Research in Computer Science II
Spring 2010

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A feasibility study for a general-purpose visual programming system

Dimitar Asenov

Supervisor: Peter Müller

19th July, 2010

Contents

Abstract

As electronics and computers become an ever-bigger part of our life there is a growing need for creating higher-quality software faster. Visualizing the structure, execution and model of programs can improve the development process. Numerous methods and styles exist for visualizing different software artifacts but, despite previous efforts, visual programming has only very limited use in industrial organizations. We present *Envision* - a framework for general-purpose visual programming. It combines many existing techniques with novel ideas to increase productivity and code comprehension. It aims to support visual programming for large scale industrial projects in a wide variety of domains.

1 Introduction and Motivation

Throughout the history of computer science many new programming languages have been developed. This is hardly surprising given the fact that computational devices are playing an ever greater role in our life. The increasing variety and power of computers present a challenge to existing software solutions. To better manage the complexity of growing applications it is necessary to develop new ways to program. There can be many driving goals when creating a new language: to make it easier to learn how to program, to introduce support for a new programming paradigm or hardware feature, to make it very efficient, etc. However, languages alone can no longer cope with large-scale development tasks. To alleviate the problem there is a continuing effort to improve the integrated development environments (IDEs) that provide a platform around a language to make it easier to create and navigate a big programming project.

During all these developments one thing has remained constant - programming languages are textual. The programmer creates a program by writing its source code. In the process, different program structures and modules are saved to different files or folders. The source code, written using a precise grammar, fully defines the structure and behavior of the program. This poses a number of important restrictions:

- The developer must learn and use a very specific language syntax. This often includes rules which are not intuitive for novice programmers. Even experienced programmers might find it difficult or cumbersome to use these rules (e.g. when writing scripts in the Bash scripting language).
- Creating new programing elements and navigating existing ones relies mostly on symbolic reasoning and memory.
- Additional resources required for developing bigger projects such as documentation, diagrams, etc. are separate from the source code. They help the programmer write the application, but do not have a direct influence on it and are not linked to program elements. This causes inconsistencies when only the code or the documentation is modified.
- Getting familiar with a larger software base by only looking at its source code is an overwhelming task. To make this possible one refers to the supporting documentation, since the source code itself does not guide the programmer in how to explore the application.

While modern IDEs help to reduce the impact of these restrictions they can not remove them since these restrictions are inherent to textual representations.

In an effort to truly remove some of these drawbacks researches have experimented with visual programming languages. Some of the advantages that they found when using visual representations are:

- Visual objects can more directly map to programming entities, removing the need for complicated syntax. Thus it is easier for novice programmers to learn to use such a language. Experienced programmers can also benefit from this and be more productive.
- Visual programming involves not only symbolic, but also visual and spatial cognitive processes. This can improve both the comprehension and retention of programs as the graphical representations more closely match a programmer's mental model. Studies [1, 2] have shown that retention for visual images is better compared to that for text.
- Two dimensional graphical representations can be more expressive compared to text which uses a single dimension.
- Figures, diagrams, tables and such, can be directly embedded in a visual program and make it easier to understand and explore.

Intuitively visual representations can help better organize a programming activity.

Despite previous efforts to develop visual languages, the term programming is still commonly associated with writing source code. The top 20 languages used in industry in June 2010 according to the popular ranking site TIBOE [1] are all textual. Other on-line rankings for languages yield similar results. While the initial hopes for visual languages were high, they did not meet researchers' expectations. Meanwhile research focus has shifted away from visual programming to software visualization and end-user programming.

We propose a new take on visual programming. Learning from the lessons of previous research and developing for today's powerful hardware we have established the concept of *Envision* - a general-purpose visual programming system. The goals we set to achieve include developing a highly flexible visual programming environment, that easily scales to large programming projects while improving the developer's experience and performance compared to modern IDEs. This paper reports on an initial study designed to asses the feasibility of our approach.

Section 2 explores related work in the fields of visual languages, software visualization and human-computer interaction. Next, the main ideas behind the visualization and user interaction in *Envision* are presented in section 3. The subsequent section 4 assesses the performance of our approach compared to existing programming techniques. In section 5 we report on our plans for further development of the system. Finally we summarize our findings in section 6.

2 Related work

2.1 Visual programming languages

Many visual programming languages (VPLs) have been developed in the past. Schiffer and Fröhlich designed Vista[3] - a VPL which builds on top of the object-oriented programming (OOP) paradigm and aims to provide support for high-level building blocks and ease of use. At the more abstract level, programming in Vista involves building entities and networks between components in a visual environment. A central notion in the language is the processor which is a logical unit in the program that has a distinct function. Processors are somewhat similar to classes and communicate with each other via data and signal tokens. Different processors can be connected to each other using a graphical user interface (GUI). In the GUI each processor is represented by an icon. At the lower level of abstraction Vista provides access to the full Smalltalk library and allows writing code fragments using text.

Another tool for visual programming is Seity[4]. It is a visual environment for the Self programming language. Seity represents objects as three-dimensional boxes that list an object's properties and

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

methods. An important concept for the environment is the focus on objects, rather than views. Each object is unique and appears at most once on the screen but an object can present different views of itself based on the current needs of the programmer. Chang and his colleagues argue that this makes it easier for the programmer to maintain object identity which is not the case with view-focused IDEs. They propose that manipulating the object directly and not through an intermediate view, makes the objects concrete and tangible. On the other hand using tools such as class browsers or object inspectors makes the objects seem secondary and distant.

Citrin et al. in [5] propose VIPR - an entirely new approach to programming based on visually nesting circles. A program in VIPR would be represented by a circle that consists of a series of nested circles, representing statements, classes, methods, etc. The execution proceeds by performing the statement connected with the outer-most circle and removing that circle. More complicated execution traces can be created by connecting different circles with arrows which indicate the next statement and possibly have an associated condition. An advantage of VIPR is that each state of a program at run-time is itself a valid VIPR program - the same language is used for programming and execution visualization.

Another entirely visual approach to writing object-oriented programs is using Prograph[6]. Classes in Prograph are represented by icons and so are each class' fields and methods. A method is defined visually by connecting various types of entities together using arrows. Entities include inputs, outputs, function calls, constants, variables, object instances, etc. Prograph comes with a fully featured environment that allows the user to explore the program using various views. This includes structure visualizations, class diagrams, property explorers and others.

More recently Grant has designed Visula[7]. The key idea behind Visula is the use of UML-like sequence diagrams to define program behavior. The diagrams are extended with control-flow and class definition elements. Since UML-diagrams are used by many professionals such an approach has the benefit of being familiar to experts.

Most VPLs are short-lived and rarely get much attention outside of research circles. One notable exception is National Instruments' LabView. It is a visual data-flow language specifically designed for data acquisition, analysis and instrument control and automation. Applications represent instruments. Each application has a virtual front panel where the user can interact with a wide variety of controls: buttons, knobs, displays, light emitting diodes etc. The logic of the application is defined by its back panel where different processing elements can be connected to define the data flow. A processing element can be an entire instrument, a built-in library function, an arithmetic or string operation, etc. Flow control entities such as loops and conditionals are also included. Programming in LabView is mostly done using the mouse, by dragging and dropping components from a palette onto the instrument panel and connecting them using wires. In [8], Baroth and Hartsough report on their experience with the language at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the US National Aeronautics and Space Administration. The same software system was developed simultaneously in both LabView and C. The two teams were given identical budget and deadlines to come up with a solution. The team using LabView managed to cover and significantly exceed the requirements of the software while maintaining constant contact with the customer. The team using C did not further communicate with the customer after receiving the initial requirements and did not make a product that fully meets them. Our personal experience with LabView in designing automated tests is also very positive. However LabView is not very well suited for general purpose programming. In particular its aim at data-flow programming makes it harder to model more reactive systems. Its support for OOP principles is also limited.

2.2 Visualization techniques and interaction tools

Although the hope had been high that VPLs would bring a revolution in programming, this was not the case. Nevertheless developers realized that visualizations can indeed improve the

software development process. Thus many tools were created which instead of substituting textual programming, complement it with visual representations. They have enjoyed a much higher success and adoption rate in industrial environments.

Major efforts in visualization tools are often implemented in new versions of popular IDEs or as IDE extensions. Modern programming environments come with color coded language syntax, auto-completion tools, code snippet libraries, UML diagram generators, etc.

The Human Interactions in Programming [2] group at Microsoft Corporation continuously evaluates new ways to improve the usability of Microsoft's Visual Studio IDE. Among others they have investigated the effect of presenting code usage information directly in the IDE to improve navigation[9, 10]; why and how software developers use graphics and whether these needs can be met by embedding images directly in the IDE[11]; improving navigation by visualizing code with automatically generated maps[12] or displaying code-thumbnails of files[13].

Being an open-source tool with a strong community and industrial support the Eclipse[3] IDE has also enjoyed a lot of attention from researchers. One of the more popular extensions developed for Eclipse is Shrimp[14]. It can visualize various aspects of Java code including class hierarchies, packages or individual classes. Additionally it has support for showing associated documentation and navigating around the different elements. Going one step further is CodeBubbles[15, 16] by Bragdon et al. This tool uses Eclipse only as a back-end and presents the user's code in a new environment that features a visual representation of code based on a bubbles metaphor. A class, a method, a JavaDoc snippet or even custom notes can be displayed in an independent frame - the bubble. Each bubble is a text editor in itself. Many bubbles can be shown on the screen and the programmer can arrange them in any way she prefers. This helps with task oriented programming activities such as debugging and maintenance.

Another interesting extension for Eclipse, that focuses on user input is Quack[17]. It allows the user to program "sloppily" by entering keywords instead of proper statements. Quack would then try its best to determine the programmer's true intent and create the corresponding statements.

2.3 Visualization evaluation and experiments

Whitely has compiled an overview of VPLs in [18]. It is immediately evident that there are only a few quantitative experiments that asses the viability of visual programming. Whitley concludes that no program representation (textual or visual) is better in absolute terms. Some representations yield better results in specific tasks, while other representations in others. It is important that the task at hand and the chosen representation match well. Further observations were that for smaller programs text might be a preferred option and that special attention should be paid to how well screen estate is utilized by visual representations. In the end visuals are said to improve the programming experience but there are not enough experimental studies or sufficient cognitive theories to guide the design of VPLs.

The classic Cognitive Dimensions(CD) framework by Green and Petre[19] has been often used to evaluate the design of a new VPL. Their work suggests a number of criteria by which a programming language and its environment can be evaluated. Initially they applied their framework to three languages: FORTRAN, Prograph and LabView. They found that generally the construction of visual programs is easier because there are less syntactic hassles and because of the use of higher-level operators. They also identified several problems with visual programming that need to be addressed: clutter, insufficient screen real-estate and high viscosity (resistance to local change of programs).

[2] `http://research.microsoft.com/en-us/groups/hip/`
[3] `http://www.eclipse.org/`

3 The *Envision* system

The purpose of this feasibility study is to introduce and evaluate the visualization and user interaction concepts of *Envision*. However the overall plan for the *Envision* system goes beyond this. To provide a better insight of the context of this study we will first present the full set of features planned for *Envision*. Afterwards we elaborate on the visualization and interaction parts of the system including demonstrative examples.

3.1 Overview

The concept for *Envision* is a combination of an IDE and an information system. Like a typical IDE *Envision* allows programmers to create software and aids in this process in various ways. Acting as an information system *Envision* maintains the software's repository, including all supporting documentation, graphs, tables etc. and keeps these development artifacts in a consistent state. Here is a list of the most distinguishing features in both domains:

IDE:

- Visual source code representation - the programmer does not work purely with text, but rather interacts with a visual representation of the program. The visualization also includes text in order to achieve maximum flexibility.
- Command enabled user interaction - apart from standard means of input, the user can also issue commands to the system in a style similar to an operating system shell. These commands are dependent on the currently selected program entity and can represent arbitrary operations such as constructing new program elements, conducting searches, invoking a new process or running system commands.

Information system:

- Semantic versioning - unlike traditional file-based version control systems *Envision* stores versions of programming entities like classes, methods, fields etc. This allows for better tracking of software evolution and enables new analysis techniques.
- Content linking - a program will not be just the instructions it executes but will also contain all supporting development artifacts. Figures, animations, tables or documents can be embedded directly in the software project. Moreover links can exist between different types of content to keep the application in a consistent state. For example the value of a constant can be taken from a table in the document describing its meaning and computation.
- Continuous analysis - the *Envision* system will continuously analyze the software in the background. This involves activities such as automatic test-case execution, formal verification and collection of statistics. This information will be reported to the user directly embedded in the visual representation of the software to aid with the development process.

Envision is primarily aimed at large-scale software development. While we also consider beginners, the main target users are expert programmers. The system is meant to be a general purpose IDE that can be used to efficiently build a wide variety of software. To this end it should also support the more popular programming paradigms. In developing the concept for *Envision* we strive to maintain a positive answer to these questions:

- Would we want to use this system every day for our coding needs?
- Can we efficiently build different kinds of software with it?

- Could we use it to create large-scale software products such as an operating system?
- Does the system feel familiar and intuitive for experienced programmers?
- Is it fun and easy to work with?

Our hypothesis is that the combination of these features in an integrated system is very powerful. The work in this feasibility study represents the first step towards testing our designs. Once the system is more mature we plan to conduct quantitative studies to objectively evaluate the fulfillment of the above criteria.

It is important to note that at the moment *Envision* is still in design phase and only a few features were implemented as part of this study. These are discussed next.

3.2 Source code representation

Despite the large number of visualization techniques developed by researches there is a lack of quantitative experiments that allow us to set rules for what are good software visualizations and what are not[18]. A few works exist that try to establish some guiding principles for creating visual languages. Here we will briefly discuss two that have helped guide our effort.

Green and Petre developed the classical Cognitive Dimensions framework[19]. It provides researches with a means to evaluate their designs. Originally the framework was applied to one textual and two visual programming languages. In our work we have applied Green and Petre's five conclusions regarding future work in VPL design:

1. *Reduced syntactic load and higher-level operators.* Building a program visually reduces the need to remember and use precise syntax rules common for textual languages. Visual source-code representations allow for higher-level operations to be expressed concisely. These features increase the concentration of the programmer.
2. *Improved secondary notation.* The capabilities of modern graphics hardware have come a long way since the Cognitive Dimensions framework was conceived. *Envision* uses a broad spectrum of visual capabilities in order to improve secondary notation in programs. This includes colors, shapes, shadows, translucency, visual effects, visual transformations, icons, animations, grouping, modularizing and others.
3. *Familiar control-flow representation.* In *Envision* control flow statements have received a visual overhaul but remain structurally close to what programmers are used to in traditional languages.
4. *Low viscosity.* Unlike many previous efforts to design a VPL, an explicit goal for *Envision* is to be at least as productive as text-based environments with respect to writing or modifying code. We hope to achieve this by relying heavily on keyboard based input for the IDE. The input mechanisms of *Envision* are discussed in detail in section 3.3.
5. *Better utilization of screen real-estate.* Although it is quite a different concept *Envision* resembles CodeBubbles[15, 16] when it comes to how the screen can be utilized to show software fragments. CodeBubbles has been shown to increase the number of function code visible on the screen by about 50% in the typical case. We hope to achieve similar results.

More recently, Petre has analyzed four previous studies on the mental imagery that expert programmers use[20]. She also summarizes lessons learned about characteristics of good visualizations. Her work has helped shape the guiding principles(3.2.1) of the visualization component in *Envision*.

3.2.1 Visualization principles of *Envision*

Here we list the major design principles that have shaped the initial software representations of *Envision*. We also provide the rationale behind them.

Software visualization in *Envision* should be ...

General-purpose *Envision* is meant to be employed in a wide variety of domains. The visualization component should match this goal.

- Many domain-specific VPLs exist and some also enjoy success in industrial settings[21, 22, 23]. However, for general-purpose applications, purely textual languages like Java, C#, C, C++, Visual Basic, PHP, Python and their text-based IDEs dominate the market. We want to promote visual programming in this segment.

Efficient The system should allow the programmer to be at least as effective in constructing software as text-based IDEs.

- There is little incentive to switch to a new type of IDE if it does not provide tangible benefits. We feel that past efforts to introduce visual programming have often failed to deliver enough with respect to programmer productivity. One of the core reasons for developing *Envision* is to offer a platform that ultimately improves the programming experience.

Fast The system should be highly responsive to the user.

- Computing a complicated visual scene has the potential to reduce the computer's responsiveness to user input. This is not acceptable for *Envision* which wants to compete with text-based IDEs where there is no such problem. The performance of modern graphics hardware should be utilized to deliver an optimal user experience.

Intuitive The programmer should find it easy to use our environment. Visualizations should look familiar and perform as expected.

- Two important aspects hinder the adoption of new tools - unfamiliarity and unconventional behavior. More often than not, past VPLs were not successful in penetrating the commercial market because they required completely new programming paradigms. Consequently any benefits they offered were outweighed by the complexity of learning how to use or interpret them. *Envision* is based on OOP - a programming paradigm that has stood the test of time and has proven to be exceptionally versatile. A programmer should be able to work with the system with a minimal introduction. Where appropriate *Envision* might provide hints to the user of how to use novel features.

Flexible The visuals in the system should be flexible to display as much detail or as higher-level of abstraction as needed. Their flexibility should at a minimum match the one of textual representations. It should also be possible to create new views and tools through plug-ins.

- In their every day job programmers face a variety of tasks. A good tool should be able to provide the necessary level of abstraction to quickly navigate and understand software. If a tool is too narrow in its selection of views or detail level it will, at best, only be used for specific purposes. This is unacceptable for a general-purpose IDE targeting large-scale development projects.

Focused Visual support should be focused on the task at hand and on the immediate needs of the programmer. Unnecessary tools or options should not be visible.

- In popular IDEs such as Eclipse or Microsoft Visual Studio there are a lot of different tools, buttons and menus visible by default. For the most time however a programmer does not need them. Showing those tools on the screen contributes little to the programming task at hand when they are not needed. Instead screen real-estate can be better utilized by displaying more of what the user is actually working on - be it a software fragment or a document.

Appropriate The representations of software entities should match the nature of the represented data.

- One of the major outcomes of Whitley's analysis of empirical studies about VPLs[18] is that visuals perform poorly when they are not suited to the entity they represent. It is vital to carefully choose what visualizations to use in which cases. Equally important is to be able to say no to a graphic when a few simple lines of text feel much more natural. *Envision* combines visuals with text in an effort to deliver an optimal experience for various programming elements.

Self-sufficient No other tool should be needed to design large-scale software. In particular no stand-alone textual editors should be used.

- *Envision* should provide enough support and levels of abstraction to cover the entire range of developer needs. Even accessing more complicated features such as memory alignment declarations or special CPU instructions should be achievable from within the IDE.

Varied The looks of the IDE should take advantage of modern hardware and offer a wide variety of visual cues and styles.

- Limited by hardware performance the VPLs of one or two decades ago could not look as appealing and polished as it is possible nowadays. It is essential to exploit the full capabilities of modern hardware to deliver a fully immersing programming experience. Secondary notations would also benefit from this.

Friendly *Envision* should be fun to use. It should just 'feel right'.

- If the programmer thinks a tool is hard to use he or she will find alternatives to replace it. This is especially true of a new IDE that someone is considering for future use. *Envision* should be characterized by a natural and inviting interface. Only then will it be considered for serious software development.

3.2.2 Entity visualizations

On the higher abstraction level the visualization of software in *Envision* is based on boxes or blocks. These are used to display entities such as applications, modules, classes and methods. More detailed levels of software fragments such as method statements or class fields are specified in textual format. Figure 1 shows a basic hello world application. The outer-most box is the application. It contains modules which are similar to packages in Java. In this case there is

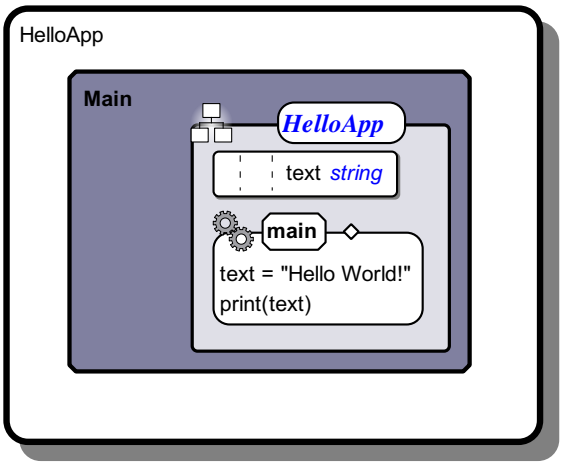


Figure 1: A Hello World application.

only one module - `Main`. The module's color and the position of the module boxes within the application is determined by the programmer.

A module contains classes. The Hello World application has only one class - `HelloApp`. Classes are indicated by an icon resembling a class hierarchy. The background color of each class is slightly transparent so that a user working on a bigger class can identify the module it belongs to. The user determines the position of a class within the enclosing module.

A class contains fields and methods. Figure 2 shows the representation of a simple circle class. The box in the upper left corner of the class shows defined fields. They are split into three categories in three columns - from left to right: public, protected and private. The `Circle` class has a boolean field `filled` which is public and three private fields: `x`, `y` and `radius`.

Methods within a class are also contained in boxes. These boxes have a white background to improve readability and can be manually placed anywhere in the class. The `Circle` class has two methods. A method is indicated by an icon depicting two gears and its name. Furthermore on the right of the name a box with the method arguments appears. There the type of an argument appears under its name. A method contains a sequence of statements. As we can see from the `area` method these statements are not limited to plain text but can include special symbols, graphics, formatting, effects, etc. Currently *Envision* only supports textual statements with a few exceptions (discussed below). We plan to extend this further. For example special symbols and

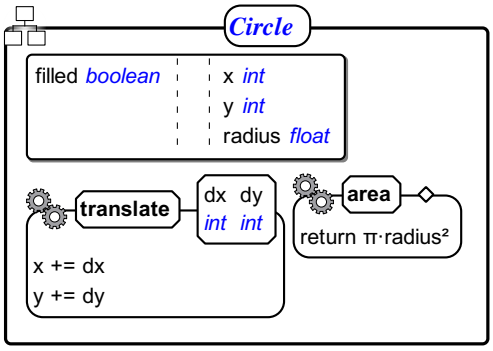


Figure 2: A simple Circle class.

graphics will be used to indicate assignments, function calls, recursive functions, matrices, return statements and others.

Control structures are a special case of statements. *Envision* currently has custom visualizations for if-else statements and loops. Figure 3 shows a factorial method that includes control statements. An 'if' control statement is indicated by a yellow diamond icon. Next to it is the condition. Below

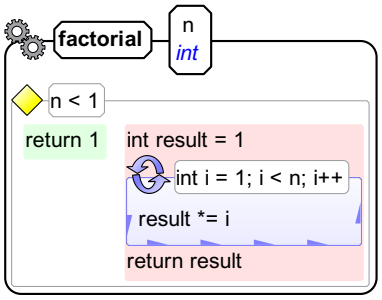


Figure 3: Control statements in a factorial function.

this horizontally arranged are the 'then' and 'else' branches indicated by light green and red background colors respectively. A 'loop' is indicated by an icon with rotating arrows and a special border. Next to the icon is the header defining the loop including terminating or continuing conditions.

As we've seen above visual abstractions exist for higher-level language features while specific details and execution statements are presented textually. However we find that adding additional visual effects or hints can be helpful even for text-based information. The reader might also notice that there are hardly any keywords visible. This is because the concepts that keywords embody are often better represented visually. For instance we can use color and/or position to indicate semantics. The structure of the 'if-else' statement and the fields of a class are good examples for this.

3.3 User interaction

In order to improve programming productivity and appeal more to expert programmers input in *Envision* is mostly from the keyboard. In the current implementation the mouse is only used to zoom in and out using the wheel and to move visual objects. The programmer can simply click anywhere on an object and drag it to a different position. One exception is clicking on a piece of text which displays the text editing cursor instead. Mimicking text-based IDEs all text visible on the screen is editable directly.

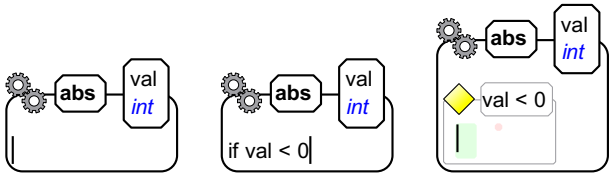
The two major features that define the feel of the user input system are discussed next.

3.3.1 Construction of visuals

Visual objects in *Envision* are not drawn or chosen from a palette. Instead they are constructed by typing. For example to create a new class the programmer would type 'class MyNewClass' and press Enter. This will create a new class within the currently selected module. The class name will be set to 'MyNewClass'. The class will be selected and ready to receive input. The programmer can then proceed to add methods or fields in a similar fashion using the keyboard. Whenever a new entity or statement is added its parent automatically expands to fully include it.

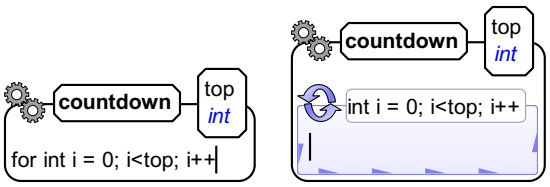
More complicated structures are also created from the keyboard. Typing 'if *condition*' will create

a new if statement with the specified condition. Figure 4 shows this process. Similarly typing 'for *header*' will create a new for loop with the specified header. This is illustrated in figure 5. Moreover it is possible to create statements based on the current context. For instance if the currently visible scope contains an integer variable named 'count' and the user types 'for count' the system would automatically generate a for loop that counts from 0 to count - 1. As a counting variable the first non-used of $i, j, k, l, m, ...$ will be used. This is demonstrated in figure 6.



(a) Waiting for input (b) Typing a creation command (c) An if statement is created

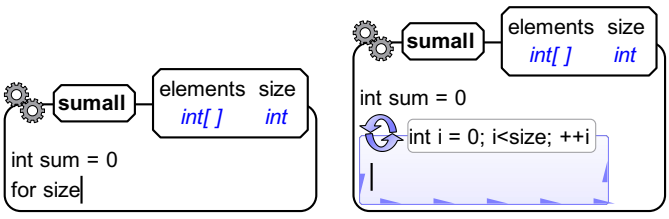
Figure 4: Creating an if statement.



(a) For loop command (b) Resulting statement

Figure 5: Creating a for loop by specifying the complete header.

The concept behind this mechanism is the separation between what the user types and what the application fragments look like. In traditional IDEs and programming languages the programmer directly writes the final structure of the software. This has the drawback that a specific grammar must be learned and used. Such a grammar typically contains many syntactical rules, symbols, delimiters, etc. This is not very convenient but is necessary in order for the parser to be able to understand the language and in order for the language to stay compact. In *Envision* syntax is kept to a minimum. The programmer expresses her desires by commands which the IDE interprets. It is responsible for figuring out what exactly the user means and for creating the corresponding



(a) For loop command (b) Resulting statement

Figure 6: Creating a for loop by specifying just the counting variable.

software fragment. This reduces the complexity of what needs to be typed especially with respect to syntax. At the same time concreteness is maintained - the visual result has a precise meaning.

3.3.2 Giving commands

By right-clicking anywhere on the screen the user shows a command prompt. It is used to quickly perform a variety of tasks which are dependent on the context:

- Create new software entities such as modules, classes, methods, fields, etc.
- Invoke IDE functions such as searching, opening views or projects and others.
- Execute system commands like copying files or playing the next song in your audio player.
- Perform user-defined actions.

The context of a prompt depends on where the programmer has clicked. After the user types a command it will be passed to the entity defining the context. This could be for example a method or a class. If the current context can not understand the command it is propagated to its parent. This process continues until some entity knows how to handle the command. If there is no such entity an error message appears directly under the prompt.

Figure 7 shows how to create a new class. The use has right-clicked on the translate method. A method does not understand the 'class' command and will pass it to its parent - a class. A class also does not understand this command and will propagate this to its enclosing module. The module interprets the class command by creating a new class with the specified name.

If the context was the Application object itself, the command would fail since an Application does not know the 'class' command. This is depicted in figure 8. Providing a wrong command would also result in a similar error message.

This prompt provides great flexibility at the programmer's fingertips. Common IDE functions can be invoked directly without opening menus or dialogs. Entire software fragments can easily be built by typing a single line.

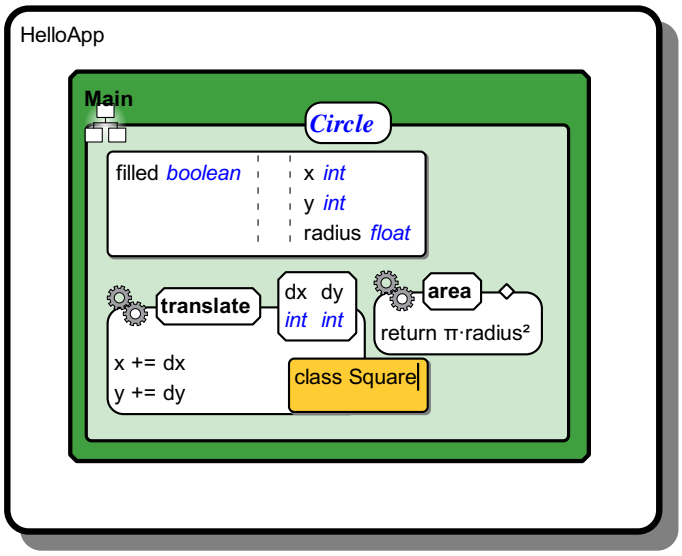
3.4 Use case examples

3.4.1 Programming a Hello World application

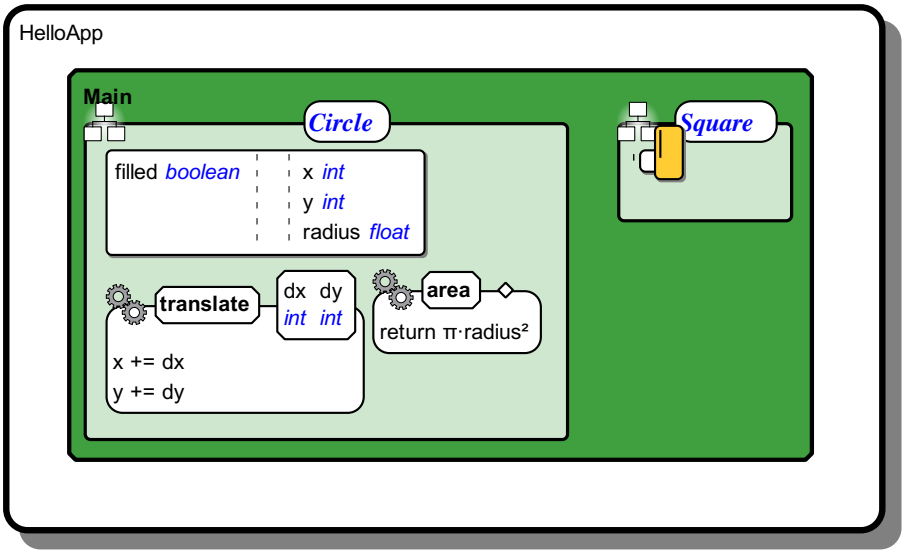
Here we will see how to create a Hello World application step-by-step. After starting *Envision* an empty canvas appears with a prompt in the middle. Type the following commands:

1. `app HelloWorld` - creates a new application called *HelloWorld*.
2. `module Main` - creates the *Main* module within the application.
3. `class Hello` - creates a new class within the module.
4. `method main` - creates the main method of the class.
5. `print "Hello World!"` - the only statement of our program.

Notice how this does not require typing any special punctuation symbols or caring about format and syntax. This process is illustrated in figure 9.



(a) A prompt



(b) The new class

Figure 7: Creating a new class by using a prompt. The context here is the `translate` method.

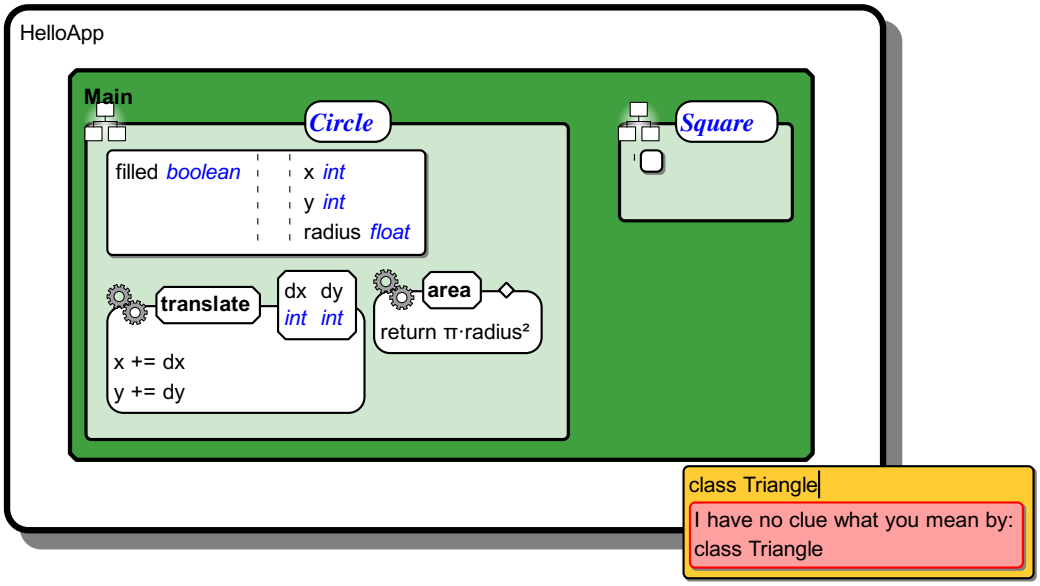


Figure 8: An error displayed at the prompt. The context here is `HelloApp` which does not understand the 'class' command.

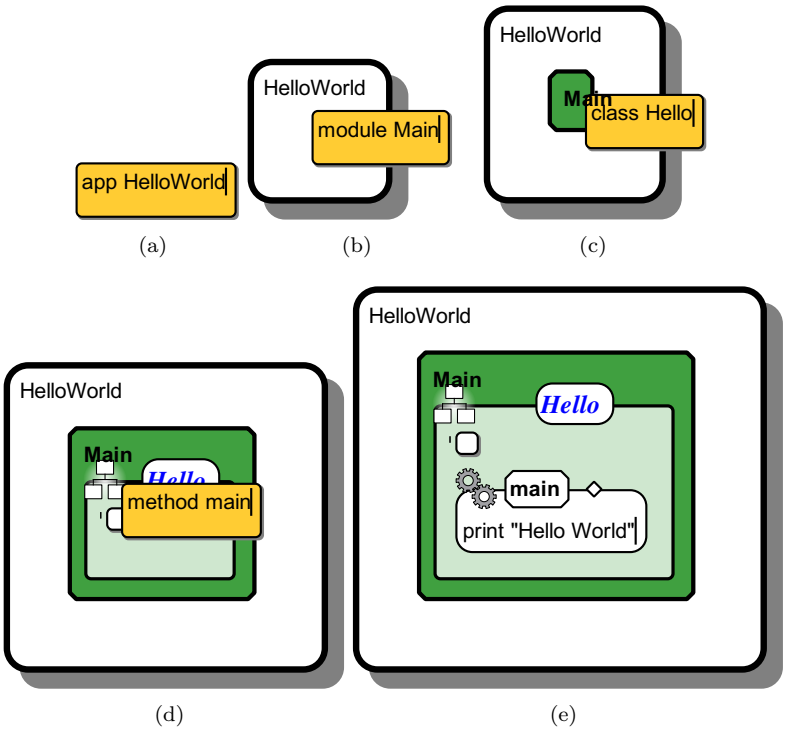


Figure 9: Creating a Hello World application.

3.4.2 Visualizing the source code of *Envision* itself

Envision is currently developed in C++. To get a better idea of how our approach scales to bigger applications we included a rudimentary C++ parser that was able to import *Envision*'s source code. Figure 10 shows the result.

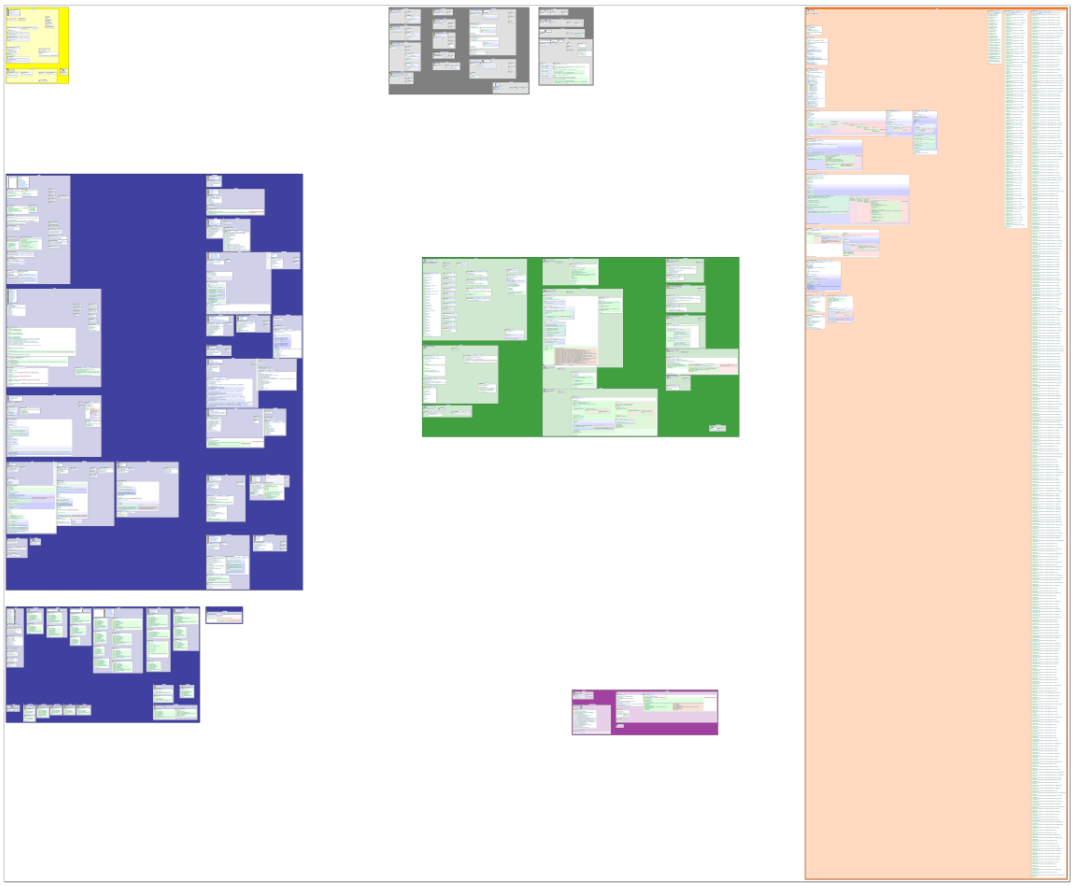


Figure 10: *Envision* visualizing itself. The canvas is zoomed out so that the entire Application is visible. A raster image was used here, since the inclusion of the vector version would increase this document's size by 100MB.

In the figure we can see in darker colors the different modules comprising the system. They are grouped with respect to color and position based on function. Blue modules pertain to visualization, gray to the program model, green to input handling, orange to C++ parsing, etc. Here we can see how *Envision* adds a new dimension to the Application. It allows the programmer to glance over the entire software piece and explore visually the relation between the different high-level modules.

Zooming in on the 'constructs' module we can see the different classes that belong to it in Figure 11.

Classes here are also arranged in a special way: from left to right and from top to bottom they start with the top-most construct types such as Application and Module and go to more detailed types such as Statement or Loop. Similarly the programmer can arrange the methods within a class. In all classes in the *constructs* module the following convention applies: constructors and destructors appear immediately below the field declarations; core logic methods appear under the

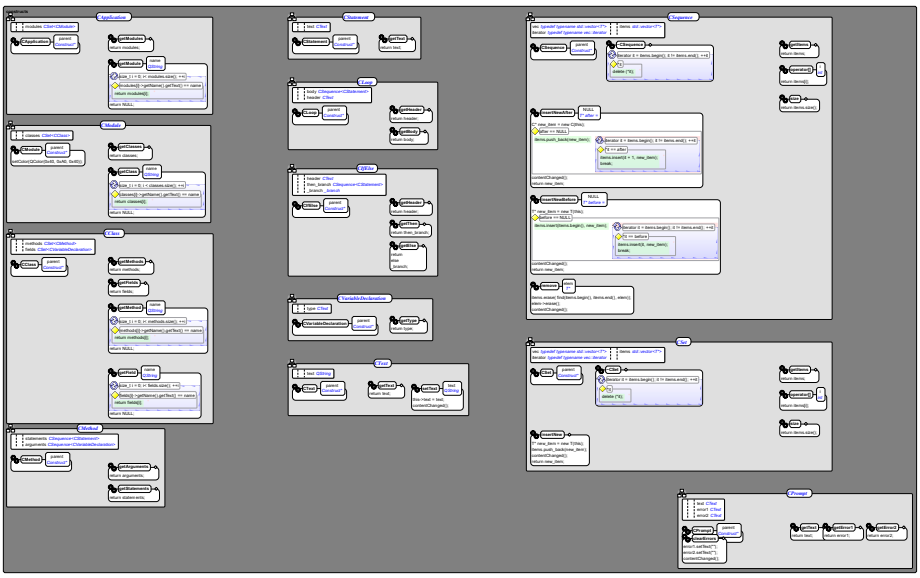


Figure 11: The *constructs* module of *Envision*. Classes here implement the logical model of programming entities such as classes, methods, statements etc. Zoom-in to see more detail.

constructors; getters and setters appear to the right; auxiliary methods, if any, appear in the lower right corner.

We can see how one can get an overview of an entire module or even the entire application by zooming out. Navigating around the application space is currently achieved by zooming in and out, using the scroll bars or using the arrow keys on the keyboard.

In our experience of visualizing *Envision*'s source code, one of the most useful features was to see methods or classes side by side.

3.5 Case study: call stack visualization

This style of software representation could offer interesting features in terms or error reporting and debugging. We explored a sample call stack visualization presented in figure 12. The methods from the stack are arranged from left to right in the order in which they were called. The highlighted line(orange) in each method is where the call to the next one occurs. Vertically the methods are aligned to this line to make it easier to trace. Information about the run-time argument values appears above each method. A useful feature here is the juxtaposition of all relevant methods on one screen. This eliminates the need to switch between different files or classes as is the case in modern IDEs.

4 Discussion

As part of this feasibility study we developed a mock-up version of *Envision* which was presented in the previous section. It focuses only on the core concepts of visualization and user input. Here we discuss our experiences with the application and comment on how it compares to traditional IDEs.

It is important to note that the mock-up *Envision* application can not be used to develop soft-

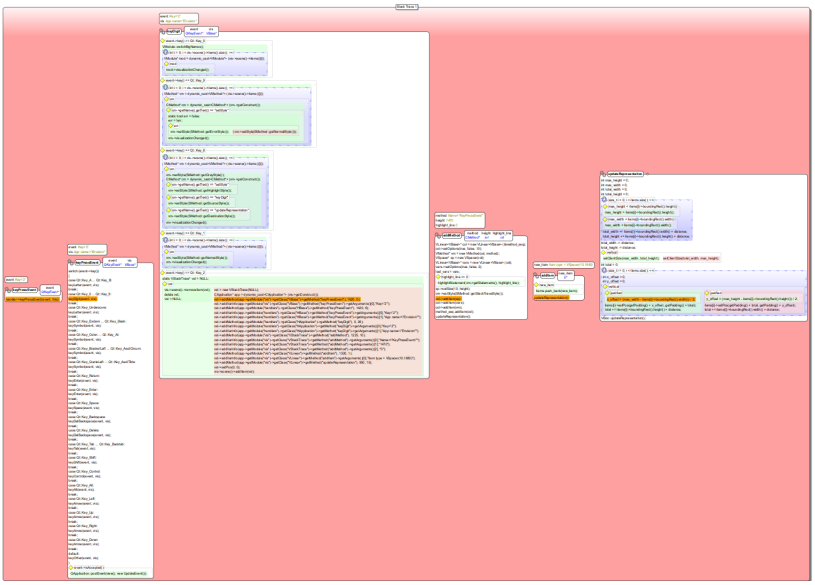


Figure 12: A mock-up stack-trace from the execution of *Envision*. A vector graphic could not be produced.

ware. Therefore all the comparisons in this section are based on preliminary results and projected features. This introspective evaluation is only an initial step. Proper quantitative studies are required to objectively asses the performance of our approach.

4.1 Implementation of the visualization principles

First we reflect on how the current concept of *Envision* implements the design principles outlined in section 3.2.

4.1.1 General-purpose

Envision is similar to a wrapper around an OOP language like C++, Java or C#. It retains the same object model and relationship between entities. It also inherits the general-purpose nature of those languages.

4.1.2 Efficient

We believe the proposed input methods which are a core part of *Envision* have the potential to be even more efficient that textual programming. This is evidently the case with the simple Hello World example presented earlier. To measure the efficiency of our approach on larger real-world programs, it would be first necessary to implement more features.

4.1.3 Fast

The current mock-up application is performing snappily even at more demanding tasks such as rendering of the entire *Envision* code base. Moreover this is done entirely in software using a single thread on a mid-range laptop. Therefore at its current stage of development *Envision*

meets the performance goal. To accommodate for future expansions we plan to employ rendering on dedicated graphics hardware.

4.1.4 Intuitive

Our system is based on OOP. So far the main visualization techniques are encapsulating entities into boxes and showing a tree hierarchy of the different entities. The former is natural when it comes to objects and data structures - enclosing them with a frame makes them more concrete and tangible. The latter is inherent in the structure of software written in the OOP paradigm. We think this contributes to the intuitiveness of *Envision*.

4.1.5 Flexible

We have shown that *Envision* is capable of covering the full spectrum of abstractions for a software application. It can show the entire application at once, outlining the relationship between different modules. Using the zoom feature it is possible to focus on a single module, class or method. At the method level one can directly observe the execution instructions of the program. This should provide enough flexibility for practically any task.

4.1.6 Focused

Envision's interface is minimal, there are no menus, toolbars or multiple visible tools. Only source code fragments are shown and arranged in the way the programmer specifies. Common tasks can be performed directly using the prompt. We find this more efficient both in terms of speed and utilizing screen real estate.

4.1.7 Appropriate

We do not limit our approach to pure pictorial visualizations. The integration of text and visuals in *Envision* aims to match a concept with a natural representation. Our system is flexible and can provide for many different types of visualizations. A more objective evaluation and long-term experimentation is needed to determine which visualization styles are suited for what software fragments.

4.1.8 Self-sufficient

The current mock-up application just showcases some interesting aspects of the *Envision* concept. Much more work is needed before the application can be used to design programs. This is discussed further in section 5.

4.1.9 Varied

Presently *Envision* is rendered entirely using zoom-able vector graphics in full 32-bit color. Various effects such as anti-aliasing, font smoothing and translucency are also used. Our visualization styles include different shapes, colors and borders and as representation entities we use boxes, gradients, symbols, icons and text. Semantics are described by both the look of an object as well as by its position. As one can see we are already utilizing a wide variety of visual objects and plan to expand this further.

4.1.10 Friendly

Our experience with *Envision*'s current interface is positive. However there are many planned features that we need to implement before the application is functional. Since friendliness is a very subjective quality, only a large-scale experiment with programmers can help make conclusions.

4.2 Comparison to traditional programming environments

Here we evaluate how *Envision*'s approach compares to one of the most popular IDEs - Eclipse. Eclipse is a mature software development platform with a complete set of features and numerous extensions contributed by third parties. It is actively supported by both industrial organizations and by the open source community. Therefore a direct comparison between *Envision* and Eclipse at this time is unrealistic. Rather we will limit ourselves to the code editing and navigating functions. Furthermore we will only consider the standard Eclipse installation that does not include any third-party plug-ins.

4.2.1 Productivity

Productivity here refers to the speed at which one can write code.

In Eclipse one has to type the entire structure of their program manually. A few exceptions are the use of the code completion feature and code wizards for e.g. new classes.

In *Envision* the programmer creates structure by issuing commands. These commands are context sensitive and have a light syntax. In essence what the programmer types is not directly the software structure but is first interpreted by the environment based on the context to determine what programming fragment should be created. This concept is similar to sloppy programming [24, 17, 25]. This separation can be very powerful. For example when typing, the user can use a compact syntax similar to scripting or functional languages, while the result can be a standard OOP structure for languages like Java. We plan to enhance the command typing mechanism to aid the programmer in a way similar to code completion. This will further make it easier to quickly create complicated program structures.

We feel that *Envision* will enable higher productivity compared to Eclipse as the input mechanisms we propose become more mature.

4.2.2 Readability

Eclipse provides a number of readability improvements over plain text file editors. These include syntax coloring, code elision and automatic formatting. The underlying representation however, remains purely textual.

Envision on the other hand shows the entire program structure in one consistent tree hierarchy. It produces images which have a minimal number of distracting features. Common semantics are represented visually rather than via syntax and keywords. Only the core logic of the application appears in textual form. There is no need for formatting since code structure is automatically and explicitly visualized by the system.

As a result, from a cognitive point of view, we believe our approach will improve readability. Graphically expressed features will be processed by the visual cognitive system. This is a very quick parallel process that is largely unconscious. Thus more capacity is left for deeper cognitive processes which are needed to process application-specific logic expressed as text.

4.2.3 Navigation

In Eclipse one navigates between different code fragments by switching to different tabs. Optionally one can search for a specific keyword, entity definition or object reference. It is also possible to click on an identifier in the editor window and navigate to its definition or uses.

These mechanisms are extremely useful when the programmer is looking for specific information. Therefore we are planning to include such features.

Additionally *Envision* allows the programmer to navigate code by looking at the software structure. Zooming out, one can see how different methods, classes and modules relate to each other. This is very useful when exploring the software for the first time. We believe that developers can use this feature to get more familiar with an unknown application faster.

4.2.4 Maintainability (Viscosity)

Maintainability here refers directly to one of the more severe problems identified by Green and Petre in [19] - resistance to local change. In their original analysis they concluded that visual programming languages reduce the user's performance when local modifications are necessary to existing programs. They found that such tasks are done several times faster in traditional textual IDEs.

On this point a textual IDE such as Eclipse facilitates high performance.

Envision aims to achieve similar results. All textual fields visible on the screen can be directly modified. Since core logic is expressed as textual instructions they should allow for equally fast changes. One problem we experienced with the mock-up application is that each time there was a change to a program fragment we needed to manually adjust the positioning of surrounding entities. This is currently a hindrance in achieving low viscosity and we plan to include automatic positioning features to alleviate this problem.

4.2.5 Teaching

While Eclipse is an extremely powerful platform for software development, it has little to offer to novice programmers. We have experience in courses where non-programmers were learning to program in Java using Eclipse. Mistakes often made by beginners include misplacing of semicolons, confusing the scope of an identifier, using the wrong kind of parenthesis or brace and bad code formatting.

Since *Envision* takes care of structure internally all of the above problems disappear. For small classroom examples *Envision* can be especially useful since the entire application can be visualized on a single screen.

5 Future work

Envision is still a work in progress. Many core concepts are planned for future development and the ones discussed here still need additional improvements. Here is a summary of possible future work:

- **Quantitative analysis** - Although this study serves to give an initial impression of the *Envision* system, it needs to be objectively validated. More information is needed on how expert programmers will interact with the system. Quantitative studies with developers must be carried out.

- **Feature-full prototype** - A feature-full prototype can be designed to enable better system evaluation and experimentation. Such a prototype would also be needed to support studies with programmers.
- **Concept refinement** - The visualization and interaction concepts need further improvement. In particular the input system requires a more complete implementation that will provide more insight about its strengths and weaknesses.
- **Error reporting** - we have barely scratched the surface of how *Envision* can benefit error reporting and debugging. More knowledge is needed about how our visual approach can improve existing practices.
- **Information system implementation** - We have not yet investigated the information system side of *Envision*. This track of research and development also presents many challenges: How will semantic versioning work? What content can be linked and how? Which analysis techniques can we combine to improve the programming experience? How do we present the results of this analysis?

6 Conclusion

In this study we have presented *Envision* - a combination of an IDE for visual programming and an information system. Our contributions include developing the concept of *Envision* and its design principles, implementing a mock-up application to demonstrate its core ideas and comparing its performance to a modern text-based IDE - Eclipse. The evaluation of our approach carried out during this feasibility study is largely positive with some inconclusive results. There is a lot of future work needed to further develop the concept of *Envision* and validate its applicability. We strive to promote visual programming in large scale software development projects. The ultimate goal of *Envision* is to allow the production of higher-quality software faster.

References

[1] R.N. Shepard. Recognition memory for words, sentences, and pictures1. *Journal of Verbal Learning and Verbal Behavior*, 6(1):156–163, 1967.

[2] L. Standing. Learning 10000 pictures. *The Quarterly Journal of Experimental Psychology*, 25(2):207–222, 1973.

[3] S. Schiffer and J.H. Fröhlich. Visual programming and software engineering with Vista. *Visual object-oriented programming: concepts and environments*, pages 199–227, 1995.

[4] B.W. Chang, D. Ungar, and R.B. Smith. Getting close to objects: Object-focused programming environments. *Visual object-oriented programming: concepts and environments*, pages 185–198, 1995.

[5] W. Citrin, M. Doherty, and B. Zorn. The design of a completely visual object-oriented programming language. *Visual Object-Oriented Programming: Concepts and Environments. Prentice-Hall, New York*, 1995.

[6] PT Cox, FR Giles, and T. Pietrzykowski. Prograph. *Visual Object-Oriented Programming: Concepts and Environments. Prentice-Hall, New York*, 1995.

[7] C. Grant. The Visula programming language and environment. In *IEEE Symposium on Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006*, pages 203–206, 2006.

[8] E. Baroth and C. Hartsough. Visual programming in the real world. In *Visual object-oriented programming*, page 42. Manning Publications Co., 1995.

[9] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 183–192. ACM, 2005.

[10] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. 2005.

[11] M. Cherubini, G. Venolia, R. DeLine, and A.J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 566. ACM, 2007.

[12] R. DeLine. Staying oriented with software terrain maps. In *International Conference on Distributed Multimedia Systems*. Citeseer.

[13] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson. Code thumbnails: Using spatial memory to navigate source code. 2006.

[14] M.A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen. SHriMP views: an interactive environment for information visualization and navigation. In *CHI'02 extended abstracts on Human factors in computing systems*, page 521. ACM, 2002.

[15] 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. In *Proceedings of the 28th international conference on Human factors in computing systems*, pages 2503–2512. ACM, 2010.

[16] A. Bragdon, S.P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J.J. LaViola Jr. Code Bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 455–464. ACM, 2010.

[17] G. Little and R.C. Miller. Keyword programming in java. *Automated Software Engineering*, 16(1):37–71, 2009.

[18] K.N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8(1):109–142, 1997.

[19] T.R.G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[20] M. Petre. Mental imagery and software visualization in high-performance software development teams. *Journal of Visual Languages & Computing*, 2010.

[21] K.N. Whitley and A.F. Blackwell. Visual programming in the wild: A survey of LabVIEW programmers. *Journal of Visual Languages & Computing*, 12(4):435–472, 2001.

[22] A. Cimino, F. Longo, and G. Mirabelli. A General Simulation Framework for Supply Chain Modeling: State of the Art and Case Study. *Arxiv preprint arXiv:1004.3271*, 2010.

[23] S. Bangsow. *Manufacturing Simulation with Plant Simulation and Simtalk: Usage and Programming with Examples and Solutions*. Springer Verlag, 2010.

[24] G. Little and R.C. Miller. Translating keyword commands into executable code. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*, page 144. ACM, 2006.

[25] G. Little, T.A. Lau, A. Cypher, J. Lin, E.M. Haber, and E. Kandogan. Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 946. ACM, 2007.