Task-specific views in Envision

Research in Computer Science Project

Cyril Steimer

csteimer@student.ethz.ch

Supervisors: Dimitar Asenov, Prof. Dr. Peter Müller
Chair of Programming Methodology
ETH Zürich

September 22, 2015

Contents

1. Introduction

Envision [1] is a visual programming tool, where a program's structure is presented using a combination of text and graphical objects. Most of today's mainstream programming interfaces work on purely textual and file based editors, often making quick navigation or comprehension of programs cumbersome. By using graphical constructs and arranging the entire program on a two-dimensional plane, Envision aims to achieve the facilitation of both quick navigation and comprehension of programs.

A drawback when using Envision is the lack of being able to adapt the content on screen to a task being worked on by the user. There is only one visualization mode, where the entire project is being displayed on screen. We will call this the *architecture view*. Within this project, we create so-called *task views*, which allow both the content which is visualized and its arrangement to be adapted to the task being solved. For example, when debugging a method, one is typically interested in seeing the method itself, its callees and its callers grouped together to efficiently analyze the method itself and call paths leading towards or from it.

Generally, we want an efficient method of visualizing the *working set* of a task to the user. This is important for keeping a developer focused on a task and in the flow. Using the existing architecture view, especially for large projects, it is easy to get lost in parts of the program which are unimportant to the task at hand. Furthermore, apart from the collection of items in the working set being important, also their arrangement needs to be considered as well as the additional information which is available. Research has also shown that depending on the task, different visualizations can be beneficial [9].

We use related work for adapting the content on screen to the task at hand as inspiration for our solution, with section 2 containing a discussion of this. In section 3 we present our design concepts which were originally created and compare them to the final design explaining the reasons for the existing differences. Section 4 highlights certain aspects of the implementation itself, and in section 5 we compare our solution to other, existing tools, highlighting the differences and discussing the perceived benefits and drawbacks of the *task views*. Lastly, section 6 provides an outlook to future work which can be done to extend or improve the presented solution, finishing with a conclusion on what was achieved.

2. Related Work

Code Bubbles
Code Bubbles [2] uses bubbles to arrange the necessary items for a task on screen. A bubble is a resizable rectangle with a border, which can contain code, but also other things, such as simple notes. They offer the programmer a high degree of freedom, as the bubbles can be arranged in any way. Later research [5] has however shown that too much arrangement flexibility can negatively influence navigational performance. By offering a large 2D plane to arrange the bubbles on, a high amount of space is offered and different tasks can be split by laying out their necessary content on different sections of that plane.

Patchworks
The Patchworks code editor [5] takes a different approach from code bubbles by specifically limiting the amount of freedom given to the programmer. While different code fragments are displayed on the screen at once, the number of such fragments is limited to six, arranged in a three-by-two grid. This grid can be scrolled both to the left and the right on an essentially limitless one-dimensional ribbon, allowing many code fragments to be open, but not appearing on screen. The results presented in the paper show that while this limits programmer freedom, it makes navigation less error-prone and reduces the time spent rearranging the content on screen when compared to Code Bubbles.

Code Canvas
Code Canvas [3] follows a similar approach to Envision, where the entire project is displayed on a 2D-plane. To vary the information which is shown on screen and make it more flexible for the user, code canvas uses a layer system where different information layers can be turned on or off. In contrast to those tools however, it does not have a strong notion of a working set, as the entire project is displayed on the plane at all times. Therefore, while its layered information concept can be used to easily switch between the information shown on screen, it suffers from similar problems as Envision.

Stacksplorer and Blaze
One of the most important pieces of information in programming is the call graph of a method, for which Blaze [8] and Stacksplorer [6] are two tools for visualizing it. As exploring the call graph of a method is an important part of many tasks, we consider efficient exploration an important part of our work. The benefits and drawbacks of these two tools are analyzed in [7]. In Blaze, one possible path through the entire call stack is shown with ways to switch between paths, while Stacksplorer instead shows the immediate neighborhood of a method.

3. Task-views concept for Envision

We created several design concepts for different parts of the system. In this section, we look at these concepts in detail, comparing them to each other and explaining our decision behind using or not using a concept.

3.1. Task view

Initially, we discuss what a *task view* is. A *task view* is essentially a collection of different information items, whereby such an information item can be something as simple as a code fragment or also further information as described later in this section. Whenever a programmer needs to perform some specific task, they can then create a new *task view* and populate it with the information items which are necessary for that task, which is essentially the working set of the programmer.

Often, a programmer will also want to have multiple views at the same time. There are different examples for when multiple views could be needed: The programmer could be interrupted from their current task and switch to something else for a while, or simply reach some sort of barrier within their task, where they want to work on something different without forgetting their other task. However, we believe the number of *task views* simultaneously open still to be low, as concentrating on many different tasks is difficult.

We decided that each *task view* should have its own isolated space. Therefore there is no connection between different *task views*. This approach seemed the best to us as it would allow the programmer to have a clear distinction between their different tasks.

3.2. Switching between views

As it was clear to us that each *task view* would be in its own separate space, there needs to be an efficient way of switching between them.

Tab-based approach
One approach is inspired by existing traditional IDEs such as Eclipse or Visual Studio. There, open files are arranged in tabs between which the user can switch. Therefore, the first concept was to use a tab-based approach for switching between the views. There are certainly some benefits to such a solution. For one, it is well-known as it is widely used in other applications, which would make it feel familiar to users as they have very likely used at least one tool using such an approach. Furthermore, it would allow for a persistent indication of the existing views, which makes it easier for the user to have an overview of the current views.

However, we also expect the number of views which are open simultaneously to be small, as usually only a limited amount of tasks is being worked on at once. Therefore,

we decided to use another solution which makes use of this assumption.

Pie cursor inspired menu
This concept is inspired by the PieCursor [4] approach. There, instead of having a normal mouse cursor, the user has a circular pie cursor separated into wedges, where each wedge represents an action that can be performed. This means that only little mouse movement is required to select an action. An additional benefit lies in the stable spatial position of each wedge, this allows users to rely on their spatial memory to perform actions without even needing to look at the wedges of the cursor.

We modified this solution based on two main considerations:

- We want to have a sufficient indication of the represented view in the menu, such as the name. Therefore, we found a circular cursor to be impractical as its space usage is not as efficient.
- Another goal is to make the selection menu more keyboard friendly. Our menu can not only be controlled by the mouse, but is also fully usable with a keyboard. This means that for one, navigation between different menu entries using the arrow keys is supported, and each menu entry has its own keyboard shortcut for switching to the view it represents. This is inspired by computer strategy games such as StarCraft or Age of Empires, where there are for example such hotkeys for switching between groups of units.

The selection menu consists of a three-by-three grid of the views, where slots which are still empty are displayed differently from ones that represent a view. By renaming such an empty slot, a new view with that name will be created. When pressing the buttons to open the menu, it is automatically opened in a position centered around the mouse cursor, also allowing efficient mouse interaction. Figure 1 shows a screenshot of the selection menu.

We believe that this approach is more keyboard friendly than using a tab-based inter-

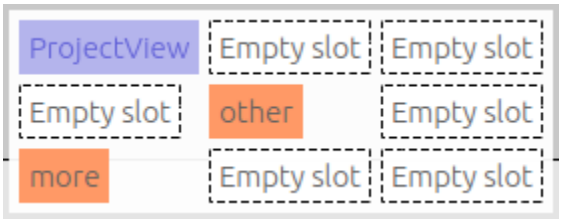

Figure 1: The menu used to switch between views. The orange highlighted entries correspond to already existing views, while the white ones are still empty. The currently selected entry is highlighted in blue.

face. Furthermore, for users who prefer using their mouse, it still requires very limited mouse movement as the menu is centered around the cursor. Spatial memory can also

improve performance of the user when switching tasks. They might even make it a habit to put the views into certain positions on the menu grid, connecting the different positions on the grid to their different *task views*. However, one negative point of the implementation is that we now miss a persistent visual indication of the existing views.

3.3. Populating views

It is important for the user to have simple and intuitive ways of populating the *task views*. For this, we look at two dimensions of such tools: First we look at context-aware and context-unaware methods of adding content. Then we contrast batch inserts against inserting a single item.

Context-aware vs. context-unaware
We look at two ways of adding content to the view. For one, we can do it in a context-aware manner: The user supplies some sort of context, based on which the content to add to the view is chosen. On the other hand, in a context-unaware way, no context is supplied to the system, Therefore the content to add is chosen from the entire project.

In Envision, we use context menus. These can be opened by selecting an item on screen and either pressing the ESC button on the keyboard or right-clicking on an item. To add content in a context-unaware manner follows the same process - however the selected item does not make a difference to the content which is added.

We decided to use the following approach: In the context menu, different methods of adding content are suggested. For example, if the context is a method, we might want to add all its callees to the view. Furthermore, commands can be entered in the context menu. We decided to use such a command - similar to the "quick open" feature in mainstream IDEs - to add content to the view in a context-unaware way, where the user supplies the full name of an item to add. We also decided to support abbreviations to the full name, to make it easier to use. For example, a method called `helloWorld` could be matched by any of the following searches: *hello*, *World* or even *lloWo*.

Batch vs. single item insertions
Independent of whether we are aware of the context when adding content, we can add one or multiple items at once. In a batch insertion, many items are added to the view in one step, while using single item insertions adds only one item to the view in any one step.
In our concept, we decided to use single item insertions for our context-unaware method of adding content. We think that this is more intuitive to the user. On the other hand, using context-aware methods it's often more intuitive to insert multiple items at once - for example when adding the callees of a method. Therefore, we decided to use batch insertions in the context-aware methods we aim to support.

Figure 2 shows ways of adding content both context-aware and unaware. The context-

unaware menu can be invoked from anywhere.

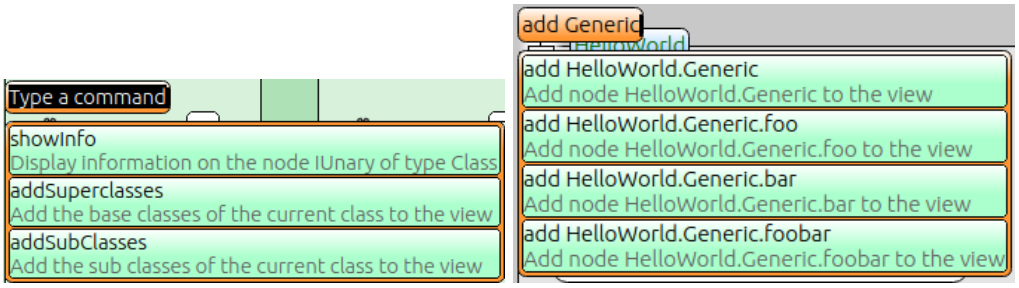

Figure 2: The left figure shows a context aware menu which appears when right-clicking on an item, specifically for a Class node here. The right figure shows the context unaware way of adding content, showing the suggestions.

3.4. Arrangement of items

An important part of the user experience is also how the items are arranged within the view. While a system such as Code Bubbles [2] allows the user full freedom in arranging their content, Patchworks [5] has shown that too much freedom can negatively influence the time taken to navigate to certain parts of the working set. The approach presented in our concept is inspired by Patchworks, while offering more flexibility in the number of items available on screen and their initial placement by using a flexible grid. The flexibility offered means that we do not set a fixed limit on the size of the grid which is visible. On the other hand, we avoid manual placement of bubbles as in Code Bubbles.

For one, we did not want to be as restrictive as Patchworks. The number of items visible on the screen at once should not be limited, but depend on both the zoom level and the size of each respective item. Patchworks does not have a zooming feature, therefore the number of items on screen can never change. We also decided not to use a grid with equally sized elements, as that would make displaying large and small elements at the same time inefficient.

The grid we use is essentially a list of columns. Each column of the grid has a fixed width, given by the widest element in that specific column. Within each column, items are then simply arranged one after another. Every item's height in the grid is given only by the height of the item itself (and not for example the highest item in a row).

This leaves us with one main issue: It is difficult to arrange items to the left or the right of an existing item, as the position of the rows is not fixed. For this, in our concept we introduce what we call "Adaptive Spacing": We can insert empty items into the grid, where their height automatically adapts to a certain piece of content around them. Furthermore, we can insert empty columns into the grid. These two functions allow us to for example display the callees of a method at the same height as the method itself.

Figure 3 shows an example of a few items arranged in a grid as explained above.

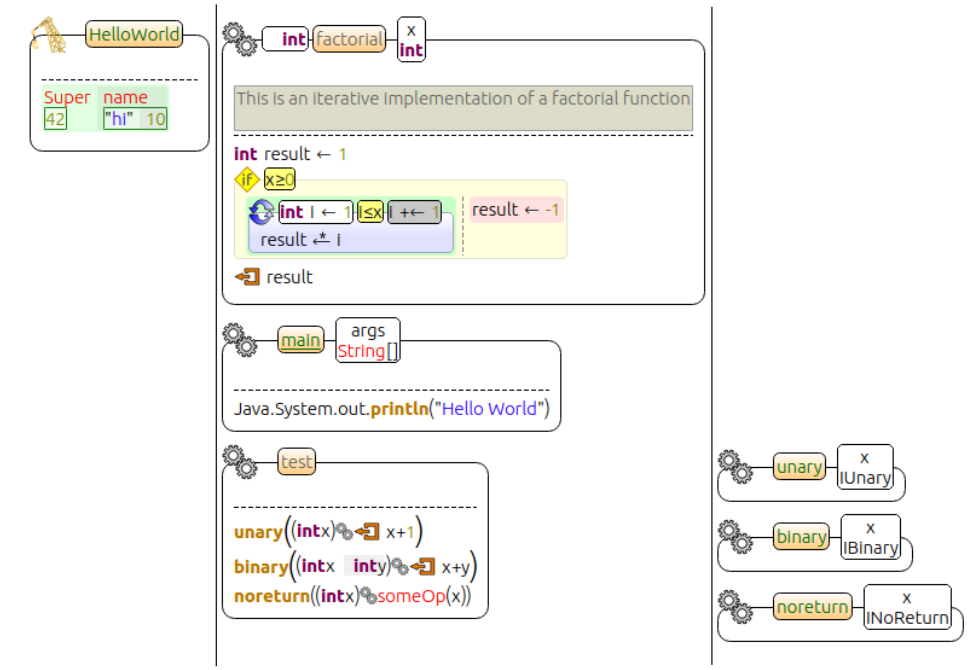

Figure 3: An example of the grid arrangement. Note that there is a spacing item, which ensures that the method `unary` is displayed at the same height as its caller, method `test`. The lines between the grid columns are to indicate the width of each column. Note that they are not drawn in Envision. We can see the width of a column being defined by its widest element.

When arranging content within the grid, we do not give complete freedom to the user. When using the context-unaware method of adding content to the view, we allow specifying the initial location of the new item within the grid, with the exact position then being automatically computed by the system. With the context-aware methods, we simply place the new items relatively to the existing item. We believe that this compromise for one gives the user a high enough amount of freedom, while still keeping them from spending time on unnecessarily rearranging content.

3.5. Different information types

In a *task view*, information can be displayed in different ways. For one, just as in Envision's original architecture view, we can visualize any AST node. In order to show only the necessary information to the user, we also wanted to ensure that nodes can be visualized using different levels of detail, for example showing only the public interface of a method and hide its body. This would allow the user to concentrate on what is

really important, and also reduce the amount of space needed by such a visualization, which allows displaying more content on screen at once.

In the concept, we also wanted to add more visual items to show information:

- Relations. For many tasks, it is important to see relations between different parts of the code, for example to visualize a relationship between a method and its callee.
- Documentation. We want to be able to show documentation on items, such as for example a method's header comment.
- Statistics. Many existing tools already support for example showing in how many places a method is called. Visual Studio for example has a persistent indication called CodeLens, while Eclipse shows it on request. We also want to be able to show such information.
- Tool information. One might also be interested in seeing information from tools displayed. For example, one could imagine that a code verifier could be executed on a part of the code, and the output should then be somehow visualized.
- Notes. The user should be able to write their own comments which do not essentially concern the code but rather the specific task into some special note node in the *task view*. This would make it easier to write down thoughts about the task and keep them stored together with the task. However, due to time constraints this was not implemented. Extending *task views* with such a system should however not present a real difficulty.

To visualize relations, we considered different approaches. One option was to use hyperlinks to show such relations, were they could be used to jump between parts of the code. While this would be efficient to display arbitrarily many relations, we decided not to use it as we believe it can get disorienting to the user when many context switches happen upon selecting these links.

We decided to instead use arrows for displaying relations. This allows an intuitive representation of relationships and also makes it easy to encode directionality. However, it is more difficult to make them scale for items which are far away from each other, as they can then occlude other content between the two items. However, we assume that users are often interested in keeping related items close to each other, at least somewhat mitigating that issue.

For documentation, statistics and tool information we decided to visualize them all in the same way. Therefore, we decided to use just a single visual item in which we could put all these types of information. Furthermore, we wanted to allow the user to adapt the information shown in such a visual item at will, turning certain layers on or off. These different information layers in our item are extendable at will. Essentially, we show a host of different, not necessarily related, information on a given node. The

available layers can be extended anywhere, meaning that both existing and future plug-ins for Envision can extend the functionality of that information item.

Figure 4 shows the different types of information that can be shown.

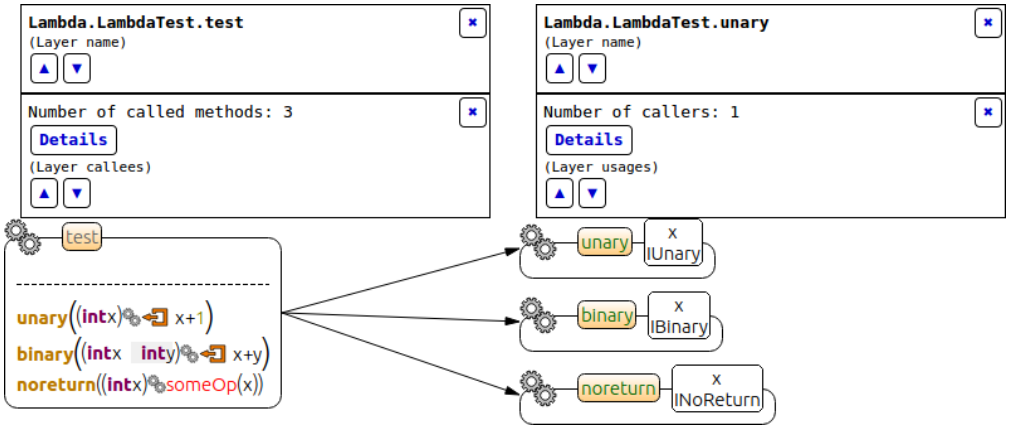

Figure 4: The different types of information that can be displayed. We have two information items, showing different content for the two methods. We also see arrows to indicate the connection between the method and its callees. These arrows exist on layers, and arrows can be turned on or off by the user in a by-layer basis.

3.6. Sharing of views

Lastly, we want to enable multiple people working on the same project to exchange their *task views*. For this, we designed a simple JSON scheme with which we can persist a *task view*'s content. As this JSON scheme does not need to keep track of actual AST nodes, but only references the ones which are used in the *task view*, it's rather simple, such that any user with knowledge of JSON could correctly use this scheme by hand. Using the persisted JSON files then enables users not only to reload their own *task views* from disk, but also allows them to send their *task views* to other users, perhaps to get some input from them if they are stuck on something. The only requirement for this to work is that all users which want to use this JSON file also have the actual project itself. An example JSON file for a small *task view* with two AST nodes and an arrow between them can be seen on the next page.

```
{
    "arrows": [
        {
            "layer": "callees",
            "node1": "{2323e58d-e864}",
            "node2": "{009e5ee1-4a55}",
            "parent1col": 1,
            "parent1row": 0,
            "parent2col": 2,
            "parent2row": 0
        }
    ],
    "disabledLayers": [
    ],
    "name": "viewName",
    "nodes": [
        {
            "col": 1,
            "purpose": -1,
            "reference": "{2323e58d-e864}",
            "row": 0,
            "type": "NODE"
        },
        {
            "col": 2,
            "purpose": 1,
            "reference": "{009e5ee1-4a55}",
            "row": 0,
            "type": "NODE"
        }
    ]
}
```

4. Implementation

In the implementation section, we take a rather high-level approach of discussing the implementation. We do not show explicit details of the code, but rather explain the methodologies used when implementing the *task views*. Some shortcomings also exist, which are explained in this section.

4.1. Changes to Envision

As some parts of the *task views* touched existing concepts of Envision, some of the core parts of the code had to be changed.

Enabling multiple views
It was not possible to have multiple views at the same time before. To enable this, the `Scene` no longer directly has ownership of all the items it contains but rather of the views themselves. These views themselves are also items and contain the nodes which make up the view. To switch between views, the old view is simply hidden within the scene and the new one shown. This meant also changes to further parts of Envision, since nodes can no longer be added directly to the scene but must be added to the views instead.

Correctly hiding overlays
Envision supports overlays, with which additional information can be displayed on the screen. For example, the arrows shown between items in the *task views* are also a type of overlay. Previously, since Envision allowed only one view, all overlays where visible by default. This meant that when switching between views, the associated overlays would still stay on the screen as they are independent from the views themselves. As associating each overlay explicitly with a view would have caused many changes, they are now simply automatically hidden as soon as any of their associated items are also hidden on screen. Therefore, as switching between views means hiding the items from the old view, the associated overlays are also hidden.

Changing commands
Envision had a strong existing command utility, which also used the auto-complete concept to offer a menu of possible actions to the user. In the *task views*, we often want to add nodes using a context-aware approach, where the commands take no arguments except for the context (which is already supplied by default). Therefore, we changed and extended the commands to allow some commands to be displayed even if no text has been entered to the command prompt - therefore essentially enabling right-click menus.

4.2. Extensions to Envision

Various extensions were made to Envision in course of the project. Here we list the main extensions that had to be made.

ViewItem.
The views are separated by making each of them their own item (which we call `ViewItem`), and adding them to the `Scene` as new top-level items. Each of these items then has its own dynamic grid, where nodes can be added and removed at will. Furthermore, these items also keep track of the arrows being displayed on screen. They are managed by a `ViewItemManager`, which is added to the scene to offer an additional level of indirection between the scene and the views themselves and keep the `Scene` class itself free from clutter.

Extending commands
We had to slightly change Envision's command system to enable some of the new commands. We created new commands using default arguments, where arguments which are left empty by the user are automatically filled in by the system. This also allows commands with arguments to be shown in a right-click menu. Furthermore, many new commands which are *task view* specific were introduced, such as the command to add nodes based on their fully qualified name or the commands which add content to the view based on their context. The new commands as well as the new shortcuts added to Envision in the course of the project can be seen in the Appendix A.

InfoNode
An important part of the work on *task views* is the new `InfoNode` class. Such an `InfoNode` can be applied to any existing AST node and displayed in a *task view*. The `InfoNode` class keeps a register of existing methods which deliver information for nodes, with a command to allow users to turn parts of the displayed information on or off. This also allows the `InfoNode`to be extended at will, as new information methods can be registered from anywhere inside the code. As such, future projects can extend the information available on nodes. For example, if such a project introduces a code verifier for Envision, it could also easily register a method to display that information within an `InfoNode`.

The `InfoNode` uses existing web technologies for its functionality. Using HTML allows us to easily change the design of certain information layers or the entire node without having to create new visualizations for it. On the other hand, using Javascript and linking it to our original Qt C++ code allows adding further functionality to the `InfoNode`, for example a mechanism of moving the screen to a callee of a method upon pressing a "Jump To" button.

JSON support
To persist views and enabling sharing of them, the *task views* need to support storing to and loading from JSON. The JSON scheme is quite simple, as storing AST nodes requires only storing their reference. In the JSON, we need to store information on the adapative spacing items, the AST nodes which are displayed, all the arrows which are shown or hidden and the content shown in the InfoNodes. Everything else is already

handled by Envision's existing functionality for persisting projects.

2D menu
To support easy and intuitive switching between views, a 2D menu which arranges the available views in a 3-by-3 grid was introduced. This menu is extensible and could be used for other purposes as well, as it allows arranging arbitrary menu items in a 2D grid. Particularly, the 2D grid can be transformed to a simple 1D menu by setting either the number of rows or columns to be one. Future work should be done to see whether this menu can be integrated with the current auto-complete menu in Envision, such that the two code bases can be unified.

4.3. Issues and limitations

Here, we list some of the known issues and limitations of our implementation.

JSON scheme
The store and load operations handling the JSON for persisting a *task view* are currently quite limited. While they easily allow extensions to the `InfoNode`, new AST nodes or also new arrow layers being added, the JSON scheme would require changes for anything further. For example, adding notes nodes to the views as discussed in section 3 which would not be persisted as part of the AST (and therefore couldn't be simply referenced) would require changes to both the save and load operations.

Limitations of arrows
Arrows can only be turned on or off in groups - single arrows cannot be toggled. Furthermore, it is currently not possible to add arrows manually - the only way to show arrows is to use the existing context-aware methods of adding content, such as adding all callees of a method. Adding arrows between two existing items however is not possible.

Limitations of adaptive spacing
Similar to arrows, spacing items are only added when context-aware methods are used. They cannot be added manually by the user. Furthermore, the spacing target of such an item is always automatically defined on creation and cannot be changed manually. As such it is not possible for the user to adapt the spacing to another item than is used by the code which originally adds the spacing item.

No recognition of existing relationships
When adding all nodes which are in some kind of relationship with an existing node, it is not checked whether these nodes perhaps already exist somewhere. For example, when adding all the callees of a method and these already exist somewhere in the view, the code does not realize this. While we often consider this not a limitation, but a design decision in the sense that it allows us to position the callees at a good position relative to the original method, it is a limitation in some cases. For example, one might imagine that the command is accidentally executed twice by the user, and now all callees (and

their corresponding arrows) are displayed twice in the two columns next to the original method. In such a case it would be better to recognize the existing relationship.

5. Discussion

In this section, we compare our solution to existing solutions which are also referenced in section 2. In a first part, we discuss how the tools compare with each other, and then summarize the discussion in a table.

5.1. Discussion of tools

To discuss the differences between our solution and four existing solutions, namely Code Bubbles [2], Patchworks [5], Code Canvas [3] and a mainstream IDE, Eclipse, we split the analysis of the differences into three parts: How can we support different tasks, how do we visualize the working set and which kind of information can our *task view* contain?

5.1.1. Tasks

For one, we can support different tasks by giving control over the content to the user. As in Code Bubbles, the user controls which and how many nodes they see on screen at once. What differentiates us from all other tools is that we allow nodes to be visualized in different ways. For example, a method can be displayed fully, including its body, or only with its signature. Likewise, a class may be displayed using only its public signature. Envision supports any type of visualization, this just being one example. This allows to keep unimportant information away from the view and not obstruct the user. The user has full control over how any node is visualized. Figure 5 shows a method visualized both fully and with only its public signature. Code Canvas is similar to Envision's *architecture view*, in that it doesn't support showing only a subset of the entire information. On the other hand, Eclipse supports showing such a subset, however unlike more visual programming tools, it does not support meaningful spatial arrangement of the content or other flexible visualizations, like Envision.

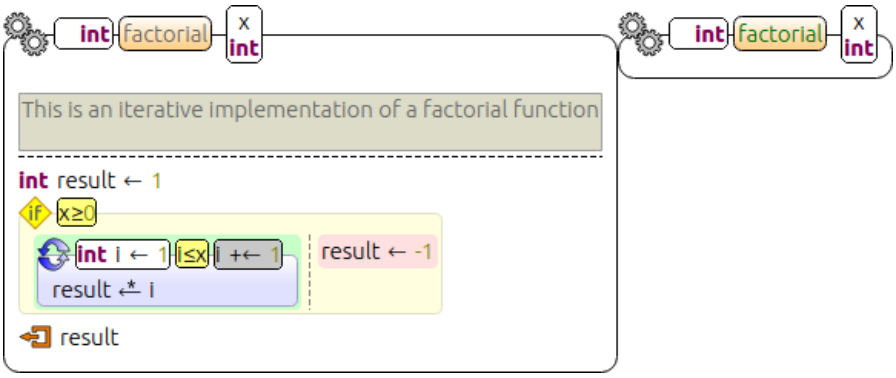

Figure 5: A method visualized in two different ways.

When it comes to supporting different tasks at once, we chose an approach that differs from both Code Bubbles and Patchworks. In both of them, it is possible to essentially arrange content with gaps, creating logical borders between tasks. In Code Bubbles it is

also possible to explicitly name those regions, giving better recognition. In our solution we explicitly keep the different *task views* separated, requiring an explicit switch. We believe this to be a better solution in some regards. Our three-by-three grid in the view selection menu furthermore offers stronger spatial meaning to the views, seeing as both Code Bubbles and Patchworks arrange their partitions on a continuous 1D ribbon (although Code Bubbles allows arranging content on a 2D space, their partitioning works only in 1D, as any partition takes the entire height of the space).
Both vanilla Eclipse and Code Canvas do not support working on different tasks at once, as there is not such a logical separation between content that is being used.

5.1.2. Working set

To visualize the working set, we use a grid layout. We can compare this to Patchworks, which uses a static three-by-two grid, allowing six different code snippets being displayed on screen at once. Our grid is much more flexible, allowing an arbitrary number of items on the screen, depending on their size. Both Patchworks and our grid allows to add items at different granularities, for example method or class. We believe that this additional flexibility benefits the user, while still avoiding some of the rearrangement issues of Code Bubbles. Eclipse uses a traditional file-based layout, while Code Canvas is similar to Code Bubbles. However, as Code Canvas displays the entire project, it does not support a working set which is a subset of the entire project.

Our grid approach is certainly more rigid than Code Bubbles' visualization of a working set, where anything can be placed at arbitrary positions. However, that can lead to many bubbles being moved if a bubble is moved to the center, which can make spatial recognition harder. Envision's grid still occasionally moves items, however often the relative ordering of connected items can be kept - also because of the adaptive spacing items

5.1.3. Information

We are capable of showing more information types on screen than Patchworks. For one, we have the InfoNode, which can display arbitrary information about some AST node. Furthermore, we use arrows to visualize relationships. Both of these ways to display additional information apart from the nodes themselves can be turned on or off by the user, giving them full control. Due to its extensibility, the InfoNode might set us apart from Code Bubbles in the future. However, at the current point the information it shows is still rather basic and exists even more extensively in other tools as well. Code Canvas has a strong layer system which allows to easily turn different information layers on and off, while allowing many different types of information being displayed on screen. Eclipse allows viewing the files themselves as well as further information in side panels, for example the type hierarchy.

Tool	On screen	Off screen	Adding content	Information	Visualization	Granularity
Envision	Single *task view*	Other *task views*	Add command, context menus	Code, meta-information, relations	Flexible, e.g. entire code, public interface	Flexible
Code Bubbles [2]	Part of 2D canvas	Rest of 2D canvas	Find, context menus	Code, meta-information, relations, notes	Fixed, always entire code	Flexible
Patchworks [5]	3-by-2 grid of snippets	Ribbon of snippets	Drag from package explorer	Code	Fixed, always entire code	Flexible
Code Canvas [3]	Part of the entire project	Hidden layers	Make layers visible	Code, information layers, notes	Fixed, always entire code	Full project
Eclipse	Tabs	Tabs	Context menus, package explorer	Code, meta-information, relations	Fixed, always entire code	File

Table 1: The "On Screen" column describes what is displayed on screen, with the "Off Screen" column describing the things which are hidden. "Adding content" looks at the different possible ways of adding content to the view, while the "Information" column highlights the different types of information that can be presented.

5.2. Summary of discussion

To compare our *task views* to existing tools more objectively, we compare different dimensions of the functionality of these tools in table 1.

6. Future Work and Conclusion

6.1. Future Work

As we saw, the *task views* that are now implemented within Envision still have some work for improvement, with some of the issues having been outlined, especially in making the content that is visualized even more flexible. Furthermore, while some parts of the *task views* are very flexible and extensible, they also haven't reached their full potential yet.

InfoNode
Certainly some steps could be taken into extending the information displayed in an `InfoNode`, as well as possibly improving their design. Another point where they could be extended is to clearly indicate which information values are actually applicable for a specific node. Currently, each of the information methods is evaluated for any node, which may return empty values. While this still allows them to be displayed correctly, it can lead to confusion for the user: When turning such information values on or off, many irrelevant values are also offered in the command's auto-complete.

Arrows
The arrows themselves show glimpses of the information layer principle which is used in Code Canvas. In future work, this could be extended and improved to a full information layer system. These layers could then not only contain arrows but perhaps also other information. This would however also need adaptations to the JSON scheme. Making the JSON scheme itself more extensible is also a point which future work could address. Especially with many items on screen which are connected by arrows, it is noticeable that the arrows do not behave particularly smart. They use the shortest path to direct two items, which can often lead through existing items, occluding possibly interesting parts of them. Future work could address a more advanced way of drawing these arrows, or possibly even finding a better way to represent relationships over larger distances altogether.

6.2. Conclusion

By adding *task views* to Envision, we have managed to include many features already available in other, similar systems. Particularly for large projects, being able to visualize only parts of the project is very helpful to solve tasks or navigate existing code. Enabling sharing of *task views* also allows multiple people to collaborate on solving a single task. We have also managed to incorporate some new features into these views, which other systems do not have.

When it comes to visualization, we have incorporated several new concepts compared to existing tools. For one, we offer a grid-based, yet flexible solution for arrangement of visualization items. Furthermore, items can be visualized in different ways and levels of detail. Lastly, our extensible info node allow adding arbitrary information to the

view as well as future projects extending the information which can be shown in these nodes. As such, we believe to have successfully combined a high degree of flexibility for the visualization of items on screen, while still keeping the user from rearranging the content on screen.

We have also come up with efficient ways of switching between different *task views*, which combined with a clear separation of the views allows a user to efficiently work on multiple tasks at once. Furthermore, arranging the available views in a grid on screen improves spatial recognition for the user, as a view can not only be connected to its name but also its position in the grid.

A. Appendix: New commands and shortcuts

This project introduced various new commands and shortcuts to Envision. The first subsection lists all new commands including their possible parameters, effects and which items they are applicable to. The second subsection then lists all the shortcuts which were introduced, their effects and when they are available.

A.1. New commands

Command	**newView** name open
Effect	Creates a new *task view* and possibly opens it
Context	Applicable anywhere
Arguments	name - name of the new *task view* open - use "open" to immediately open the new view (default = don't open)
Command	**switch** name
Effect	Switches to an existing *task view*
Context	Applicable anywhere
Arguments	name - name of the *task view* to switch to
Command	**removeNode**
Effect	Removes the current node from the current *task view*
Context	Applicable on any node contained in the current *task view* as a top-level node
Arguments	-
Command	**toggleLayer** name
Effect	Toggle the given arrow layer in the current *task view*
Context	Applicable anywhere
Arguments	name - name of the arrow layer to toggle
Command	**showInfo**
Effect	Add an `InfoNode` for the current node
Context	Applicable on any node defining a symbol
Arguments	-
Command	**toggleInfo** name
Effect	Toggle the given info layer on the current `InfoNode`
Context	Applicable on any `InfoNode`
Arguments	name - name of the info layer to toggle
Command	**saveView**
Effect	Save the current *task view* to disk
Context	Applicable on any node with a manager
Arguments	-

Command	**add** name
Effect	Add a node by name to the current *task view*
Context	Applicable anywhere
Arguments	name - name of the node to add, supports using only part of the name

Command	**addNode** name column row
Effect	Add the current node to some *task view* at some position
Context	Applicable on any node defining a `METHOD` or `CONTAINER` symbol
Arguments	name - name of the *task view* to add the node (default = current) column - the column to add the node at (default = 0) row - the row to add the node at (default = 0)

Command	**addCallees**
Effect	Add the callees of the current method to the current *task view*
Context	Applicable on any method
Arguments	-

Command	**addCallers**
Effect	Add the callers of the current method to the current *task view*
Context	Applicable on any method
Arguments	-

Command	**addSuperClasses**
Effect	Add the super classes of the current class to the current *task view*
Context	Applicable on any class
Arguments	-

Command	**addSubClasses**
Effect	Add the sub classes of the current class to the current *task view*
Context	Applicable on any class
Arguments	-

Command	**inspectMethod**
Effect	Create a new *task view* to inspect the current method
Context	Applicable on any method
Arguments	-

A.2. New shortcuts

Shortcut	**CTRL + G**
Effect	Open the *task view* switching menu
Context	Applicable anywhere

Shortcut	**CTRL + 1** to **CTRL + 9**
Effect	Switch to the *task view* indexed by numbers one through nine
Context	Applicable anywhere

Shortcut	**ARROW keys**
Effect	Navigate in the *task view* switching menu
Context	Applicable within the switching menu

Shortcut	**ENTER**
Effect	Select the current entry in the *task view* switching menu or leave renaming mode
Context	Applicable within the switching menu

Shortcut	**F2**
Effect	Enter renaming mode in the *task view* switching menu
Context	Applicable within the switching menu

Shortcut	**CTRL + I/J/K/L**
Effect	Moving to the top/left/bottom/right of a `VViewItemNode`
Context	Applicable within a `VViewItemNode`

Shortcut	**ENTER**
Effect	Open a command prompt with the text "add"
Context	Applicable within a `ViewItem` (not including its children)

Shortcut	**CTRL + F5**
Effect	Fully refresh an `InfoNode`
Context	Applicable within an `InfoNode`

References

[1] D. Asenov and P. Müller. Envision: A fast and flexible visual code editor with fluid interactions (overview). In *Visual Languages and Human-Centric Computing (VL/HCC)*, pages 9–12, 2014.

[2] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: A working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2503–2512, New York, NY, USA, 2010. ACM.

[3] Robert DeLine and Kael Rowan. Code canvas: Zooming towards better development environments. In *Proceedings of the International Conference on Software Engineering (New Ideas and Emerging Results)*. Association for Computing Machinery, Inc., May 2010.

[4] George Fitzmaurice, Justin Matejka, Azam Khan, Michael Glueck, and Gordon Kurtenbach. Piecursor: Merging pointing and command selection for rapid in-place tool switching. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1361–1370, New York, NY, USA, 2008. ACM.

[5] Austin Z. Henley and Scott D. Fleming. The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2511–2520, New York, NY, USA, 2014. ACM.

[6] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 217–224, New York, NY, USA, 2011. ACM.

[7] Jan-Peter Krämer, Thorsten Karrer, Joachim Kurz, Moritz Wittenhagen, and Jan Borchers. How tools in ides shape developers' navigation behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3073–3082, New York, NY, USA, 2013. ACM.

[8] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. Blaze: Supporting two-phased call graph navigation in source code. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, pages 2195–2200, New York, NY, USA, 2012. ACM.

[9] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, June 1995.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Task-specific views in Envision

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Steimer	Cyril

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Ehrendingen, 22.09.2015	C. St.

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.