Supporting Program Development Comprehension by Visualising Iterative Design

Charles Boisvert
City College Norwich
{cboisver@ccn.ac.uk}

Abstract

eL-CID (e-Learning by Communicating Iterative Design) demonstrates computer programs' iterative design using computer animation. It translates descriptions of iterative editing into an animated demonstration. An analysis of the work of expert programming trainers shows that successive versions of a program are shown statically. eL-CID attempts to visualise the changes dynamically as if code was being edited in front of the user.

Several example demonstrations have been developed. To the author's knowledge, this is the first system designed to visualise the iterative process of program development.

Keywords --- Software Visualisation, Iterative Development, Human-Computer Interface, e-Learning.

1 Introduction

This paper describes the design and implementation of eL-CID, a system to support learning by visualising iterative design. eL-CID (e-Learning by Communicating Iterative Design) takes examples of computer programs' iterative design and uses computer animation to show them to students.

A review of expert programming teachers' iterative design techniques was first carried out to develop a language to describe the steps of program code editing. eL-CID translates it into successive frames in an animated demonstration of the code's iterative development. The system thus helps expert programmers communicate their software development skills by showing a process that is difficult to present statically.

This paper first reviews the background of software tools for visualisation, development, and instruction in software development. It then analyses practices in teaching iterative development and how a time-dependent activity such as iterative design, has been mapped to various visualisations. Based on this analysis, it proposes an iterative development description language and explains how the design descriptions are translated into animations. Finally, it reports on design examples that have been developed thus far and concludes with a discussion of empirical evaluation and some possible extensions of the work.

2 Background: software visualisation, development, and instruction

Software tools are frequently used to support and help teach iterative program development. Two fields are especially worth reviewing for their relevance and for their use in tutoring: software visualisation systems and software development or engineering environments.

2.1 Software development and visualisation tools

2.1.1 Software visualisation

Software visualisation (SV) focuses on the development of techniques to present data and algorithm execution. Many teaching, research and industrial systems use data visualisation, as the survey of object-oriented programming environments by Romero et al. [20] shows. Algorithm visualisation systems are used in teaching as they can show traces of complex execution [4, 11, 12, 21].

SV systems have brought advances in the representation of complex information. Particularly relevant to this project are the techniques used to visualise algorithm execution. The steps of execution are usually shown in an animation to allow a viewer to construct a mental model of how the process leads to the required result.

Although the transformation of a program in the iterative development process seems not to have attracted the interest of the SV community, the tools and technique used in algorithm visualisation could be adapted to visualising program development.

2.1.2 Software development environments

Integrated development environments (IDEs) and software engineering environments (SEEs) support the development of complex systems. IDEs exist to support program coding in practically any language. SEEs extend their support to other aspects of the development process, such as object modelling in UML (at which Rational Rose excels), process management with process-centered SEEs [9], or co-operative development [6]. Many of these systems were developed for the industry but they are also used in teaching.

Two aspects of IDEs/SEEs are particularly relevant to eL-CID. First is the use of data management to record the state of software projects and successive versions of source code. Often used to support teamwork, backup and versioning [2], the principle could be exploited to demonstrate iterative development by replaying improving versions of code. Second is the development of intelligent editors supporting refactoring. Refactoring actions, such as renaming or creating variables or moving methods in object-oriented development, originally identified to facilitate iterative development and improve editors [10], could be used to exploit the semantics of code changes.

In this area as in SV, the use of successive versions of code to support program development comprehension has not been considered. It appears that as yet, there is no system for making and showing examples of iterative development. However, this direction could yield great benefits.

2.2 The Cognitive Science of program development

Research in Human-Computer Interaction [15, 16] Computer-Assisted Instruction [17, 23] and Software visualisation [4, 12] all concords to show that *communicable systems*, systems that are built on cognitive models, are the key to successful knowledge communication.

Starting with Papert's development of the Logo language [17], many tutoring systems have been developed and used to facilitate constructivist learning of program development. Parallel to this development, the close relation between development tools and cognitive models remains a major preoccupation of the research community [19, 3].

Studies of the psychology of programming inform researchers with cognitive models of program comprehension [8]. However, they show more interest in *program* comprehension than in program *development* comprehension.

A system to mimic the software development process could help investigate the cognition of software development. A simple objective could be to investigate the hypothesis that iterative development is a better cognitive model of software development than the waterfall alternative.

Iterative programming is a software development methodology that relies on 'lightweight' processes and a model of development based on many successive iterations [7]. Its popularity has grown with a movement for *agile development* [1] and it is gaining importance and a lot of interest in the computing industry.

A look at this development model reveals ample anecdotal evidence that it is understood naturally: the enthusiasm that this model generates throughout the profession; the relative unpopularity of other software engineering processes; and the large body of online and printed teaching materials that attempt to mimic in print the piecemeal development of a computer program, all tend to how that this is the case.

eL-CID is intended to help teachers provide the dynamic visualisations required to show students examples of iterative development. It can be used as a tool by teachers to develop examples to demonstrate; by students to support and structure their understanding of software development; and finally it could also provide evidence to study this development methodology and its comprehension.

3 Visualising program development

3.1 Iterative development in the classroom

Teaching programming usually introduces students to a form of "waterfall" software development model – a linear, unidirectional view of development from specification to design, implementation then testing. Iterative development is not so easy to communicate, even though it is a popular and essential skill that programming students at all levels need to acquire.

Lecturers find that agile development has to be learnt by experience rather than by 'being told'. There are several reasons for this. Formal teaching defers to the historical importance of the waterfall model; it also uses textbooks and written material, which are by nature static and do not lend themselves well to describing iterative development. Iterative development, therefore, is difficult to impart in the lecture theatre. It is also awkward to split into small, simple pieces for practice through bite-size exercises.

Instead teachers use large exercises and formative assessment to let students build an understanding of the need to work iteratively towards an application.

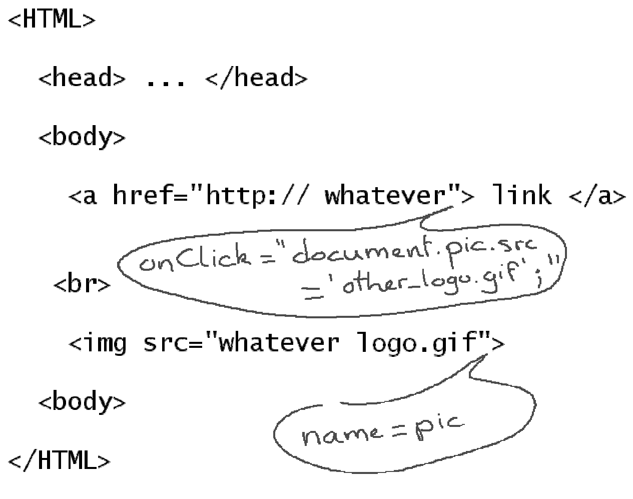
In other words, as lecturers have little appropriate means of demonstrating iterative software development, they teach it using an almost exclusively constructivist approach. While constructivism is an excellent way to allow the students to take ownership of their new knowledge, it is both time and resource consuming. Also with the growing numbers in higher education the pressure is strong to abandon it as an inefficient approach. The risk therefore is to fail to impart an essential skill in order to continue responding to the educational needs of a growing student population.

3.2 Static visualisations of dynamic development

The difficulties of showing students agile development stems primarily from attempting to show statically --- in diagrams or in print --- a technique which is dynamic. However, it is possible to map dynamic program changes to static representations. Two common techniques are to mark changes using annotations and to show selected successive versions of a program.

3.2.1 Using annotations

Iterative programming can be demonstrated by annotating programs to show how they are developed over multiple iterations. *Figure 1* (below) shows a basic example of such a demonstration by showing code being inserted in an HTML page.



```
<HTML>
  <head> ... </head>
  <body>
    <a href="http:// whatever"> link </a>
   <br>
    <img src="whatever logo.gif">
  <body>
</HTML>
```

Figure 1: annotating code to show its iterative improvement

Such annotations do demonstrate on a static diagram the dynamic changes to a program. The technique, however, has two immediately visible drawbacks.

- First, annotations only remain clear for a limited number of program changes --- too many program revisions turn an initial code listing into a morass of insertions, striked out code lines and block moves.
- Second, the quality of such diagrams in making explicit the dynamic aspects of the development process depends highly on the individual abilities of the lecturer.

An alternative to annotations is to show a program's transformation by providing multiple successive versions of it. This will show the dynamic aspects of program development in print. A review of printed program development descriptions in programming textbooks reveals that it is a frequent occurrence.

3.2.2 Multiple program versions.

To identify how expert programmers and trainers explain development, we checked numerous programming textbooks. Series of textbooks explaining how to program in many languages offer excellent examples of teaching techniques. The succession of programming examples in a text follows one of three techniques.

The cookbook: a cookbook offers a list of program examples that are mainly unrelated to each other. Written for proficient programmers, they are used a reference to apply a known technique in a given language. Good examples of cookbooks are the O'Reilly reference texts [5, 14].

The evolving program: particularly useful for novices, these texts rely on an initial program which is often quite simple, and show successive improvements until the systems explained use a wide range of techniques and satisfy demanding requirements. A good example is [18], although more texts follow a hybrid approach.

The hybrid approach: this is the presentation followed by a vast majority of texts. The authors choose several programs to illustrate different techniques. Each program is then showed in successive, increasingly elaborate versions. That technique is for instance used by [13, 22].

Overall, printing multiple versions is clearer than annotations where program changes are extensive. However, only a limited number of versions can be reasonably shown in a textbook chapter, so that care has to be taken to select the significant steps of a given iterative design. The clarity of the work is also strongly dependent on an accompanying explanation, so that as for annotations, the value of a collection of program versions in print primarily depends on the qualities of the author as a communicator.

4 Research and Implementation

To overcome the difficulties of showing dynamic changes in a static presentation, a straightforward solution is to use computer animation techniques to show the changes dynamically, ensuring a more natural mapping between the effective process of iterative design and the view that is proposed to the students. That is what eL-CID proposes to do in this research.

To model the iterative development process, a descriptive language and XML application are proposed that describe the basic steps of code editing. This application is then used to model the iterative development of some well-known programming examples. A computer system was built that uses those descriptions to show iterative software development.

4.1 A simple language for iterative editing

Program editing can straightforwardly be reduced to syntactic changes to program code. For eL-CID, six elementary changes are proposed:

- *Cursor move:* Move the cursor to a given position;
- *Insert:* Insert given lines and characters at the current position;
- *Select:* Select the given number of characters and lines;
- *Delete:* Delete the current selection;
- *Copy:* Copy the current selection to a clipboard;
- *Paste:* Insert the clipboard contents at the current cursor position.

```
<elcid>

<source>
&lt;html&gt;
&lt;body&gt;

&lt;a href="http://images.google.com/"&gt;Over here&lt;/a&gt;
&lt;p&gt;
&lt;img src="images/charles.jpg"&gt;

&lt;/body&gt;
&lt;/html&gt;
</source>

<iteration>

   <move>
      <linenumber>6</linenumber>
      <colnumber>29</colnumber>
   </move>

   <insert>
      <chars> name="photo"</chars>
   </insert>

   <move>
      <linenumber>4</linenumber>
      <colnumber>35</colnumber>
   </move>
   <insert>
      <chars> onMouseover = "photo.src='images/bull.gif';"</chars>
   </insert>

</iteration>

</elcid>
```

Figure 2: an example of iterative development with eL-CID

These six operations suffice to simulate any changes to a text file. Stored alongside the source code in an XML file, they describe program modifications. *Figure 2* (opposite column) shows the syntax on an example which starts inserts JavaScript code into an HTML source.

These six editing operations are not the only possible choices. The shortest list excludes the *select*, *copy* and *paste* operations, but these are now so common that they have been included to model the editing process.

Other operations could be added that reflect key presses, such as back and forward delete and up, down left and right cursor moves. The possible benefits of these operations did not justify the integration of many more text editing operations in an initial version of eL-CID.

Some common editing operations are obtained by compounding the basic ones:

- Placed insertion = cursor move + insert
- Cut = copy + delete
- Move text = move + select + cut + move + paste

It may prove useful to eventually include shorthand notations for such common operations; an analysis of which basic combinations recur frequently could also prove fruitful.

Each editing operation requires basic parameters, such as the row and column to move the cursor to or the number of characters to select. It can also accept an optional note to display explanations about program changes as they are shown to the learner.

4.2 eL-CID implementation

When eL-CID opens an XML data file, it parses the data to prepare the demonstration. It then displays the initial listing, and shows the step by step changes as if the program was being edited in front of the user. At each step the user can stop to observe the changes and run the work in progress.

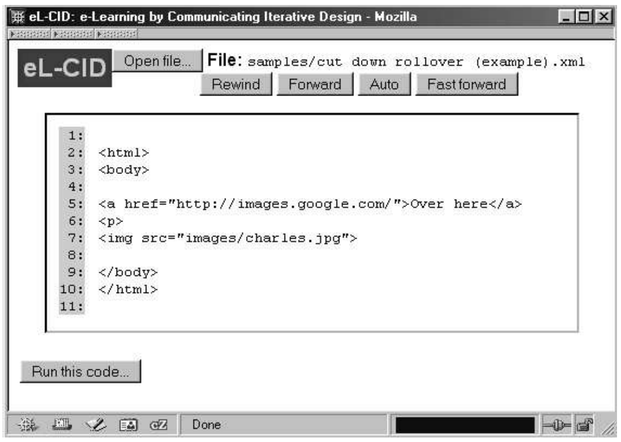
eL-CID is written in JavaScript. Deployment can be either on a web site, or on a client system with the application showing in a web browser.

An expert programmer writes his iterative design in an XML file. eL-CID uses the World-Wide Web Consortium's XML DOM parser to parse the file to find the source code and its incremental changes. The changes are stored in an array of steps.

The system then displays the program source and plays the code changes at the student's request. It shows different options available to the user: to move forward one step at a time, play the changes as a timed animation, or instantly show the initial or the final program version. Any version of the code can also be executed separately.

The changes appear as if the expert was editing the code in front of the student. Each step is played forward according to its characteristics and the eventual corresponding annotation is displayed in a separate text box (a 'post-it') that can be moved away or turned off.

The screenshot (*Figure 3*, below) shows the interface of eL-CID.



File: samples/cut down rollover (example).xml

```

<html>
<body>

<a href="http://images.google.com/">Over here</a>
<p>
<img src="images/charles.jpg">

</body>
</html>

```

Figure 3: eL-CID interface

The system is currently running from a web site at the address http://www.boisvert.uklinux.net/

The complete system source code and the examples developed so far are also available from the site.

4.3 Design examples

eL-CID can demonstrate examples of iterative program development in any programming language. To test and demonstrate its capabilities, two examples have been developed for JavaScript:

- `sort.xml` uses eL-CID to visualise the development of a classic algorithm. The initial version displays an array in a web page. It then turns into a program to find the largest item in the array. It is the starting point for developing the well-known *selection sort* algorithm.
- `rollover.xml` shows how to improve a web page with *image rollover*s --- images that change over when the mouse is moved on them. A common web design technique, it is often used to enhance user interaction. The example used throughout this paper is a cut-down version of rollover.

Both examples were created with specific students in mind. They are common examples of program development and show well how more could be created to teach program development.

5 Conclusions

The project started with the hope of developing a tool to demonstrate iterative program development and examples for classroom use.

The eL-CID software could be used by many academic centres or adapted to the teaching needs of a wide range of different computing courses. The program design examples developed at this point are limited. However, programming examples and exercises are often very course-specific; the possibility to develop new ones is more valuable.

The project could also bring valuable teaching and research results in the medium and long term.

5.1 Licensing and commercial application

The software and examples are available from a website,

http://www.boisvert.uklinux.net/

It is distributed under the GNU/GPL license.

The licensing terms of eL-CID were chosen to facilitate its development.

The data for the iterative development examples is excluded from the General Public License. The exclusion means that examples could be developed and exploited commercially – sold, licensed, or kept to protect the exclusivity of training facilities. The examples developed thus far are in the public domain and available on the Website.

5.2 Application

The review of programming textbooks and the present examples show that eL-CID is a promising technique. To pursue its development, a pilot evaluation would be needed.

5.2.1 Empirical evaluation

An evaluation would help to determine, on the one hand, how useful eL-CID is in teaching and whether and how it needs to be improved, and on the other, help show how iterative development is understood among the students.

As the system is intended for teaching, evaluation should take place in real classroom conditions. It is envisaged to use a participant action researcher approach. Students will be taught using the new software in a module where the delivery will be adapted to the new technology. Concurrent research will study and evaluate student reactions.

An existing system for collecting course feedback will be adapted to provide better feedback opportunities and facilitate analysis of the results.

5.2.2 Further development and perspectives

This work has many wider implications. Several natural extensions would develop the software usefully while shedding light on the cognitive science of program development.

- Implementing an editor for eL-CID, so that trainers could easily build examples of iterative design to show to their students;
- Using such an editor to help users communicate iterative programming examples and ideas;
- Analyse the patterns of code editing recorded in eL-CID examples to study the development techniques of novice and expert programmers;
- Identifying and exploiting regular patterns in code editing from the same eL-CID examples to recognise semantic editing units, study editing techniques, and improve the visualisation of iterative development;
- Extending the eL-CID model to include the use of code from multiple files in program development and modeling the lattice of relationships between programs in a body of design examples.

The development and use of eL-CID for teaching and research could be a useful tool to support teaching iterative development and a rich source of information to understand the cognitive processes involved in software creation.

6 Acknowledgements

This work has been made possible thanks to support by both APU and City College. I would like to thank all those who have made this possible: Rob Fiddy and David Lovell-Badge at Norwich City College, and Chris Greene at Anglia Polytechnic University.

Throughout the research, the encouragement of my colleagues has been a great source of enthusiasm. My thoughts go to all the staff who have borne my repeated demonstrations and made countless cups of tea.

7 References

[1] Agile Manifesto (2001): *Manifesto for Agile Software Development*. Online at http://www.agilemanifesto.org/

[2] Barghouti, Naser, Wolfgang Emmerich, Wilhem Schäfer and Andrea Skarra (1995): *Information Management in Process-Centered Engineering Environments*. In Process-Centered Environments, John Wiley and sons.

[3] Ben-Ari, Mordechai (1999): *Bricolage Forever!* 11th annual workshop of the special interest group on the psychology of programming, University of Leeds.

[4] Boisvert, Charles (1995): *A Learning environment for Natural Language Processing*. In Proceedings of the CS-NLP ' 95 conference on the Cognitive Science of Natural Language Processing, Dublin, 1995.

[5] J. Bradenbaugh, Javascript Application Cookbook, O'Reilly 1999;

[6] Fougères, Alain-Jérôme, P. Canalda, and P. Chatonnay (2002): Pédagogie de Projets tutorés Basée sur la Synchronisation de fragments de Procédés Coopératifs: Motivation, Modélisation et Expérimentation.

[7] Fowler, Martin (2003): *The new methodology*. Online paper maintained at http://www.martinfowler.com/articles/newMethodology.html

[8] Green, T.R. (2000): *Instructions and descriptions: some cognitive aspects of programming and similar activities.* Invited paper, in Di Gesu, V., Levialdi, S. and Tarantino, L. (Eds.) *Proceedings of Working Conference on Advanced Visual Interfaces (AVI 2000).* New York: ACM Press, pp.21-28. Also available in pdf form via www.ndirect.co.uk/~thomas.green/workStuff/papers/

[9] Gruhn (2002): *Process-centered software engineering environments – a brief history and future challenges*. In proceedings of the annals of Software Engineering, 14, pp. 363-382, 2002, Kluwer.

[10] IntelliJ (2002): *IntellijIDEA overview*. Jetbrains ed. Published online at http://www.intellij.com/

[11] Mullholland, Paul (1997): *Teaching programming at a distance: the internet software visualization laboratory*. In journal of interactive media in education, KMI 1997. Published online: http://www-jime.open.ac.uk/

[12] Mullholland (1998): *Sharing Programming Knowledge over the Web: the Internet Software Visualization Laboratory*. In Marc Eisenstadt and Tom Vincent (eds.), the Knowledge Web: Learning and Collaborating on the Net. Kogan Page 1998.

[13] Negrino, T., Smith, D.: *Javascript for the World-Wide Web: visual quickstart guide*, Peachpit, 1999;

[14] J. Niederst, Web Design in a Nutshell, O'Reilly 2001

[15] Norman, Donald Arthur (1986). *Cognitive engineering*. In D.A. Norman and S.W. Draper (eds.), User Centred System Design. Hillsdale, New Jersey: Lawrence Erlbaum.

[16] Norman, Donald (1988): *The design of everyday things*. ed. MIT Press 1988.

[17] Papert (1980) Mindstorms: Children, Computers, and Powerful Ideas. Basic Books, New York.

[18] Pereira, F. C. N., Shieber, S. M.: *Prolog and Natural-language Analysis*. CSLI, 1987

[19] Retowsky, Fabrice (1998) *Software reuse from an external memory: the cognitive issues of support tools.* 10th annual workshop of the special interest group on the psychology of programming, Knowledge Media Institute, Open University.

[20] Romero, Pablo, Richard Cox, Benedict du Boulay and Rudi Lutz (2002): *A survey of external representations employed in Object-Oriented programming environments*. In Journal of Visual Languages and Computing (Special Issue on Program Visualization).

[21] di Scala, Robert-Michel (2000): *Un package interactif d'assistance par ordinateur appliqué à un enseignement d'initiation à l'informatique-programmation.* Presentation a l' Atelier conception de contenus pédagogiques du colloque international TICE 2000, 18-20 octobre 2000.

[22] Walther, S., Banick, S., Levine, J.: *Teach yourself ASP and e-commerce in 21 days*, SAMS, 2000

[23] Wenger (1986): *Artificial Intelligence and Tutoring Systems*. Morgan Kaufmann Publishers.