Envision

A Fast and Flexible Visual Code Editor with Fluid Interactions

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

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 a wide range of visualizations, tailored to a developer's needs in the context of specific tasks. Going beyond the tool maintainers, professional users of a tool might also have the need to customize it, for a particular organization or project. Existing tools for VP rarely satisfy this demand for flexibility and extensibility.

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 offer interactions that are cumbersome and limiting for skilled users — a problem that Green and Petre [5] called "high viscosity". Oftentimes the problem comes from a strong focus on mouse-based interactions. The issue especially affects VP tools that manipulate non-textual representations, such as LabView or Scratch. Some VP 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.

This paper contributes a solution to these three fundamental issues. We are not aware of any other VP tool that solves them simultaneously for general-purpose languages. To address the issues we built Envision, a visual programming environment for large-scale object-oriented programs. Our solutions are sufficiently general to address these issues also in other VP tools. The contributions of this paper are threefold:
(1) We present an experimental VP tool that is fast and flexible.
(2) We provide a detailed description of interactions that enable efficient manipulation of programs in Envision.
(3) We show the efficiency of these interactions by using

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

CogTool [7], [8] to compare Envision to Eclipse on three typical code editing tasks.
Videos of Envision can be found at the project's home page: http://www.pm.inf.ethz.ch/research/envision.

The paper has the following organization. In Sec. II, we explore Envision's user interface, demonstrate some of its flexibility features and performance, and define requirements for the tool's interactions. In Sec. III, we describe in detail the most essential interaction mechanisms in Envision. Sec. IV gives more information about Envision's design and architecture and provides guidelines for achieving good performance with large projects. In Sec. V, we present a CogTool simulation that shows that editing code in Envision is as quick as in Eclipse. We discuss related work in Sec. VI and conclude in Sec. VII. A shorter version of this paper appeared at VL/HCC 2014 [9].

II. Feature and UI Overview

In this section, we introduce the basics of code visualization in Envision and show different flexibility and performance aspects of the tool. We also define interaction requirements needed to effectively support these features.

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`.

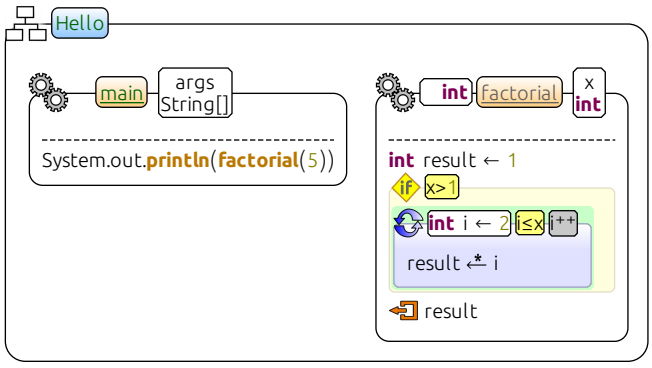


Fig. 1. Envision showing a Java class that prints the factorial of 5. The class has an icon depicting a class hierarchy and a blue background behind the class name. Methods have an icon depicting two cogs and an orange background. A green name indicates a public symbol and a gray name one with default visibility. The underline indicates a static method. The $\overset{*}{\leftarrow}$ icon represents the $*=$ 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. In general, we are exploring visualizations that abstract from the concrete language syntax and focus on the language features instead. 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. Objects like methods or classes, can be freely arranged in two dimensions within the body of their container. This allows one to visually group related objects together. 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, like everything else in Envision. In fact these visualizations are decoupled from the syntax of the language and it is possible to render something completely different, such as an icon or an interactive widget. For example, two-dimensional arrays are visualized as matrices as shown in Fig. 2.

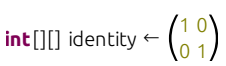


Fig. 2. A two-dimensional array rendered as a matrix. The matrix is editable. The textual equivalent is `int[][] identity = {{1,0},{0,1}}`.

To enable the quick and convenient manipulation of code visualized as in Figs. 1 and 2, Envision satisfies the following two interaction requirements:
Req-keyboard: All edits should be achievable by using just the keyboard, like in a standard text editor. This includes creating new structures such as methods, editing expressions, and navigating between different parts of the visualization. Developers can be very efficient using the keyboard and it should not play a secondary role in VP tools.
Req-feedback: After the AST of the program has been modified based on the user's input, visualizations should be updated immediately, to provide feedback on the new state of the AST. Processing input in key-stroke batches delays feedback which may confuse users and make them less productive.

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 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 [10].

Going beyond executable code, we have started experimenting with rich documentation support. Fig. 5 shows an example

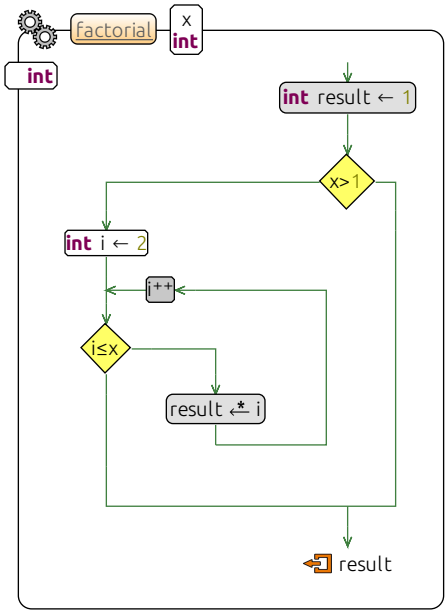
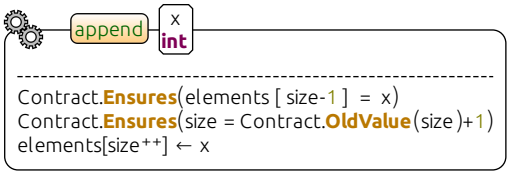
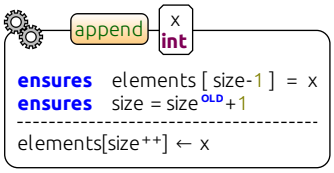


Fig. 3. The control flow visualization of the factorial method. The visualization is not just a static image, that is, code can be edited in this form.



(a)



(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.

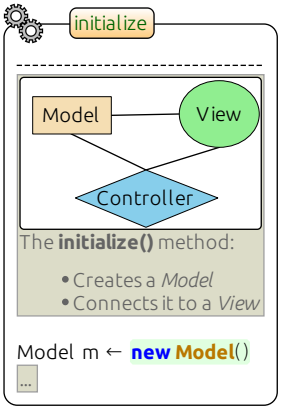


Fig. 5. An early prototype of our rich comment editor supporting diagrams.

of a rich comment. Users can write comments using a version of the popular Markdown[4] syntax. For viewing, comments are visualized in their HTML equivalent. In addition to formatted text in comments, Envision features a built-in diagram editor that enables the creation of simple diagrams directly inside a comment. Other comment features are support for displaying images from disk, displaying a specified web page in an inline browser, and even running custom HTML/JavaScript code in an inline browser.

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. Two additional requirements implemented in Envision help provide efficient interactions:

Req-generic: Built-in generic interactions, such as copy and paste, should work across all visualizations. Having a set of core interactions makes the system more intuitive and transparent for the user. Furthermore, generic interactions reduce the implementation overhead for new visualizations, thereby promoting alternatives.

Req-customizable: In addition to the functionality provided by generic interactions, it should be possible to create custom interactions. In this way, existing visualizations could be adapted to user preferences or new domains, and new visualizations with complex behavior could be created. This facilitates direct manipulation by avoiding a switch between simpler, but editable, visualizations and more elaborate ones that are read-only.

C. 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].

Fig. 6 shows a zoomed-out view showing a part of the canvas. Users can zoom in/out using the mouse wheel. At any zoom level objects can be selected, moved, and edited normally. The rectangles in the middle show classes using the same visualization as Fig. 1, but due to the high zoom level, it is not possible to see their contents in detail. Text overlays show the class names. In the bottom left corner there is a mini-map that shows the entire project at a glance. Programmers familiar with the spatial layout of the project can simply click on the map or zoom in on a location to navigate to it.

Even when showing a complex visual scene, Envision does not compromise on the quality of the visualizations or the interactions. At any zoom level, visualizations are drawn with vector graphics and are editable, provided they are visible. This dictates one final requirement towards the tool:

Req-responsive: As the complexity and number of visualizations grow, the environment should remain responsive.

[4] http://daringfireball.net/projects/markdown/
[5] http://xerces.apache.org/
[6] http://www.pm.inf.ethz.ch/research/envision

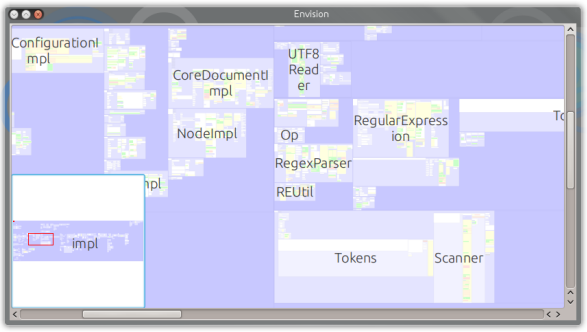


Fig. 6. A zoomed-out view showing a part of the Apache Xerces project.

Professionals are often involved in big projects and an editor's performance must scale well to such projects. This includes both interactions for navigating and for editing.

III. Interactions

In this section, we detail the core interactions provided by Envision. Professional developers are used to quickly and directly editing the source code of a program using the keyboard. Therefore, we put a strong focus on keyboard-based interactions. In particular, we avoid using drag-and-drop gestures for creating or editing structure and limit mouse operations to navigation and zooming.

A. Universal Visual Cursor

A prerequisite for any application that uses keyboard input is the cursor. In textual environments, it indicates where the text that the user types will go. Developers are able to position the cursor anywhere in a text editor, regardless of the syntax of the edited document. Visual environments typically do not offer this freedom, and limit the cursor to selecting a visual object or manipulating a text field.

For Envision, we designed a cursor that provides the freedom of a text editor in a visual setting. Using the keyboard arrows or mouse clicks, it is possible to position the cursor virtually anywhere on the canvas:
(1) On top of an item, thus selecting the item. Selecting a visual item is a typical interaction in visual tools. For example, in Fig. 1, we can select the class's icon to copy or delete the class.
(2) Inside text. This is typical for text editors and text box widgets. For example, we can place the cursor in the middle of the `identity` label in Fig. 2.
(3) Between items, in empty spaces. This is an unusual cursor location, which provides a lot of possibilities for interactions. Being able to mark empty space is especially useful for creating new structures. For example, in Fig. 4, we can place the cursor between the method's name and icon, marking a location that allows us to directly enter the method's return type.

Next, we will provide a few more technical details about how cursors are defined. Each visual item in Envision has a bounding rectangle and can declare regions within this rectangle that can be occupied by the cursor. Typically, regions occupied by child items (e.g., an icon) are declared so that children can be selected. Regions, such as between children, or at the corners

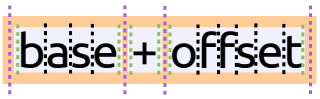


Fig. 7. An expression where all cursor regions are indicated with dashed lines. Adjacent regions are treated as equivalent.

or edges, can also be declared to enable additional interactions, such as removing a child by pressing Del. Pressing an arrow key moves the cursor in the desired direction to the closest cursor region within the same visualization. If the cursor is already at the edge, it is moved to the closest region of the parent visualization and so on. For example, in Fig. 1, if the cursor is at the end of the `println` expression, pressing → will move the cursor to the beginning of `int result ← 1` in the `factorial` method. This mechanism also works for more complex visualizations like the one from Fig. 3, where it is possible to switch between all parts of the method body by simply using ↑ and ↓. In many common cases, appropriate regions can be automatically declared even for new visualizations. For example, when using the standard list layout provided by Envision, it will automatically declare cursor regions for list items and for empty spaces in the list. This automatic behavior supports generic interactions that work across visualizations. It is also possible to manually define cursor regions in a visualization to achieve a custom interaction.

When using the arrow keys to move the cursor, there are two interesting situations that need special treatment. The first one arises when objects are placed far apart and the next cursor position could be quite far away on the screen. Large jumps in the location of the cursor can confuse the user, so we limit the distance the cursor is allowed to move when a key is pressed, that is, the key press is ignored if the cursor would jump too far away from its current position.

The second situation arises when there are many adjacent cursor regions. A common case is illustrated in Fig. 7. It is an expression consisting of three visualizations (two identifiers and an operator) arranged horizontally. Each of them has cursor regions at its edges, colored in green. There are also cursor regions between the visualizations, colored in purple. Imagine, for example, that the cursor is positioned right after the letter "a" and a user wants to use the keyboard to position the cursor right after the "o". Going through all the cursor regions will take 8 key presses instead of the desirable 4. Thus Envision provides a way to flag adjacent cursor regions as equivalent and treat them as one region during navigation. Using this feature moving the cursor in the case from Fig. 7 behaves as desirable.

The cursor is solely based on the structure of the visualizations and is independent of the program's AST. This makes the cursor a universal tool for editing in Envision and a key enabler of keyboard-centric interactions. The cursor enables three fundamental interactions in Envision: (1) marking positions for the insertion or deletion of objects via a single key press such as Enter or Back←; (2) providing a familiar text-like edit functionality for objects that resemble text, such as expressions; (3) selecting a context for invoking a context-sensitive command prompt; We present the latter two next.

B. Free Expression Editing

In a text editor, edits are unrestricted and allow the developer to temporarily break the code structure to quickly achieve the desired results with only a few keystrokes (e.g., typing `{{1,0},{0,1}}` from left to right). Providing such a quick interaction has traditionally been a problem for visual editors, especially if complex visualizations like the one from Fig. 2 must be editable. Quick editing is especially important for expressions since they are the leaves of ASTs — the larger part of a program — and are edited very frequently.

Following JPie [11] and Barista [6], we made expressions freely editable, like text. We relaxed the constraint that an expression has to be always syntactically correct by adding special error nodes to represent incorrect tokens. This design provides the flexibile text-like edits that developers are familiar with. For example, we can create the entire expression in Fig. 2 by just typing it from left to right.

As shown in Fig. 2, we decouple how expressions are visualized from the concrete language syntax. To achieve this we use a bi-directional mapping between visualizations and text. On every keystroke, the visualizations of expressions are unparsed into a textual equivalent, which is implicitly modified based on the current cursor position, and re-parsed to update the AST. The immediate updates implement the requirement for immediate feedback. The bi-directional mapping mechanism is essential for providing flexibility. In essence it maps the individual components of a visualization, such as children or edges, to strings that represent expression nodes from an AST. For example, the values of the matrix from Fig. 2 map to literal expressions, and the surrounding parenthesis to the top-level initializer expression. The ability to define such a mapping between any visualization and text is a difference to other tools like Barista, which has a tighter coupling between syntax and editable visualizations.

C. Context-sensitive Command Prompt

Compared to expressions, higher-level nodes of a program's AST are edited less frequently and are more stable. This stability allows interactions on a structural level to be efficient, which makes it possible to keep higher-level AST nodes always in a consistent state, and eliminates the need for error nodes like in expressions. To edit high-level AST structures, we use a context-sensitive command prompt. It provides access to actions specific for the currently selected item as well as all of Envision's commands.

Each visualization type has an extensible set of commands, which can be typed in the prompt. For example, the class visualization from Fig. 1 has commands to add methods or fields to the class. Pressing Esc or clicking the right mouse button shows the command prompt right below the cursor. This allows the execution of: (1) commands associated with the object at the cursor or any of its ancestor objects, and (2) core tool commands such as *find*, *save project*, and *exit*. A command has access to the AST and the visual canvas. Therefore, there are no restrictions on what a command does: it can manipulate the AST, change properties of visualizations, or even execute a system command. The prompt works across all visualizations and fulfills the requirement for generic interactions. Envision provides a number of built-in commands, but it is also possible to register new ones, according to the customizability requirement.

When typing commands, Envision provides a list of auto-completion suggestions and a brief description of each command as can be seen in Fig. 8. This helps explore the available commands and reduces the need to remember the list of arguments. If the user invokes an incorrect command, an error message will be displayed directly in the prompt. A further feature simplifying command entry and reducing the need to remember the precise syntax of commands is the support for abbreviations. For example, instead of typing the complete form of the `method` command to create a method (e.g., `public static method Main`), it is sufficient to abbreviate the command name and arguments as long as the abbreviations are not ambiguous. This is illustrated in Fig. 8.

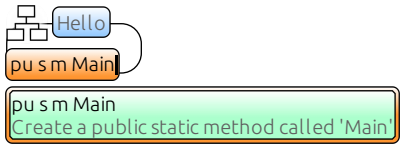


Fig. 8. An abbreviated command in a prompt invoked on a class object.

D. Other Interaction Features

Fulfilling the requirement for generic interactions, Envision provides a number of additional interactions that work across all visualizations such as the standard copy, paste, and undo operations, as well as a way to manipulate the AST independently of how it is visualized, by using primitive tree operations. Implementing the requirement for customizability, Envision allows the system to be customized in two ways to achieve specialized behaviors. The first one is to use customization features of existing interaction mechanisms. For example, new commands can be added to the command prompt. The second way is to implement new interaction handlers in order to replace existing ones, or use them with new visualizations. A handler processes all mouse and keyboard events coming from visualizations and can be used to implement complex behaviors on top of Envision's generic interactions. We use this method, for example, to implement free expression editing or to enable intuitive editing of matrices like the one from Fig. 2. This method is also particularly suitable for customizing interactions when working with embedded DSLs like we show in our previous work [10].

IV. Design and Architecture

In this section, we discuss Envision's architecture in more detail. We discuss what mechanisms form the foundation of the tool's flexibility and provide guidelines for achieving good performance in complex visualizations.

A. Plug-in Architecture for Flexibility

Envision is built using a modular plug-in architecture. All of the tool's functionality is implemented by plug-ins with well-defined interfaces. The different plug-ins are logically stacked into layers, where each successive layer can use the services of previous layers. An overview of the available plug-ins and layers is shown in Fig. 9.

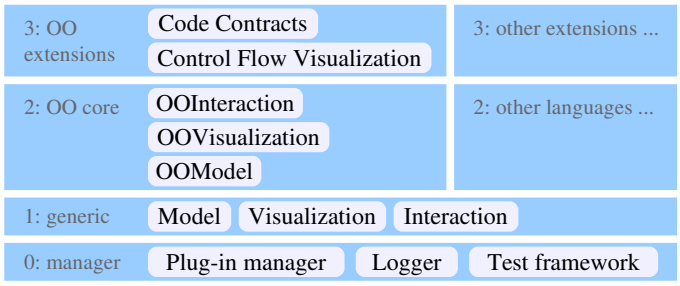


Fig. 9. The different layers and plug-ins of Envision's architecture.

Layer 0 contains the basic functionality of the system such as plug-in management, logging, reflection mechanisms, and a self-testing framework.

Layer 1 consists of three plug-ins that implement the different parts of a Model-View-Controller (MVC) framework that is the foundation of Envision. The *Model* plug-in implements generic functionality for specifying a program AST. The *Visualization* plug-in implements Envision's visualization engine on top of Qt's Graphics View[7] framework. The *Interaction* plug-in implements the generic interactions available in the system. These three plug-ins define many extension and customization mechanisms that are used in higher layers. This layer provides services that could be used for supporting any programming language.

Layer 2 is for groups of extensions for a particular language or programming paradigm. At the moment we support only OO languages. We have developed three plug-ins, which are the OO extensions of the generic MVC implementations: *OOModel*, *OOVisualization*, and *OOInteraction*. They define what AST nodes form an OO program, their default visualizations (Figs. 1 and 2), and interactions. These plug-ins also provide extension mechanisms that can be used from higher layers.

Plug-ins in further layers can customize language features and provide alternative visualizations and interactions. For example, a library designer could optionally bundle an Envision plug-in with the library in order to customize how code from the library is visualized and how developers interact with it. The example in Fig. 4b is implemented with such a plug-in. This can be useful for designers of embedded DSLs. Another plug-in at this layer implements the control flow visualization in Fig. 3.

Envision's architecture promotes a clear separation of concerns and facilitates extensibility. The separation into layers and plug-ins makes it easy to identify where a new feature should be implemented. The overhead of implementing new features is reduced by the reuse of existing building blocks, plug-in services, and customization concepts. For example, throughout Envision, we use a declarative API to create new visualizations by combining existing ones.

Envision's design has also been beneficial for us as researchers. It allows us to experiment with a variety of visualization and interaction techniques, which lead to the ones presented in Secs. II and III. For example, when creating a new visualization, it is possible to define many of its properties such as colors, shapes, distances, font properties, icons, etc. using the built-in styles API. Using the API, property values are automatically read from an XML file on disk and can be changed without recompiling the program. Other researchers are also welcome to experiment with the tool — its C++ source code is open[8]. Development is done in Linux, but Envision is a cross-platform tool and the code is occasionally updated for Windows compatibility.

B. Performance Guidelines

Envision achieves good performance with large programs thanks to its performance-sensitive design and optimizations. Some of these optimizations, related to collision detection (CD), are inspired by video game rendering techniques; others, such as reducing the number of memory allocations, are made possible by the technology we use to implement Envision. Below we formulate the most important ones as guidelines for tool designers. Through these optimization, Envision remains responsive with large projects.

Structure visualizations in a visual tree. Qt uses CD algorithms to determine which objects: are visible on the screen, are under the mouse cursor, should be updated, etc. Although visual objects in Qt are logically organized as a tree, by default the bounding box of each object depends only on what the object draws itself and disregards children. Thus, to calculate collisions, Qt iterates over all objects, which is slow. CD can be much faster if the bounding box of each item completely encompasses all of its children. Qt has support for enforcing this, which results in some performance gains, but also introduces overhead due to the enforcement. We modified Qt's source code to skip this enforcement, which is not necessary in Envision since all visualizations respect that condition. This resulted in significant benefits when drawing a large number of objects at once. Note that this optimization does not restrict the flexibility of Envision's visualizations, since objects are allowed to overlap.

Draw only what is necessary. Qt already provides support for culling off-screen objects by using CD to decide what is not visible. However, when observing a canvas with many visual objects at a high zoom level, most of them are visible and therefore drawn. This is especially of concern to Envision, since the mini map that we provide in the tool (see Fig. 6) is actually just a second rendering of the same canvas, where all objects are always visible. With millions of objects, the standard Qt approach was too slow and unusable. At a very high zoom level, many visual objects (especially text which is slow to draw) are practically invisible, but were nevertheless drawn. We made a further modification to Qt's drawing routines, to skip the drawing of any visual item that occupies less than one pixel in any dimension. Note that this also automatically skips the drawing of all child items. This results in a significant speed up when drawing a large number of objects, like the canvas from Fig. 6.

Cache text renderings. At medium zoom levels, thousands of objects could be within the visible region of the screen and not small enough to be discarded by the optimizations discussed so far. This is especially a problem for textual elements (labels, expressions, etc.) as drawing so much anti-aliased text at once

[7] http://qt-project.org/doc/qt-4.8/graphicsview.html

[8] https://github.com/dimitar-asenov/Envision

can be slow. We used the functionality of Qt's `QStaticText` class to speed up text drawing by caching results of drawing operations. In this way, text has to be redrawn only when it is updated or when the zoom level changes. This resulted in a noticeable speed-up, without a significant memory cost.

Decouple updates from rendering. Modern graphics frameworks minimize the number of painting operations, but painting still occurs quite often, for example when scrolling, zooming, panning, etc. In many of these cases, all or almost all objects on the visual canvas remain unchanged and do not need to be updated. For achieving good performance, it is essential that updates are decoupled from the regular painting operations, and are only performed when necessary. When the user modifies a program, only visualizations that are affected should be updated. Using a tree organization for visualizations, an edit typically affects only a particular visualization and all of its ancestors in the tree.

Use mature graphics frameworks and libraries designed for performance. Envision is based on the Qt framework, which features a mature C++ graphics framework. It has a large community of developers and often receives optimizations or even redesigns that greatly improve performance. This choice has had a big impact on Envision's performance. It enabled us to build an environment that remains responsive even with large projects.

V. CogTool Evaluation

To evaluate the fluidity of the interactions of our system, we used CogTool [7], [8]. Our goal is to show that the speed of low-level edits in Envision is comparable to text-based tools and, therefore, editing in Envision, remains an insignificant part of software development. This is in contrast to existing visual programming tools, which often suffer from cumbersome interactions. Our evaluation resembles what Green and Petre did for their Cognitive Dimensions framework [5] when they called this bottleneck "high viscosity". CogTool is well suited for such an evaluation. 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 the performance of skilled users, who know how to achieve the given task without hesitation.

We modeled how three typical code editing tasks are accomplished in Envision and Eclipse, similarly to what other researchers have done in the past [8]. The steps to complete each task and the final results are assumed to be known to the developer — the programs just need to be edited without the need to deliberate. Next, we briefly describe the three tasks.

HelloWorld — This task involves the creation of a new project, a class within the project, and a simple `main` method that prints the string "Hello world". In Envision, we create the project, class, and the method using the command prompt and abbreviated commands. The call to the `println` method is created without using auto-completion. In Eclipse, we use the main menu to create the new project and a context menu to create the new class. We did not use the option to automatically create the `main` method in the class wizard dialog, as this is not a part of the usual Eclipse interactions and will skew the results. Envision does not have such a shortcut but one could be easily added. When editing the method, we used Eclipse's built-in code snippet facility to speed up writing. As in the Envision case, we did not use auto-completion.

Rocket — This task is an adaptation of a task from the work on Cognitive Dimensions of Green and Petre [5] and is originally based on work by Curtis et al. [12]. The task is to modify a method that computes the trajectory of a rocket to account for air resistance. It requires the addition of five new statements to a method. Using this task, Green and Petre showed that visual programming environments can suffer from high viscosity — resistance to local change — and require several times the amount of time for such changes compared to a textual environment. We adapted the task to the Java programming language and modeled it in Eclipse and Envision. The actions required to complete the task in both tools are almost identical.

Tetris — In this task, a new shape is added to a game of Tetris. This consists of adding a new element to an existing enumeration and editing a three-dimensional array, a loop condition, and another array. In Envision, the three-dimensional array is edited as a matrix of arrays, and there is no need to manually insert space breaks to align array elements. The minimap is used to navigate from one edit location to another. In Eclipse, space breaks are inserted to nicely align the elements in the three-dimensional array. Navigation between different edit locations is done via the file tabs and the scroll bar.

TABLE I. CogTool estimations of task completion times.

Task	Estimated time in seconds ($\pm 10\%$ 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 three tasks the differences in completion times between Eclipse and Envision are within the empirically observed error margin of $\pm 10\%$ in CogTool's underlying KLM cognitive model. The results indicate that editing code in Envision is as fast as in a text editor. We are not aware of another visual code editor that provides such fluid interactions and offers visualizations as flexible as the ones in Envision. The CogTool models used for this evaluation can be downloaded from www.pm.inf.ethz.ch/research/envision/vlhcc2014.cgt.

The goal of this evaluation is very specific and is independent of code comprehension, developer accuracy, and learnability. To test these qualities, further experiments with professionals are needed.

VI. 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 [10], we focus strongly on customizability, through explicit mechanisms in the existing implementation and through our plug-in based architecture. Thirdly, unlike Envision, Barista is not built to support large software projects. The authors consider only small examples and do not make any performance claims.

MPS [13] and Intentional software [14] are two domain workbench tools that also provide visual structured editors. Both of these tools take a non-standard approach to software engineering where many domain-specific languages are defined and combined together. This is in contrast to Envision's goal of supporting mainstream programming.

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.

There are many visual programming tools for beginners such as Alice [1], Scratch [2], and JPie [11]. Such tools focus on easily constructing executable programs via visual means and displaying a live execution while programming. Users can use drag and drop gestures to put different program fragments together, which eliminates most if not all syntactical errors. Unlike tools for professionals, there is no strong focus on keyboard interactions, freedom of editing the program, performance, or customizability.

Another big group of visual programming tools are applications designed for end-users. These environments are designed primarily for ease of use and smaller programs. They mostly use their own programming model and often lack good support for large, complex projects or flexible visualizations.

VII. 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 complementing Envision's geometric zoom with a semantic zoom feature. We also plan to explore different ways to more efficiently navigate Envision's canvas. 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 Andrea Helfenstein for her work on a declarative API for defining visualizations and Jonas Trappenberg for his work on rich comments in Envision, as part of their 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] B. E. John, K. Prevas, D. D. Salvucci, and K. Koedinger, "Predictive human performance modeling made easy," CHI '04, pp. 455–462.

[8] R. Bellamy, B. John, J. Richards, and J. Thomas, "Using CogTool to model programming tasks," PLATEAU '10, pp. 1:1–1:6.

[9] D. Asenov and P. Müller, "Envision: A fast and flexible visual code editor with fluid interactions (overview)," to appear in VLHCC '14.

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

[11] B. E. Birnbaum and K. J. Goldman, "Achieving flexibility in direct-manipulation programming environments by relaxing the edit-time grammar," VLHCC '05, pp. 259–266.

[12] B. Curtis, S. B. Sheppard, E. Kruesi-Bailey, J. Bailey, and D. A. Boehm-Davis, "Experimental evaluation of software documentation formats," *J. of Systems and Software*, vol. 9, no. 2, pp. 167 – 207, 1989.

[13] M. Fowler. (2005, June) A language workbench in action - MPS. [Online] Available: http://martinfowler.com/articles/mpsAgree.html.

[14] C. Simonyi, M. Christerson, and S. Clifford, "Intentional software," OOPSLA '06, pp. 451–464.