Envision: A Fast and Flexible Visual Code Editor with Fluid Interactions (Overview)

Dimitar Asenov
Department of Computer Science
ETH Zurich
dimitar.asenov@inf.ethz.ch

Peter Müller
Department of Computer Science
ETH Zurich
peter.mueller@inf.ethz.ch

***Abstract*—While visual programming has had success in some areas such as introductory or domain specific programming, professional developers typically still use a text editor. Designing a visual tool for professionals poses a number of challenges: visualizations must be flexible to support a variety of different tasks, interactions must be fluid to retain productivity, and the visual editing must scale to large software projects. In this paper we introduce Envision, a visual structured code editor that addresses these challenges using an architecture that supports flexible, customizable visualizations, keyboard-centric controls for fluid interaction, and optimizations to ensure good performance for large projects. Experiments with CogTool indicate that Envision's code manipulation techniques are as efficient as those of Eclipse, thus overcoming a major usability barrier for visual programming for professional developers.**

Keywords*—*programming environments, structured editors, visual programming, human-computer interaction

I. Introduction

Visual programming (VP) tools are very successful in specific domains (e.g., LabView[1]), in end-user programming (e.g., spreadsheets), and in teaching (e.g., Alice [1], Scratch [2]). However, the great majority of professional programmers are stuck in a textual world. A look at two popular web-sites[2,3] keeping track of the popularity of programming languages easily reveals that all mainstream languages today are textual. Most developers program in them using IDEs such as Eclipse, or just with a text editor. Compared to tools from other domains, the influence of VP techniques on tools for mainstream programming is minor, limited to features such as syntax highlighting and error underlining. The recent work on showing code fragments in a two-dimensional canvas in Code Bubbles [3] and the related Debugger Canvas [4] (part of Visual Studio) is an important step forward, but these and other attempts to improve existing IDEs build on top of their solid textual foundation, which limits what visualizations are possible. To our knowledge there is no VP tool that: (1) is fundamentally a visual tool, (2) supports mainstream languages, (3) is designed for professionals, and (4) can handle large projects. Building such a tool poses a number of challenges:

Flexibility. Professional developers are involved in many different tasks including designing and documenting software, exploring unfamiliar code, implementing features, testing and debugging, working with domain-specific languages (DSLs), etc. Different tasks have different information needs and navigation strategies. Therefore, visualizations and interactions need to be flexible and customizable to enable tailoring the tool to a developer's needs in the context of specific tasks and domains. Existing tools for VP rarely satisfy this demand for flexibility.

Fluid interactions. A key strength of text editors is that they provide a universal set of interactions that developers are well familiar with and can use efficiently. Thanks to these efficient interactions, editing code takes only a small fraction of the overall development time. A successful VP tool must be at least as good as text-based editors when it comes to manipulating source code. It should allow developers to edit the code quickly and directly. Many existing VP tools focus on mouse-based interactions that are cumbersome and limiting for skilled users — a problem that Green and Petre [5] called "high viscosity". Some tools, like Barista [6], do provide keyboard-based interactions closer to text, but at the cost of more closely coupling the visualizations used for editing to the grammar of the language, which hurts flexibility.

Performance. Unlike beginners or most end-users, professional programmers work on large projects that can span millions of lines of code. Accordingly, the tools supporting developers in their tasks must perform well with large programs. A VP environment must remain responsive as visualizations get more complex and as more of them are drawn simultaneously. VP tools targeting mainstream languages such as Barista or Alice, do not share our goal to support professional developers and have not been shown to perform well with large projects.

To address these three issues we built Envision, a fast and flexible visual programming environment for large-scale object-oriented programs. In this paper we provide an overview of Envision and our solutions, which are sufficiently general to address these issues also in other VP tools. We are not aware of any other VP tool that solves all three issues simultaneously for general-purpose languages. The paper has the following organization. In Sec. II, we explore Envision's user interface and demonstrate some of its flexibility, interaction, and performance features. In Sec. III, we present a CogTool evaluation that shows that editing code in Envision is as efficient as in Eclipse. We discuss related work in Sec. IV and conclude in Sec. V. Envision's source code and videos of the tool can be found at the project's home page: http://www.pm.inf.ethz.ch/research/envision. The extended version of this paper [7] provides additional detail about Envision's interactions and architecture.

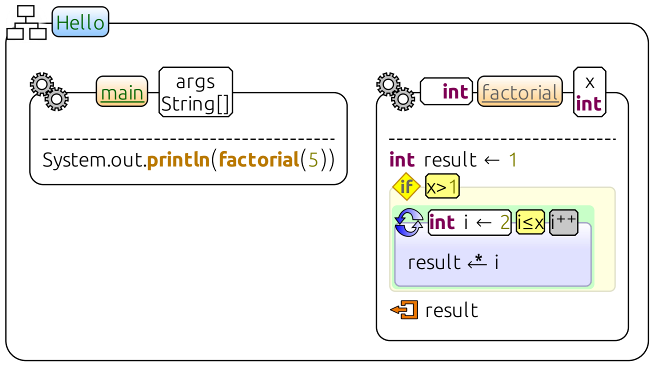
[1]http://www.ni.com/labview/
[2]www.tiobe.com/index.php/tiobe_index
[3]http://langpop.com/

© 2014 IEEE doi: 10.1109/VLHCC.2014.6883014

II. Overview of Envision

A. User Interface

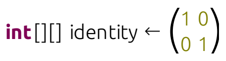
Envision renders code fragments on a two-dimensional canvas. One such fragment can be seen in the screen shot in Fig. 1. It shows a Java class `Hello` containing two methods, `main` and `factorial`.



```
Hello
  main  args String[]
    System.out.println(factorial(5))
  int factorial  x int
    int result ← 1
    if x>1
      int i ← 2  i≤x  i++
        result ←* i
    result
```

Fig. 1. Envision showing a Java class that prints the factorial of 5. The class has an icon depicting a class hierarchy and a blue background behind the class name. Methods have an icon depicting two cogs and an orange background. A green name indicates a public symbol and a gray name one with default visibility. The underline indicates a static method. The $\overset{*}{\leftarrow}$ icon represents the $\star=$ operator.

Envision uses the semantics of the underlying textual language, but in contrast to text editors, it can display graphical objects to represent language constructs. We are still experimenting with different visualizations and can freely choose how a code fragment is rendered — whether to use text, icons, or shapes; how to compose visualizations; what colors to use; etc. Fig. 1 shows the current defaults of the system. High-level code structures and declarations like classes, methods, and some statements are visualized with a box and icon. We can use visual properties, instead of text, to encode meaning. For example we use icons instead of keywords, and spatial arrangement to express control structures. Low-level code structures such as expressions are visualized in a linear sequence of visual items, mostly just text. Even though expressions might look like text, they are just standard visual objects and are decoupled from the syntax of the language. Therefore, non-textual visualizations are also possible, for example, two-dimensional arrays are visualized as matrices as shown in Fig. 2.



$$\textbf{int}[][]\ \text{identity} \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Fig. 2. A two-dimensional array rendered as a matrix. The matrix is editable. The textual equivalent is `int[][] identity = {{1,0},{0,1}}`.

To enable the quick and convenient manipulation of code visualized as in Figs. 1 and 2, all edits in Envision are achievable using the keyboard as described in Sec. II-C. This includes creating new structures such as methods, editing expressions, and navigating between parts of the visualization.

B. Flexibility

Envision allows the creation of multiple alternative visualizations for the same type of AST nodes. For example, in Fig. 3 you can see the `factorial` method from Fig. 1 rendered using an alternative visualization showing the control flow. Such alternative visualizations could be tailored to a particular task by presenting different information or rendering information differently. Alternative visualizations can be used not only for different tasks, but also for different domains or libraries. For example, Envision can automatically use an alternative visualization instead of the default one, based on programmable conditions that may depend on the program's entire AST. This is illustrated in Fig. 4. The `append` method declares two postconditions using Microsoft's Code Contracts. The postconditions are expressed by calls to the static methods `Ensures` and `OldValue`. This approach enables the encoding of specifications without extending the underlying language. In Fig. 4a, we see how this looks using the default visualizations, and in Fig. 4b, we see the same method with alternative visualizations. Here, Envision is customized to automatically show all calls to the `Ensures` and `OldValue` methods using

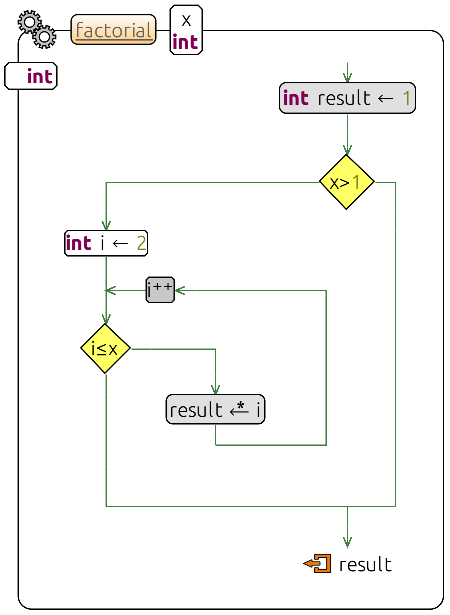
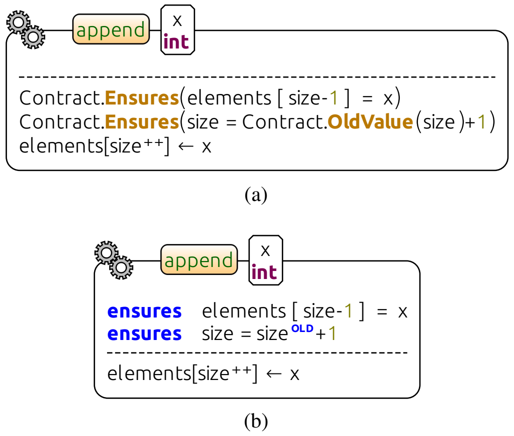


Fig. 3. The control flow visualization of the factorial method. The visualization is not just a static image, that is, code can be edited in this form.



```
append  x int
  Contract.Ensures(elements [ size-1 ] = x)
  Contract.Ensures(size = Contract.OldValue(size)+1)
  elements[size++] ← x
```

(a)

```
append  x int
  ensures   elements [ size-1 ] = x
  ensures   size = size^OLD+1
  elements[size++] ← x
```

(b)

Fig. 4. (a) A method that appends an element to a list using .NET Code Contracts to specify postconditions and (b) the same method with custom visualizations for the contract method calls. Both forms are editable.

a keyword-like visualization that is easier to read. The contracts are shown as part of the method signature, above the dashed line. This type of customization is especially useful for designing embedded DSLs as discussed in our previous work [8].

We are also experimenting with rich documentation support. Fig. 5 shows an example of a rich comment. Users can write

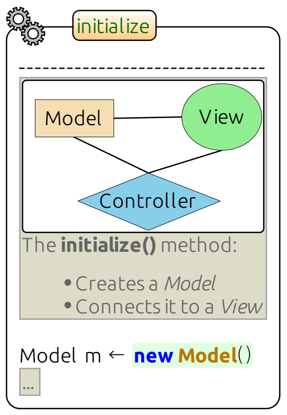


Fig. 5. An early prototype of our rich comment editor supporting diagrams.

comments using a version of the popular Markdown[4] syntax, which is rendered as HTML for viewing. Additionally, comments can contain diagrams, created directly inside Envision, images, web browsers, and custom HTML/JavaScript code.

As we have seen, Envision enables the creation of visualizations that differ significantly from text. The effectiveness of such visualizations depends directly on the interactions that they provide. On the one hand, to make the tool more intuitive to use and reduce the overhead of extending it, Envision supports a number of generic interactions that work across all visualizations. These are briefly discussed in Sec. II-C. On the other hand, Envision allows the customization of visualizations and interactions for a particular user, organization, or domain. This makes it possible to create new visualizations with complex behavior, and facilitates direct manipulation by avoiding a switch between simpler, but editable, visualizations and more elaborate ones that are read-only.

This flexibility is enabled by Envision's plug-in architecture. All of Envision's functionality is implemented as plug-ins, which are stacked in layers and provide services to each other. Plug-ins can add support for new interactions, visualizations, languages, and customizations. For more detail on Envision's design, please see the extended version of this paper [7].

C. *Interactions*

Professionals are used to quickly and directly edit source code using the keyboard. Therefore, Envision puts a strong focus on keyboard interactions. Unlike many VP tools, we avoid using drag-and-drop gestures for creating or editing structure and limit mouse operations to navigation and zooming. Users can also navigate with the arrow keys, moving a visual cursor anywhere on the canvas to select items, text, or points where new programming fragments can be inserted.

Similarly to a text editor, it is possible to freely edit expressions just by typing. On every keystroke expressions are unparsed into a textual form, which is adjusted based on the pressed key and then re-parsed to visualize the new expression. Special error nodes represent syntactically invalid forms, which often occur temporarily during editing. This mechanism also works for non-textual visualizations such as the one from Fig. 2.

Another interaction feature is Envision's context-sensitive command prompt. When an item is selected, the user can invoke a prompt where they can type commands. The available commands depend on what item is selected, and offer a convenient way to modify the AST or invoke tool commands. We use this feature to conveniently create top-level AST nodes such as classes and methods.

Envision has a number of built-in generic interactions that apply to all visualizations, such as copy and paste, undo, inserting and deleting objects, and the command prompt. Supporting extensibility, it is possible to create custom interactions, including commands for the context sensitive prompt, and entirely custom event handlers for any visualization. An event handler can process mouse clicks or key presses at a low level and can implement any complex behavior. We discuss Envision's interactions in more detail in the extended version of this paper [7]. For a demonstration of the interactions we invite the reader to watch the videos[6] of the tool.

D. *Performance*

Envision is capable of visualizing large amounts of code, or even entire programs at once. For instance, it can visualize the entire Apache Xerces[5] Java project with its more than 200kLoc and more than 800 classes and interfaces — comparable to simultaneously viewing all of the project's source files in a text editor. The AST of the project contains more than 1.1 million nodes, which are visualized by more than 1.4 million visual items. To get a better feeling for the scale of this project and how it is handled by Envision we invite readers to see the introductory video on the project home page[6].

Even when showing a complex visual scene, Envision does not compromise on the quality of the visualizations or the interactions. Visualizations are always drawn with vector graphics and are editable. Envision achieves good performance with large programs thanks to its performance-sensitive design and optimizations. Most important for quickly handling millions of objects are fast collision detection algorithms, which can be used to greatly reduce what visual objects need to be updated or repainted, based on whether they are visible or not. Using Qt's mature Graphics View framework has also had a major impact on Envision's design, enabling it to perform well. We provide specific optimization guidelines in the extended version of this paper [7].

III. CogTool Evaluation

To evaluate the efficiency of low-level edits in Envision, we used CogTool [9]. In CogTool, one creates a model of several user interfaces and specifies how a task can be achieved using each one. The tool then uses a cognitive model to simulate

[4]http://daringfireball.net/projects/markdown/

[5]http://xerces.apache.org/

[6]http://www.pm.inf.ethz.ch/research/envision

the performance of skilled users. The steps to complete each task and the final results are assumed to be known to the user — the edits are performed without the need to deliberate. Our evaluation resembles what Green and Petre did for their Cognitive Dimensions framework [5] when they measured ”viscosity”. We modeled how three typical code editing tasks are accomplished in Envision and Eclipse. The **HelloWorld** task is to create a new project and a method that prints the string “Hello world”. The **Rocket** task is to modify a method that computes the trajectory of a rocket to include air resistance, adapted from the work on Cognitive Dimensions. The **Tetris** task is to add a new shape to the game.

TABLE I. COGTOOL ESTIMATIONS OF TASK COMPLETION TIMES.

Task	Estimated time in seconds ($\pm10\%$ range)	
	Eclipse	Envision
HelloWorld	41.6 (37.5 - 45.8)	37.5 (33.8 - 41.3)
Rocket	75.0 (67.5 - 82.5)	72.1 (64.9 - 79.3)
Tetris	44.1 (39.7 - 48.5)	36.6 (32.9 - 40.3)

The simulation results in Table I show that for all tasks the differences in completion times between Eclipse and Envision are within the empirically observed error margin of $\pm10\%$ in CogTool's underlying cognitive model, KLM. The results indicate that editing code in Envision is as fast as in a text editor, and therefore remains an insignificant part of software development. We are not aware of another visual code editor that provides such efficient interactions and is as flexible as Envision. The CogTool models used for this evaluation can be downloaded from www.pm.inf.ethz.ch/research/envision/vlhcc2014.cgt.

IV. Related Work

Barista [6] is an implementation framework for creating visual code editors. It has features similar to Envision, such as augmenting the code with HTML documentation, the ability to present source code in different ways, keyboard navigation, and text-like editing. Our work goes beyond Barista in a number of important aspects. Firstly, not all visualizations in Barista are editable. The tool's authors demonstrate pretty-printed views of mathematical operators, but for editing revert to visualizations that closely match the tokens of the concrete language. This is not necessary in Envision, since more powerful interactions allow all visualizations to be edited directly. Secondly, as we have previously shown [8], we focus strongly on customizability. Thirdly, unlike Envision, Barista is not built to support large software projects, and its authors do not make any performance claims.

Targeting the same audience as Envision are tools like Code Bubbles [3] and the related Debugger Canvas [4]. Building on the well-established Eclipse and Visual Studio platforms, these tools provide an alternative way to navigate code in a two-dimensional canvas. This canvas can be populated with different “bubbles”, which represent methods, classes, documentation, and other artifacts. However, these visualizations are not flexible. Ultimately both tools use text as their foundation, and there is a classical text editor in a code bubble. This precludes alternative visualizations for most language constructs, for example for expressions, as in Fig. 2.

LabView is perhaps the most successful visual programming tool used by professionals. Unlike Envision, LabView targets the domain of measurement and control systems, and is based on a non-mainstream, data-flow driven programming model.

Unlike tools for professionals, visual tools for beginners such as Alice [1] or Scratch [2] do not place a strong focus on keyboard interactions, freedom of editing the program, performance, or customizability.

V. Conclusion and Future Work

In this paper, we introduced Envision — a visual code editor for OO programs. It offers flexibility in defining how programs are visualized and provides fluid interactions that are familiar to professional developers. Using CogTool, we demonstrated that the interactions of our visual structured editor are as efficient as those of a standard text editor. Envision is also a platform for experimenting with new ideas in the area of visual code editors.

We are currently working on a semantic zoom feature and also plan to explore additional ways to more efficiently navigate Envision's canvas with large projects. Since a visual editor provides a lot of freedom for visual effects, arrangements, and overlays, we want to investigate how these can improve a developer's productivity and help them find information quicker. Finally, we plan to update the look of the default visualizations based on user input and good visual design principles.

Acknowledgments

We would like to thank Jonas Trappenberg for his work on rich comments in Envision as part of his Bachelor theses.

References

[1] S. Cooper, “The design of Alice,” *Trans. Comput. Educ.*, vol. 10, pp. 15:1–15:16, November 2010.

[2] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, “The Scratch programming language and environment,” *Trans. Comput. Educ.*, vol. 10, no. 4, pp. 16:1–16:15, Nov. 2010.

[3] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., “Code Bubbles: a working set-based interface for code understanding and maintenance,” CHI ’10, pp. 2503–2512.

[4] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, “Debugger Canvas: industrial experience with the Code Bubbles paradigm,” ICSE ’12, pp. 1064–1073.

[5] T. Green and M. Petre, “Usability analysis of visual programming environments: A ”Cognitive Dimensions” framework,” *JVLC*, vol. 7, no. 2, pp. 131 – 174, 1996.

[6] A. J. Ko and B. A. Myers, “Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors,” CHI ’06, pp. 387–396.

[7] D. Asenov and P. Müller, “Envision: A fast and flexible visual code editor with fluid interactions,” ETH-Zürich, Tech. Rep., 2014, available at www.pm.inf.ethz.ch/publications.

[8] D. Asenov and P. Müller, “Customizing the visualization and interaction for embedded domain-specific languages in a structured editor,” VLHCC ’13, pp. 127–130.

[9] B. E. John, K. Prevas, D. D. Salvucci, and K. Koedinger, “Predictive human performance modeling made easy,” CHI ’04, pp. 455–462.