished product — and it is unlikely that it ever will be. Just as the SimpleBuilder is intended to support development of applications in constant evolution, I hope that the SimpleBuilder itself will undergo several revisions and constant refinement. The code may appear messy, as functionality to support the new cloning features has been grafted onto existing code in several places. The source tree would likely benefit from code cleanup and refactoring of cloning functionality into separate classes where possible. Several features are missing or incomplete in the current version of the tool. A non-exhaustive list follows:

- A help system is needed. Research shows that while exploration of a program is a powerful way of learning how to use a new system, this is often done with the aid of available documentation (Rieman 1996). Today, there is no documentation or help system for the SimpleBuilder aside from the BeanBuilder tutorial and the feature descriptions presented in this thesis. This lack of formal documentation clearly needs to be adressed, both to ensure continued work on the SimpleBuilder, and to supply test users with an aid they most likely would have had available in a "real world" situation. This also includes documentation of accompanying example components. The use of standard JavaBeans components is outside the scope of this documentation process, but the SimpleBean components accompanying the SimpleBuilder need to be documented.
- Proper consideration should also be given as to whether one should implement typed ports, such as those of EVOLVE (Stiemerling et al. 1999). While this can simplify the process of connecting, or "wiring", the components, it also introduces an additional level of complexity, namely that of data types. The current approach of only one object type being passed between components is certainly simplest for the user, but it places the burden of knowing how to handle different val-

ues on the component implementation. This is consistent with our goal of taking the burden of computer-specific terms (i.e. data types) off the end-users shoulders, but it may limit functionality in unforeseen ways. Further research on this area is needed, and feedback from the team behind EVOLVE will surely be of use.

- The class extension mechanisms of the SimpleBuilder today are rudimentary, to say the least. A copy is performed on the source code of the original component, and the resulting source file must be edited by hand by means of traditional Java programming. To allow actual end-user developers to safely implement new functionality in a SimpleBean, a new type of tool is needed. In the SELF programming environment ( 2.3.2 on page 17), adding functionality to an object is a simple process of either editing an existing method object, or adding a new one - which can be based on an existing method. I hope that a similar mechanism can be implemented for the SimpleBuilder, i.e. a graphical interface for extending Java classes. By limiting the scope to SimpleBeans components, some delimiting can be made. For example, I envision that adding data fields or methods to a class can lead to automatic updating of the corresponding BeanInfo class, to reflect the changes. Such an editing tool would certainly be an exiting project, and may lower the threshold for actual Java programming.

- In chapter 4, I brushed briefly on the distinction between selection from the palette, and selection from the application. As mentioned in the previous point, extending classes (from the palette) needs more work to be practical. An exciting angle for future research could also be to focus on the selection and cloning from components in the application. Experience from the use of an application will likely inspire the end-user developer to examine the components of a running application, and figure out ways to improve these. Figuring out how to improve functionality this aspect of component cloning would be an intriguing challenge, both theoretically and technically.

- When I set out to create the SimpleBuilder, my vision was that of a multiplatform, flexible tool for working with cloneable components. As it is written entirely in Java, it should run on all platforms with a working Java Virtual Machine and class library. However, all development has been done under Mac OS X. I have not had the time or resources to check the tool's performance and behavior on other platforms, such as Microsoft Windows, GNU/Linux or Solaris. There are no obvious reasons as to why it should not work, but appearance, placement of buttons etc. needs to be checked. Likewise, although the file manipulation done in the application is written in platform-independent code, this has not been tested.

# Chapter 7

# Conclusion

> The outcome of any serious research can only be to make two questions grow where only one grew before.
>
> — *Thorstein Vebler*

The goal of this thesis, as formulated in 1.3 on page 4, was to provide a simple toolkit for component assembly using cloning techniques and user modification of components. Some literature had been written on the subject, but there was no working toolkit available to demonstrate the theories as concrete actions in an actual application builder. This thesis describes the making of such a toolkit.

The work described herein is a work in progress, not a professional application ready for shrink-wrapping and sale to the masses. The work has consisted of much research, both in literature and API documentations, but also programming work on the "SimpleBuilder", a modification of the Sun Bean Builder. I must confess that I have grown increasingly fascinated with the concepts of prototype-based object orientation, of end user empowerment and of component technology. I have learnt a great deal about software development methodologies, the philosophy of software engineering and user participation, and, on a more technical level, I have learnt a lot about the internals and externals of the Java programming language.

The thesis work started out with a considerable amount of research into existing solutions, and literature on the subjects of evolutionary development, software components and the prototype/clone concept. Following this, I sat down to decide which features I wanted in a tool for supporting these components in a manner simple enough for users with no programming experience. In the problem definition of this document, my main concern was whether it was feasible to create a multiplatform component assembly tool supporting cloning of components, that was also simple to

use. Looking at the results, I think that at least a small step has been taken in that direction.

In chapter 4, I presented the SimpleBuilder, which along with this thesis is the chief result of my work. The SimpleBuilder serves as a proof of concept that can serve as a basis for discussion on the topic, as well as for further development. It is probably not mature enough to be used for in-depth research on the topic of how useful these programs can be in real-life situations. However, it was my intention that the SimpleBuilder could be a useful artifact for concretization of terms such as cloning, customization and extension of components. I feel that at least this goal has definitely been reached. In this document, I have also presented a model for clone-based development that surely should be an interesting basis for further discussion. The topic of evolution by cloning familiar components is not explored in great detail in the available literature. I hope that I have contributed material that can serve as foothold; a starting point for this exploration.

# Bibliography


Apple Computer, Inc. (1993), 'The NewtonScript Programming Language'.

Bansler, J. (1989), 'Systems development research in scandinavia: Three theoretical schools', *Scandinavian Journal of Information Systems* **1**(9), 3–20.

Benington, H. D. (1987), Production of large computer programs, *in* 'Proceedings of the 9th international conference on Software Engineering', IEEE Computer Society Press, pp. 299–310.

Bjerknes, G. & Bratteteig, T. (1988), 'Computers utensils or epaulets? the application perspective revisited', *AI & Society* **2**(3), 258–266.

Bjerknes, G., Bratteteig, T. & Espeseth, T. (1991), 'Evolution of finished computer systems the dilemma of enhancement', *Scandinavian Journal of Information Systems* **3**, 25–45.

Boehm, B. W. (1988), 'A spiral model of software development and enhancement', *IEEE Computer: Innovative Technology for Computer Professionals* **21**(5), 61–72.

Budde, R., Kautz, K. & Kuhlenkamp, K. (1992), *Prototyping - An Approach to Evolutionary System Development*, Springer Verlag, New York.

Bødker, S. & Grønbæk, K. (1991), 'Cooperative Prototyping - Users and designers in mutual activity', *International Journal of Man-Machine Studies, special issue on CSCW* **34**(3), 453–478.

Chang, B.-W., Ungar, D. & Smith, R. B. (1995), Getting close to objects: object-focused programming environments, *in* M. Burnett, A. Goldberg & T. Lewis, eds, 'Visual Object-Oriented Programming', Prentice-Hall, pp. 185–198.

Costabile, M. F., Fogli, D., Fresta, G., Mussio, P. & Piccinno, A. (2003), Building environments for end-user development and tailoring, *in* 'IEEE Symposium on Human Centric Computing Languages and Environments', IEEE Press, pp. 31–38.

Cox, B. J. (1986), *Object oriented programming: an evolutionary approach*, Addison-Wesley Longman Publishing Co., Inc.

Curtis, B., Krasner, H. & Iscoe, N. (1988), 'A field study of the software design process for large systems', *Commun. ACM* **31**(11), 1268–1287.

Dahl, O.-J., Myhrhaug, B. & Nygaard, K. (1968), Some features of the simula 67 language, *in* 'Proceedings of the second conference on Applications of simulations', IEEE Press, pp. 29–31.

Dahlbom, B. & Mathiassen, L. (1993), *Computers in context: The philosophy and practice of systems design*, Blackwell Publishers, Cambridge.

David, E. E. & Fraser, A. (1968), Software: The state of the art, *in* P. Naur & B. Randell, eds, 'Software engineering: report on a conference sponsored by the NATO Scientific Committee', Scientific Affairs Division, NATO, Garmisch, pp. 119–125.

Ehn, P. (1993), Scandinavian design: On participation and skill, *in* D. Schuler & A. Namioka, eds, 'Participatory Design: Principles and Practice', Lawrence Erlbaum, New Jersey, pp. 41–77.

Fischer, G. (2002), 'Beyond "Couch Potatoes": From Consumers to Designers and Active Contributors', *First Monday* **7**(12).
**URL:** *http://www.firstmonday.dk/issues/issue7_12/fischer/*

Floyd, C., Reisin, F.-M. & Schmidt, G. (1989), Steps to software development with users, *in* C. Ghezzi & J. A. McDermid, eds, 'Proceedings of ESEC '89, 2nd European Software Engineering Conference', Berlin, pp. 48–64.

Fowler, M. & Scott, K. (2000), *UML distilled (2nd ed.): a brief guide to the standard object modeling language*, Addison-Wesley Longman Publishing Co., Inc.

Ghiselin, B., ed. (1985), *The Creative Process, a Symposium*, University of California press.

Gosling, J., Steele, G. & Joy, B. (2000), *Java Language Specification*, Addison-Wesley.

Java Community Process (2003), 'Java Specification Requests #057 Long-Term Persistence for JavaBeans Specification'. 7 oct. 2003.
**URL:** *http://jcp.org/en/jsr/detail?id=057*

Kaasbøll, J. (1997), 'How evolution of information systems may fail: many improvements adding up to negative effects', *European Journal of Information Systems* **6**, 172–180.

Kaasbøll, J. & Øgrim, L. (1994), Super-users: Hackers, management hostages, or working class heroes? a study of user influence on redesign in distributed organizations, *in* P. Kerola, ed., 'Precedings of the 17th Information systems Research seminar In Scandinavia', pp. 784–798.

Lewis, C. & Rieman, J. (1994), 'Task-Centered User Interface Design'.
**URL:** *http://hcibib.org/tcuid/chap-5.html*

Lieberman, H. (1986), Using prototypical objects to implement shared behavior in object-oriented systems, *in* 'Conference proceedings on Object-oriented programming systems, languages and applications', ACM Press, pp. 214–223.

Mathiassen, L., Seewaldt, T. & Stage, J. (1995), 'Prototyping and specifying: Principle and practices of a mixed approach', *Scandinavian Journal of Information Systems* **7**(1), 55–72.

McCracken, D. D. & Jackson, M. A. (1982), 'Life-cycle concept considered harmful', *ACM Software Engineering Notes* **7**(2), 29–32.

McIlroy, D. (1968), Mass produced software components, *in* P. Naur & B. Randell, eds, 'Software engineering: report on a conference sponsored by the NATO Scientific Committee', Scientific Affairs Division, NATO, Garmisch, pp. 138–150.

McIlroy, D. (1972), 'The outlook for software components', *International state of the art report: software engineering* **11**, 243–252.

Microsoft Corporation (2004), 'COM: Delivering on the Promises of Component Technology'. 5. feb 2004.
**URL:** *http://www.microsoft.com/com/*

Mørch, A. (1997), Three levels of end-user tailoring: customization, integration, and extension, *in* 'Computers and design in context', MIT Press, pp. 51–76.

Mørch, A. (2003), Evolutionary growth and control in user tailorable systems, *in* 'Adaptive evolutionary information systems', Idea Group Publishing, pp. 30–58.

Mørch, A. I. & Mehandjiev, N. D. (2000), 'Tailoring as collaboration: The mediating role of multiple representations and application units', *Comput. Supported Coop. Work* **9**(1), 75–100.

Nielsen, J. (2004), 'How to conduct a heuristic evaluation'. 21 jan. 2004.
**URL:** *http://www.useit.com/papers/heuristic/heuristic_evaluation.html*

Nygaard, K. (1979), The iron and metal project. trade union participation, *in* Å. Sandberg, ed., 'Computers dividing man and work', Utbildningsproduktion.

Nygaard, K. & Dahl, O.-J. (1978), The development of the SIMULA languages, *in* 'The first ACM SIGPLAN conference on History of programming languages', ACM Press, pp. 245–272.

Paul, S., Prakash, A., Buss, E. & Henshaw, J. (1991), Theories and techniques of program understanding, *in* 'Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research', IBM Press, pp. 37–53.

Petroski, H. (1992), *The evolution of useful things*, A. Knopf, New York.

Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S. & Carey, T. (1994), *Human-Computer Interaction*, Addison-Wesley.

Rieman, J. (1996), 'A field study of exploratory learning strategies', *ACM Trans. Comput.-Hum. Interact.* **3**(3), 189–218.

Royce, W. W. (1987), Managing the development of large software systems: concepts and techniques, *in* 'Proceedings of the 9th international conference on Software Engineering', IEEE Computer Society Press, pp. 328–338.

Smith, R. B. & Ungar, D. (1995), Programming as an Experience: The inspiration for Self, *in* 'ECOOP '95 Conference Proceedings'.
**URL:** *http://www.sun.com/research/self/papers/programming-as-experience.html*

Smith, R. B. & Ungar, D. (2001), 'Programming as an Experience: The inspiration for Self'. 6. feb 2004.
**URL:** *http://research.sun.com/research/features/tenyears/volcd/papers/ungar.htm*

Stiemerling, O., Hinken, R. & Cremers, A. B. (1999), The EVOLVE Tailoring Platform: Supporting the Evolution of Component-Based Groupware, *in* 'Proceedings of EDOC'99', IEEE Press, pp. 106–115.

Sun Microsystems (2003*a*), 'Enterprise JavaBeans Technology'. 8 oct. 2003.
**URL:** *http://java.sun.com/products/ejb/index.jsp*

Sun Microsystems (2003*b*), 'JavaBeans'. 7 oct. 2003.
**URL:** *http://java.sun.com/products/javabeans/*

Sun Microsystems (2004), 'Java Remote Method Invocation (RMI)'. 5 feb. 2004.
**URL:** *http://java.sun.com/products/jdk/rmi/index.jsp*

Szyperski, C. (1998), *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley.

The Object Management Group (2003), 'Introduction to OMG Specifications'. 8 oct. 2003.
**URL:** *http://www.omg.org/gettingstarted/specintro.htm#CORBA*

*The Jargon File 4.7.7* (2004). 20 feb. 2004.
**URL:** *http://www.catb.org/~esr/jargon/*

Ungar, D. & Smith, R. B. (1987), Self: The power of simplicity, *in* 'Conference proceedings on Object-oriented programming systems, languages and applications', ACM Press, pp. 227–242.

Wittgenstein, L. (1953), *Philosophical investigations*, Basil Blackwell & Mott, Oxford.

Wulf, V. & Rohde, M. (1995), Towards an integrated organization and technology development, *in* 'Proceedings of the conference on Designing interactive systems', ACM Press, pp. 55–64.

Åsand, H.-R. H., Mørch, A. & Ludvigsen, S. (2004), Superbrugere: En strategi for ikt-omstilling, *in* A. M. Kanstrup, ed., 'E-læring på arbejde', Roskilde Universitetsforlag. Learning Lab Denmark, pp. 131–147.

# Appendix A

# Code listings

## A.1 An example of an application stored as XML

Listing A.1: This is the XML representation of a simple application consisting of a textfield, and a label that displays the last text to be entered in this field.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_03" class="java.beans.XMLDecoder">
 <object class="javax.swing.JFrame">
  <void property="size">
   <object class="java.awt.Dimension">
    <int>300</int>
    <int>322</int>
   </object>
  </void>
  <void id="JPanel0" property="contentPane">
   <void method="add">
    <object id="JLabel0" class="javax.swing.JLabel">
     <void property="bounds">
      <object class="java.awt.Rectangle">
       <int>72</int>
       <int>174</int>
       <int>37</int>
       <int>16</int>
      </object>
     </void>
     <void property="text">
      <string>JLabel</string>
     </void>
    </object>
   </void>
   <void method="add">
    <object id="JTextField0" class="javax.swing.JTextField"
   >
```

```
  <void property="bounds">
   <object class="java.awt.Rectangle">
    <int>60</int>
    <int>105</int>
    <int>70</int>
    <int>22</int>
   </object>
  </void>
  <void property="scrollOffset">
   <int>1</int>
  </void>
  <void property="text">
   <string>JTextField</string>
  </void>
  <void method="addActionListener">
   <object class="java.beans.EventHandler" method="
 create">
    <class>java.awt.event.ActionListener</class>
    <object idref="JLabel0"/>
    <string>setText</string>
    <string>source.text</string>
   </object>
  </void>
  <void property="document">
   <void property="documentProperties">
    <void id="Boolean0" method="get">
     <string>filterNewlines</string>
    </void>
   </void>
  </void>
 </object>
</void>
<void property="preferredSize">
 <object class="java.awt.Dimension">
  <int>300</int>
  <int>300</int>
 </object>
</void>
<void property="bounds">
 <object class="java.awt.Rectangle">
  <int>0</int>
  <int>0</int>
  <int>300</int>
  <int>300</int>
 </object>
</void>
<void property="layout">
 <object id="SpringLayout0" class="javax.swing.
 SpringLayout">
```

```
  <void method="addLayoutComponent">
   <object idref="JTextField0"/>
   <object class="java.beans.Expression">
    <object idref="SpringLayout0"/>
    <string>getConstraints</string>
    <array class="java.lang.Object" length="1">
     <void index="0">
      <object idref="JTextField0"/>
     </void>
    </array>
    <void property="value">
     <void property="x">
      <object class="javax.swing.Spring" method="
constant">
       <int>60</int>
      </object>
     </void>
     <void property="y">
      <object class="javax.swing.Spring" method="
constant">
       <int>105</int>
      </object>
     </void>
     <void method="setConstraint">
      <string>East</string>
      <null/>
     </void>
     <void method="setConstraint">
      <string>South</string>
      <null/>
     </void>
    </void>
   </object>
  </void>
  <void method="addLayoutComponent">
   <object idref="JLabel0"/>
   <object class="java.beans.Expression">
    <object idref="SpringLayout0"/>
    <string>getConstraints</string>
    <array class="java.lang.Object" length="1">
     <void index="0">
      <object idref="JLabel0"/>
     </void>
    </array>
    <void property="value">
     <void property="x">
      <object class="javax.swing.Spring" method="
constant">
       <int>72</int>
```

```
      </object>
     </void>
     <void property="y">
      <object class="javax.swing.Spring" method="
constant">
       <int>174</int>
      </object>
     </void>
     <void method="setConstraint">
      <string>East</string>
      <null/>
     </void>
     <void method="setConstraint">
      <string>South</string>
      <null/>
     </void>
    </void>
   </object>
  </void>
  <void method="addLayoutComponent">
   <object idref="JPanel0"/>
   <object class="java.beans.Expression">
    <object idref="SpringLayout0"/>
    <string>getConstraints</string>
    <array class="java.lang.Object" length="1">
     <void index="0">
      <object idref="JPanel0"/>
     </void>
    </array>
    <void property="value">
     <void property="x">
      <object class="javax.swing.Spring" method="
constant">
       <int>0</int>
      </object>
     </void>
     <void property="y">
      <object class="javax.swing.Spring" method="
constant">
       <int>0</int>
      </object>
     </void>
     <void property="width">
      <null/>
     </void>
     <void property="height">
      <null/>
     </void>
     <void method="setConstraint">
```

```
        <string>East</string>
        <object class="javax.swing.Spring" method="
  constant">
         <int>0</int>
         <int>0</int>
         <int>2147483647</int>
        </object>
       </void>
       <void method="setConstraint">
        <string>South</string>
        <object class="javax.swing.Spring" method="
  constant">
         <int>0</int>
         <int>0</int>
         <int>2147483647</int>
        </object>
       </void>
      </void>
     </object>
    </void>
   </object>
  </void>
</void>
<void property="glassPane">
 <void property="bounds">
  <object class="java.awt.Rectangle">
   <int>0</int>
   <int>0</int>
   <int>300</int>
   <int>300</int>
  </object>
 </void>
</void>
<void property="layeredPane">
 <void property="bounds">
  <object class="java.awt.Rectangle">
   <int>0</int>
   <int>0</int>
   <int>300</int>
   <int>300</int>
  </object>
 </void>
</void>
<void property="name">
 <string>frame0</string>
</void>
<void method="addWindowListener">
 <object class="apple.laf.AquaRootPaneUI"/>
</void>
```

```
  <void property="visible">
   <object idref="Boolean0"/>
  </void>
 </object>
</java>
```

## A.2 The SimpleBean and SimpleBeanObject classes

Listing A.2: SimpleBean.java

```
//
//  SimpleBean.java
//  SimpleBean framework
//
//  Created by Balder ¿Mrk on Tue Feb 24 2004.
//  Copyright (c) 2004
//
package simplebeans;

public interface SimpleBean{
    public void setValueSBO(SimpleBeanObject value);
    public SimpleBeanObject getValueSBO();
}
```

Listing A.3: SimpleBeanObject.java

```
//
//  SimpleBeanObject.java
//  SimpleBean framework
//
//  Created by Balder ¿Mrk on Wed Feb 25 2004.
//  Copyright (c) 2004
//

package simplebeans;


public class SimpleBeanObject{
    static final int TYPE_INT = 0;
    static final int TYPE_DOUBLE = 1;
    static final int TYPE_STRING = 2;
    static final int TYPE_OTHER = 3;

    Object o;
    int type;
    boolean array;
    public SimpleBeanObject(Object o, int type){
        this(o, type, false);
    }

    public SimpleBeanObject(Object o, int type, boolean
   array){
        this.o = o;
        this.type = type;
        this.array = array;
    }
```

```
    public Object getObject(){
        return o;
    }

    public int getType(){
        return type;
    }

    public boolean isArray(){
        return array;
    }
}
```

## A.3 Sample SimpleBean and corresponding BeanInfo

Listing A.4: SimpleTextField.java

```
//
//  SimpleTextField.java
//  SimpleTextField
//
//  Created by Balder ¿Mrk on Mon Feb 23 2004.
//  Copyright (c) 2004
//
package simplebeans;

import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.beans.*;
import simplebeans.SimpleBeanObject;
import simplebeans.SimpleBean;

public class SimpleTextField extends JTextField implements
    SimpleBean {
     private PropertyChangeSupport changes = new
    PropertyChangeSupport(this);
     private SimpleBeanObject valueSBO = new
    SimpleBeanObject("", SimpleBeanObject.TYPE_STRING);

     public SimpleTextField(){
         addKeyListener(new KeyAdapter(){
             public void keyReleased(KeyEvent e){
                 char inntastet = e.getKeyChar();
                 if(inntastet == KeyEvent.VK_ENTER)
                     changes.firePropertyChange("valueSBO",
    valueSBO, new SimpleBeanObject(getText(),
    SimpleBeanObject.TYPE_STRING));
             }
         });
     }

     public static void main(String[] args){
         JFrame j = new JFrame("Test");
         j.getContentPane().add(new SimpleTextField());
         j.setVisible(true);
     }

     public void setText(String text){
         String oldText = getText();
         super.setText(text);
         changes.firePropertyChange("text", oldText, text);
```

```
    }

    public String getText(){
        return super.getText();
    }

    public void setValueSBO(SimpleBeanObject s){
        valueSBO = s;
        switch(s.getType()){
            case SimpleBeanObject.TYPE_STRING:
                setText((String)s.getObject());
                break;
            // Other types handled here
            default:
        }
    }

    public SimpleBeanObject getValueSBO(){
        return new SimpleBeanObject(getText(),
    SimpleBeanObject.TYPE_STRING);
    }

    public void addPropertyChangeListener(
    PropertyChangeListener l){
        if(changes == null)
            super.addPropertyChangeListener(l);
        else
            changes.addPropertyChangeListener(l);
    }

    public void removePropertyChangeListener(
    PropertyChangeListener l){
        if(l != null)
            changes.removePropertyChangeListener(l);
    }

    public PropertyChangeListener[]
    getPropertyChangeListeners(){
        return changes.getPropertyChangeListeners();
    }

}
```

Listing A.5: SimpleTextFieldBeanInfo.java

```
//
//  SimpleTextFieldBeanInfo.java
//  SimpleTextField
//
//  Created by Balder ¿Mrk on Mon Feb 23 2004.
```

```
//  Copyright (c) 2004
//
package simplebeans;

import java.beans.*;
import simplebeans.*;
import java.lang.reflect.*;

public class SimpleTextFieldBeanInfo extends SimpleBeanInfo
    {
    private static final Class beanClass = SimpleTextField.
   class;

    public BeanDescriptor getBeanDescriptor(){
        BeanDescriptor bd = new BeanDescriptor(beanClass);
        bd.setDisplayName("A␣simpler␣textfield");
        bd.setPreferred(true);
        return bd;
    }

    public PropertyDescriptor[] getPropertyDescriptors(){
        try {
            PropertyDescriptor text = new
   PropertyDescriptor("text", beanClass);
            text.setPreferred(true);
            text.setBound(true);
            PropertyDescriptor value = new
   PropertyDescriptor("valueSBO", beanClass);
            value.setBound(true);
            value.setHidden(true);
            PropertyDescriptor rv[] = {text, value};
            return rv;
        } catch (IntrospectionException e) {
            throw new Error(e.toString());
        }
    }

    public EventSetDescriptor[] getEventSetDescriptors(){
        try{
            EventSetDescriptor changed = new
   EventSetDescriptor(beanClass,
                                                          "
   propertyChange",
                                                          java
   .beans.PropertyChangeListener.class,
                                                          "
   propertyChange");
            changed.setDisplayName("Value␣changed");
            EventSetDescriptor[] rv = {changed};
```

```
            return rv;
        }catch(IntrospectionException e){
            throw new Error(e.toString());
        }
    }

    // Return descriptors for the connection wizard in the
    SimpleBuilder.
    // The wizard will ignore hidden methods. (setHidden(
    true)).
    public MethodDescriptor[] getMethodDescriptors(){
        try{
            Method getValueSBOMethod, setValueSBOMethod,
    propertyChangeMethod;

            Class[] getValueArgs = {};
            getValueSBOMethod = SimpleTextField.class.
    getMethod("getValueSBO", getValueArgs);
            MethodDescriptor getValueDescriptor = new
    MethodDescriptor(getValueSBOMethod);
            getValueDescriptor.setHidden(true);

            Class[] setValueArgs = {SimpleBeanObject.class
    };
            setValueSBOMethod = SimpleTextField.class.
    getMethod("setValueSBO", setValueArgs);

            MethodDescriptor[] rv = {getValueDescriptor,
                new MethodDescriptor(setValueSBOMethod)};

            return rv;
            }
        catch(Exception e){
            throw new Error(e.toString());
        }
                }


    public java.awt.Image getIcon(int kind){
        System.err.println("getting␣icon␣for␣
    SimpleTextField");
        java.awt.Image img = loadImage("SimpleTextField16.
    gif");
        if (img == null)
            System.err.println("Couln't␣load␣icon");
        return img;
    }

}
```