Final Thesis

PyMorphic - a Morphic based Live Programming Graphical User Interface implemented in Python

by

Anders Österholm

LIU-KOGVET-D--06/12--SE

23th August 2006
PyMorphic - a Morphic based Live Programming Graphical User Interface implemented in Python

Final Thesis

performed at Human Centered Systems division
in the Department of Computer and Information Science
at Linköping University

by

Anders Österholm

LIU-KOGVET-D--06/12--SE

23th August 2006

Examiner: Dr. Mikael Kindborg
Department of Computer and Information Science
at Linköping University
Programming is a very complex activity that has many simultaneous learning elements. The area of Live-programming offers possibilities for enhancing programming work by speeding up the feedback loop and providing means for reducing the cognitive load on the working memory during the task. This could allow for better education for novice programmers. In this work a number of systems with a shared aim of providing educational tools for scholars from compulsory level to undergraduate college were studied. The common approach in the majority of the tools was to use program abstractions like tangible morphs, playing cards, capsules for code segments, and visual stories. For the user these abstractions and tools offer better focus on the constructive and creative side of programming because they relieve the user from the cumbersome work of writing program code, but they also sacrifice some of the expressiveness of a low-level language.

A Live programming system, called PyMorphic, based on the Morphic model was built in the Python programming language. Two different solutions, based on the Wx toolkit for Python, were constructed and evaluated. The results show that Morphic and Python go well together because Python is a programming language that allows for compact and dynamic code. PyMorphic was evaluated with the cognitive dimensions framework and theories on cognitive load and working memory. A user attitude test was performed and the results showed that the users had a positive attitude towards the PyMorphic system. The PyMorphic project is open-source and it is hosted on Sourceforge. The code can be downloaded from the project web-site: http://pymorphic.sourceforge.net. Anyone is welcome to take part in further development of PyMorphic.
First I would like to thank my supervisor Mikael Kindborg for his invaluable assistance and guidance. I would also like to thank Tomas Larsson who became my primary discussion partner during the work.
# Table of Contents

## 1 Introduction

1.1 Motivation ................................................. 1
   1.1.1 Programming ........................................... 1
   1.1.2 Live programming ..................................... 2
   1.1.3 Python .................................................. 2

1.2 Scope ...................................................... 3
   1.2.1 Research Issues ....................................... 3

## 2 Theory

2.1 Cognitive Aspects of programming .......................... 4
   2.1.1 Cognitive load ......................................... 5
   2.1.2 Schemas ................................................ 8
   2.1.3 Program comprehension ................................ 9
   2.1.4 Learning barriers ..................................... 10
   2.1.5 Differences between expert and novice programmers 11
   2.1.6 Design-theoretical approaches to software design .... 12
   2.1.7 Programming activities ................................. 14
      The edit-compile-run cycle ............................... 14
      Designing ................................................ 16
      Coding ..................................................... 16
      Debugging ............................................... 17

2.2 Evaluating the usability of a programming system ....... 18
   2.2.1 The cognitive dimensions framework .................. 18
   2.2.2 Designing novice programming systems ............... 20

2.3 Program development systems .............................. 20
   2.3.1 Textual vs Visual representation of languages ....... 21
   2.3.2 Integrated Development Environments. ................. 22

2.4 The Python programming language ........................ 23
   2.4.1 Object orientation ..................................... 24
   2.4.2 The Wx library ........................................ 25
3 Live Programming
3.1 Rapid application development . . . . . . . . . . . . . . . . . . . . 27
3.2 Live programming systems . . . . . . . . . . . . . . . . . . . . . . . . 27
   3.2.1 Morphic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
   3.2.2 Alice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
   3.2.3 HANDS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
   3.2.4 JPie . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

4 PyMorphic Project
4.1 Philosophy and vision . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
   4.1.1 The intended use and user . . . . . . . . . . . . . . . . . . . . . . 37
   4.1.2 Design principles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
   4.1.3 Development Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.2 Evaluation framework . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3 Solution 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
   4.3.1 Object views . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
   4.3.2 Squeak as a role model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
   4.3.3 Design decisions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
   4.3.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
   4.3.5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.4 Solution 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
   4.4.1 Design decisions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
   4.4.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
   4.4.3 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

5 Discussion
5.1 Application areas for live programming . . . . . . . . . . . . . . . . . . . . . 56
5.2 Live programming in education . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.3 Graphical representations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.4 Usability in the design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.5 Morphic in Python . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.6 Lessons learned from PyMorphic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.7 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

6 Conclusion
6.1 Final word . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

Bibliography
1.1 Motivation

In order to educate computer programmers education in computer science must be stimulating and rewarding. Computer programming is a complex field and every programmer starts out as a novice and has to be trained. In the past decades vast research has been done in the area of creating programming systems that assist novices in learning how to program, see for example [PM96] for a summary.

The Live-programming approach to software design have existed as an alternative way to create programs, but has not gained much publicity outside it’s own community. An early User Interface (UI) for a direct-manipulation programming environments is Morphic. The interactive model of Morphic has never become a mainstream choice among programmers for various reasons, but with the growth of new ways of using computers, in education and for rapid prototyping, Morphic could possibly become a useful tool for users who value aspects of programming languages like speed and quick feedback.

1.1.1 Programming

Programming is an activity with many faces. Fundamentally it is process of mapping a problem from a problem-domain to a program-domain. Programming is in theory a very structured activity and structure is what theorists on the area have strived for. Empirical studies though, show that programming is more of an opportunistic activity [GP96, D01].

Programming is considered to be one of the most complex activities a person can undertake. Developing expertise in the area takes time. A software designer has to simultaneously jump between many levels of abstraction in order to produce good quality software. Programming has traditionally been done in cycles. First you write code, then you compile it and finally you run the program. This edit-compile-run cycle has existed since the birth of computers. It has not really been questioned if this is the
best approach to programming. Other ways of programming have emerged and one is Live-programming.

1.1.2 Live programming

Live programming (see Chapter 3) is actually a concept that has received different meanings. It has been used for describing the art of composing music by invoking algorithms during performance. Another use is the act in which a tutor constructs a program live in front of an audience. A third area that uses the term is programming by direct manipulation of objects. The creators of Morphic used the terms directness and liveness (see Section 3.2.1) to describe the philosophy and the aim of their work.

Direct-manipulation of running objects is not usually referred to as live-programming, rather as adding liveness to an application. In order to create liveness the program must be built in a way that allows for the source code to be updated during execution. This is different from the classic edit-compile-run cycle, where the program is edited before it is tested.

One of the avant-gardes in Live-programming and especially of the study of Live-programming is Alan Blackwell at the Cambridge University. Alan Blackwell is interested in studying the more unusual approaches and techniques in programming for the reasons expressed as:

The real benefit in studying unusual populations of programmers, whether they are representative of end-users or not, is that in addressing more unusual needs we may find more creative solutions.[BC05, p. 2].

This work will continue in this spirit by introducing new solutions to the art of Live-programming with the hope of strengthening the view of programming as a very creative activity.

1.1.3 Python

Python (see Section 2.4) is becoming a popular language for education. It is a dynamic language that should be suitable for Live-programming, but few Python systems take advantage of this. Python is an interactive programming language in which it is possible to interact with an interpreter. This offers the programmer the possibility to interactively work with the program and thus not having to follow the edit-compile-run cycle. Python offers wide introspection features which make Live programming possible.
1.2 Scope

The scope of this work is to survey the area of Live-programming and to contribute to the community with an implementation of a Morphic-like system in Python. In contrast to the work of my peer precursors in the area [Lyn04, Sve04], this work begins in the human cognition and the constraints that the information in the user interface creates on the knowledge acquisition process of programming. The area of cognitive science and education will be combined with that of computer science, especially object oriented programming. A part of the work will handle the general usability considerations (see Section 2.2), but this will not be done by following established usability heuristics. Instead general usability considerations concerning extrinsic cognitive load and a cognitive dimensions framework will be used.

It is very important early in any study like this one to handle and decide on the issue of whether to view programming as an individual and as a social activity. This work focuses on the interaction between one programmer and the computer. Collaborative programming activities are not addressed.

1.2.1 Research Issues

This work will result in a Morphic-like system implemented in Python. The system design will make use of different theories of programming. The aim with this work is to bring an answer to the following questions:

- How is good usability reached in the design?
- How can a PyMorphic system be designed and implemented?
- Is Python a suitable language for implementing a Live-programming Morphic-like system?

The central concept in this work will be Live-programming. The PyMorphic project will be based on Live-programming theories. One part of the work will define the concept of Live-programming as it will be used in the development of PyMorphic. There are two questions that will be used to elaborate on to the concept: Which are the application areas for Live-programming and which live-programming tools exist? How is Live-Programming used to make education in programming more effective?
Programming is a word that is associated with many different activities. Consequentially a definition of programming has to be broad and general. As Alan Blackwell notes, programming is seldom defined in modern research publications [Bla02]. Nevertheless, a general definition can be taken from Pair stating that programming is

the activity in which one transforms specifications that describe a function into a program [Pai93].

that is, a text that can be interpreted by a machine in order to calculate that function. According to this definition programming is about making the computer do what you want it to do in order to calculate the result you want. Brooks use the term program design [Bro99] to refer to basically the same activity as programming. He chose to describe program design as the mapping between problem domain and program domain. The two words programming and program design will be referred to in this work as synonymous. Programming will in some cases be used as a more general term when it includes software design as an activity. In a few places software design will be used as a more general and abstracted term than programming. This will be the case if programming is used in the sense of the physical writing of program code.

2.1 Cognitive Aspects of programming

From a cognitive-science point of view the interesting aspects of the programming area are the human problem solving and knowledge acquisition aspects involved in programming.

Francoise Détienne [DÔ1] has studied and summarized research in the area of cognitive aspects of software design. She concludes that there are many different theories and more or less productive approaches applied in the study of the human cognitive side of programming. One of the most productive approaches has been to compare the work of novices and experts.
Many of the studies reviewed in her work have focused on how programmers organize their activities and strategies for program comprehension.

According to theories on cognitive load (see Section 2.1.1) the success of any knowledge acquisition task is dependent on the cognitive load put on the learner. An important part of usability engineering is to avoid too much cognitive load in the user’s tasks. Section 2.1.1 reviews the theories of cognitive load and its dependency on the working memory. Section 2.1.2 will go into the theories of the fundamental units of symbolic knowledge - the schema, the relation between schemas and expertise, and the implications for programming. Later in Section 2.1.2 schemas are used to describe how working memory and cognitive load affects knowledge acquisition and the differences between novices and experts. Section 2.1.3 gives an overview of the different theoretical frameworks that have been applied to the comprehension and understanding of computer programs. Section 2.1.4 surveys theories on Learning barriers and Section 2.1.5 broadens the view on what distinguishes experts from novices in programming. Section 2.1.6 approaches the programming activity from a design-theoretical perspective.

2.1.1 Cognitive load

The cognitive load theory [Swe94] was introduced to study the cognitive processes of learning and knowledge acquisition. The cognitive load theory identifies the working memory as both a prerequisite and a bottleneck for learning.

The general memory capacities of humans have no limit. No person has ever filled their memory completely. However, following the theories of short-term memory (or the newer working memory) and long-term memory it is obvious that the long-term memory is possibly unlimited, but that the working memory on the other hand is very limited. Famous tests have shown that the working memory can only hold 7±2 objects simultaneously [Mil56]. Typically, material remains in the short-term store for about 30 seconds, if it is rehearsed [Ste05]. Some theories have suggested that information is also stored acoustically, rather than visually. This has however been debated and the latest theories suggest that different systems connected to different senses work together. The bottleneck of the working memory has led to a general agreement that many human errors in problem solving and decision making can be traced to working memory failure [And84].

The term working memory was introduced by Alan Baddeley (see for example [Bad92] as a way to integrate the short-term store and the long-term memory. According to Baddeley and most memory researchers today [Ste05], the working memory is an activated part of the long-term memory.
Previous theories have seen the short-term memory as an own and separate storage with a number of slots to be filled. According to the old theories, elements in the short-term memory could be replaced by other elements from the long-term memory and thus; causing a memory failure if the old information was needed. In the working memory theory a failure occurs when the activation level of the element goes below a threshold value.

Some theories suggest that the total activation of the working memory is constant and that activation is shared between the tasks of maintaining elements and processing information. Given this constraint, a total processing and storage demand for activation that exceed the total amount of working memory capacity has two effects. One is that it slows down processing of thoughts. The other is that it causes a sort of forgetting. The sharing of resources means that the memories in the working memory can easily be lost by distracting information. If a person for instance is forced to read certain facts from one display and keep these facts in mind while we switch to another display, the load on the short term memory can become heavy. Perhaps we can’t even manage to keep the information in our memory, and thus have to switch back and forth between the two displays. If such a situation demands that we have to remember key combinations or commands to manage the computer, the interference with the short term memory will be considerable.

Cognitive load and learning

The process of learning requires the working memory to be actively engaged in the comprehension (and processing) of instructional material to encode to-be-learned information into long term memory. A heavy burden on the limited working memory can inhibit learning.

According to the cognitive load theory schemas (Section 2.1.2) can serve a function of reducing the burden on the working memory. A fluent reader can rapidly read a text because all the words correspond to previously acquired schemas. Compared to the unskilled reader the fluent reader might bypasses the working memory and instead make direct use of our superior long-term memory stores.

The cognitive load theory does not explain how learning takes place or how schemas are formed. The aim is rather to explain what hinders learning from taking place.

General cognitive load can be divided into intrinsic load and extraneous load. The total cognitive load is the sum of the two. The intrinsic load is the difficulty of the general task and is not dependent on the representation of data. Intrinsic load is dependent on the interactivity of learning elements. A learning element is a learning item in its simplest form. Learning elements that do not interact can be handled in isolation and causes
less intrinsic load. For example the task to close a program-window by pressing the X in the header of the window is independent of the program contents. However the task to write computer programs involves many interacting learning elements, especially for a novice user. The user has to simultaneously handle both a set of dependent syntax elements, complex rules and semantics of the programming language. The extraneous load is dependent on the presentation of the data. This addition to the total cognitive load is affected by the computer interface design.

The important thing to have in mind when addressing cognitive load is that if a task has a heavy intrinsic cognitive load from a high level of element interactivity and a heavy extraneous load because of poor instructional design, then learning is likely to be inhibited.

**Implications for the design of educational tools**

Sweller have applied the theory of cognitive load on different areas, with a general focus on the instructional processes of learning tasks. The theories have also been applied to the task of programming; see for example [Hee01].

Among the instructional techniques that have successfully been developed from the cognitive load theory can be found in [Coo98]:

- **The goal free effect**: In many training exercises for math, you have the goal and the given. You then follow a multiple step procedure from the given to the goal. The goal, the given and the possibilities in between easily overflows the learners working memory. If the learner has only the given and the task to go in any direction the learner performs and learns much better.

- **The worked example and problem completion effect**: If students are going to build proper knowledge schemata they benefit greatly from having worked examples to study.

- **The split attention effect**: Students should not have to look up related information at different places. For example: when working with a textbook and the computer screen at the same time, or when the explanation of the elements in a diagram is put as text outside the diagram instead of explaining these elements directly with notes on the diagram.

- **The redundancy effect**: This technique is about not showing the same content on more than one place on the screen which causes unnecessary confusion.

- **The modality effect**: Findings show that the working memory can be expanded by the use of multiple modalities of information. For
example: Information could be presented as a combination of visual and audible modes.

2.1.2 Schemas

The fundamental unit of symbolic knowledge is the concept [Ste05]. Concepts can often be captured in single words, like house, and different concepts are related. One of the main approaches to understand how concepts are related in the mind is through schemas. A schema is a mental framework for organizing knowledge, which creates a meaningful structure of related concepts. Sternberg presents the fundamental characteristics of Schemas:

- **A schema can include other schemas:** A schema for animals includes for example a schema for cows, a schema for apes etc.

- **Schemas encompass characteristic features rather than rules:** Mammals typically have fur, but humans do not and are still considered to be mammals.

- **Schemas come in different levels of abstraction:** A schema for justice is more abstract than a schema for fruit.

Schemas also include information about relationships. Some of the most interesting relationships for cognitive psychologists are the casual (“if-then”) relationships. These prevent us from breaking fragile things by telling us that they can break if they are for example dropped. The information schemas can also be used to draw inferences in novel situations. Many theories on expertise state that the expert approaches a novel situation with a set of schemas that can successfully be applied.

Programming schemas

The theory of schemas has been widely used to describe knowledge and expertise in programming [D01]. Schemas in programming are thought to be created from the experience that comes from solutions to earlier problems. These schemas are applied to new situations that resemble a previous problem. Studies summarized by Détienne show that expert programmers keep programming schemas in their memory. Détienne classifies these schemas into:

- **Elementary programming schemas:** These are built from the user’s frequent use of common control structures like loops. They have been described as having slots that need to be filled with information. For a schema that handles loops there is a goal, an initialization value, an update condition and an end condition slot.
2.1 Cognitive Aspects of programming

- **Algorithmic schemas:** These are related to elementary schemas but used for more advanced algorithms and often composed of several elementary schemas.

- **Tactical and strategic schemas:** These are schemas that are language independent.

- **Implementation schemas:** These are dependent on specific programming languages.

- **Structural schemas:** In studies of the production and understanding of text, schemas referred to as structural schemas are often used to describe the process. The term superstructure is used to describe the way we remember how different types of texts are structured. In the programming field the term role has been used to describe something similar to structural schemas. A structural schema for classes of object-oriented programs is composed of the roles: creation, initialization, reading access, writing access, input and output.

Schemas offer very tangible and plausible explanations to many cognitive processes. This can itself be a problem since the easy solution that Schemas can give rise to stereotypes [Ste05]. Another thing about that need to be considered about schemas is that the theory of schemas is somewhat limited by the fact that it does not focus on the processes that generate schemas [D01].

**From novice to expert**

Different studies from different domains (e.g. chess, Go, radiology and physics) all show that experts used larger and more developed schemas which they applied on the problem [Ste05]. The schemas of experts also involve procedural as well as declarative knowledge. Another difference between experts and novices is that experts have reached a higher degree of automatization for solving the task. This has the effect that the working memory freed up and instead the long-term memory is used. This enables experts to better monitor their progress and their accuracy during problem solving. In conclusion: In order to become an expert in an area, two processes are required; **schematization** and **automatization**. Schematization involves developing rich, highly organized schemas. Automatization is a process which involves consolidating sequences of steps into unified routines that require little or no conscious control.

**2.1.3 Program comprehension**

To be able to add code to a computer programs one needs to be able to comprehend the code already written in the program. Green and Petre
[GP96] have found that at the coding stage, the programmers develop their code in small segments which is knitted in to what has been written so far. Because of this behavior programmers need to read and understand what has been written so far, in order to knit new material in. They called this the ‘parsing-gnisrap’ cycle (gnisrap = parsing backwards).

The different theories on how programmers comprehend programs are central to the cognitive aspects of programming. Détienne focus on the textual representation of computer programs and notes that researchers who have studied the understanding of software have generally considered a program as a text. Therefore many of the theoretical frameworks that were developed for text understanding have been borrowed. Her survey of the area led her to the following categorization of the different theories of program comprehension.

- **Program comprehension as text understanding:** Programs that are created with a textual language are represented for the programmers as written and readable text. One related area that can be applied to program comprehension is the theories on general text comprehension.

- **Understanding by constructing a network of relations:** This can be further divided into propositions or schemas. In the propositions view the comprehension of computer programs can be described in terms of a small number of control structures: sequence, iteration, and selection. These fundamental units are often referred to as prime programs can be decomposed into them. The schemas theories have not been explicitly in the study of program comprehension since they origin from the study of general written texts.

- **Program comprehension as problem solving:** Another way to look at program comprehension is to apply the analogy of problem solving. The programming task is composed of solving big and small problems on different levels.

### 2.1.4 Learning barriers

One approach in the study of education of novice programmers is to study the learning barriers that programmers encounter. Research has identified several aspects of programming languages that pose barriers for programmers. Andrew J Ko et. al. [KMA04] have identified six learning barriers in End-user programming systems:

- **Design barriers:** The programmer doesn’t know how to translate the problem to the program domain.
• **Selection barriers**: The learner can not determine which programming interfaces are capable of a particular behavior. One way to overcome this barrier working examples may help in reaching the “worked example and problem completion effect” (see section 2.1.1 on cognitive load).

• **Coordination barriers**: The systems limits on how different programming interfaces can be combined to achieve complex behaviors.

• **Use barriers**: The system obscures how different interfaces can be used and which effect such use will have.

• **Understanding barriers**: The programs external behavior that does not reveal what the program did do and what it did not do. The understanding is inhibited when the external behavior does not match the user’s expectation.

• **Information barriers**: The programming environment does not provide sufficient information about a program’s internal behavior. The places to search for the information can easily become copious.

The lowest common denominator of the list of barriers is that they are all affected by the design of the GUI and in the presentation of information in the programming system.

### 2.1.5 Differences between expert and novice programmers

In the section about programming schemas 2.1.2 a discussion was started about expertise and a definition of expertise as the possession of strong and complete knowledge schemas was presented. In this section the discussion is broadened. There is a need for a broader discussion because in order to become an efficient programmer the user needs more than the right schemas. According to Détienne schemas are limited because the do not address how the knowledge from the schemas are put in use and how learning takes place. A more strategy oriented approach is needed to understand this. This section and the next, which uses design theory to explain the cognitive aspects of programming, will discuss further on the differences between expert programmers and novices.

The difference between experts and novices is very apparent in the programming activity. Studying these differences is very rewarding since the differences are very tangible and easily measured.

Like in other areas, becoming an expert in programming involves the processes of schematization and automatization. Recall tests have shown
similar results in programming code as in chess-positions and other domains. An expert can recall the code better if they are presented in correct and coherent order. Research on the area of expertise in computer programming has shown that the following aspects are what separate novices from experts. The survey done by Détienne [D01] summarizes that:

- Experts construct more complete representation of the problem before they start the process of solving it.
- Experts use more rules of discourse.
- Experts are capable of generating several alternative solutions before making a choice.
- Experts use more external devices, particularly external memory such as drawings on papers.
- Experts design strategy is top-down and forward for familiar and not too complex problems, while novices is bottom-up and backward.
- Some aspects of programming tasks are carried out completely automatically by experts.

Novice programmers need feedback from evaluation on small amounts of code where experts can code larger amounts between evaluations [GP96]. A study of novice LISP programmers showed that novices need ‘progressive evaluation’ [AS84].

2.1.6 Design-theoretical approaches to software design

Programming can be seen as a design activity. The fact that the problems to be solved are ill-defined makes design-theories applicable. Design is an activity where a designer deals with ill-defined problems and communicates with a situation and redefines the problem in order to be able to produce a concrete result that meets the many requirements on many levels. Designing is a very knowledge-intensive activity where the designer has to attain to many activities, and constantly evaluate the work. Design is an iterative process where iterations are made up of the creation of solutions and the evaluation of the same solutions. This process often causes a heavy cognitive load and designers need to find ways to deal with this. Détienne have found that different forms of externalization of memory can come in handy:

Like writers, program designers make many notes. This is linked to the iterative nature of the process and to the limited capacity of the working memory, whence the need for an
external memory. It is also linked with the need for programmers doing maintenance work to understand why a particular design was adopted [DÔ1, p. 33].

In the study of program design several different approaches have been used based on different view-points. Détienne categorizes the different approaches into Knowledge-, Strategy- or Organization centered.

Knowledge-centered approaches Many studies referred to by Détienne have seen the knowledge the programmer possess as the variable that decides how effective the outcome of the software-design process will be. The approach has been very influences by the AI-area and the human memory has been referred to as a data-storing and retrieving object. The experiments that have been conducted have focused on programmers that are supposed to have knowledge about programming and the interesting aspect of study has been the way the knowledge is used in programming. The theories of Schema (see Section 2.1.2) have been widely used to describe the knowledge of experienced programmers.

Strategy-centered approaches The strategy-centered approaches are less focused on expert’s knowledge and more into how knowledge is put to use. Strategies for reaching a solution to a programming problem can be classified along several axes. They have been summarized by Détienne as either:

- **Top-down vs. Bottom-up:** Top-down development goes from more abstract to less abstract and bottom up vice versa. In the first case the programmer develops the solution at an abstract level and the refines it by adding more and more detail.

- **Forward vs. Backward:** Forward development means that the program is written in the same direction as the execution. Forward development has been associated with the application of known schemas by the programmer. Backward development is more frequently used in explorative work.

- **Breadth first vs. Depth first:** Breadth first development consists of developing the complete program at one level of abstraction before moving to the next.

- **Procedural vs. Declarative:** Procedural development focuses on the procedure and process of the solution and declarative on the parts of the solution like objects and roles.
**Organization-centered Approaches** The organization-centered approaches focus on how the programmer organizes his/her activities. The theories on the area have focused on the hierarchical organization of the work where the programmer jumps between different levels of abstraction when working out a solution. The more empirically based studies have focused on the opportunistic organization of software design. Program designers often organize their work with a hierarchical model of the task and a solution plan. A solution plan is a sequence of actions in a program which will achieve the goal of the program. A plan for a program that calculates a mean value can consist of calculating the sum and counting the number of elements and a second calculation that calculates the mean.

The work however is more of opportunistic character according to Green and Petre. There is not consensus among researchers on this point however [PM96]. Nevertheless the opportunistic character of programming has been shown in different studies. The opportunistic strategy can be triggered by many different factors such as resource limitations for example in the working memory. Information about the hierarchical plan and the way it is being followed is lost when the capacity of the working memory is exceeded. One of the reasons for the designer to leave the hierarchical plan may be the economic utilization of the resources available. When required information for one part of the solution is not available, the programmer puts it to the side. The solution is often developed with many parts being prematurely developed compared to the hierarchical plan.

### 2.1.7 Programming activities

The development of a program does not take place in a linear fashion [GP96]. Programmers neither write a program in text order from start to finish, nor work top-down from the highest mental construct to the smallest. In the following sections the division of programming activities has been divided into designing, coding, debugging and documenting. The division could have been done in many other ways, for example programming could be divided into analysis-design-implementation or the activity of testing or refactoring or even maintenance tasks could be included. The following division of activities is used because it focuses on the focuses on the user’s activities and the interaction with the program.

**The edit-compile-run cycle**

Classical programming activities in opposition to Live-programming activities (Chapter 3) can naturally be explained as composed of cycles. The programmer creates program code according to some more or less explicit design. Then the user needs to test the code and invoke the compiler for the programming language to compile the program into machine code so
that it can run on the computer. When all syntax errors are corrected the program is run by the user and the user performs tests to assure that the program is free from semantic errors. This ends the cycle and the user may start on a new cycle by editing the program and adding functionality or correcting semantic errors. The edit-compile-run cycle is a natural way to organize the programming process. It is driven by the creative ideas on how to realize an idea and to check if the output is of the desired type. The larger the program is, the more time will be spent on testing. When testing a program there are many different possible methods available. In good programming practice, a complete automated testing suite is desirable. The thing about testing is that some problems are of the “Wicked kind” This means that the correctness of the solution can not be tested formally since the correct solution changes with new knowledge about the task. This is often the case in the design task. Nevertheless when the user is testing a program’s functionality the program’s state must be set to the desired for the testing. This can involve several actions that have to be conducted by the user. This is especially the case for beginners whose work is exploratory, and who have not reached the expertise, and who don’t posses the tools to write a good automated testing suite, to create a complete testing platform for the program. For these users it can be very cumbersome to perform a set of operations needed to test a small part of the program.

Goldman has studied novices and observed that the compounded and delayed feedback that the edit-compile-cycle imposes on the process is very problematic:

The problems are compounded by delayed feedback that results from the edit-compile-execute cycle. Inexperienced programmers are fearful of describing things incorrectly (“What should I type?”). Inevitably, when they do describe things incorrectly, they become frustrated (“That’s not what I meant!”). To make matters worse, delayed feedback can result in a student investing significant time and effort, perhaps making similar errors repeatedly, before discovering the problem. A student in an introductory programming course may become so bogged down in details to miss the important ideas and reach the mistaken conclusion that computer science is nothing more than arcane syntax.[Gol04a, p. 5-6].

The results from Goldman’s studies suggest that there is a problem with the learning elements in the classic way of designing software in an edit-compile-run fashion.
Designing

The concept of design is in itself a container for other concepts. As mentioned in the beginning of this chapter the use of the term design as a programming activity is a little problematic and it needs a little elaboration. Design is usually referred to as an activity in a software engineering discourse. Programming is usually seen as a lower level activity than design and as part of the implementation phase of a software engineering process. This has however changed in the more agile and iterative approaches to software engineering where design and implementation are integrated. For more information see for example [Bec99].

In this section the term design is used to describe a lower level and less complex activity than software engineering. Design is here being used the way Brooks [Bro99] use it to describe program design the mapping between problem domain and program domain. Brooks view of program design has been reinforced by subsequent research referenced in [GP96]. Design as it is used here is the first step in every iteration of the edit-compile-run cycle. This is not a tremendously solid definition, since design is not equal to part of the “edit”, “compile” and “run” categorization. However in the edit stage the code needs to be invented and the program needs to be designed. Designing is about coming up with solutions. When the ideas are concrete enough the user place their hands on the keyboard.

Coding

This activity involves the actual generation of program code. It is difficult to draw the two overlapping activities generating and maintaining code. The problem of division of activities is very typical for programming. Andrew Ko have found that many new tools for programming focus on the creation of code, although research have shown that 60-90% of software development costs are taken up by reading and navigating through code, which is referred to as maintenance tasks [KAM05].

The results of Ko’s own study where programmers were observed while using a modern Integrated Development Environment (IDE) show that maintenance work interleave three fundamental activities [KAM05]:

- Collection of a group of task-relevant code fragments, called a working set
- Navigation of these code fragments dependencies such as uses and declares relationships
- Repairing or creating the necessary code.

According to Ko’s results there are many trends that can be identified. One is that programmers spend an average of 35% of their time navigating
2.1 Cognitive Aspects of programming

between dependencies. As much as 46% of the time spent is used to inspect task-irrelevant code. These figures are quite high and the give good motivation to the design of new tools.

**Debugging**

Humans are not perfect creatures. Not even programmers are. An important and major part of programming will always be to figure out why things didn’t go as the imperfect programmer expected. According to the classic “Brook’s rule” [Bro75] more than 50% of all time and effort in a software project is spent in testing and debugging activities.

Much of research has been done in the area of debugging. Many of the researchers agree that although the programming systems have evolved in the past decades, the debugging systems are still following the same principles as they did 30 years ago. The tools that programmers have for doing debugging are breakpoints, code-stepping and print-statements. This suggests that the area of debugging has either reached perfection or has been somewhat neglected. Henry Lieberman at the MIT Media Library at one point got so frustrated about what he called “The debugging scandal” [Lie97].

We’re sad to report that there is not as much related work in this area as there should be. Even recently implemented programming environments seem to provide only the same set of tools that have been around for the past 30 years: trace, breakpoints, stack inspection, and perhaps a line-by-line stepper [LF98, p. 18].

In the years before and after Lieberman showed his frustration the research area the search for better debugging tools has certainly not been neglected. Henry Lieberman himself designed the ZStep 95 debugging system accompanied by Christopher Fry [LF98]. ZStep 95 keeps a complete history of the computation so that the program can be run backwards for debugging, animates the source code of the program as it executes, and keeps a correspondence between expressions, their values, and graphic output.

Many other different approaches have been tested including automatic debugging, relative debugging, program slicing [Wei82] and visualizations. According to Andrew Ko and Brad Myers [KM04] few of these attempts have been very usable or able to reduce debugging time. Myers Pane and Ko suggest that programmers make errors when they suffer long chains of breakdowns [MJFP04]. Based on this hypothesis they have directed their work on debugging to the prevention of breakdowns in debugging.

Their own tool is called Whyline [KM04]. It is based on the idea that the debugging activity starts with a question and the debugger is supposed to be able to reply to given questions.
2.2 Evaluating the usability of a programming system

Into the mid 80’ies the study of programmers and the programming activity consisted much of making laboratory tests with dependent and independent variables. The test persons were mostly students which as a consequence did not give validity to other programming contexts. Détienne [Dó1, p. 9] summarizes the problems in three points:

- the small size and low complexity of the problems used;
- the straightforward nature of the programming environment used;
- the purely individual character of the activity studied

The effects of these issues were that the collaborative aspects of programming were not considered. Also the straightforward nature of the task meant that the specification and design parts of programming were not considered although they often make up a bigger part of the programming activity than the source code editing. In recent years new research has focused more on these issues and there has been a shift in the paradigm for that reason.

2.2.1 The cognitive dimensions framework

Green and Petre have purposed a framework to be used as an evaluation technique for specifically evaluating visual programming environments [GP96]. The framework has been used in numerous studies (see i.e. [Bla02]). Green and Petre are very careful about pointing out what their cognitive dimensions is not a set of guidelines that should be followed in order to succeed. They are rather discussion tools, intended to raise the level of discourse. The framework is as they themselves point out broad brush, meaning that it is not a set of heuristics that are concerned about the details. They are rather task-specific and rest on a common proto-theory.

The authors suggest that the framework is intended for developers who make design decisions in their implementation work. It supplements the highly specific analyses typical of contemporary cognitive models in HCI.

Many protagonists of HCI have tried the direct route; they have explicitly attempted to develop methods of design. Such is not our aim. Indeed, we feel it would be impertinent to suggest that cognitive psychologists can tell professional designers what to do; we have no pretensions that we can design programming languages [GP96, p. 2].
2.2 Evaluating the usability of a programming system

The framework consists of a number of cognitive dimensions [GP96, p. 8-9]:

- **Abstraction gradient**: What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?

- **Closeness of mapping**: Programming requires mapping between a problem world and a program world. The closer the programming world is to the problem world, the easier the problem-solving ought to be.

- **Consistency**: When some of the language has been learnt, how much of the rest can be inferred?

- **Diffuseness**: How many symbols or graphic entities are required to express a meaning?

- **Error - proneness**: Does the design of the notation induce ‘careless mistakes’? Hard mental operations: Are there places where the user needs to resort to fingers or penciled annotation to keep track of what’s happening?

- **Hidden dependencies**: Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?

- **Premature commitment**: Do programmers have to make decisions before they have the information they need?

- **Progressive evaluation**: Can a partially-complete program be executed to obtain feedback on ‘How am I doing’?

- **Role - expressiveness**: Can the reader see how each component of a program relates to the whole?

- **Secondary notation**: Can programmers use layout, colour and other cues to convey extra meaning, above and beyond the ‘official’ semantics of the language?

- **Viscosity**: How much effort is required to perform a single change?

- **Visibility**: Is every part of the code simultaneously visible (assuming a large enough display), or it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

The different dimensions should ideally be used to describe the structure of the artifact. The analysis should be task oriented and different usage
like safety-critical or exploratory design will require different profiles which are set up with the different dimensions.

The cognitive dimensions framework has as it’s main purpose not to take too much time from the design process. In a few hours a developer should be able to make a good-enough analysis of the situation.

2.2.2 Designing novice programming systems

The caption of this section is inspired by the title of a work by Pane and Myers [PM96]. The two authors have summarized most of the work on usability in programming system that has been done at the Carnegie Mellon University. The comprehensive report contains numerous references to other studies in the field. In the section on user control and freedom they agree with the cognitive dimensions framework to what is important in what should be strived for. A part of the report deals with the program metaphor. According to the authors the choice of metaphor is very important. A metaphor must be easy to grasp. It is also important that the system is consistent with it’s metaphorical model. The authors also point out that metaphors are never perfect and difficulties will be encountered if too much is drawn from the metaphor. A badly chosen metaphor will also seem arbitrary for the users and make the use of their external knowledge from other areas more difficult to apply.

2.3 Program development systems

This section describes the most common paradigms in how to represent programs in different programming systems. The representation of the program and the programming language forms the interface between the programmer and the computer and provides information about data to the programmer. As Green and Petre point out:

- data is not information. Data must be presented in a usable form before it becomes information [GP96, p. 4].

In a humanistic perspective the interesting aspect is how different representations of computer programs are understood by users. The most common way to represent a computer program is with written instruction in a programming language. Another popular way to represent programs is with graphical metaphors. The next section will present these different approaches.
2.3 Program development systems

2.3.1 Textual vs Visual representation of languages

The most common visual representation of a computer program is the lines of text that are written according to the grammatical rules of the programming language. The following line in Python should be interpreted as a call to the print function which is passed the argument of the string “Hello World”.

```python
print 'Hello World'
```

When reading a text file, the interpreter of the program reads the program line by line and interprets the instructions.

Programs that appear like text on the screen are categorized into textual languages. There is also another type of programs: the visual programs. They are usually represented graphically for the user with boxes connected with lines. This makes it possible to illustrate the different states the program can be in. There are many possible advantages with visual programming languages. Research on the area however has shown that visual programming languages do not improve the programmer’s efficiency [GP96]. Therefore the dominant languages today are the ones that have a textual representation.

Colleges and universities generally have not adopted visual languages. This is not only because most professional programmers prefer textual languages over visual languages for general-purpose programming, but more importantly because the thought process involved in constructing programs within visual languages is often so strikingly different from that used in writing textual programs that knowledge transfer would be limited [Gol04a, p. 4].

The different activities of programming (editing, running, debugging) require different representations. When the program is executed it often launches it’s own interactive graphical interface that reacts to the users actions.

Another way to represent a computer program is by visualizing the flow through a flow chart. This can be accomplished by interpreting both textual and visual languages. The representations then show different possible executions of the same program. One approach is to visualize a textual programming language with a visual part to complement the view.

As long as the program runs without errors there is no apparent need to show the textual representation of the program, except for comprehension reasons. However, if the program has errors, or bugs, there is a need for debugging. In debugging mode the program is often represented as both the running application and the code-view where the current line that is
being executed is marked. Additional data of the call stack and heap are usually also represented. There has been some research done on other ways of representing and visualizing the program during execution for the purpose of debugging, see for example [LF98].

2.3.2 Integrated Development Environments.

Ten years ago programming systems were relatively simple. A system mostly consisted of the compiler or interpreter and a text editor that possibly had some syntax highlighting. The compiler was often launched from a shell and had no graphical interface. In the years up till today much development of new systems and research on these has been done in the area.

As with other areas that did benefit from the advances in GUI’s in the 90’ties the programming environments evolved into systems referred to as Integrated Development Environments (IDE’s). Today every programming language sooner or later gets an IDE. The purpose of the IDE is to keep the necessary tools to support the different activities easily accessible for the programmer. These tools include a source code editor (which often has
syntax colouring and pop-up method suggestions), a compiler, a graphical user interface (GUI) builder, a class-browser and a graphical debugger.

Some IDE’s also have support for making UML Class-diagrams and for both generating code from these and reverse-engineering diagrams from code. Popular IDE’s with the above mentioned features are Microsoft Visual Studio and the open source Eclipse project (Figure: 2.2).

These environments, as several others less common, can handle different programming languages. The GUI is often created by using drag-and-drop techniques. The essential functionality of the program must still be coded by the programmer in a textual language.

2.4 The Python programming language

Python is an object-oriented, high-level general-purpose scripting programming language. The creator of Python, Guido van Rossum wanted Python to be a language for rapid development.
The language comes with a large standard library that covers areas such as string processing (regular expressions, Unicode, calculating differences between files), Internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programming), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, file systems, TCP/IP sockets) [pyt].

Python is considered an easy to use programming language. A program written in Python is generally shorter than a program with similar functionality written in Java. The reason for this is that Python is dynamically typed and that it has control structures with simple syntax. The language interpreter is also dependent on indentation in the code in order to identify code blocks.

2.4.1 Object orientation

Object orientation is the dominant design paradigm in computer programming today. Popular programming languages like Java, C++, C# and Ruby all support or require object oriented programming. The success and impact that Python has had is due to the fact that it follows the object oriented programming paradigm. Object oriented programming is usually considered to comprise the following features:

- **Encapsulation**: Objects reveal their “outside” through tangible behavior, but keep their “inside” hidden. Another way of saying this is that we do not need to know how an object works to know how it behaves. Python does not have data encapsulation. The programmer must follow naming conventions to make use of the feature.

- **Composition**: The behavior of an object may result from the collective behavior of objects from which it is composed or constructed.

- **Classes**: Groups of objects can share common characteristics or can be cloned from a common prototype. Objects are said to be instances of their classes.

- **Inheritance**: The behavior of a superclass can be “handed down” to (or inherited by) its subclasses. Python allows for multiple inheritance: a class can inherit from more than one superclass.

- **Polymorphism**: Object behavior in response to the same action or message may take many forms, depending on the class of the object. Python makes polymorphism particularly easy to write, since the language is not strongly typed: a variable can hold objects of any type or class, and the elements of a list can be of mixed classes too. The benefits from this freedom of use for the programmer come with an
inevitable trade-off though: a language that generates fewer compile-time errors tends to generate more run-time errors.

Another related paradigm is the object by prototype techniques that are used instead of classes. In the SELF programming language prototypes are used to build objects. The reason for this is that the creators wanted to make the experience of programming close to the general patterns of human reasoning. Concepts that are thought to be very fundamental in humans, like tangible objects and containers, are used as language metaphors.

One of the special characteristics of Python is its vast introspection features. The language has built in functions for accessing attributes of running objects. For example the `dir()` built in function returns a list with all the attributes of an object. In Python attributes can also be added to objects on-the-fly. This is not something that is available in many object-oriented programming languages and it is a feature that can come in handy in Live-programming.

### 2.4.2 The Wx library

Standard Python distributions include the Tkinter wrapper for the TK graphics package. The package supports the most common windows widgets. The downside of Tkinter is that it is becoming somewhat outdated. A newer graphics library that is considered to be more up-to-date is the Wx package. The Wx package is still an add-on package for Python, but many voices suggest that it should become part of future standard distributions.

If the developers choice is to go with Python there are still many different ways in which different Python add-on packages can come in handy.
Live programming is a quite loosely defined concept that has received different meanings. Alan Blackwell uses the concept when describing live coding as music composition [BC05]. By browsing the Internet the author of this report has found another use: the act in which a tutor constructs a program live in front of an audience. A third area that uses live programming, but refers to it as something else is the area of most interest to this work; the area that is more often is referred to as direct manipulation of running objects. The term direct manipulation was founded by Ben Shneiderman at a time when most computer interfaces were textual [Shn83].

Alan Blackwell who is avant-garde in new programming paradigms including live programming gives a concrete and general description of how a direct manipulation system should work.

In a direct manipulation system, the current status of the system should be continuously represented to the user, a single action should have a single visible effect in the representation, and restoring the state of the representation to that before the action should restore the situation. [Bla02, p. 3].

The founders of Morphic used the terms directness and liveness (see Section 3.2.1) to describe the philosophy and aim of their work. Direct-manipulation of running objects is not usually referred to as live programming, rather as a way to add liveness and directness by interactivity to an application. In this work the term Live-programming should be referred to as dynamic change of objects and functions in running programs. An important aspect that differentiates it from classical programming is that there is no necessary edit-compile-run cycle making up the programming activity. Live-programming is a good way to describe this phenomenon. In Live-programming the feedback cycle is shortened since changes in the source code are available in the current execution of the program.
3.1 Rapid application development

Rapid application programming is a methodology that focuses on speed and quick testing of ideas instead of building robust solutions from a fixed specification. The term Rapid Application Development shares a common philosophy with Live programming, but the concept in itself has become blurred and branched since it’s invention in the mid-80’s and will therefore be used with awareness in this work. One software development methodology that originated from RAD is the more contemporary term Agile Development and it’s instantiation in for example Extreme Programming. This use of the term focus is on the software engineering level since the agile approach on software engineering was created to come to terms with the problems of rigidness in earlier development processes. In the 70’ties software engineering was seen as a process of making complete specifications and then implementing those for a complete system. According to the agile approach software engineering is a knowledge process where the knowledge about the problem increases during the design of the solution and the flaws in the initial design are revealed. In the rigid waterfall model changes could not be made one the specification was written. In any programming system that is used for educational purposes the process will naturally be a knowledge process. Therefore there is a good motivation for making educational programming systems embrace agility and testing of ideas and hypothesizes.

A related concept to RAD is Continuous design. In the continuous design the program that is being developed is constantly admitting changes to the design. This differs from agile approaches like iterative design where design decisions are made at the beginning of the iterations. A robust live programming development system with dynamic classes allows for RAD in that it eliminates the process of restating the program after changes and restoring the program state to where the new code is put into use.

As mentioned in the previous section the art of composing music live for an audience could be considered to be called rapid application development.

In modern IDE’s there are many tools that speed up the process of for example refractory of code and GUI-design. One of the first tools that were considered to be a rapid application development tool is Visual Basic for Windows. Today many other programming systems also provide the kind of functionality that Visual Basic offer which is on of the reasons why the concept of Rapid application development has become blurred.

3.2 Live programming systems

Much work in Live programming concerns music composition [BC05]. This work will focus on direct manipulation of physical objects which is a dif-
Since programming is a very complex field that is difficult for beginners to grasp, much work has been done on implementing systems that should offer a gentle first introduction for children and students to programming. This work will not be able to cover all systems implemented. For a summary of many of the systems see [Pan02]. I will focus on the systems that build the foundation for this work which of course includes the different Morphic implementations. Other related systems that influence the work are Alice 3.2.2 and HANDS 3.2.3 and the JPie environment refsec:jpie which is implemented in Java.

3.2.1 Morphic

The Morphic system was initially created for the Self programming language by John H. Maloney and Randall B. Smith [Mal95, MS95, SU95].

Philosophy

The goal with the development of Morphic was to create a Graphical user interface with Liveness and Directness [MS95]. In their own words the two central concepts get the following descriptions:

- Liveness means the user interface is always active and reactive – objects respond to user actions, animations run, layout happens, and information displays update continuously [MS95, p. 1].

- Directness means a user interface designer can initiate the process of examining or changing the attributes, structure, and behavior of user interface components by pointing at their graphical representations directly [MS95, p. 1].

Directness and liveness are properties of the physical world and the creators of Morphic tried to use the metaphor of communicating objects. One of the use goals with Morphic has been to shorten the iterations in the process of creating user-interfaces. Since design is an iterative process, Morphic can speed up the process by implementing directness and liveness. The cognitive load on the designer also decreases since the designer don’t have to correlate graphical components of the interface with their alternate representations in the programming language or to switch between run and edit modes.
How Morphic works

The central abstraction of the Morphic system is the morph. The word morph comes from Greek and means essentially “thing”. When using Morphic in programming you create and manipulate morphs. A morph is always visible and can be picked up and moved. Morphs also have the following properties: [Mal95]:

1. It can perform actions in response to user inputs,

2. perform an action when a morph is dropped onto it or when it is dropped onto another morph,

3. perform an action at regular intervals, and

4. control the placement and size of its sub-morphs.

In the SELF programming language in which Morphic was first implemented, it was used as the standard user interface (UI). The special and novel thing about this UI was that it could be run and edited at the same time. Compared to other User interface editors that let you drag-and-drop components and behavior into the interface, Morphic makes no run/edit distinction. The advantages of this are according to Maloney and Smith that the overhead of frequent mode changes is eliminated (i.e. you don’t have to restart the program to see your changes and you don’t have to close the program to make new changes) and that it decreases the cognitive burden or load of remembering the current mode [MS95].

As mentioned earlier, the central abstraction of Morphic is the morph. In a complete Morphic environment all visible objects in the GUI are actually morphs. Morphs can contain other morphs to make up composite morphs. This way the system maintains a small and portable kernel and a programmer can redesign the GUI completely with relative ease.

The UI-loop of Morphic can be described by Figure 3.1. Morphic uses *Ubiquitous Animation* [MS95] which encompasses ideas to create liveness and liveliness in the system. Morphs should have lightweight autonomous behavior for animation. This means that they control their own animations. To accomplish this, every morph has a step method which is called in every cycle of the World (illustrated with the runStepMethods in Figure 3.1). This also allows multiple animations to be active concurrently. World is the root morph in which all other morphs are placed in a tree-like structure. The Morphic system should also provide a kit of animation behaviors that can be applied to any morph. The type of behaviors could be motion, color changing, and animation.
Morphic has since its birth had a small, but dedicated, community of users. It has never reached the mainstream population of programmers like for example Java has. One reason that Morphic did not reach the broad area of programming could be that Smith and Ungar, when they created the SELF language according to their own vision of what programming should be, took one step away from the general view of the purpose computer programming.

Programmers are human beings, embedded in a world of sensory experience, acting and responding to more than just rational thought. Of course to be effective, programmers need logical language semantics, but they also need things like confidence, comfort, and satisfaction—aspects of experience which are beyond the domain of pure logic [SU95, p. 1].

A part of any discussion about Morphic needs to be dedicated to the issue about classes on prototypes; some authors refer to it as a debate [SU95]. In object orientation there are two main ways of building objects: by using classes that serve as templates that are instantiated into objects or by using...
prototypes that can easily be copied or cloned and modified to created new objects. Morphic is considered to be more of a prototype based system in it’s SELF implementation.

The advantages of class-based object orientation are that the powerful functionality in inheritance can be used. Prototype-based object orientation can also support inheritance as in the case of SELF, but the strength lies more in the flexibility of the objects. The motivation for the SELF language was to use a metaphor of tangible objects and use that as far as possible. The first implementation of Morphic, which was done in SELF was therefore implemented with a model based on prototypes.

Apart from the general talk about the programming experience Smith and Ungar were not specific about the actual advantages in application programming of their new System. When other object oriented languages became the language of use for Object oriented software engineering, which is the one and only paradigm today for large scale systems, SELF has only a few enthusiasts.

A reason for the success of for example Java could be that it handles abstract and less tangible object-orientation better than SELF. The very powerful techniques with inheritance, interfaces and polymorphism and the choice to make Java completely object oriented might contribute as well.

**Morphic in Smalltalk**

Morphic was at one point completely rewritten in the Smalltalk language for the *Squeak* [squ] programming system. The reasons for this change are not really clear, but the developers might somehow have found the Morphic framework more suitable for their visions about the future than the standard Model-View-Controller in Squeak. Today, both frameworks are implemented in Squeak, but there will no further development on the MVC.

In the Squeak project, the desire is to make it a tool for education of novice-programmers and children, and for creating web applications with the *Seaside* [sea] continuation of Squeak.

The Smalltalk language is originally based on different design principles than SELF. When designing Smalltalk, the motivation was communication. The developers wanted humans to start communicating with computers in a way that is grounded in the human cognition. Therefore Smalltalk language is built on the Metaphor of communicating objects. This is the same metaphor as Morphic is built on and the unification between them was therefore a natural one. Morphic has a more class-based system in the Smalltalk implementation.
Schemorphic and MorphR

Morphic has also been implemented in smaller scale in Ruby [Lyn04] and Scheme [Sve04] as MSc projects in Computer science.

3.2.2 Alice

Alice is an object-oriented direct-manipulation programming system that has been tested in introductory courses in computer science at the Carnegie Mellon University. The targeted user for Alice is the middle schooler. The designers of Alice system have had the ambition to make programming more attractive for girls. The intended process when working with Alice is that programmers which are the users first craft a story. Then they make lists of participating objects and actions that will be needed in the story. This way they learn to break a large problem into smaller ones. Studies have shown that Alice have had a positive influence on the amount of computer courses the students continue to take after working with Alice [CDP03].

One thing about Alice that can both be strength and a weakness is
that students do not come in contact the syntax in the Java programming
language which Alice is built upon and the skill of dealing with syntax
errors like missing semi-colons or mismatched braces etc. Interaction with
the interface is handled with the mouse and the program is generated by
dragging and dropping code segments into the code panel. The positive side
of this is that the cognitive load decreases since there a fewer simultaneously
interacting learning elements and less jumping between different levels of
abstraction (see section 2.1.1 on cognitive load).

Alice has also been implemented in the Squeak environment (see section
3.2.1 on Morphic).

3.2.3 HANDS

HANDS, Human-centered Advances for Novices to Develop Software, is a
programming system designed for children. The creator of HANDS, John
Pane did the work as his PhD work at the Carnegie Mellon University.
Usability has been the primary focus during development. The central
Metaphor and abstraction in HANDS is the card, the deck of cards that
allows more complex programs and the agent that manipulates the cards.
Users write attributes on cards and then place them under a surface. The
only thing that is visible through the surface is the attributes. This im-
proves the visibility of data since it is only represented on the cards. The
aim with the system is to improve usability by minimizing the need for
control structures and local variables.

3.2.4 JPie

At Washington University, a visual programming system called JPie (Java
Programmer’s Interactive Environment), [jpi] have been implemented in
Java [Gol04a, Gol04b]. The positive side of doing this quite cumbersome
work is that Java is strong typed, object oriented language with built-in
data encapsulation. The motivation for the JPie project is a comparison
to the art of writing. A long time ago, only a few people, the scribes, could
write. Today only a few people can write computer programs, the pro-
grammers. The JPie team hopes that more people can use programming
as a creative way of expressing themselves. When it comes down teaching
programming on college level where most people get serious about pro-
gramming. Goldman has identified a serious problem with introductory
courses in programming in that they focus too much on programming de-
tails in favor of the concepts. This leaves an impression in students that
programming is a shallow discipline only concerned with transcribing ideas
into code, and not the ideas themselves. Goldman is a writer and there-
fore it is best to quote his work in order to show his view on programming
College-level introductory computer science courses serve two competing objectives, education and training. On the education side, we want to expose students to the beauty of computer science, to provide insight into important ideas and ways of thinking, and to cultivate their intellectual curiosity about the field. We want beginning students to understand and appreciate powerful and inspiring ideas, in areas such as software architecture, algorithm and data structure design, and concurrency. On the training side, we want to instill programming and debugging skills in order to reinforce concepts and prepare students for later courses and beyond. To satisfy training objectives, however, Introductory computer science courses are often reduced to language courses, in which students spend the bulk of their time learning linguistic constructs and programming mechanics. Low-level concerns undermine deeper educational objectives, leaving introductory computer science courses as ‘rites of passage’ in which only those willing to contend with a steep learning curve in an artificial syntax survive. Such courses demand a bottom-up learning style and appeal to a small fraction of the college population, leaving the rest without an attractive opportunity to gain meaningful hands-on experience with
3.2 Live programming systems

computer science concepts [Gol04a, p. 1-2].

Goldman and his team have implemented what seems some be quite impressive. The idea with dynamic classes in a strongly typed language like Java can offer a very fast feedback loop in programming.

JPie separates the model of the program from the view, by providing a GUI with a fixed set of allowable manipulations of the program and thus prevents syntax errors from occurring. This approach is similar to that in Alice.

The GUI of JPie as seen in 3.3 is manipulated by dragging and dropping capsules that make up the program. Capsules represent variables, methods, and other JPie program units. The different colors represent the scope of the unit.

Debugging

In JPie, emphasis is put on the debugging capabilities which must be live in order for the system to be a Live programming system. The system must allow for modification of programs that contains erroneous expressions. The JPie debugger allows for breakpoints and stepping of program threads. Errors that occur do not require the thread to be restarted. During debugging, the programmer may pause the program and correct detected errors before resuming execution.

JPie also support On-the-fly exceptions. When exceptions that are not explicitly thrown or caught by a method occur, the programmer may choose how she want to deal with it. The user can then catch, throw or propagate the exception.
Chapter 4

PyMorphic Project

This section describes the PyMorphic implementation which is the major part of this Thesis work. The disposition for the chapter is a top-down description of the work. Section 4.1 describes the vision and the intended user of the system. Succeeding sections goes deeper into details where section 4.3 and 4.4 describes the implementation of PyMorphic which was realized as two different solutions. The solutions are called solution one and two. Solution two is an elaboration of solution one and should be considered as the main branch of the development which hopefully does not come to an end with the completion of this thesis.

4.1 Philosophy and vision

Programming is, as it has shown throughout this work, a very complex activity. It is easy for novices to become overwhelmed and discouraged if too many learning elements; rules and regulations are introduced at the same time. It’s a fact that, although it can be very troubling for an expert to watch, novice programmers will always come up with bas solutions to programming problems before they learn the better ones. If the programmer is to choose to continue her venture into the art of programming, it must be stimulating, fun and rewarding. Programming is, or should be, a very creative activity since it is essentially an activity where one starts out with an idea and then constructs a digital artifact that accomplishes things and realize the idea. Great things can be accomplished with programming.

The idea that programming should be a creative and fun experience is the leading star for PyMorphic. In order to make programming a pleasant journey towards expertise the user has to form good programming schemas through schematization (Section 2.1.2) and the cognitive load must not be too great. These to principles are the focal points on where the efforts in the PyMorphic project will be put.

PyMorphic focuses on the object oriented Live-programming which entail that there is no clearly defined starting or ending point in the program. If a metaphor is to be applied it must be closer to a running system or even
a world with properties and objects that is continuously evolving. A classic metaphor like the pipe in which data is run through and manipulated is not as applicable as the main metaphor in a Morphic system.

### 4.1.1 The intended use and user

The intended user for PyMorphic is the user who has some experience with programming, but values graphical feedback in favor of purely textual.

![Figure 4.1](image)

**Figure 4.1**: The bar should be read from top to bottom where the novice starts at the top and the absolute expert is somewhere at the bottom of the red segment. The initial mandatory “Hello World” print out is followed by the PyMorphic system. When the user wants to implement more abstract features than the PyMorphic system provides the user moves on to the bottom section (which is good).

There are many definitions of programming and many different levels and paradigms as well. The practice of engineering large systems differs from the more small scale implementations that aim to test different programming techniques and theoretical concepts. In working and discussing the area of programming the scope must be narrowed to a certain domain in order not to make it specific enough to reason about. What kind of programming is PyMorphic about then? As shown in Figure 4.1 PyMorphic is supposed to take on where the simple programs like the mandatory printing of “Hello World” on the screen leaves off. Advanced low-level systems developed in notoriously difficult languages like C++ lies outside the aim for PyMorphic. This kind of programming is part of bottom segment of
the graph that represents the programmers with high level of expertise.

The user of PyMorphic is a person with ambition to become a good programmer, but who has some way to go and who is interested in progressing with a gentle learning curve and who doesn’t like to get stuck with not being able and understand and solve problems in programs. There is an educational purpose for the user and a big interest in always being able to progress in the work without getting stuck.

The aim with PyMorphic is to satisfy users who are at a stage with impatience over simple text-input programs and who want to work with graphics and animation and in the same time get an understanding of object-orientation and the advantages with this approach. At this level the user probably has gained knowledge in specific areas of programming, but at the same time the user has not built schemas of who every concept relates to the other. The user can therefore become very frustrated with what seems to be unexpected behavior based on false assumptions. A Live system like Morphic helps the impatient user by providing fast and direct visual feedback. The introspection capabilities of PyMorphic also helps the user to check for error in a live debugging sense. A term that is often used for describing impatient programming is the Rapid programming approach. This approach can describe PyMorphic. The user is in a very exploratory phase when working with PyMorphic.

A programmer can benefit greatly from using a debugger for more advanced program debugging. According to the vision of PyMorphic the use of a debugger can suppress the creative side of programming if it is introduced to early. Too many things to think about at the same time creates a cognitive overload, which can harm the users general interest in programming and in the worst case scare the user from continuing to learn more.

There should be a way to learn incrementally and to learn small portions of programming. The user always has lots of question. This is an experience that the author of the thesis learned when working as a tutor in a Computer Science 1 course at the Linköping University. The best thing is if the program can give answers to questions to decrease the load on human tutors (which are not always available).

Judging from the name of this project the decision to go with Python could seem to have been taken at a very early stage in the process, but thoughts about choosing another language for the implementation existed long into the project.

4.1.2 Design principles

Following the theory on the cognitive aspects of programming (section 2.1) a set of design principles were formulated early in the design process. These
principles were of course also inspired by the different Live Programming systems surveyed in chapter 3.

The major principles are as follows:

- **Stimulate creativity:** Allow for the user to play and interact directly with the objects.

- **Support sensible schematization of programming concepts:** This is an absolutely vital point in education and the major goal with PyMorphic. The users build their initial concepts of objects by having the object represented as tangible morphs.

- **Be consistent with the metaphor:** The tangible morph is the main metaphor for an object in the system and it will be used as much as possible.

- **Allowing for incremental work:** Introduce programming concepts at the users own pace. This is related to the theories on cognitive load. An overloaded working memory inhibits learning and the user must not be tempted or forced to try and take short-cuts in the learning and try to grasp too many learning elements at once.

- **The ability to isolate problems:** One big goal with this project has been to create a tutorial with added exercises for first year graduate students at the Cognitive Science education program at Linköping University. The Python language is the first programming language the students get to work with. Important things that are needed in order to make a good programming environment is good feedback to the user. A user that is not used to programming is more dependent on fast and understandable feedback is even more important.

- **Availability for opportunistic design:** Since this is how programmers work. This is also based on the assumption that users need to isolate small problems and make assumptions about them and test them. This principle could also be called freedom in design.

- **Good possibilities of externalization of working memory:** The number of learning elements will be many enough in a complex activity like programming. The user interface must support the user to externalize the working memory. The paper and the pen are very important tools in programming for externalizing ideas. When the programmer moves on to start working with the computer screen it is important that the visualizations on the screen do not require the programmer to go back to return to the paper in order to externalize things on the systems that could be externalized on the screen.
• **The ability to reduce the effort of building a working set:**
  This principle is based on the vast research from the Carnegie Mellon University.

• **Keep it simple:** The PyMorphic environment is aimed at novice to intermediate programmers. This design decision is quite uncommon in programming systems. Many system engineers don’t want to admit that their work is not suitable for advanced programming. They want to embrace every programmer. In the opinion of this work the Squeak system has become too complex for novices and too messy for experts. The creators have grasped for a lot of functionality which makes user puzzled and possibly unmotivated.

• **Feedback that does not introduce many novel concepts:**
  Instead of just printing out the error message there should be a connection to the user’s model of the program. The user is looking at the source code in a different way than the system. “The system should always keep users informed about what is going on, through appropriate feedback within reasonable time” [Nielsen 1994]. One thing that can be improved in the feedback is the error-messages.

### 4.1.3 Development Process

The choice of development model is important in any software development process. This work will be using an iterative and agile approach. All iterations of the work have a specification, an implementation and an evaluation part. There are many advantages with this approach. In an explorative project like PyMorphic, new insights will lead to the implementation of new hypotheses that are not available before the first iteration. By using an iterative approach the new knowledge can be incorporated in the work in a systematic way.

The work on PyMorphic naturally divided into iterations. The iterations are referred to as solutions in this work since this description is closer to the truth because both solutions have different strengths and weaknesses. The first solution (Section 4.3) proved to have a few serious technical setbacks, which were corrected in the second solution (Section 4.4).

The orthodox process when designing in a user-centered manner is to go from lo-fidelity prototypes to hi-fidelity prototypes and then to a working product. The work on PyMorphic has not been orthodox. It is closer to the truth to call it a RAD project (see Section 3.1). In the initial stages of any design work the greater the diverging of concepts and ideas the better. The PyMorphic work initially jumped over the first step since the issue was to test the Morphic philosophy, which was already implemented in the Squeak system and that served as a very influential role-model. In
that sense a step away from the supposedly ideal design process was taken. The project was initially more exploratory about the technical possibilities of Python and Morphic than user-centered design. On the other hand this approach got a more hi-fi prototype running sooner and offered more insights into Live-programming. The thought behind this was that in a complex task like programming the real advantages and disadvantages of an implementation are not discovered until a certain level of complexity and fidelity in the developed prototype is reached. The context in which PyMorphic is being developed is in an academic university education. The University undergraduate student is also an intended user of PyMorphic. As mentioned earlier the author of this thesis worked as tutor in a computer science 1 course in programming and this gave first-hand information and many insights into every aspect of programming at the level of expertise that PyMorphic aims for.

The project is hosted by the Sourceforge software development web site. The official project site is pymorphic.sourceforge.net. The Sourceforge CVS service is used in this project. The official PyMorphic project site can be found at http://pymorphic.sourceforge.net.

4.2 Evaluation framework

For the evaluation of the iterations in the work the cognitive dimensions framework was used (see section 2.2.1) together with the report on the design of novice programming systems developed at the Carnegie Mellon University (see section 2.2.2).

The cognitive dimensions framework provides a broad brush discussion tool which is suitable in this kind of iterative work. The reasons for the choice of method are that:

- A deeper analysis would slow down the development process.

- The iterative model of development suits quick evaluation methods.

- The prototypical and explorative nature of this work suits a broad brush tool since focus lies more on principles than on fine-grained HCI details.

The evaluation for the final solution also included user acceptance testing. This testing was conducted in an informal walk-through where the user carried out a tutorial that offered a set of small tasks to get a general understanding of the system.
4.3 Solution 1

In the presentation of the solution focus will be on the design decisions and on the results and less on the process.

Initially the development of PyMorphic was exploratory in order to grasp the applicable features of the Python programming language in a Morphic system and to set out major conceptual design decisions. The metaphor was the use of physical morphs with properties such as size, shape and movement. There were some thought about which design patterns that were going to be implemented or if the process would be more ad-hoc to focus on the rapidness. Since the work initially has been iterative, the central metaphor was not developed during the first iteration where the work was exploratory.

The first solution in the PyMorphic project focused on the idea of using widgets for source code editing and the workspace where the user executes expressions as composite morphs. This attempt seemed promising early on, but later it proved to be too cumbersome to implement with the time resources available for this project.

4.3.1 Object views

Reducing the cognitive load is quite a challenge in a complex activity like programming. The solution to that problem that PyMorphic attempts to provide is the Multiple View System. The metaphor used in the program is the morph- the physical object. In order to teach programming in Python the objects can not only be represented as visual graphics. The Multiple View system was enhanced in the second solution, but started out in the first.

The idea of the Multiple View System was inspired by the philosophy behind the SELF programming language. Focus is on giving good possibilities for the user to build strong schemas of basic object orientation. This is done by providing tangible visible object on the screen. The idea was also that the user was going to be able to inspect every tangible object and see all the attributes of the corresponding object. The third view was the source-code view. Programming is essentially about editing and comprehending source-code. There is no point in obscuring that part for the user.

4.3.2 Squeak as a role model

The Squeak system initially served as a strong role-model for the PyMorphic system. The Squeak system is complete and has already evolved through numerous development cycles. Squeak inspired the creation of the Halo, the workspace and the System Browser.
4.3.3 Design decisions

An initial loose specification was created early in the development. As mentioned earlier much inspiration was taken from the Squeak programming tool implemented in Smalltalk. The list of requirements looked like the following. The list of widgets would be the following:

- **A Background Panel:** This is the Device Context for the world where morphs are to be drawn. The panel is represented as a Window in the GUI.

- **A Workspace:** This is for executing source code.

- **A System Browser:** for editing method source code and recompiling objects at runtime.

- **A world morph:** which would be used as the root to the morph tree

The functionality of the program is described with the following list of functional requirements:

- Traversing of the morph-tree for every program cycle. (See figure 3.1 of Morphic).

- Every widget should be movable and resizable. A Halo like the one implemented in Squeak might be suitable for this.

- The user should be able to execute code with Pythons exec method.

4.3.4 Results

In the first attempt to make PyMorphic the design was based on the idea that everything in the Programming system was going to be represented as a morph object. *Figure 4.2* shows a Screenshot of the PyMorphic system.

The top right corner shows a rectangle morph with a ball morph inside it. The ball is activated and a halo with three icons shown around it. The high-level Wx-components (Workspace, System Browser) had to be wrapped into morphs in order to achieve polymorphism in the solution. These composite objects proved to be difficult to interact with. This is because the Wx-widgets have their own event-handling with the operating system. The Morphic system used the same event-handling system so the event handling in the composite morphs could not be wrapped in an easy way. There was also a problem with overlapping windows since the Wx components were used in an unconventional sense. The design of the different widgets is presented below.
**System Browser:** A minimalist system browser was implemented. This is a solution that is similar to the System Browser in Squeak. One of the possible downsides of this type of system browser is that only one method can be shown at the time. Other IDE's like Eclipse show the whole source file at once provides functionality for minimizing methods and classes so that the file does not become too spacious. The Wx-library has a StyledTextCtrl widget. This was used in the System Browser. The widget has many of the state-of-the-art features of source-code editors. It has syntax highlighting and the possibility to minimize and maximize methods and classes.

**Workspace:** The workspace has been implemented, like the System Browser, with a clean and minimalist design using a StyledTextCtrl. In the workspace the user can copy and paste text using the standard windows handlers for copy and paste. The role model for the workspace is the Squeak workspace. The primary function of the workspace is that it allows for the code to be executed in the system.
Halo: A first attempt on a Halo was implemented. It had the Pick-up and resize functions. These were both lacking somewhat in performance.

4.3.5 Evaluation

The first solution of PyMorphic offers a complete working system. Some features, like the editing of methods and the execution of expression proved to be easy to implement thanks to Python built in functions and the Wx-library. Some unexpected problems also came up. As mentioned earlier the event-handling and screen updating did not work as smoothly as expected. There were some problems with the unconventional solution involving overlapping panels in Wx. The solution disabled all layout controls in order to take control of the drawing. There was also a performance problem in the resize operations of the composite morphs. These problems were worked on for a while, but they were on the technical level and it was not easy to find good solutions to them.

Cognitive load and Cognitive dimensions framework

The program starts with an empty screen. This way the users start in an exploring mode. The positive effect of this is that the user is not overwhelmed by information and the cognitive load starts out at a minimum. The user has to be active in making decisions and does so in a goal free creative manner.

The different points of the Cognitive Dimensions framework and their contribution to the evaluation are listed below. The cognitive dimensions framework is suitable for making a quick evaluation of the first iteration before forming the requirements for the second solution.

- **Abstraction gradient:** The system is quite abstraction hungry in that the System Browser requires an abstraction in order to create executable code.

- **Closeness of mapping:** The only way to edit the program is to write source-code. There is no real closeness of mapping in that sense. The Halo which appears around morphs provides better closeness of mapping.

- **Consistency:** The consistency for easy tasks is quite good. For more complex tasks the base morph class needs to be extended and that is not as easy and consistent to achieve.

- **Diffuseness:** The syntax in Python allows for compact and readable code which on a syntax level decreases the diffuseness. The step to working with the Morph abstractions is a little tricky. This is mainly
because the module name where the morph class or derivate class is defined has to be included in any call to it.

- **Error - proneness:** The central point with PyMorphic is that no unnecessarily hard mental operations should be required in the work. Any change in the code should immediately give visual feedback to the user. Any careless mistakes should be detected quickly. The system is not robust enough in Solution 1 to allow for a complete testing of the Error - proneness.

- **Hidden dependencies:** Dependencies are one of the downsides with the first solution. Since only one System Browser can be visible at once it is difficult to view dependencies between methods. Although in the Python language the dependencies are visible.

- **Premature commitment:** Since PyMorphic is object oriented the code can be developed in both a top-down and a bottom-up fashion. This quality entail that programmers can work on the part of the solution where they have the information.

- **Progressive evaluation:** This is probably the most vital property that PyMorphic must have to reach success. Since the program is in run- and edit-state at the same time the programmer is free to test any methods at any time.

- **Role - expressiveness:** This part was not well developed in Solution 1. The Squeak system provides an object inspector, but that was not implemented in PyMorphic solution 1. Visually the user can keep track on the different Morphs, but their programmed relations can not be viewed.

- **Secondary notation:** Thanks to the StyledTextCtrl the code in PyMorphic has syntax highlighting. Python also requires indentation for every code block.

- **Viscosity:** Thanks to the Autoreloader changes are incorporated into instances automatically. This means that the viscosity is low since no new instances need to be created.

- **Visibility:** It is possible to have all important widgets visible at once. Since every widget is a composite morph the Halo-operations are available for every morph. The big downside is in an attempted juxtaposing of source code. This is not possible in solution 1.

The system browser makes the system abstraction hungry since every new method or class that is added to the system modifying the class hierarchy and the user must consider this.
Python

The solution with every widget represented as a derivate from the morph class and composite morphs with morphs as wrappers for Wx-widgets became a too difficult solution. Since Wx has its own Event-handling system the approach became problematic because a conflict came up between the Wx-widgets own event-handling and the Device context which was the intended Morphic view. The events could be caught without involving the Wx Event handling system.

Another problem concerned the overlapping panels which became flickery since they are not handled satisfactory by the Wx-layout and update controls and overlapping panels is not part of the intended use of Wx.

The Python exec function has been very useful for the PyMorphic project. The Wx-library also came in handy. One of the issues about Python that can pose intricate problems is the two different styles in which modules can be imported. A prerequisite for the Live programming approach modules need to be updated in run-time. To accomplish this they need to be reloaded. The two different approaches to import modules are either the expression:

\[
\text{import <module>}
\]

In this case calls to the classes in the other module is reached by writing:

\[
\text{module.class.method()}
\]

In the other case the import statement looks like:

\[
\text{from <module> import <class or function>}. 
\]

In a call the modules name is omitted which allows for shorter code in the call:

\[
\text{class.method()}
\]

This is possible since all the classes, functions, and variables are imported individually from the module. One might wonder why not always use the \text{from <class> import <module>} clause. The advantage is clear in that you don’t have to write the module name before every call. But the downside is that reloading modules becomes very cumbersome since every entity needs to be reloaded individually and another possible hitch is that naming conflicts may appear if functions in different modules share the same name.

In order to achieve a Live-updating system an Autoreloader class [aut] was used as one of the parents for every Morph object. The Autoreloader was found as Open-source code on the Internet and consists of approximately 50 lines of code. The Autoreloader uses weak references and an
instance-tracker to keep track of every Morph instance. When a class definition for a Morph derivate class is updated the instance tracker sets the correct attributes to the updated class. This approach proved to work without much pains.

**Error messages**  Error messages are presented in the Bottom Shell-prompt. This is built on the standard Wx shell-widget. The error messages are not formatted and are presented the way Python prints the Traceback. The Shell-prompt can also handle input to the system.

### 4.4 Solution 2

The second solution started when the first solution ran into difficulties which motivated a different approach. As mentioned in the previous section the desired features were too cumbersome to implement.

#### 4.4.1 Design decisions

In the second solution the design was not going to mess with Wx and work against the layout methods offered there. Instead the idea was to use the Wx-package with standard layout controllers. The positive side with this was that the development could be completed faster and for the user the recognition of familiar widgets that are used as standard in the Operating System will improve usability. The solution with composite morphs and the accompanying performance problems were not allowed into solution two. The problem was that it is bulky to wrap the Wx-component into the morph since it provides it’s own event-handling. It is not an easy solution to inhibit its handling of mouse events.

While in the first attempt every expression written by the user in the workspace was executed in the main namespace. In the second attempt better control and encapsulation of relevant data was desired. One possible solution to this was to make a separate namespace in which only the user’s Python expressions are executed and thus separated from Python’s standard namespaces. This way the user has control over the names objects in the world. One of the hopes with this approach was that the intended user would grasp the concept of namespaces more easily and get a better understanding of namespaces in general.

One of the design decisions was that the toolbar in the parent window where there would be a button that references the API for the modules. This way the user should get an introduction to API’s and to the built-in PyDoc. The Python auto-documentation tool PyDoc offers the same general functionality the same basic principles as the popular and standard JavaDoc for Java.
The system browser in solution 1 mimics the Squeak system browser. This attempt was very abstraction hungry according to the cognitive dimensions framework. This attempt might work well for Smalltalk, but when it comes to Python it is easy to become skeptical. In a Python program the code is divided into modules and it is not desirable to obscure this principle of source files which is the common data-organization principle in most languages. Modules are the primary way to organize the code in the file system since a module is represented by a file. Because of this and the fact that PyMorphic is a middle station in the journey towards becoming an expert in programming the second attempt changes the editing interface from the System Browser into a more standard IDE Code editor. This way the user gets to see the source file as one view in the Multi-view system.

Another feature that was intended for solution 1 but not realized was the Code Inspector View. There was a possible advantage in changing the System Browser into a Source-code viewer. The system browser from solution 1 offered limited visibility and juxtaposing. The problem of juxtaposing would also be helped by implanting a Model-View-Controller design pattern which allows for multiple observers of different source-code files.

In summary the following list of features represents the desired continuations in the development after solution one.

- A Top View composed of a standard Windows Parent window with Child windows for every Morphic view (i.e. world, workspace etc.).
- A Model-View-Controller design pattern.
- A separate namespace in which user entered expressions are evaluated.
- An API view.
- An IDE code editing view following a more modern custom.
- A Code Inspector View.

In the next section reviews the implementation of these design decisions and presents how the solutions were realized.

4.4.2 Results

The results section is the final results for the project since solution two is the latest release of the project. The solution followed the Model View Controller design pattern and several different views were created.
File Editor

In the Squeak environment there is no apparent source file editor and the file-browser is not designed like contemporary file browsers found in other popular IDE’s. The design decision to change the system browser from solution 1 into a file browser and editor in accordance with the standards today is based on the assumption that computer users today are very accustomed to the file browser and use it to navigate the file hierarchy. Every

![Figure 4.3: The Source-file editor view.](image)

IDE requires a source file editor if it is going to handle any kind of code writing. This is of course a compromise. The illusion of the manipulation of physical objects is not apparent in this view. However the user gets acquainted with the normal view of program code. This way the step from PyMorphic to other programming environments will not be so confusing.

World View

The same halo that was used in solution 1 was kept in the second. The only difference was that the delete feature was added. In a functionality aspect the difference between the solutions was that in the second one every icon in the Halo had actual working functionality corresponding to it. Compared to the first solution this world view widget has a toolbar from which basic morphs can be created interactively. This raises the level of abstraction from solution 1.
4.4 Solution 2

![World view](image)

**Figure 4.4:** The World view.

**Object inspector**

The object inspector view is inspired by Squeak, but has a more generous selection of different object attributes although not all the general attributes of objects are presented, that would have been too many, but the most important for manipulating morphs. The whole idea with the object inspector is to follow the theories of cognitive load and reveal the attributes and the new object view different from the graphical view on the user's initiative. The inspector is not thrown in the user's face.

The standard Wx TreeCtrl widget was used to create the tree. A link to the source code for the object is also included in the toolbar.

**Namespace view**

In the namespace view all the names that have been used in the PyMorphic code execution are listed. These are the names that are accessible from the Workspace. When executing code this namespace is provided as the local namespace. The intention with this view is that the user should get the connection from presentation of objects in the world to the presentation in the computer's memory. It is also a helpful feature in debugging. The user can see what kind of objects are bound to different names.

4.4.3 Evaluation

In this section the specific parts that differentiate the second solution from the first one are discussed.
The cognitive dimensions framework was used to evaluate the solution just as in the first solution. Only the points that are different from the first solution are handled here.

- **Abstraction gradient:** The Hunger for abstractions still exists, but it is less with the second solution. The Morph is still the abstraction that is needed in order to create visible results, but in the second solution the user has access to the entire source-code file in the source-code view.

- **Closeness of mapping:** With the new view with A Parent and many Child-windows the closeness of mapping is generally provided. There is a possible issue with overlapping windows. If all the different
windows the make up the multiple-view system are simultaneously open there is a risk of crowding among the child-windows.

- **Error - proneness:** The reload button in the source-code view provides a good way of controlling the modules for errors. When it is pressed the new code is checked for errors and errors are displayed in a popup error view.

- **Progressive evaluation:** This is probably the most vital property of PyMorphic to reach success. Since the program is in run and edit state at the same time the programmer is free to test any methods at any time. The scripted nature of Python makes it easy to write independent code segments. The combination of the Autoreloading and the Python language gives promises for good progressive evaluation once the system had become more complete.

- **Role - expressiveness:** This is the dimension that has evolved the most between solution one and two. With the new namespace view the user can see which names and objects that the system knows of. The Morph tree is also visible through the object inspector view.

- **Visibility:** A lack-of-space problem naturally appears when there are two many important views at once. It could be argued that some views could benefit from being merged.

### User attitude testing

Involvement of the user in the development process is vital for success in any software development project that has a user interface. General guidelines and heuristics can take you far, but not all the way.

A tutorial where the user was going to construct a pong game was created for the user tests of the second PyMorphic solution. The tutorial consisted of a total of 16 steps (see figure 4.7). The progress was displayed in the dialog windows header. The choice of the Pong-game as the tutorial’s task was that it consists of multiple morphs, animations and basic conditioning. This was believed to be well suited for the intended user’s level of expertise. The intention behind this design was to reduce the amount of simultaneous learning elements according to the theories on cognitive load. The information in the tutorial was presented in a dialog view with a limited piece of information associated with one user task displayed simultaneously. A total of 5 users ran the complete tutorial. The age span was from 15 to 25 years. The programming experience ranged from novice to professional programmer. All of them completed the tutorial with only minimal assistance from the test-leader. The assistance consisted of some directions where the test-leader wanted feedback on opinions about certain
features like the placement and appearance of certain controls. The users were observed during the test and were encouraged to think aloud. Notes were taken by the test-leader during the test.

The users were all very focused during the task and there was no visible breakdown in the interaction. During the test, many of the users came up with suggestions for how to improve the system.

When asked about their opinions, most of the users answered that the PyMorphic system was interesting and fun to work with and many of the test-persons had many suggestions on how to extend the functionality of
the pong game beyond what was covered in the tutorial; for example that there should be many balls that could bounce on each other. Some users had suggestions on design improvements in layout and the design of the icons in the toolbar. Not all the implemented features in PyMorphic could be tested in the tutorial. The object inspection and namespace views were not tested, but a few users found them anyway.

**Python**  Python remained a good choice in the second solution. There were some issues about the Autoreloader which prevented the use of it as a parent for every class. Instead the Autoreloader was used for every subclass of the Morph class. The implementation of the separate namespace where the users expressions were evaluated worked smoothly.

**Error messages**  Error messages are presented in a Pop-up window in solution 2. This follows the technique applied in Squeak. The error messages are not formatted and are presented the way Python prints the Traceback.
Chapter 5

Discussion

In this chapter I discuss the results from the project, and present ideas for future work.

5.1 Application areas for live programming

The application areas for live programming probably exceed what has been shown in this work. For example the area of live music composition has only been mentioned briefly in the work. This work has focused on live programming and it’s application for educational purposes. In the Live approach the user can test code in the system and explore the possible constructs in the language and get fast feedback. The user might be less hindered by learning barriers and details that call find it easier to come up with creative ideas and this might create a desire for the powerful features of for example object orientation and other programming techniques. Using Live-programming systems like PyMorphic should be suitable for educational purposes just like the other Live-programming tools that have been surveyed in this work. A conclusion that can be drawn from this work is that live-programming seems to have a future in explorative programming as often is the case when the purpose is education. With it’s fast feedback on changes in the code the Live-programming approach may work well for Rapid Application Development (RAD). All programming activities might benefit from faster feedback. Both the design and the testing activities could get better support from the visible feedback.

The visionaries in different academic and other proto-programming projects generally set the bar high when it comes to picking a domain of users in the general publication that should be the targets of their visions. As it has been continuously shown in this work programming is a very diverse and complex activity. Different users and organizations see different values and have different desires with their programming work. There will always be different approaches to programming depending on a number of parameters such as speed, robustness, readability of code and testing requirements. The Live-approach may not be the best solution in every case.
and that is probably not desired by anyone, but if the live approach can offer some improvement on the quality of work in any area it is interesting to investigate how this can be done. As the results from use of systems like Alice and JPie for educational purposes the live approach combined with the Objects-first methodology in teaching programming has proven to be fruitful. The tests of the PyMorphic system also give indices on that the experience of working with the system is stimulating the creativity among users. It is not easy to categorize programming practices and programming users. Some people start programming on their own and have become experts at the time when others write their first line of code. The self taught experts might not have as much use for the PyMorphic system as users who are more dependent on a more gentle learning slope.

5.2 Live programming in education

As we can see in this work many approaches have been tested in the area of Live-Programming. This trend will probably continue. The question about how well Live-programming is suited for education is addressed and stressed by all the authors of the systems that have been surveyed in this work. Live-programming has definitely established itself as a promising paradigm in education. The studies on Alice and JPie show that users are very stimulated by this approach.

In PyMorphic the theories of cognitive load, the building of schemas and learning barriers have been combined with the Live-programming approach. The cognitive load theories originate from theories on working memory and learning. PyMorphic is attempting to incorporate this set of theories.

The testing of the system was dialog based and the response from the users on this approach was good. The users were stimulated by working with the system and were curious about continuing to do more work after the scenario ended. Of course more testing and longer testing sessions would be desired than was possible with the time resources available.

One possible cause for confusion for intended user was discovered in the demonstration of the program, especially for people with no previous programming experience; In Live-programming there is no real start and end to every programs execution. The thought behind PyMorphic was that a separate namespace with its own widget and view would give the user solid representation of what the system is working with. In a confusing world the list in the namespace is the thing the user can utilize as a visualization of the current active objects in the system. This is a very interesting idea, but it would require longer testing sessions.
5.3 Graphical representations

All the systems reviewed in this work have put much work into the GUI and the graphical representation of program entities. This is also the case with PyMorphic. In PyMorphic the graphical primitives are only one of the views of the morph object. The PyMorphic multi-view system consists of the graphical presentation in the World Viewer, the Inspect Viewer where the user can see the values of attributes. The inspector updates with the general step method. The third view is the source view where the entire Python programming language is visible. The difference between Python and the other systems like Alice, JPie or HANDS is that there is no obscuring of the syntax of the program. The other surveyed systems use entities to eliminate syntax errors from the user. This way the object oriented reasoning is enhanced. The philosophy of PyMorphic was that the code-view should not be abstracted away, but complemented with other views. From the age when children learn to read and write, they get accustomed to an advanced syntax in their first language. The Python syntax should be possible to teach to pretty young children. There is no need to make abstractions in one area (Python) that would be considered to be unnecessary in another (spoken language).

As the creators of SELF remarked the metaphor with tangible objects as representations for objects in object oriented programming holds as long as not to many abstract entities are introduced. For a student, these abstractions will sooner or later come, and the student will probably start working with more feature rich languages. There is no need to change the metaphor for this reason. It is better to see PyMorphic as a natural step in the process of becoming a programming expert.

5.4 Usability in the design

The aim with the work has been to use a concrete and tangible object metaphor for programming. The workflow in general is supposed to go from the concrete to the abstract. The first solution proved to be technically very tricky to make a complete system of, but aesthetically it was perhaps more promising than the second solution.

The cognitive dimensions framework provided many insightful approaches to weigh different aspects of the design against each other.

The usability in the system follows the standard widgets in the Wx library. As mentioned earlier in the work the mid-nineties was the time when large amounts of literature and empirical results on how usable interfaces are built emerged. The art of representing graphical interfaces has matured considerably and has become general heuristics.
5.5 Morphic in Python

Morphic and Python definitely work well together. With the powerful Wx graphics it is possible to build a full featured IDE in the Python language.

There proved to be a problem in mimicking the Squeak UI where everything is a morph. With Python/Wx a compromise had to be made. The strength in the reached compromise was that the user get’s a GUI to work with that consists of standard components seen in other programs (File browsers, List controls etc). This probably has the effect that it is easier for the user to get acquainted with the environment and less suspicious. The user will probably also feel that the knowledge acquired in PyMorphic can be applicable in other areas.

5.6 Lessons learned from PyMorphic

As it turned out, PyMorphic does support fast and live feedback to the user which was a requirement, but one area in which the solution falls short is when it comes down to revealing the fundamental structure of computer programs. The user of PyMorphic interacts with a system built up around a metaphor of tangible objects, as it was desired. The system contains code and morphs without any level in between. The representation of the execution of code in the computational threads are not shown.

It would take a lot of effort to build a completely new graphics toolkit like what was done when Morphic was rewritten in Smalltalk. It might not be worth the effort concerning all the other things the required time could be spent on. In the PyMorphic project the Wx-package was used. A standard general purpose graphic package like Wx or Swing for Java is quite a match to master completely. Much of this work has consisted in learning to master the Wx package and using it to come as close as possible to a Morphic system. The first attempt was aiming for a stronger Morphic like implementation, but it proved that the Wx package did not respond well to the intended use. The performance of the system went down and overlapping panels without layout control were not supported. An ideal Morphic system in Python would be built from scratch. But that would take time. Today standardized widgets are dominating. The experiences from using Wx as a standard Python graphics package are very good. It has quite a few very nice widgets.

This project implemented a Model-View-Controller design pattern which proved to be very valuable.
5.7 Future work

There are four main areas that were identified as key areas for future work.

- **Additional metaphors:** It would be interesting to study how it would be possible to complement the morph as a physical object with other metaphors for programming. Perhaps the program flow could be represented in some visual form. This would also become a good debugging tool if executions could be recorded.

- **Emphasize the user’s acquiring of programming concepts:** This could be done by providing explanations to the different terms that programming involves. This would follow the idiom "be a programmer by talking like a programmer".

- **Error messages:** As the system appears in its current form it prints error messages the way Python presents them in a pop-up window. These messages can be difficult to decode for a novice and the information from the traceback can be quite abundant.

- **Aesthetic appearance:** In order for programming to be rewarding the environment must be friendly and inviting. The standard Wx-components are very functional and users recognize themselves in it, but standard is often not good enough. In order to draw attention something new is needed.

There are also other things that could be considered as future work beside these four points. Better graphics with 3-dimensional morphs like the Alice system would draw more attention to the system, but the risk when following that path is that the development becomes too technically driven and as a result the complexity may rise to an undesired level.
Chapter 6

Conclusion

In this chapter I return to the research questions the were presented in chap-
ter 1 and present a summary of the results of the work. The first questions
deal with the Live-programming concept which has been a strong guide-
line for the PyMorphic project. The later questions address the research
questions that were answered with the PyMorphic system.

Which are the application areas for Live-programming and which
live-programming tools exist? There are numerous of different pro-
gramming tools that qualify as Live-programming tools. This work has
surveyed a few of the most well known and widely used. The spirit in the
area is good and the tools are constantly achieving new goals. The main
area for Live-programming is for education of mid scholars. Programming
does not have to introduce too many learning elements at once which helps
the users to overcome the learning barriers that are associated with pro-
gramming. Judging from the success of the systems mentioned in this
work there seems to exist a real need for programming tools that offer a
more gentle and human-centered approach to building early programming
schemas and metaphors.

The Live-programming approach may also work well together with a
Rapid Application Development methodology. Since the feedback is im-
proved the activities that make up the programming activity can be speeded
up. In designing the user get’s feedback that gives new knowledge to sup-
port new design decisions. In the coding the user can write smaller seg-
ments of code and testing becomes integrated with coding and errors are
shown faster since the programmer does not have to change from edit to
run-mode.

How is Live-Programming used to make education in program-
mimg more effective? The main approach in all the systems reviewed
is to focus on one important part of programming. The benefits from this
is that the programmer can build understanding with fewer simultaneous
learning elements which improves learning according to the theories on cognitive load. The JPie system focuses on abstraction and prevents the user from making syntax errors. This entails that the user can get further and build more advanced software in the introductory programming course at the undergraduate level. The Alice system has a similar approach, but it offers an even more abstracted view. The programming in Alice is completely independent from any low-level programming language. Alice also provides visual and graphical feedback which increases the role-expressiveness. The Squeak environment is probably the most complex and advanced Live-programming system. The Squeak environment forces the novice to make many premature commitments in its inherently abstraction hungry design. This need for abstractions can be a good pedagogical method, but one has to be careful of too much cognitive load on the user since another learning element is introduced. The HANDS system is an ambitious work which uses an interesting Pack of cards metaphor. The system is primarily designed for children since the used metaphor cannot be abstracted far and the programs that are developed can only utilize concrete objects and entities.

**Which graphical representations are suitable for live-programming?** All the systems reviewed in this work offer strong metaphors and utilise Drag-and-drop functionality in the GUI. The entities which the user works with are graphical abstractions. In Morphic the abstraction is the Morph. JPie uses functional components represented by graphical pieces that are connected. Alice uses a similar approach. In HANDS the graphical component is the Card with properties written on it.

**How is good usability reached in the design?** By utilizing the drag-and-drop functionality the interface becomes tangible and the hidden dependencies are minimized. The concrete and tangible approach to the GUI-design comes with a price. As soon as the programmer starts dealing with more advanced programming practices the metaphor will hinder the desired solutions. This is the reason that these programming systems have had their biggest success among young and novice programmers.

**How can a PyMorphic system be designed and implemented?** The PyMorphic system that this work has produced uses the Wx-library. The first solution tried to make a stronger Morphic kernel with composite morphs wrapped around the Wx-components. This proved to be cumbersome to implement and too difficult from a technical aspect. An important part of the W was to get a system running that could offer a complete user interface that could be tested on users. This was done in the second attempt. In this solution Wx-widgets were used directly for complex compounds like text editors and hierarchical outliners. An advantage with
this was that all the test persons felt comfortable and could recognise the use of all the widgets from other systems. The Liveness in the system was accomplished by using an Autoreloader class. This made it possible for all Morph objects to be updated with new class definitions, since the program made live changes to the system. The Cognitive dimensions framework was applicable to the project and the evaluation of Live-programming systems like PyMorphic should benefit from being measured according to this framework.

Is Python a suitable language for implementing a Live-programming Morphic-like system? Python is definitely a good choice for implementing a Morphic-like systems. With the resources available for this project a working solution was developed. This is thanks to the standard widgets and introspection features that Python and the Wx package offer. It is important to stress that the second solution was a Morphic-like system in many aspects, but the different widgets were not morphs in themselves as in the first solution.

Python is a popular language for writing fast applications with little code. The Autoreloader class which is essential for Live-programming worked well for the Morph class and it’s derivates. The final systems offers Liveness in that it updates instances with changes in the class definition. This did not require many lines of code in Python and proves to be a sufficient solution. On this point Python shows that it is a good programming language for creating liveness in a Programming system.

6.1 Final word

Live-programming offers a friendly approach to computer programming. The art of writing software can be and should be a creative and stimulating process. The systems reviewed in this work all have an aim to follow that spirit. They all focus on not overwhelming the user with too many simultaneous learning elements and details. This is also the intention with the PyMorphic project. At some point every scholar in programming makes the choice if they are going to continue into more advanced software engineering or if they are content with being able to understand and build simple applications. No one should have to make the wrong choice just because they were scared away with too many rules and regulations that made programming seem like something rigid and boring. Ideally everyone should be able to take control of the computer and use it as the powerful tool it is. It is likely that with ubiquitous computers, people will come closer to computers and naturally embrace programming as a mean of expressing themselves. The Live-programming approach might be
a way for end-users to customize and evolve computer systems to suit their needs. The PyMorphic system is still in an early phase of development and it is free for anyone who is interested in the field to participate in the development of PyMorphic. The project source files are available at http://pymorphic.sourceforge.net.
BIBLIOGRAPHY


[jpi] jpie.cse.wustl.edu. 33, 34


[pyt] www.python.org. 24

[sea] www.seaside.st. 31


[squ] www.squeak.org. 31


Upphovsrätt

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/.

© Anders Österholm
Linköping, 2006