Master Thesis

CrawLogo
An Experiment in End-User Programming for Web-Enabled Applications

Johan Nilsson
2003-12-12
LIU-KOGVET-D--03/16--SE

Supervisor and examiner:
Kevin McGee
Department of Computer and Information Science
Linköping University
Abstract

With the rise of the Web, there is more interest among end-users to create different kinds of software that use elements from the Web or allow for networked interaction between users. Currently however, many available tools for this purpose are hard to use or lack a sufficient level of expressiveness. To provide new insights on the construction of tools that allow end-users to create their own Web-enabled software, this thesis explores design issues and consequences of applying the Turtle metaphor from the Logo-programming language to an end-user programming environment for Web-enabled applications.

In order to explore this, CrawLogo was created – a programmable end-user tool that supports the creation of Web-enabled applications using a Turtle-like control metaphor and language adapted from Logo. As a proof-of-concept, several Web-enabled applications were created using this new tool, including CrawLogo Pong, a somewhat alternative version of Atari’s classic Pong game, and a collaborative browsing environment, in which users can browse the Web together. The resulting CrawLogo environment allows for creating Web-enabled applications that – using more traditional programming languages – would be quite complex and require deep technical programming skills. Further, while utilizing a Turtle-like control metaphor in CrawLogo allows for the creation of some new types of applications and some new ways of interacting with the Web, it also raises new problems such as how to successfully design within the CrawLogo metaphor and how to create a meaningful representation of Turtle-geometry-based navigation on the Web.
Acknowledgements

I would like to express my gratitude to a number of people for their support throughout the work and writing of this master thesis.

First of all, I particularly would like to thank my supervisor Kevin McGee. It is hard to imagine an advisor with a better mix of encouragement, knowledge and interesting ideas than Kevin McGee. As well as guiding me through this work, he taught me a new way of thinking and writing about design research, and I value his help and guidance very much.

I also would like to thank my colleagues Jody Foo, Kattis Karlsson and Sebastian Andersson for making the days and nights in the interaction design studio a lot more fun. Special thanks to Jody for being a lighthouse in the sometimes foggy seas of Java.

A big thanks to Mum and Dad, for all their support and for always being there.

And finally, thank you Frida, for putting up with me during this year, providing great comments and ideas and for saying "You can do it!".

Linköping, December 2003

Johan Nilsson
Table of Contents

CHAPTER 1  INTRODUCTION .................................................................................. 1
  1.1  OVERVIEW OF THE THESIS...................................................................... 1
  1.2  WHO ARE END-USER PROGRAMMERS?....................................................... 2
  1.3  DIMENSIONS OF PROGRAMMABILITY.......................................................... 3
  1.4  WEB-ENABLED APPLICATIONS..................................................................... 4
      1.4.1  Definition: Web-enabled....................................................................... 5
      1.4.2  Finding and Retrieving Information..................................................... 5
      1.4.3  Publishing and Sharing Information.................................................... 6
      1.4.4  Interacting With Other Users............................................................... 7
      1.4.5  Gaming............................................................................................... 7

CHAPTER 2  SURVEY ............................................................................................... 9
  2.1  HOW ARE WEB-ENABLED APPLICATIONS CREATED TODAY?................ 9
  2.2  EMPOWERING END-USERS TO BUILD SOFTWARE APPLICATIONS........ 10
      2.2.1  Programming Language Design......................................................... 11
      2.2.2  Programming Environment Design.................................................... 14
      2.2.3  Alternative Approaches to Programming............................................ 16
  2.3  EXISTING END-USER TOOLS FOR WEB-ENABLED APPLICATIONS.......... 18
      2.3.1  Finding and Retrieving Information.................................................... 18
      2.3.2  Publishing and Sharing Information.................................................... 19
      2.3.3  Interacting With Other Users............................................................... 19
      2.3.4  Gaming............................................................................................... 19
      2.3.5  What about HTML?.......................................................................... 20

CHAPTER 3  PROBLEM STATEMENT .................................................................... 21
  3.1  PURPOSE .............................................................................................. 22
  3.2  LOGO AND THE TURTLE METAPHOR..................................................... 22
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Design and Implementation</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Development Tools,</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Design Protocol</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>CrawlLogo</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Environment Description</td>
<td>29</td>
</tr>
<tr>
<td>5.1.1</td>
<td>GUI and Workspace</td>
<td>30</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Command Line and Editor</td>
<td>31</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Language and Primitives</td>
<td>32</td>
</tr>
<tr>
<td>5.1.4</td>
<td>CrawlLogo Objects</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>Putting it all Together: Programming in CrawlLogo</td>
<td>38</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Creating and Controlling Objects</td>
<td>38</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Creating and Managing Procedures</td>
<td>39</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Interacting With Other Users</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Demonstrational Applications</td>
<td>42</td>
</tr>
<tr>
<td>5.3.1</td>
<td>CrawlLogo Pong</td>
<td>42</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Collaborative Browsing</td>
<td>43</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Image Slideshow</td>
<td>45</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Guess Who</td>
<td>46</td>
</tr>
<tr>
<td>5.4</td>
<td>Unresolved issues</td>
<td>47</td>
</tr>
<tr>
<td>5.4.1</td>
<td>General Purpose Constructs</td>
<td>47</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Editor Usability</td>
<td>47</td>
</tr>
<tr>
<td>6</td>
<td>Analysis: Design Insights</td>
<td>49</td>
</tr>
<tr>
<td>6.1</td>
<td>Important Functionality</td>
<td>49</td>
</tr>
<tr>
<td>6.2</td>
<td>Turtle Goes Crawling</td>
<td>50</td>
</tr>
<tr>
<td>6.3</td>
<td>Syntonicity and Design of Primitives</td>
<td>51</td>
</tr>
<tr>
<td>6.4</td>
<td>Programmable Filters</td>
<td>53</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>55</td>
</tr>
<tr>
<td>7.1</td>
<td>Summary and Contributions</td>
<td>56</td>
</tr>
<tr>
<td>7.2</td>
<td>Directions for Further Research</td>
<td>57</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Web-Enabled End-User Programming</td>
<td>57</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Towards a Web Geometry</td>
<td>58</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Web-Driven Emergent Software</td>
<td>59</td>
</tr>
<tr>
<td>7.3</td>
<td>Final Words</td>
<td>59</td>
</tr>
<tr>
<td>8</td>
<td>References</td>
<td>61</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

End-user programming tools aim to provide computer users that lack programming expertise with the possibility to create their own software. With the rise of the Web, there is more possibility and interest to create applications that use elements from the Web or allow for networked interaction between users. As children and non-professional programmers enjoy making different kinds of software, they would probably like to create their own Web-applications. Currently however, many of the available tools for this purpose suffer from deficiencies such as still being too hard to use or lacking a sufficient level of expressiveness. This thesis sets out to provide new insights on the design of tools that allow end-users to create their own Web-enabled applications, and in particular, explore consequences of applying aspects of the “Turtle metaphor” from the Logo programming language to an end-user programming environment for Web-enabled applications.

1.1 Overview of the Thesis

In this introductory chapter we look at some general topics of the field of end-user programming such as a) characteristics of end-user programmers and b) dimensions of programmability, and also c) what types of Web-enabled applications end-users might want to create. To be specific, four broad classes of such applications are described; finding and retrieving information, publishing and sharing information, interacting with other users, and playing and creating games. All of these will be described in more detail below.
The remainder of this thesis is structured as follows. The survey in Chapter 2 gives a review of how Web-enabled applications are commonly created today, different techniques to empower end-users to create their own software, and what end-user tools for the creation of Web-enabled applications that already exists. Chapter 3 presents the goals and purpose of this thesis and also provides a brief description of the Logo programming language and the Turtle metaphor. Chapter 4 describes the design and implementation protocol that was used when developing CrawLogo – a programmable environment for the creation of Web-enabled applications with a Turtle-like control metaphor and language adapted from Logo. The CrawLogo environment itself is described in detail in chapter 5. Chapter 6 presents an analysis of some of the consequences of applying the Turtle metaphor in a Web-context, followed by a brief summary and conclusions in chapter 7.

1.2 Who are End-User Programmers?

Many people engage in programming-like activities every other day, often without thinking about what they do in terms of programming. These activities can include making a graph within a spreadsheet-application like Microsoft Excel, creating rules in an e-mail-client for how mail from different senders with different subjects should be handled, or setting the VCR to record a TV-program.

Some twenty years of widespread use of application-development environments such as spreadsheets, statistical packages and CAD-systems shows that end-users can be empowered to create their own – often surprisingly sophisticated – applications. While end-user programming environments often lack the expressive power of general purpose programming languages, they also lack the steep learning curve [47]. Various research and tools have shown that even children can successfully engage in programming-like activities such as creating their own games or simulations [14][34][50].

An important point here is that end-user programmers are not professional programmers, but they are also not to be regarded as naïve, or novice users [47]. They are musicians, physicists, children, students, economists, designers or anyone who might want to create or customize their own useful, interesting or fun piece of software.
1.3 Dimensions of Programmability

Current definitions of programming often state something like: “to write a series of instructions which make the computer perform a particular operation” [9]. As we will see, definitions like this might be somewhat obsolete and would exclude many tools commonly referred to as “programmable”. Thus, a more comprehensive definition of programming could be stated as instructing the computer to perform a task or series of tasks. A detailed compilation of historic and research definitions of programming is provided in [6].

In programmable applications, there can also be various types of programming. Nardi classifies programming tools according to their degree of expressiveness and interactive construction [47], where expressiveness refers to how powerful a tool is in terms of what it can represent, and interactive construction to the extent to which the tool has interactive interface elements such as icons, buttons, menus, toolbars, etc. Rough as this taxonomy might be, it provides some sort of classification tool for different types of programmable applications.

![Expressiveness Diagram]

*Figure 1: Programmable systems organized by degree of interactive construction and expressiveness. Reprinted with permission from [47], Copyright © 1993 Massachusetts Institute of Technology.*
To exemplify, spreadsheets have a high interactive construction while many traditional textual programming languages have a low interactive construction but are very expressive. With these dimensions of programmability in mind, tools suitable for end-user programmers would preferably have both a high degree of interactive construction and expressiveness.

Another, more purpose-based, classification of programmable environments divides tools into systems that attempt to teach programming for its own sake, and those that attempt to support programming in pursuit of another goal [32]. In general, end-user programming tools would fall in the latter category, since their primary purpose often is to empower users in a specific task that they are trying to perform.

In the following section we will look at some Web-related tasks where end-user empowerment would be useful and what kind of applications end-users might want to create.

1.4 Web-enabled Applications

The World Wide Web (WWW or “the Web”) provides a vast resource for finding and retrieving information, sharing files, communicating with other people and much more. With the rise and evolution of the Web, there is also more possibility and interest to create Web-enabled applications, and just as children and non-professional programmers enjoy creating, customizing and extending different kinds of software, they probably would like to make their own Web-enabled applications, with the Web providing various ways of collaborating with others, sharing work and reaching a broad public. However, many of the commonly used tools for doing this are complicated and require deep technical understanding and knowledge of various programming languages.

To distinguish between different uses of Web-related terms, this section presents a definition of how the term Web-enabled is used in this thesis. It also provides a short review (sections 1.4.2 - 1.4.5) that aims to give and idea (and not in any way a complete list) of what kind of activities users are engaging in when using the Web and different types of Web-enabled applications end-users might want to create or customize. The review is divided into a) finding and retrieving information from the Web, b) publishing and sharing of information, c) interacting with other people and d) gaming.
1.4.1 Definition: Web-enabled
The term Web-enabled in this thesis refers to applications that have a Web interface, that is, some way to retrieve or operate on content from the Web and use the infrastructure of the Internet to allow for interaction with other users. Thus, two very important properties of a Web-enabled application are a) permanent access to vast amount of constantly changing and increasing data, and b) networked connections to other users. As we will see, these properties are emphasized in many interesting and useful tools and approaches for performing tasks on the Web.

The definition of “Web-enabled applications” stated above excludes some types of applications that are commonly included when talking about “Web applications”, such as many Java applets. This is not to say that no Java applets can be Web-enabled, but, in this definition, running within a Web page does not automatically mean that an application is Web-enabled.

1.4.2 Finding and Retrieving Information
Using the Web to find and retrieve information can sometimes be difficult and time-consuming, due to the complexity in organization and quantity of information stored. This has led to a lot of research and development of tools that helps users find what they want.

The most common way of finding things on the Web is via search engines, such as Google\(^1\), Alta Vista\(^2\), and AllTheWeb\(^3\). The term "search engine" is often used generically to describe both crawler-based search engines and human-indexed directories. The latter represent a very small percentage of search engines; in fact, all search engines mentioned above are crawler-based. This means that they automatically crawl the Web and store their findings in large databases, which people then can search.

Although search engines are very helpful when looking for information on the Web, it can still be difficult to find the information wanted since a) it is hard for users to accurately express what they want, and search engines do not adapt their search strategies according to different users, and b) most Web pages are irrelevant to a

\(^1\) http://www.google.com
\(^2\) http://www.altavista.com
\(^3\) http://www.alltheweb.com
particular user’s interests [10]. Web crawlers (also referred to as agents, spiders or robots) aim to be useful for users by providing personalized and customized information, and can be used for, e.g., data mining, visualization or Web-browsing assistance [37][10].

While search engines and Web crawlers can be seen as standard tools for finding and retrieving various kinds of information or files on the Web, there are also tools that provide alternative approaches to these tasks. Such tools include CollageMachine⁴ [33] and N.A.G.⁵. CollageMachine is a tool which retrieves media elements from the Web and arranges them into a collage. By organizing and managing the retrieved elements in different ways, the user informs the systems of what he or she is interested in, and the collage evolves over time. N.A.G. creates a similar collage, however with audio files. Based on user queries, N.A.G turns the process of searching and downloading mp3 files into a “chaotic musical collage”⁵.

1.4.3 Publishing and Sharing Information

Today, any user with access to a computer and an Internet connection can become an informal online author and publisher. Recently, the interest for doing this seems to have exploded worldwide which have given birth to phenomena such as blogs (Web logs) and Wikis (online editable encyclopedias). Blogs are Web pages, in some sense similar to diaries, which largely consist of citations or pointers to other Web pages, often along with some thoughts and reflections. Returning to the issue of authoring, some research has realized the usefulness of the possibility to interact with other users and developed tools for distributed authoring [26].

A frequently performed activity is file-sharing. The basis for sharing files via the Internet is the file transfer protocol (FTP), thus the prototypical of a file sharing application is an FTP-client, but today many tools and online services provides built-in mechanisms for the exchange of files (e.g. ICQ⁶, Kazaa⁷, Direct Connect⁸).

⁴ http://www.csdl.tamu.edu/ecology/combInformaation/
⁵ Network Auralization for Gnutella: http://turbulence.org/Works/freeman/
⁶ http://www.icq.com
⁷ http://www.kazaa.com
⁸ http://www.neo-modus.com
1.4.4 Interacting With Other Users

With many online places for social interaction, from text-based newsgroups and mailing lists to virtual cities or communities where users can interact and talk to each other through avatars, the Web can play an important part in enabling people to interact with each other [24]. There are also many native applications that provide possibilities to communicate and interact with other people in text, audio or video. These include e-mail, chat clients (e.g. ICQ and IRC⁹), various types of voice communication tools and more advanced real-time collaboration tools such as Microsoft NetMeeting¹⁰ and Groove¹¹.

1.4.5 Gaming

The convergence of computer- and videogames and the Internet continues, and interest from game-developers in using the Internet to add to the gaming experience is constantly growing. Many games today – from text-based MUDs¹² to 3D-MMORPGs¹³ – use the Internet for supporting e.g. multiplayer modes and in-game real-time voice communication. Further, there are probably hundreds of Web sites where a user can play a game directly through their browser. As technology to develop such games becomes easier to use, more and more people will be able to visit as well as creating such sites themselves.

---

⁹ http://www.irc.org
¹⁰ http://www.microsoft.com/windows/netmeeting/
¹¹ http://www.groove.net
¹² Multiple User Dimension, Multi User Dungeon, or Multi User Dialogue
¹³ Massive Multiplayer Online Role Playing Games
Chapter 2
Survey

We now have an idea of what types of Web-enabled applications end-users might want to create. In order to understand the current state of work that has been done on providing end-users with the means to create their own Web-enabled applications, the following review describes a) how such applications are commonly created today, b) various techniques for empowering end-users to build software applications, and c) existing end-user tools for creating Web-enabled applications.

Since almost no tools provide end-users with more general means to create a wide range of different types of Web-enabled applications, section 2.3 accounts for tools that support more specific areas of Web-enabled application development, divided into the previously described four classes of applications that end-users might want to create.

In general, what we see is that there are some powerful and some easy-to-use tools for developing Web-enabled applications – but almost none that are both. We also see how different techniques can make programming languages and environments easier to use and how metaphors and “intelligent” elements can help users in different ways when creating their own applications.

2.1 How are Web-enabled Applications Created Today?
There is a wide range of development tools commonly used for creating Web-enabled applications such as games, tools for finding or sharing information or communicating and interacting with other people. The notion of Web programming languages is really wide and
quite vague but usually includes languages, protocols and tools such as Java and JavaScript Perl, PHP, CGI, ASP, XML, HTML to name a few.

These tools will not be described in detail here. A more elaborated review and evaluation of the most important Web programming languages is available in [60]. In general, using these tools to create Web-enabled applications such as various kinds of games, applications for finding or sharing information or for communicating and interacting with other people often require deep technical and programming knowledge.

We now turn to look at what work that has been done on providing non-programming experts with the means to create their own software.

2.2 Empowering End-Users to Build Software Applications

Many researchers suggest the main reason why traditional programming is hard as the presence of a gap between a user’s mental model of a problem and this in a form accepted by a computer [6][55][45][49]. To bridge this gap and make programming easier there are basically two ways: moving the system closer to the user or moving the user closer to the system [55].

End-user programming aims to provide computer users without a formal programming background with the possibility to create their own software by utilizing various techniques to move the system closer to the user. Designed to be easy to learn and write, SmallTalk [31] was, together with the Logo programming language [50] (described in section 3.2), among the first end-user programming languages when it first appeared in 1972. SmallTalk included a simple yet highly expressive syntax and also set the standard for object-oriented programming.

The essence of end-user programming is nicely captured by Alan Kay, who in the foreword of [16] states that “One should not have to learn about Carnot cycles or of combustion engines \(--/ \) in order to drive a car”, the point being that end-users who want to create or customize their own software should not have to learn about technicalities such as memory allocation or thread handling. More specifically, motivation for programmable applications for end-users is that they allow a)
exploration of concepts never conceived by the system designers [20] and b) the creation of applications in an easy way [19][47]. A third central idea is to c) change the role of the users from consumers to producers; from passive absorbers of information to active creators [1][18][20][21].

Since the requirements of end-user programming environments differ widely depending on the users and their purposes, there is no single general method or approach of end-user programming [25]. Instead, there is a variety of techniques, which we will look at below. In general, these techniques and their attempts to make the creation of applications for end-users easier can be divided into a) simplifying the programming language, b) making the programming environment more usable and c) introducing “alternative” programming techniques.

2.2.1 Programming Language Design
A universal characteristic of programming situations is that a program is represented in some notation [6]. As traditional text-based programming is hard it would probably be a bad idea to embed a language such as C/C++, Java or Perl in a tool for non-programming experts. To make programming more accessible for end-users, attempts have been made to simplify the language used in programmable applications, by a) making programming more similar to ways of expression that people use in natural spoken or written languages and b) developing “lightweight” scripting and macro languages. We also look at some advantages and drawbacks with general purpose and domain specific programming languages.

2.2.1.1 Making the Language More Natural
Leveraging users’ existing language knowledge has been frequently suggested as an effective strategy when designing programming languages for children, end-users, or others new to coding [8]. Since natural language (vocal or written) mirrors our normal way of interacting with our environment we do not have to learn a new syntax or grammar; we already know how to talk and write.

While it has been argued that informal conversation is a “paradigm case” for human computer interaction [47], research on the use of unconstrained natural language has reported numerous difficulties with actually implementing this into a system. Two major obstacles include semantics and world knowledge [40]. The problem of semantics has to do with the extent to which “meaning” (to a word, phrase or
sentence) is attached to the context in which the conversation takes place. And as for world knowledge, much of what happens during any conversation-like activity is dependent on shared experiences and previous knowledge, also referred to as common ground [11]. Most humans seem to have very little common ground with computers.

There are some general suggestions on how properties of natural languages can be employed within programmable applications [40][8], and many deal with various restrictions and constraints. As just pointed out, there are problems with using spoken or written English as a programming language, but with some constraints, such as limited vocabulary, sentence length, prohibitions of complex constructions [40] and a somewhat formalized syntax [8][49] the use of a natural-like language may prove beneficiary for end-user programming environments.

Some tools (e.g. MOOSE [7], HyperTalk14 and Macromedia Director15) have shown how this approach can be successfully employed. In Macromedia Director, for instance, the user can switch between a Java or C++-like dot-syntax and a more English-like way of expression. The following Lingo-code shows how to make a graphical object 50% transparent using both ways:

```
sprite(8).blend = 50

set the blend of sprite 8 to 50
```

This allows beginners to create their own behaviors and scripts in a simple way, and at the same time, experienced C++ or Java-programmers will feel at home, using the dot-syntax.

2.2.1.2 Scripting and Macro Languages

The boundary between scripting languages and traditional ones is somewhat blurry [30][5] but basically the former are “lightweight” languages, designed for quick, short programs, and meant to be easier to learn and simpler and faster to use [5]. These properties make them interesting in the context of end-user programming. Scripting

15 http://www.macromedia.com
languages that are commonly used for creating Web-applications include Compaq’s Web Language\(^{16}\) JavaScript and Perl.

However, though all of these strive to be easy-to-use means for creating Web-applications, they are still quite complex and by some argued to be not really well suited for non-programming experts [41].

An attempt to make JavaScript more accessible for end-users is bookmarklets\(^{17}\); small JavaScript applications that are contained within a bookmark, which can be edited, saved and used in the same way as normal bookmarks. These are very useful for simple tasks such as changing the background color of a Web page, but when it comes to more complicated tasks like counting words on a Web page the lack of feedback and little overview in the URL-field of a bookmark makes it hard to create bookmarklets of more complex nature. Further, the syntax for a given behavior may differ between different Web browsers.

Applications that support *macro recording* provide users with a way to record a sequence of actions. This is done by invoking a “Start Recording” command, performing a series of actions, and then invoking a “Stop Recording” command. The user can then “replay” the action sequence with a “Redo” command, using a shortcut such as a mouse click or a key-press. Besides recording sequences of direct manipulation actions, macros can also be programmed textually. The strength of macros is that they require little technical knowledge – the user simply records normal interaction with the application. The primary benefit of macros is that the user can automate common tasks.

### 2.2.1.3 General Purpose vs. Specific Purpose Languages

When developing a tool in which users can make their own programs, there is the option of whether to base it on an existing general programming language, or to create a new, special-purpose language.

The opinions regarding this matter differ widely. Some previous research dismisses the more general-purpose way of programming and suggests that the only approach that will succeed in practice is domain-specific programming environments [55]. This point of view

\(^{16}\) Formerly known as WebL.

\(^{17}\) [http://www.bookmarklets.com](http://www.bookmarklets.com)
is primarily supported by the argument that a more specialized representation more closely maps to the users’ conceptualization of the problem, thus having more problem solving power [55].

However, one problem with this approach is that end-users will find it difficult to bring knowledge from one environment to another, and have to learn a new syntax or method of interaction for every new tool they use. Eisenberg [19] for instance, has stated strong advantages of starting with a “standard” language and extending it for use within a particular domain, as opposed to creating a unique ad hoc language. Besides immediately having access to a vast amount of general programming constructs, this also means that any user who learns the basics of for instance Pascal would quickly be able to gain proficiency of any Pascal-based programmable tool.

What can be seen as a third approach to the problem of whether to use a general or a specific language is to set meaningful tasks within the domain of the user’s interests as a starting point and in the process of designing programming languages have the overriding goal to support users’ skills and interests [47][19]. Thus, rather than stating that one approach is generally preferable over the other, these user skills and interests should guide the design of both form and properties of the language.

2.2.2 Programming Environment Design

In the previous section we looked at some important considerations regarding programming language design. Although there are some useful and interesting ways to make the language of programmable applications simpler, some of these tools have been criticized for still being too hard to learn for beginners [46]. A different approach is to focus on environment rather than language design, which often includes compromising some expressiveness for ease-of-use, utilizing a variety of metaphors and graphical interface elements.

This section describes how graphical user interfaces and metaphors can empower users when interacting with a system.

2.2.2.1 Graphical User Interfaces

Systems that are easy to learn and use often have a direct manipulation front end, which “significantly reduces the amount of necessary scripting” [46] or programming. While a well designed programming language may provide a good match between language primitives and our
language of thought, two important benefits of direct manipulation interfaces are ease-of-learning and ease-of-use [19]. It is for instance both easy to learn and easy to remember that a button with an image of a pen has something to do with drawing, and that a button with an image of a printer means printing the document.

There are many programmable tools that illustrate the benefits of graphical user interfaces. Not previously referred to in an end-user programming context is PDP++ [48], a system for creating artificial neural network simulations. The system is easy enough for novice users (students of neural networks), who via the graphical interface can set up, train and test a network without having to deal with much of the complex mathematics and algorithms of neural networks, but at the same time, it is very powerful and flexible enough for research use.

The major disadvantage of direct manipulation interfaces has been argued as the difficulty to express conditionality and iteration graphically [46].

2.2.2.2 Metaphors
As previously indicated, people tend to perform better at tasks when the problem is framed within a familiar context. Thus, the general idea of basing a tool and the interaction with that tool on a metaphor is that users will often find it easier to interact with the tool, because they can take advantage of previous knowledge and experiences.

Further, as argued by Blackwell [6]:

“If we can exploit commonality between conventionally recognized programming activity and other contexts of human tool use, this offers the opportunity both to apply insights from the world of programming to other fields of tool design, and also to adopt creative solutions for programming tools that have been identified in atypical programming situations.”

This suggests that meaningful metaphors can be especially important in programmable tools for non-professional programmers and children, since it can provide them with a more concrete way of thinking about often abstract and complex computational constructs.

Some tools such as Logo [50] and LiveWorld [59] have used metaphors to make objects more real for the user. Travers [59] for instance, introduces the animate metaphor, using animate terms to describe the activity of a machine or other non-living object and
suggests that picturing the components of programs as objects that can have explicit goals, a degree of autonomy and the ability to dynamically react to their surroundings can be useful to make the operations of complex systems such as computers more comprehensible. In Logo, the user controls a Turtle to draw graphics or create animations (The Logo programming language and the Turtle metaphor will be described in section 3.2).

2.2.3 Alternative Approaches to Programming

Approaches that can be regarded as “alternative”, with respect to traditional programming, aim to frame the activity of programming in a more intuitive, usable or more easily grasped manner. Examples of such approaches include gesture-based programming [62], visual programming [29], programming by physical manipulation [44] and even using a mixture of several different concurrently active programming paradigms [13].

This section describes an approach that seems especially interesting and useful within a Web context; the use of intelligent elements or agents to aid users in creating their own software. First some general ideas behind adding intelligence to programmable tools are described, and then we look at the Programming by Example paradigm in particular.

2.2.3.1 Adding Intelligence

As shown by the extreme programming (XP) paradigm [3], the presence of a partner\footnote{What Beck refers to as pair programming.} can be very valuable in the activity of programming. While one user types, the other one thinks tactically about related things and how what is currently being created fits with the “big picture”, this resulting in much higher work quality. Although not explicitly trying to follow the rules and practices of XP, similar thoughts have been expressed with the idea of programming with “agents”. The other part of the pair, in this case, is a part of the programming environment with “explicit goals, a degree of autonomy and the ability to dynamically react” to the surrounding [59].

The idea behind using intelligent agents as collaborative partners in tools is that it allows users to “think in terms of purpose and function, and thus to understand without having to know the ‘implementation details’ of
"the system" [59], i.e., users can to a higher extent focus of what is really interesting with the task at hand, instead of time-consuming, boring or difficult parts.

Other tools that include computational collaborative partners are Letizia and its successor Let’s Browse [37][39]. Letizia is an agent that that assists a user browsing the Web by inferring a rough approximation of the user’s interests from his or her past Web browsing activities. This differs from much other work on “intelligent” browsing aids such as Web Crawlers, where users often have to explicitly state their interests in form of rules or keywords.

2.2.3.2 Programming by Example

Programming by Example (PBE, also commonly referred to as Programming by Demonstration) is an approach to end-user programming that can be seen as elaborating on the ideas behind embedding intelligent elements, visual programming and the use of macros. The general motivation for PBE is that:

“[I]f a user knows how to perform a task on the computer, that should be sufficient to create a program to perform the task. It should not be necessary to learn a programming language like C or BASIC. Instead, the user should be able to instruct the computer to ‘Watch what I do’, and the computer should create the program that corresponds to the user’s actions” (Allen Cypher, in [16]).

PBE aims to solve this problem by combining the simple interface of macros with the expressiveness of a scripting language [35]. There are a number of different approaches encompassed by PBE, but they all focus on the use of concrete examples rather than describing procedures abstractly [38].

A commonly noted shortcoming of macros is that they are too literal; if the repetitive task has minor variations the macro has limited utility [35]. For example [(scenario from [16]), if a user select cells A1 through A15 within a spreadsheet, he or she may actually want to select this month’s employee sales, which may later occupy cells A1 through A16 if a new employee joins the department.

Thus, what PBE actually sets out to do is inferring and generalizing users’ intentions from a given set of examples. Unlike macros, however, PBE programs can contain control structures such as iteration and conditionals [35].
Actually implementing the intention-inferring part is probably the most difficult problem to solve, this having resulted in many PBE tools being very task specific (used in restricted domains). Hence, criticism against PBE often takes hold of this point, arguing that this is the reason why PBE will not be replacing traditional programming for large-scale programs. This issue is addressed by Lau and Weld [35], who frame PBE as a machine learning problem and have focused on developing domain-independent methods for performing generalization in a wide range of applications.

2.3 Existing End-User Tools for Web-Enabled Applications

This section reviews existing tools and approaches that in various ways employ the techniques described in the previous sections to empower end-users to create different types of Web-enabled applications. In essence, we now look into what answers to the question “How can end-users create their own Web-enabled applications?” that are already provided. An extensive survey of more general programming environments and languages for non-expert programmers is available in [32].

2.3.1 Finding and Retrieving Information

Creating applications that find and retrieve information from the Web is still something quite complex, and not much work has been done on providing end-users with easy-to-use means to construct their own search tools or Web crawlers. There are some toolkits for creating Web crawlers such as “The Web Crawler Toolkit” in IBM’s “Intelligent Miner for Data”\(^\text{19}\), but these are directed towards professional programmers. Tools like WebSPHINX [42] and FeedDemon\(^\text{20}\) however, provide intuitive GUIs and a simplified syntax for creating scripts/rules with which users can customize tool-behaviors and settings to create a personalized Web crawler or newsgathering tool.

Since the common way of interacting with the Web is via a Web browser, a very natural approach when creating a Web programming language is to extend a Web browser with some kind of programming capabilities [43]. This allows for various highly expressive operations

\(^{19}\) http://www.software.ibm.com/data/iminer/

\(^{20}\) http://www.bradsoft.com/feedemon/
and procedures to be applied to the content of a Web page in order to, for instance visualize content or count occurrences of a phrase.

2.3.2 Publishing and Sharing Information
Some work provides means for non-programmers to be authors and publishers of media for the Web, such as digital libraries [58] and Wikis. Although not especially designed with children in mind, a Wiki is “the simplest online database that could possibly work”[21] and consists of a set of Web pages that can be freely edited by anyone, using a Web browser. Wikis encourages democratic use of the Web and promotes content composition by non-technical users.

2.3.3 Interacting With Other Users
As mentioned in the introduction, the Web provides useful means for enabling people to interact with each other online. Recently, online meeting spaces have gained interest among researchers, educators and business people [28], something that has lead to further development of new virtual places and tools for interaction between users.

Some end-user tools support networking tasks such as messaging and networking. In ToonTalk[29], a bird can fly to bird nests on other computers and deliver messages, and in Net Logo[23], a whole school-class can take part in running and controlling a network-simulation. Other tools, such as MOOSE Crossing [7] and Pet Park [17] emphasize the importance of community support for application construction and allow users to build virtual environments, where they can meet, collaborate and interact with other users from all over the world. Common for these tools is that they empower users by providing an easy-to-use GUI combined with meaningful control metaphors.

2.3.4 Gaming
As for creating games, some research has been done on developing construction kits primarily directed towards children [4][7][17] that allows for easy creation of simulations and games via visual programming techniques and rules/behaviors-scripting. Moving away from ease-of-use towards expressiveness, there are more advanced

[23] http://ccl.northwestern.edu/netlogo/
toolkits such as the RPG Toolkit\textsuperscript{24} and DarkBASIC\textsuperscript{25}. These tools provide a “low threshold – high ceiling” that allows users to combine various “blocks” of functionality and as they grow more experienced, create their own graphics or behaviors.

2.3.5 What about HTML?

As described, there are a lot of tools and research dedicated towards making programming easier for non-programmers and children, the widest spread of all time probably being HTML. HTML allows users to create various kinds of Web-content in no time, and learning the language’s syntax is no longer required thanks to an abundance of HTML construction kits (Macromedia’s Dreamweaver, Adobe’s GoLive, or Microsoft’s FrontPage to name a few). However, HTML is a markup language rather than a complete programming language, and is as such limited in expressive power and more suitable for authoring \textit{content} for the Web than providing a tool for the creation of more sophisticated Web-enabled applications.

\textsuperscript{24} http://www.rpgtoolkit.com
\textsuperscript{25} http://www.darkbasic.com
Chapter 3
Problem Statement

As we saw in the survey, there have been a number of attempts to provide end-users with interesting and “intelligent” network applications – and to make the creation of applications easier by simplifying the programming language or making the programming environment more usable. Some tools provide interesting interfaces for retrieving and operating on pieces of information from the Web, but users are often pretty much left with the built-in algorithms and behaviors of the system, and thus, these tools lack the expressiveness to perform tasks beyond those imagined by the designer. In general, few existing end-user tools for the creation of Web-enabled applications provide an expressive programming language in combination with a meaningful metaphor and ease-of-use.

Introducing the “Turtle” as an object to think and program with, the Logo programming language (further described in section 3.2 below) shows that a tool that provides a meaningful control metaphor and takes existing (body) knowledge into account can help users do powerful and interesting things with that tool [50]. Originally, Logo and its Turtle provided an alternative way to explore and comprehend geometrical concepts. To empower users in areas other than geometry, properties of Logo have been employed in tools for e.g. musical perception [2], modeling decentralized systems [51], game design26 and exploring mathematics [12]. Using a similar approach, it would be interesting to see how aspects of Logo could be

26 SeeLogo: http://www.ithaca.edu/seelogo/
utilized in a programmable tool for creating Web-enabled applications.

3.1 Purpose
The general aim of this thesis is to explore a new approach and provide insights for the design and construction of end-user tools for Web-enabled applications. In particular, the purpose of this thesis is to explore consequences of applying aspects of the "Turtle metaphor" from the Logo programming language to an end-user programming environment for Web-enabled applications.

Extending and adapting the Turtle metaphor and Logo language for a Web context raises important issues of language and environment design, such as creating new types of Turtles and new computational constructs. It also involves identifying advantages and problems with this approach, and to see what types of Web-enabled applications that can be created using a Logo-based end-user programming tool.

In order to give a better understanding of the Turtle metaphor – and to help the reader understand ways in which the Logo paradigm has been extended in the work of this thesis – the rest of this section provides a brief description of Logo and the Turtle metaphor.

3.2 Logo and the Turtle Metaphor
In the Logo programming language [50], a Turtle gives the user control over geometry in order to make complex geometric designs, drawings or animations. In Euclidian geometry, a point in a coordinate system is an entity that has a position but no other properties, and geometrical shapes can be described by specifying the relationship between a set of points. The Logo Turtle is in many ways like such a point, but it also has additional properties, such as a size and a heading, i.e. it faces some direction. The user can create drawings by controlling the Turtle’s movement and rotation, using the commands FORWARD and BACK [a number of steps] and TURNLEFT and TURNRIGHT [a number of degrees]. In essence, what the Turtle does for the user is:

- making a mathematical point concrete (thus adding a meaningful metaphor)
- adding a notion of heading (e.g. facing NORTH)
• providing a control language that is meaningful for the user
  (FORWARD, RIGHT, etc.)

Thus, the Turtle is a “computer-controlled cybernetic animal that exists
within the cognitive minicultures of the Logo-environment” [50] and serves
as an “object-to-think-with”, that is, an externalization of
computational processes, that is good to program and good to think
with. When controlling the Turtle’s movements, users can relate to
their own bodies and how they move in space.

![Figure 2: A Turtle that started with a heading of 0 (north) and has done FORWARD 100, TURNRIGHT 90, FORWARD 100, TURNRIGHT 90 and finally FORWARD 100.](image)

Leveraging existing knowledge in the process of learning something
new is referred to as syntonic learning, and syntonicity is sometimes
used in a programming context to describe language-design that is
harmonious with the cognitive style or thinking of the programmer
[50]. Thus, syntonicity can be seen as representing the “resonance
between external forms and concepts and what people know about
themselves” [63].
Chapter 4
Design and Implementation

In order to explore design issues related to the creation of end-user tools for Web-enabled applications and extension of the Logo programming language and metaphor for use in such tools, the following was done:

- design and implementation of CrawLogo – a tool with an embedded programming language that allows for the creation of Web-enabled applications
- design and exploration of a Turtle-like control metaphor, based on concepts and frameworks from the Logo programming language, for the construction of such applications
- creation of several Web-enabled applications using this new programmable tool

CrawLogo and the demonstrational applications that were created are described in detail in the following chapter. This chapter describes and motivates a) development tools and b) the design protocol that was used when developing the CrawLogo environment.

4.1 Development Tools
The CrawLogo environment was developed in Java 1.4.2\(^7\) using Eclipse 2.1.1\(^8\). Java was chosen because of its platform independency

---
\(^7\) [http://java.sun.com](http://java.sun.com)
\(^8\) [http://www.eclipse.org](http://www.eclipse.org)
and good support for networking and concurrency. The application’s GUI was created using the Java Swing package.

The CrawLogo interpreter was based on an existing Java implementation of Logo called rLogo. rLogo was chosen since it provided a well documented open-source Java implementation of Logo’s core primitives that could easily be extended with custom primitives and modules.

4.2 Design Protocol
The design and implementation phase was very exploratory driven and consisted of iterations of approximately one week length. Each iteration was followed by a reflection session, where the tasks for the next iteration were decided.

Thus, the CrawLogo environment itself and the use of this tool constituted a very important part in suggesting the next course of action. In a sense, the design and implementation process can be seen as a dialogue with the situation – problem material and context at hand – where the situation “talks back” and the designer then

---

29 http://www.embry.com/rLogo/
responds to the situation’s back talk. This approach is grounded in thoughts of Donald Schön [54], who further states that:

“In a good process of design, this conversation with the situation is reflective. In answer to the situation’s back-talk, the designer reflects-in-action on the construction of the problem, the strategies of action, or the model of the phenomena, which have been implicit in his moves“.

One significance of such a design strategy is that one may have an idea of how something should be realized, but it is not until these thoughts are actually externalized and seen in action that one can get a proper understanding of whether they are good or not.

To summarize these thoughts in a tool-development context, utilizing short iterations and actually use the tool in between can provide valuable insights and ideas on how to proceed with the work.
Chapter 5
CrawLogo

This chapter provides a detailed description of CrawLogo, and is divided into four parts that describe a) the structure, functionality and capabilities of the CrawLogo environment, b) how to create and manage procedures and objects, c) a number of demonstrational applications created in CrawLogo, and d) some problems and unresolved issues with the implementation in its current state.

In general, applications are developed in the CrawLogo environment by creating and managing procedures and various objects, such as Web browsers and filters, using Logo-like syntax and semantics. As a demonstration of what types of Web-enabled applications can be created, a number of demonstrational applications developed in CrawLogo are described, including CrawLogo Pong; a somewhat “alternative” two-player Web-enabled version of the classic Atari game, a collaborative browsing application that allows users to browse the Web together, and a slideshow that displays images from the Web.

5.1 Environment Description

In CrawLogo, users can create their own applications that run within the CrawLogo workspace. These applications can be local – running on one computer, or distributed – running across two or more computers.

The core parts of the environment are the user interface, the CrawLogo interpreter and a number of programmable objects. Figure 4 below shows a schematic overview of the system and Figure 5 shows a screenshot from the CrawLogo workspace. Users interact with the tool via the
command line or the editor, or by direct manipulation of objects placed in the workspace.

Besides these core parts, the system is designed as a set of individual modules each of which provides different functionality to the system. Thus, adding new features, objects or functionality to the system would simply involve plugging in the new modules to the system and adding corresponding CrawLogo primitives.

The rest of this section will describe the different parts and aspects of the environment.

5.1.1 GUI and Workspace

The GUI in CrawLogo consists of a workspace where the user can create and control various CrawLogo objects. Using the system often involves a combination between direct manipulation (using the mouse) and giving textual commands, although the programming part is purely textual.
5.1.2 Command Line and Editor

The most important objects in the GUI are the command line and the editor (see Figure 6 and Figure 7). From these, the user has access to all commands and primitives of CrawLogo.

**Figure 5**: CrawLogo workspace with procedure editor and command line (to the left) and various CrawLogo objects.

**Figure 6**: The Command Line
The command line is primarily intended for instantly invoking procedures and shorter commands or series of commands, without adding them to a procedure. This is useful for controlling and manipulating CrawLogo objects “on the fly”, e.g. during a game.

The editor is used to create and manage longer procedures. The editor window consists of four parts – a procedure editing frame, a list of existing procedures, a message frame (where messages from other users and system messages are displayed) and a panel with action buttons for evaluating, executing, editing, saving and loading procedures.

5.1.3 Language and Primitives
As CrawLogo is based on rLogo, an open source Java implementation of Logo, it contains the core set of Logo primitives, including support for creating procedures, subprocedures and variables, a basic set of Turtle graphics primitives, flow control (e.g. REPEAT and WHILE clauses) and some logical expressions (e.g. AND, OR, NOT).

A set of custom primitives were created, partly for making up for shortages in rLogo, and partly for adding new functionality that was not previously available in Logo tools. These custom primitives
include support for threading and networking and creating and manipulating CrawLogo objects. Some graphics primitives for drawing on the screen in rLogo (e.g. controlling the pen) were removed.

The new CrawLogo primitives were designed to map closely onto task-specific operations that users might want to perform in their applications, but at the same time be general enough to support the construction of a wide range of applications. For example, when starting a server that other users can connect to, most end-users would probably not want to, nor should they have to, get into details like socket handling and sending packages, but just get the server up and running and start interacting with other users.

5.1.4 CrawLogo Objects

One sense in which CrawLogo builds on the tradition of Logo is that it tries to make objects real to the user. As we saw in the survey, this can be done by giving the objects properties of physical objects, such as appearance, location, responsiveness to action [59] and names.

Originally, the Turtle in Logo was primarily a “drawing Turtle” which internal state was limited to having a heading and a pen that was either up or down. In subsequent versions and variants of Logo, the Turtle progressed to being more of a “behavioral Turtle” that could have additional and more sophisticated properties and interact with other Turtles and with the world around them [52]. In CrawLogo, various kinds of objects can be created. In analogy to more subsequent Logo tools, these objects can be conceptualized and controlled as “behavioral Turtles” with different properties and capabilities. Object concurrency is supported in the sense that all objects can act simultaneously.

By using various CrawLogo “Turtles” as “objects to think with” [50], creating and playing with an object is meant to stimulate creativity and give rise to ideas on what it can be used for and what role it could play in a larger application.

5.1.4.1 Web Browsers

The Web Browser in CrawLogo allows users to retrieve and view Web pages. Users can interact with this Web browser in the same manner as with traditional Web browsers (entering URLs in the address field
and clicking on hyperlinks), but they can also control it using textual CrawLogo commands.

Figure 8: A CrawLogo Web browser.

When a new Web page is displayed, the Web browser Turtle automatically extracts all hyperlinks contained in that page and maps them around itself as “exits” to other Web pages (see Figure 9). Telling the Turtle to move forward will take the Turtle to the Web page that it is currently facing, and telling the Turtle to turn right will direct it towards another Web page, or “exit”.
Thus, browsing the Web in CrawLogo means conceptualizing Web pages as nodes with exits that leads to other nodes. Since Web domains can have directories and subdirectories, the Web browser turtle was equipped with vertical movement (movement between directories), adding a 3rd dimension to CrawLogo Web movement. In some sense, a Web domain can be thought of as a house, with numerous doors (hyperlinks) to rooms (other Web pages) on different floors (directories and subdirectories). The relationship between Web pages on different directory levels in a Web domain is illustrated in Figure 10 below.

As the “width” of each exit is calculated by $360/n$, where $n$ is the number of hyperlinks of the current Web page, the Turtle will always be facing one specific exit, and all exits will have the same width. As stated above, changing the heading of the Turtle will (unless there is only one exit) turn it towards a new exit, and invoking the FORWARDWEB (FDW) command will tell the Turtle to go through that exit into a new Web page.
5.1.4.2 Shapes

Shapes are the objects that are most similar to the traditional Logo Turtle. They are simple graphical objects that can be instructed to move on the screen and interact with other objects. Currently, quadrangles and ellipses of different sizes and colors can be created. The shape-objects were created with the intention that they could function as “carriers” of information to be sent between other objects. Since one common property of CrawLogo objects is to hold an URL, one possible use of shapes is to store bookmarks.

Figure 10: Conceptual model of a Web domain, with some possible hyperlinks between pages.

Figure 11: Shapes in CrawLogo – an ellipse, a circle and a square.
5.1.4.3 Filters

As all other CrawLogo objects, filters can move around in the workspace and interact with other objects, but they also graphically modify anything that they come across – except other filters – placing one filter on top of another will currently not result in what is under to be e.g. both blurred and embossed. A CrawLogo filter can be thought of a Turtle with video cameras attached to its stomach, and a TV-screen on its back. It records whatever it crawls across, performs some modification to it, and displays the result in the screen on its back.

Currently, a filter Turtle can be told to use the following different filters:

- blur
- emboss
- zoom
- sharpen
- detect edges
- mirror

The type of filter that a filter Turtle uses can be changed during runtime, and thus, a filter Turtle can be programmed to filter some elements differently that other. For instance, it can be instructed to zoom only when it comes across a Web browser and blur everything else.

The degree of for instance “blurring” is set by changing the power of the filter, where a power of 0 means no blurring, and a power of 20 means a very high amount of blurring.
5.2 Putting it All Together: Programming in CrawLogo

This section describes some programming aspects of CrawLogo; how to create and manage procedures and subprocedures, interact with other users and how different CrawLogo Turtles can be created, controlled and used within applications.

5.2.1 Creating and Controlling Objects

Different CrawLogo objects are created by invoking different make-commands. For instance, the following command creates a Web browser with the name myBrowser.

```
makebrowser myBrowser
```

All objects in CrawLogo must have a name which is referred to for controlling the object. In order to make an object move on the screen, we first state which object we are talking to, and then tell it to move forward:

```
talkto myBrowser fds
```

Invoking the TALKTO command on object puts the object in listening mode, so subsequent commands will affect this object until we choose
to address another object. The \texttt{FDS} command (short for \texttt{FORWARDSCREEN}) is used to control Turtle movement on the screen, and is thus different from \texttt{FORWARDWEB} (or \texttt{FDW}) that controls movement on the Web. (The problem of primitive naming for different kinds of semantically similar movement will be discussed in chapter 6.) In contrast to turtle movement in Logo, where the \texttt{FORWARD} command is followed by a number (how far the turtle will travel), turtles in CrawLogo have a speed property, so given that a turtle has a speed of 10, invoking the \texttt{FORWARDSCREEN} command will cause it to move forward 10 steps. Since CrawLogo uses Turtle geometry, the outcome of \texttt{FORWARDSCREEN} will depend on the object’s heading. Given a heading of 90 degrees and a speed of 10, the object will then move 10 steps to the right.

Some objects require specifying additional parameters on creation. When creating a filter for instance, we have to specify what kind of filter we want to create:

\begin{verbatim}
makefilter blur myBlurFilter
\end{verbatim}

\textbf{5.2.2 Creating and Managing Procedures}

This section describes how to create and manage procedures in CrawLogo. As an example, we will create a simple Web crawler that moves between and displays randomly selected Web pages. In order to have a Web-browser Turtle called myBrowser follow a randomly selected hyperlink on a Web page, we can do:

\begin{verbatim}
talkto myBrowser
turnright random 359
fdw 1
\end{verbatim}

Creating procedures in CrawLogo can be seen as adding words to the Turtle’s vocabulary. So, in order to learn a Web-Turtle a new word for following a random hyperlink, we can create the following procedure:

\begin{verbatim}
to go_to_random_page
    turnright random 359
    fdw 1
end
\end{verbatim}

Every time we call \texttt{go_to_random_page}, the Web-Turtle will go to a new Web page and display its contents. Further, \texttt{go_to_random_page} is now a procedure, and can be called by other
procedures. We can now create a superprocedure, \texttt{run\_web\_crawler} that calls \texttt{go\_to\_random\_page}:

\begin{verbatim}
to run_web_crawler 
  go_to_random_page 
end
\end{verbatim}

Now, to complete our simple Web crawler, we just have to add a few lines to tell the Turtle to keep moving, which can be done either \textbf{a)} recursively or \textbf{b)} incrementally. We also add a short pause, to have time to view the page before the crawler continues to a new Web page.

\textbf{a)}

\begin{verbatim}
to run_web_crawler 
  go_to_random_page 
  pause 4000 
  run_web_crawler 
end
\end{verbatim}

\textbf{b)}

\begin{verbatim}
to run_web_crawler 
  make run 1 
  while (:run) [ 
    go_to_random_page 
    pause 4000 
  ] 
end
\end{verbatim}

In \textbf{b)} we also added a variable that tells whether to keep running or not, thus here the crawler can be easily stopped by setting the variable \texttt{run} to 0.

A different way of teaching a Turtle a new behavior is to “show it how it’s done”. By invoking the \texttt{STARTRECORD} command, users can drag the browser around on the screen and record its movement. The Turtle can then repeat this movement. However, no CrawLogo code is generated for this behavior, and thus, the Turtle’s path cannot be edited or changed by the user. Also, currently only movement on the screen is recorded and thus not other actions, such as clicking on hyperlinks.
5.2.3 Interacting With Other Users

The network support in CrawLogo allows for multiple users to interact with each other. Any user can set up a CrawLogo server and let other users connect to it. Once connected, users can:

- share and remotely control CrawLogo objects
- share procedures
- send images
- write messages to each other

Thus, users can take advantage of the network support to collaborate when creating applications, share their objects or entire applications, or just chat.

5.2.3.1 Establishing Connection

In order to establish communication among a number of remotely located users, one user first sets up a server by giving the command:

```
startserver <port number>
```

Thereafter, other users can connect to this server by typing:

```
connect <IP of server> <port number> <nickname>
```

5.2.3.2 Sharing Objects and Communicating

All CrawLogo objects are shareable. This means that objects in the CrawLogo workspace can be sent to any user currently connected to the same CrawLogo server. A user who shares an object still “owns” this object. Receivers can manipulate it, however there are certain restrictions. Shared objects are always connected to the original on the owners desktop; should the owner decide to change any of the object’s property (size, position, URL, etc) using CrawLogo commands, these changes will be propagated to all shared versions of this object and override any changes made by the receiver. The issue of ownership does not apply to shared procedures.

In order to share a Web browser called myBrowser, one would give the command:

```
share myBrowser
```

As indicated above, when the creator of myBrowser changes its content, for instance by clicking on a hyperlink, the new Web page
will be displayed in all shared versions of myBrowser in other user’s CrawLogo environments.

Talking to other users is done by using the TALKTO command, specifying that it is a client being talked to (and not a Turtle) and who is being talked to (client’s nickname):

```
talkto client <nickname> <[message]>
```

Talking to all currently connected users is done by substituting a nickname for “all”. For example:

```
talkto client all [Can anyone help me with my bouncing-behaviour?]
```

### 5.3 Demonstrational Applications

The CrawLogo environment supports the creation of applications that in various ways use elements from the Web or allow users to interact with each other. This section provides a description of four Web-enabled applications that was developed in the CrawLogo environment.

#### 5.3.1 CrawLogo Pong

As described in the survey, the Web provides useful means for creating and playing different types of games. In CrawLogo, users can connect to each other to play and collaboratively create games.

CrawLogo Pong is a Web-enabled version of the classic Atari Pong\(^3\) game with network-support. Currently, it is a two-player-game, but extending it for more players would not be difficult. Each player uses a Web browser as a paddle to hit the ball. By setting the heading of the paddle and invoking the `FORWARDSCREEN` and `BACKWARDSCREEN` (FDS and BKS) commands, users can control the movement of the paddle to protect their goal and hit the ball. The object of the game is the same as in the original Pong: to protect one’s own goal from being hit, and to hit the opponent’s goal.

The speed of the ball is affected by how many images that is shown in a Web browser (or paddle) when the ball bounces against it; more images – higher speed.

\(^3\) [http://www.atari.com](http://www.atari.com)

\(^\text{x}\) See for instance [http://www.pong-story.com](http://www.pong-story.com)
Users can play sound files, either by creating a sound using makesound <soundname> <path>, and then using play <soundname> to play it, or play a local file directly using play <path>. This was used for playing a bouncing sound when the ball hits a wall or a paddle.

Figure 13: CrawLogo Pong game running.

CrawLogo Pong is an example of distributed programming in CrawLogo in the sense that both users are involved in creating the application. Although most of the rules and object behaviors reside on one computer, both users must, for instance, create and share their own paddles. Also, it is an example of a Web-enabled application in that properties of a Web page is used to control aspects of the game (in this case, the speed of the ball).

5.3.2 Collaborative Browsing
The collaborative browsing application is an application in which users can browse the Web together, show each other interesting Web pages and discuss what they see.
Figure 14: Collaborative browsing session with four participating users and two active Web browsers.

All Web browsers that are included in the collaborative browsing session have an owner (the user that created and shared the browser) that controls the “base URL” of that browser. While the degree of freedom in reading shared material is quite high – as well as reading and scrolling through the current document, users can also visit linked pages and browse these decoupled from the browsing session – the owner can synchronize the view for all participants. For an elaboration on issues related to different degrees of freedom in collaborative browsing tools, see [61].

Further, this application can also be used for broadcasting slideshows of Web pages. A Web-page slideshow can be controlled by a CrawLogo Web crawler (see section 5.2.2) or by feeding the stream with a local text-file containing a list of URLs.
5.3.3 Image Slideshow

Say a user wants to view a slideshow of images, using local images would be possible, but perhaps not too exciting, since most of these images have probably been viewed before. One feature of CrawLogo is the possibility to create streams of material from the Web. This was used to create a slideshow that shows images from the Web.

![Image Slideshow](image.png)

*Figure 15: Slideshow running, currently with “cats” as the preferred content.*

While some properties of the Web-movement primitives (e.g. FORWARDWEB, TURNRIGHT) makes them not particularly suitable for actions such as “display all images on this Web site” (for an elaboration on this problem, see section 6.1), the Web browser object could not be used for this task. Therefore, a separate object, an Image Viewer, was created for the purpose of displaying images.

The main feature of the slideshow application is that a user can instruct it to “prefer” certain images (using the `PREFER <STRING>` command). This allows the user to create a slideshow displaying only e.g. dogs. Further the user can change the preferred content at any time, e.g. change the preferred content from “dogs” to “cats”.

The slideshow can function as random input for creativity sessions, or just run until it comes across something that the user likes. Once an interesting or fun image is found, the user can right-click the image to visit the Web page from where the image has been extracted, or send it via the network to another CrawLogo user. Since the Image Viewer
retrieves images by querying a search engine\textsuperscript{32}, although the preferred content has been specified, any pictures available via the Web can occur in the slideshow. This can be regarded as good for unbiased creativity sessions, but this also means that people unwillingly might be exposed to offending images.

5.3.4 Guess Who

Guess Who (alternatively Guess What), is a Web-enabled multiplayer guessing game. The user who is the game manager creates a Web browser and then finds an image of a (preferably) well-known person. A filter, for instance a blur filter, is then created and placed over the image. Both the Web browser and the filter are then shared to all participants.

Starting at a blurring ratio of 20 (what is under the filter will be highly distorted) the user who is the game manager will gradually reduce the amount of blur until someone recognizes who is on the picture.

\textbf{Figure 16:} A game of Guess Who in action. Can you guess who?

\textsuperscript{32} http://www.alltheweb.com
5.4 Unresolved Issues
Described here are some known-of shortcomings in CrawLogo in its current state. Some possible solutions are discussed in section 7.2.

5.4.1 General Purpose Constructs
The single major drawback of the current implementation is probably the lack of support for strings and lists. This prevents the creation of more semantic objects and behaviors. Support for this provided, besides constituting powerful general purpose computational constructs, a Web browser Turtle could for instance decide which Web pages to visit or display based on textual content, and alike, filter Turtles could be instructed to filter Web pages based on their content.

5.4.2 Editor Usability
The CrawLogo editor currently provides little assistance in the process of editing and debugging procedures. Due to the lack of error messages, general printed feedback and parenthesis matching, longer procedures can be hard to debug and overview.
Chapter 6
Analysis: Design Insights

This chapter presents and discusses insights related to utilizing and extending the Logo model and the Turtle metaphor in an end-user tool for Web-enabled applications.

The development of a programming environment that allows for the creation of Web-enabled applications raised several important design issues, such as embedding and extending a Turtle-like control metaphor adapted from Logo in such a tool in general, creating a meaningful representation for Turtle movement on the Web, and leveraging the syntonicity from Logo to the CrawLogo environment. Using a CrawLogo Turtle to travel the Web brought out the importance of a representation for Turtle movement on the Web. The model currently used is useful for automating Turtle movement on the Web, as for instance in a Turtle Web crawler, but lacks a meaningful visualization. There are some important differences between Turtle geometry on a two-dimensional plane and Turtle geometry on the Web, and this poses problems for leveraging the syntonicity of Turtle geometry in Logo to Turtle movement on the Web in CrawLogo. Also, we see that movable and programmable filters have some interesting advantages compared to browser built-in means of Web-element manipulation.

6.1 Important Functionality
During the development of CrawLogo, some functionality was identified as useful, not to say mandatory, for the creation of Web-enabled applications. This includes providing users with means to:
• find, retrieve, display and manipulate pieces of information from the Web

• interact with other users in order to communicate, share material or create applications together

Supporting these tasks might be regarded as obvious in a tool for creating Web-enabled applications. More intriguing is issues of how support for some of these tasks were realized, something that we will look into in the following sections.

6.2 Turtle Goes Crawling

Granted, in CrawLogo the Web browser has a significant role and is an important object for interaction with the Web. Section 5.2.2 described a Turtle employed as a crawler that moves between Web pages and displays them in a Web browser. Although selecting which hyperlink to follow is done on a random basis, this application shows that aspects of Turtle geometry can be used for navigation and movement on the Web in order to create something often quite complex as a Web crawler. To be more useful however, this Web Crawler would have to be extended with some way of acquiring knowledge of what information the user is interested in, e.g., a way to instruct the Turtle to go to Web pages with specific contents such as images, keywords or Web page meta-data. Currently, the lack of a string data type in CrawLogo prohibits this.

The Slideshow application (described in section 5.3.3) has some semantic properties, allowing the user to specify what images that should be displayed by using the PERFER command. However, this only allow for slideshow manipulation on a very high level, and users cannot tell the slideshow to display images based on any properties other than filename, such as size, colors or URL.

Regarding movement and navigation, the Web is a highly dynamic space. For instance, any node (Web page) or path between nodes can change at any given time. Thus, there is no guarantee that executing the same series of CrawLogo movement commands from the same starting point at different times will produce the same path or have the Turtle end up on the same Web page.

Having a Turtle moving around the Web between and within Web pages, along complex and unpredictable paths, calls for a meaningful and intuitive representation of such movement. Currently, the Web
geometry used in the CrawLogo environment is a 3D model, where the Turtle geometry of Logo has been extended with vertical movement. As previously described, each domain can be conceptualized as a *house* with a number of *rooms* (Web pages within that domain) on different *floors* (directories and subdirectories), where each room can have *exits* (hyperlinks) to other Web pages within or outside the domain.

The creation of a meaningful representation for navigation and orientation on the Web is a complicated task. Web services such as the Internet Archive\(^{33}\) and possibility to cache Web pages might for instance add a dimension of time, and thus it is unclear whether a two- or three-dimensional representation would be sufficient. Some problems regarding syntonicity in the currently used representation are discussed in the following section.

### 6.3 Syntonicity and Design of Primitives

Much of the learnability of Logo’s Turtle geometry lies in its syntonicity – the possibility for a user to identify with the Turtle and mentally (or physically) “act out” a procedure as if being the Turtle. One problem dealt with leveraging the syntonicity in Logo to the CrawLogo environment and how the language should be designed to maintain the Turtle metaphor. Currently, the CrawLogo primitives can be seen as related to the Turtle metaphor on three different levels, dependent on to what extent they are “in the metaphor”.

**Level 1**: Primitives that can be acted out “playing Turtle”. These include for instance movement commands such as `FORWARD WEB`, `UP`, `TURN LEFT`, `TURN RIGHT` and are clearly in the metaphor.

**Level 2**: Primitives that are in the metaphor to the extent that users can imagine that they are talking to a Turtle. These include for instance operators such as `SET COLOR`, `SET POWER` and `SET URL`.

**Level 3**: Primitives that are outside the metaphor in that they do not relate directly to the Turtle, as for instance `START SERVER` and `CONNECT`.

Primitives outside the metaphor raise the issue of whether these should be redesigned to something more Turtle-like. Starting a server

\(^{33}\) [http://www.archive.org](http://www.archive.org)
for instance, could be seen as a creating public Turtle-hive, a place where to send all Turtles that should be accessible by other users.

In Logo, users can take advantage of their knowledge of how they move their bodies in the real world when controlling a Turtle on the screen. In a 2D-plane, this works very well. For a Turtle moving between different Web pages, this works somewhat different. While there exists tools that allow for moving up\(^{34}\) in a Web domain directory hierarchy, moving down poses bigger problems. While moving up is unambiguous, e.g. moving up from the URL http://www.liu.se/ffk/ will always lead to http://www.liu.se/, moving down from the same URL is highly ambiguous, since there a directory often have several subdirectories. This suggests adding another dimension to the Turtle’s heading. Further, in Logo the Turtle responds to movement commands such as FORWARD 100 with immediate visual feedback, i.e. the Turtle moves on the screen. This works the same way in CrawLogo as far as on-screen movement is concerned, but movement on the Web lacks such immediate feedback. Since this probably makes it hard to conceptualize what a Turtle moving on the Web is actually up to, it probably also makes it difficult to “put oneself in the Turtle’s shoes” and understand movement between different Web pages as something else than just random movement.

A language design related problem raised by introducing Turtle-movement on the Web were giving primitives for movement meaningful names. While Web browser-Turtles can move forward and backward both on the screen in the CrawLogo workspace and between sites on the Web, it was difficult to decide what e.g. telling a Web browser-Turtle to do “forward 1” should mean. To complicate this further, “back” and “forward” in an ordinary Web browser means navigating between previously visited Web pages. In the current implementation, this problem was solved by naming primitives for these different types of movement differently, but an alternative solution could include having several Turtles associated with a single Web browser object, each responsible for different types of movement.

---

\(^{34}\) The Google Bar: http://toolbar.google.com
6.4 Programmable Filters

Besides creating and customizing the existing Logo language, extending Logo for use in a Web-context also involved the creation of new objects, or new Turtles. The movable filters were developed with the purpose to provide useful means of manipulating various kinds of elements from the Web and elements in the CrawLogo workspace, and being controlled as Turtles, doing so within the CrawLogo metaphor. The benefits of using moveable filters as a user interface tool have been previously emphasized [56], however not as end-user programmable tokens in a Web-enabled context.

Information from the Web is often retrieved via a Web browser “as is”, that is, without possibility to perform any major alterations to its form or content. Most Web browsers let users change the font size on a Web page, but this functionality is highly dependent on the nature of the Web page and does not work very well with, for instance, images, video, Flash or Shockwave material. One advantage of a movable filter is that it can operate on any material on a Web page and regardless of how the Web page is implemented.

Having programmable filters means that there is the possibility to customize and add sophistication to their behavior. In CrawLogo, users can equip filter Turtles with sensors, allowing them to discriminate between things that they come across in the CrawLogo workspace. This makes it possible to tell a filter to magnify or enhance the contrast of all Web browsers, but blur everything else.

Some steps have been taken lately to promote creation of customizable Web pages [22], primarily to allow people with special needs to make Web pages more readable (viewable) or use alternative interaction techniques. However, these efforts focus on teaching designers and programmers of Web pages how to create customizable Web pages. Needless to say, there will always be poorly designed Web pages not following guidelines of accessibility. Here, moveable filters can be useful – as shown by the implemented filters – for magnifying text or enhancing contrasts, but in the future possibly also for translating text, reading text aloud or, based on image recognition technology, describe images.
Chapter 7
Conclusions

The CrawLogo environment was developed in order to explore consequences and design issues of applying aspects of the Turtle metaphor from Logo in a programmable environment for Web-enabled applications. This approach of combining a Turtle-like control metaphor with Web usage and end-user programming of Web-enabled applications do allow for the creation of some new types of applications and some new ways of interacting with the Web. Further, as shown by the demonstrational applications, the CrawLogo environment allows for the performance of tasks and creation of applications and object behaviors that – using a more traditional programming language – would be quite complex and require some deep technical programming skills.

As discussed in the previous chapter, in order to design within the CrawLogo metaphor and provide users with the possibility to identify with Turtle objects and procedures, a meaningful representation is needed. With respect to the currently used model, it is doubtful whether the syntonicity of Logo with the benefits that follows can be leveraged to its full extent to Turtle movement on the Web without some kind of visual feedback of what the Turtle is up to.

What follows is a) a summary of the major contributions of this thesis and b) some suggestions for and thoughts about future research.
7.1 Summary and Contributions

To summarize, the work of this master thesis has provided three main contributions:

- a programmable environment, CrawLogo, that is designed to allow end-users to create their own Web-enabled applications
- an analysis of design issues regarding language, environment and representation, related to utilizing the Logo model and a Turtle-like control metaphor in such a programmable environment
- a number of demonstrational applications, created in CrawLogo

While there are some Logo versions that provide different types of network functionality, such as communicating with other users or sharing objects and Logo-simulations (e.g. Imagine Logo and NetLogo), “Web-enablement” in many existing Logo tools refers to the possibility to publish Logo applications or simulations online, or the fact that the tool runs within a Web browser. Currently none has focused on including elements from the Web within applications and taking the Turtle-geometry metaphor online.

In the design and implementation of CrawLogo, particular attention was paid to utilizing and extending the Turtle metaphor from Logo. As shown by the demonstrational applications, the approach of utilizing an extended Logo model for Web-enabled programming allows for the creation of several different applications. These included both more traditional applications such as a collaborative browsing environment and new types of games such as CrawLogo Pong. Further, the CrawLogo environment do allow for the performance of tasks and creation of applications and object behaviors that – using a more traditional programming language – would be quite complex and require some deep technical programming skills. The work with the CrawLogo environment also provided insights on important functionality that a tool for creating Web-enabled software should support.

We also saw that it seems doubtful whether the syntonicity of Logo can be leveraged to its full extent to Turtle movement on the Web

---

35 http://www.logo.com/imagine/
36 http://ccl.northwestern.edu/netlogo/
without a meaningful representation of and feedback on what the Turtle does (i.e. how it moves on the Web). Although such a representation might not have to be visual, it is likely that this would be helpful.

Teaching Turtles new behaviors using macro recording techniques was briefly explored. No strong conclusions can be drawn regarding this other than that combining direct manipulation and macros with a Turtle-like control metaphor seems interesting.

One major challenge to the end-user programming community lies in creating programmable environments that are easy to use and have a high level of expressiveness, with meaningful interaction and interface metaphors. CrawLogo makes it possible to perform various programming tasks in few steps, compared to traditional programming languages, using a high-level Logo-like syntax. Though, since no formal user testing was made, no statements can be made regarding how CrawLogo would make things easier or better for users, or how this model would affect how users think about the Web or navigation and orientation on the Web. This remains, in addition to several other topics provided by the work on CrawLogo, pointers for possible future research and projects, which we will look at in the following section.

7.2 Directions for Further Research

This master thesis work has in its exploratory approach come across several interesting side paths. Naturally, and somewhat unfortunately, there was not time to travel down all these paths, so provided here are some thoughts on further work with the CrawLogo environment and some unexplored realms of Logo influenced Web usage and end-user programming.

7.2.1 Web-Enabled End-User Programming

With the Web and various content on the Web continuously expanding in amount as well as form and function, the possibility to retrieve and operate on different elements from the Web is very interesting. The work of this thesis has explored design issues related to providing end-user programmers with the means to create their own Web-enabled applications. Interesting and useful further steps would include (user) evaluation and further development of CrawLogo primitives, syntax and semantics and extending the
support for finding, retrieving, managing and manipulating different elements from the Web for use in users’ own applications. In particular, when it comes to finding information, the CrawLogo environment needs strings and lists to allow for the creation of more “content aware” Web crawlers and information gathering applications.

As previously described, what end-user programming really sets out to do is making the activity of programming easier for non-expert programmers, something that includes adding usability to both language and environment. For CrawLogo, there is need for some particular usability enhancements to the procedure editor, such as more informative feedback and error messages, and user interface components such as matching parenthesis or color-coded primitives and variables.

Also, it would be interesting to explore how other programming paradigms, such as agent-based programming or programming by example, or other interface and interaction metaphors could be utilized in a Web-programming context. As a lot of material on the Web would be suitable or already comes in streams (e.g. streaming audio and video), stream-based programming seems to be a particularly interesting approach in a Web-programming context.

7.2.2 Towards a Web Geometry

How do people handle the Web? What does their representation of the Web look like, and how do different pieces of information on the Web relate to each other?

The tools we use to interact with and manipulate elements in our environment affect the way we think about these elements (e.g. [14]). Since the primary tool for retrieving and displaying material from the Web is a Web browser, this can be seen as the main artifact mediating information and knowledge from the Web to the user, thus likely affecting the way we think about and use the Web. It would be interesting to see how new means and metaphors for navigating on and retrieving information from the Web would affect how people think about the Web.

The need for a meaningful representation of how Turtles move and act on the Web raises the issue of creating a Web geometry based on Turtle geometry. As we have seen from the analysis of the model for
movement on the Web currently employed in CrawLogo, using a 2D representation for Turtle heading posed some problems. A 3D model would be better, but such a representation might also face some difficulties.

As there have been a lot of research and development of tools for visualizing the Web and large data collections (survey available in [23]) in 3D, 4D and even 8D, a first interesting step could be to explore how these efforts could be combined with the concepts of Turtle geometry in order to construct a new tool for navigation, orientation and information retrieval on the Web.

7.2.3 Web-Driven Emergent Software
As pointed out, an interesting and important issue when talking about Web-enabled applications is the constant connection to an incredibly vast space of dynamic, constantly changing and, at least often, unpredictable data. As we have seen, this can be used as input to applications or for creating unpredictable streams or slideshows of images or Web pages. It is easy to imagine other types of applications where these properties could be used to create Web-enabled applications with emergent content. One specific example could be online multi-player games, where the form and content of various Web-pages could generate room-descriptions, monsters, treasures, missions, etc. Such a game would truly be different every time.

7.3 Final words
The aim of this work has been to explore consequences of applying the Turtle metaphor from the Logo programming language in an end-user programming tool for Web-applications, and to provide design insights for the construction of such programmable tools. While the CrawLogo environment and the demonstrational applications showed some advantages and problems with this approach, user studies are needed to further explore these issues. Also, further work with CrawLogo is needed, especially concerning creating a meaningful representation of a Turtle geometry-based Web geometry.

In conclusion, the CrawLogo environment and the insights provided by this thesis can be regarded as steps towards a “Web Logo” – a tool for non-programmers and children, in which they will be able to design and create their own useful and fun Web-enabled software using an enhanced Logo language.
Chapter 8
References


References


References


[38] Lieberman, H. What is Programming by Example? URL: http://web.media.mit.edu/~lieber/PBE/what-is-PBE.html


References


**Title**  
CrawLogo: An Experiment in End-User Programming for Web-Enabled Applications

**Abstract**  
With the rise of the Web, there is more interest among end-users to create different kinds of software that use elements from the Web or allow for networked interaction between users. Currently however, many available tools for this purpose are hard to use or lack a sufficient level of expressiveness. To provide new insights on the construction of tools that allow end-users to create their own Web-enabled software, this thesis explores design issues and consequences of applying the Turtle metaphor from the Logo-programming language to an end-user programming environment for Web-enabled applications.

In order to explore this, CrawLogo was created – a programmable end-user tool that supports the creation of Web-enabled applications using a Turtle-like control metaphor and language adapted from Logo. As a proof-of-concept, several Web-enabled applications were created using this new tool, including CrawLogo Pong, a somewhat alternative version of Atari’s classic Pong game, and a collaborative browsing environment, in which users can browse the Web together. The resulting CrawLogo environment allows for creating Web-enabled applications that – using more traditional programming languages – would be quite complex and require deep technical programming skills. Further, while utilizing a Turtle-like control metaphor in CrawLogo allows for the creation of some new types of applications and some new ways of interacting with the Web, it also raises new problems such as how to successfully design within the CrawLogo metaphor and how to create a meaningful representation of Turtle-geometry-based navigation on the Web.

**Keywords**  
CrawLogo, end-user programming, Web-enabled applications, Logo, Turtle geometry, Turtle metaphor