Declarative API for Defining Visualizations in Envision

Bachelor's Thesis Report

Andrea Helfenstein
Supervised by Dimitar Asenov, Prof. Peter Müller

May 5, 2013

Contents

List of Figures

Listings

1 Introduction

Envision is a visual programming environment for object-oriented languages, written in `C++`. Its development has begun as the master thesis of Dimitar Asenov in 2010 [10], and is still being actively developed.
Envision moves away from the classical text-editor based programming interface, towards an environment, where every piece of the code is visualized in a way that is not restricted to text only, but may include symbols, images, visual indications for scope or control flow, and many other visual elements. The human brain can process visual impressions much faster than it can process written words. Augmenting the code with visual elements shall help the programmer to understand the code more quickly and more intuitively.

1.1 Motivation

In order to visualize every piece of code inside a developing environment, Envision needs to be able to display thousands of visualizations at the same time. This is why we use a custom visualization framework based on the Qt GUI module [5] directly, and not a standard GUI framework like Qt Widgets [6]. Such a framework is designed for displaying only a few objects at a time, and does not scale well.

However, using the current Envision approach, every visualization needs to implement fine-grained control of the rendering process, which makes adding or modifying even trivial visualizations a tedious task due to the need to write a lot of boilerplate code.

Our goal is to make it easy to add and modify visualizations for Envisions. The motivation behind this is the following:

- We want to enable software developers to write their own visualizations for Envision. This would allow them to create specific visualizations for embedded Domain Specific Languages or libraries, potentially making their use much easier and more intuitive.
- Also, in order to improve the existing visualizations, we need to be able to determine which visualizations work best for which visual components. To this end, we need to be able to quickly adjust the visualizations and compare the different variants.

This thesis concentrates on the design and implementation of a visualization API on top of the already existing framework, providing a higher level of abstraction. This abstraction will allow standard visualization components (section 2) to be written and combined in a declarative way, with emphasis on flexibility, good readability, and minimal amount of code. Nonetheless, it will still be possible to implement non-standard visualizations, avoiding the higher level abstraction and using the underlying framework, or even to mix the two approaches. We will take the design of other GUI frameworks into consideration (section 3) when designing the API.

2 Standard Features and Properties of Visualizations in Envision

In this chapter we discuss what features and properties the visualizations in Envision currently have, and what features we would like them to have in the future. This then gives us a list of what functionality the new API needs to support.

2.1 Static Visualizations

In figure 1 there are a lot of visualizations that are static. Those are for example the icons and variable type names (such as `int`, `void`, and `float`).

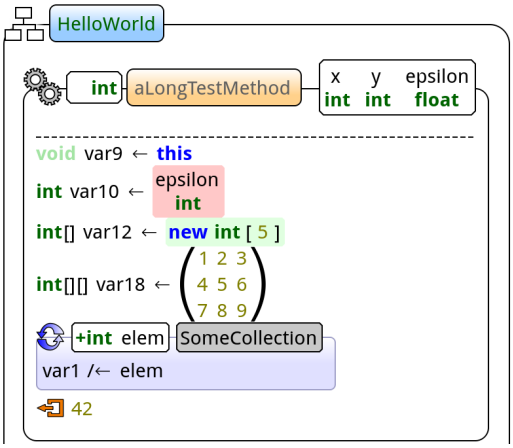


Figure 1: A screenshot showing how code may be visualized by Envision

2.2 Sequences of Visualizations

The most common way of displaying visualizations, is to show them one after another in a sequential manner, either vertically or horizontally. Depending on the orientation, the visualizations can be aligned horizontally to the *left*, the *right*, or the *center*, and similarly on the vertical axis to the *top*, the *bottom*, or the *center*.
Examples for this type of laying out the visualizations are the list of arguments (horizontal sequence), and the lines of code (vertical sequence).

2.3 Header and Body

Some visualizations, like those of classes, methods, and loops, contain other visualizations. Most of these *container visualizations* have a header at the top, displaying some general information, and a body below, where all the contained visualizations are shown.

2.4 Layers and Background

The *container visualizations*, mentioned in the last section, all have a background shape which does not just encompass all the contained visualizations, but only a subset of them.

2.5 Visualizations in a Grid

In figure 1, some visualizations are arranged in a grid structure. E.g. in the body of the method, there is a matrix. Also the parameters of the method (rightmost visual item of the method's header) may be arranged in a grid.
An important point to note here, is that the size of the grid might not be the same for all visualizations of the same type. E.g. the matrix visualization may be a general one, and not just for 3×3 matrices. Also, methods may have an arbitrary number of arguments.

2.6 Conditional Layouts

When visualizing code it is often useful to support multiple possible layouts, and then dynamically choose which one to use.
E.g. one might want to visualize a public field differently than a private one. When displaying an if-statement one may want to alter the layout depending on the contents of the different branches. It may even be desirable to display a series of if-statements like a switch statement, if they are used like one.
Currently Envision does not have an easily usable mechanism to support dynamic choosing of a specific layout based on user-defined conditions.

2.7 Visualizations with Size Depending on their Parent

Some visualizations may depend on their parent's size. E.g. in the list of statements, separators in the form of lines may need to be stretched to match the widest statement visualization in this list.

3 Layout Management in Other GUI Frameworks

There exist many other GUI frameworks, hence we analyzed how some of those enable the user to define and manage layouts. This helped us determine, which layout types will, or will not be useful for visualizations in Envision. Based on this knowledge, we can choose some layout types, and implement them adapted to the specific needs of defining a visualization for Envision.

The GUI frameworks we looked at in detail were the following:

Qt Widgets [6] A GUI framework written in `C++`.

Tikz [7] A graphics package for Latex.

wxWidgets [9] A GUI framework written in `C++`.

Gtk Layout Containers [1] A GUI toolkit written in `C`.

Tkinter [8] A GUI package included in all Python distributions.

All of those frameworks have their own concepts of how to define where a widget should be drawn. However, most of the concepts appear in more than one framework. We can divide those concepts into *Sequential Layouts*, *Grid Layouts*, and *Anchor Layouts*.

3.1 Sequential Layouts

Almost every framework we analyzed has a sequential layout. Here, sequential means that one component follows another, arranged either horizontally or vertically. In most of the frameworks it is referred to as a *Box Layout* (Qt Widgets: `QBoxLayout`, wxWidgets: `wxBoxSizer`, gtk: `HBox/VBox`, tkinter: `pack`[1]).

A sequential layout is the most basic layout. You can just add one component after the other, and align them appropriately. More complex structures can be achieved by nesting. However, using a lot of nesting comes at the cost of reduced readability.
The sequential layout is a good choice for visualizing simple sequences, but we need more intuitive solutions for more complex constructs.

3.2 Grid Layouts

As the name suggests, in a grid layout the components are arranged in a grid. In most of the frameworks supporting grids, one component can occupy more than one cell of the grid. Also, it is allowed to leave some cells empty. In general, one can define minimum width and height for columns and rows respectively. In addition, a row or column can be marked as stretchable. This means, if there is more space available than needed by the whole layout, those stretchable columns and rows will grow. In some frameworks a factor can be specified to indicate how the additional space will be distributed among the stretchable rows/columns.

The grid layout is very powerful but also intuitive, even for more complex structures. It allows natural alignment over two dimensions. Most GUI frameworks provide some form of grid layout (Qt Widgets: `QGridLayout`, wxWidgets: `wxFlexGridSizer`, gtk: `Table`, tkinter: `grid`).
Although a grid layout can make the definition of particular layouts easy, it is not very flexible. If

[1] `pack` is not strictly a sequential layout, but it can easily be used as one.

a more flexible layout is needed, one will end up designing a very fine-grained grid, and then again merging a lot of cells. This would not be readable anymore, and it also might affect the drawing performance.

3.3 Anchor Layouts

In an anchor layout, the components' positions are specified relative to each other. E.g. a component can be specified to be *left of* or *below* another component. This approach of laying out components seems not to be very common among GUI frameworks. Of the frameworks we looked at, only Qt Widgets[6] and Tikz[7] support an anchoring system.

Anchoring System in Tikz In Tikz[7] components can be placed using an anchoring system. Each component has north, north west, south, south east etc. properties, called *anchors*. Those anchors refer to a specific *point* relative to the component. When placing the component, the specified position can be a coordinate, or an anchor of an already placed component (which can be resolved to a coordinate). Additionally to that, one can specify which of the component's anchors should be put at this position. Note, that you can only use the anchor for placing the components, and not for resizing them.

`QAnchorLayout` in Qt Widgets In the `QAnchorLayout` components are placed by *aligning* their edges (like *left*, *right*, *south*, *north*, but also *center*) to the edges of other components, including the containing layout.
Here, a single anchor only affects the position on either the x- or the y-axis of the concerned components. Also it does not by itself place an component, it just constrains the components to have the concerned edges at the same place on the affected axis.

The Tikz approach is not as flexible as the Qt approach, because in Tikz you can only relate the position of a component to an already fixed coordinate. Using the Qt Widgets, you have the possibility to only fix the x or y coordinate of a component. Also, you do not place a component directly, but only specify relationships between some component's edges, which additionally allows you to specify that components should have the same size. The latter is only possible in Tikz, by explicitly setting the two components' sizes to the same value.

We have now examined how some GUI frameworks enable the user to manage layouts in a convenient way, and how powerful each of those layouts can be. We can use the principles of those layouts as a guideline when designing our own declarative API, enhancing the already existing visualization framework of Envision.

4 The Underlying Visualization Framework of Envision

In this section, we give an overview of the visualization framework in Envision. It is important to know the basic principles behind this framework, in order to understand how our new declarative API (section 5) can be used, and how the functionality behind the API is implemented (section 6).
Envision uses a *Model View Controller* (MVC) architecture. In this thesis we are mostly concerned with the *view* part of this architecture. Although, since the view is visualizing the model, we also need to know the basics about the *model* of Envision.

`Model::Node` The *model* of Envision is represented by a *tree of nodes*, the type `Model::Node` being the base of all those nodes.

`Visualization::Item` The *view* of Envision is represented by a *tree of items*. The base class of all these items is `Visualization::Item`. An item can be responsible for visualizing a model node, but it could also be used to display e.g. an icon, which is not part of the model.

4.1 The Class `Visualization::Item`

`Visualization::Item` is the base class of all visualization items. All derived classes need to implement those two main methods:

- `void determineChildren()`
- `void updateGeometry(int availableWidth, int availableHeight)`

Both of those methods are relevant when the visualization item needs to be rendered. The method `determineChildren` is called first. The method is used by the visualization item to update, create, or destroy contained visualization items, according to the possibly updated model.

In a second step, the method `updateGeometry` is called with `availableWidth` and `availableHeight` of zero each. This indicates, that the visualization item's minimum size should be computed. When this method is being called, the minimum sizes of all the visualization items child items are already available. The visualization item needs to set the positions of all its child items. It can also expand the child items, if they support it. In the end, the visualization item has to set its own size.

If a visualization item is requested to expand, its method `updateGeometry` is called a second time, with non-zero `availableWidth` and `availableHeight`. Those arguments need to be at least as large as the visualization item's minimum size. After this call of `updateGeometry` the visualization item's dimensions will be exactly the given `availableWidth` and `availableHeight`.

5 The New Declarative API for Visualizations

Envision uses the Qt framework, and we found the its API to be very intuitive in general. Therefore, when designing the declarative API for visualization in Envision, we followed the design principles as they are stated by the developers of the Qt framework in [4] and [2]:

Be minimal *A minimal API has as few public members per class and as few classes as possible. This makes it easier to understand, remember, debug, and change the API.*

Be complete *A complete API means the expected functionality should be there. This can conflict with keeping it minimal. Also, if a member function is in the wrong class, many potential users of the function won't find it.*

Have clear and simple semantics *Common tasks should be easy to do. Rare tasks should be possible but not the focus. Solve the specific problem; don't make the solution overly general when this is not needed.*

Be intuitive *An API is intuitive if a semi-experienced user gets away without reading the documentation, and if a programmer who doesn't know the API can understand code written using it.*

Be easy to memorize *To make the API easy to remember, choose a consistent and precise naming convention. Use recognizable patterns and concepts, and avoid abbreviations.*

Lead to readable code *Code is written once, but read (and debugged and changed) many times. Readable code may sometimes take longer to write, but saves time throughout the product's life cycle.*

In other words, we want the user to be able to easily produce readable code that does what the user expects.

5.1 Usage of the Declarative API

As discussed in section 4.1, the base class of all visualization items, `Item`, lets the user define the layout and behavior of descendant classes, by overriding the methods `determineChildren` and `updateGeometry`. We introduce a new visualization item, acting as the base class of all items using the declarative API: `DeclarativeItem`. This class implements the two methods mentioned above, and lets the user specify the layout and behavior of descendant visualization items in an alternative, more declarative way.

The user specifies one or more *forms*, where a form represents one possible layout for this visualization type. If the user specifies more than one form, he also needs to specify in what state of the visualization which form to use. This mechanism can be used to switch between different layouts based on user defined conditions, as described in section 2.6.

In this section we explain the basic structure of a visualization item that uses the declarative API.

```
1 class MyVisualizationItem : public DeclarativeItem<MyVisualizationItem> {
2   public:
3     static void initializeForms();
4     int determineForm() override;
5 }
```

Listing 1: Class Declaration using `DeclarativeItem`

It is mandatory to implement a static method `initializeForms`. As the name suggests, this method is used to initialize the forms mentioned above. These form definitions are then shared among all instances of this particular visualization type.
The method `determineForm` needs to be overridden only if the user defines more than one form for this visualization item type. It is used to decide which form to use when the item needs to be rendered.

```
1 void MyVisualizationItem::initializeForms()
2 {
3   /* form with index 0 */
4   addForm(/* define an element */);
5
6   /* form with index 1 */
7   addForm(/* define another element */);
8 }
```

Listing 2: Method `initializeForms()`

Inside the method `initializeForms`, there needs to be at least one call to the method `addForm`. The added forms will be identified by increasing numbers in the order they were added, starting with 0 for the first form.

If there are multiple forms defined, the method `determineForm` needs to decide which one of them should be used. This decision method is called each time the visualization needs to be rendered. It returns an integer, which is interpreted as an *identifier of the form to use* for rendering. The default implementation of `determineForm` always returns 0, meaning the first form will be taken. This suffices for the case, where there is only one form available.

What is a Form? In the method `addForm`, an object of type `FormElement` needs to be supplied. This object is the root of a tree, describing a way to arrange the child items of the visualization item we are defining.
We use the composite pattern, such that every form element can potentially contain other form elements. There are multiple classes derived from `FormElement`, some used for wrapping visualization items (section 5.2), and some used as layouts to arrange other form elements (section 5.3). The following two sections will show how those form elements can be used to control the way a visualization item is displayed.

5.2 Item Wrappers

An item wrapper is a form element that takes care of rendering a child item of the visualization item we are defining. In a tree of form elements, most of the tree's leafs are item wrappers, since visualization items are the only objects to be rendered. The form element trees are just there to compute where and how large those visualization items should be displayed.
There are three types of item wrappers:

(1) `NodeItemWrapperFormElement`
visualizes a *model node* using its *default visualization*

(2) `NodeWithVisualizationItemWrapperFormElement`
visualizes a *model node* using a *specific visualization* and style

(3) `VisualizationItemWrapperFormElement`
renders a *specific visualization* using a provided style (without any model node)

The names of those classes are very long and each of them is a template class with up to two template arguments. Using these inside the form definition would reduce the readability by a lot. Therefore we added three *factory methods*, that can be used by any class derived from `DeclarativeItem`. Each of those methods corresponds to one of the three item wrapper types. Next we show how to use those factory methods.

The first parameter to specify for any item wrapper, is a place to store the wrapped visualization item. This needs to be a *pointer-to-member* to an item inside the visualization item we are defining. In addition to that, (1) and (2) need to be able to get a *model node* to construct a new visualization item if needed. Also, (2) and (3) need to know what type of visualization item should be created, and what style to use for it.

In the following, we describe how exactly those three parameters need to be specified, using the visualization for the return statement (`VReturnStatement`) as an example.

(a) Original visualization for the return statement, created with listing 3.

(b) Adapted visualization, to illustrate what happens if the visualization of the return value is rendered with the default style for `VList`.

Figure 2: Visualization of the return statement

```
1  void VReturnStatement::initializeForms()
2  {
3    addForm((new GridLayoutFormElement())
4      ->setTopMargin(5)->setBottomMargin(5)->setHorizontalSpacing(5)
5      ->setVerticalAlignment(LayoutStyle::Alignment::Center)
6      ->put(0, 0, item<Static, I>(&I::symbol_,
7                                  [](I* v){return &v->style()->symbol();}))
8      ->put(1, 0, item<VList,I>(&I::values_,
9                                [](I* v){return v->node()->values();},
10                               [](I* v){return &v->style()->values();})));
11 }
```

Listing 3: `initializeForms` method of the class `VReturnStatement`

In listing 3, two kinds of item wrappers are used: One with visualization and style only (lines 6-7), and another one with a model node and a style (lines 8-10).

The one on lines 6-7, with the *style only* (3) displays the icon:

```
item<Static, I>(&I::symbol_, [](I* v){return &v->style()->symbol();})
```

`Static` is the visualization type for the icon. It can be used for displaying images. `I` is an alias for the class `VReturnStatement`.
The first argument, `&I::symbol_` is a pointer to the private member `symbol_` of the class `VReturnStatement`. This member needs to be defined in the class declaration as `Static* symbol_`.
Note that the base class `DeclarativeItem` of `VReturnStatement` takes care of destroying all items used in any item wrapper upon destruction of the containing declarative visualization item. Since the child items are the only fields of `VReturnStatement`, this class does not need any destructor.
The second argument is a *lambda-function*, taking an argument of type `VReturnStatement*`, and returning a *reference to a style*. Note that the style is acquired by calling the visualization item's `style` method. In our case, the lambda-function returns the symbol style, holding e.g. the path to the icon picture.

The item wrapper definition on lines 8-10, displays a list of return values. It renders a *model node with the given visualization type and style* (2):

```
item<VList,I>(&I::values_, [](I* v){return v->node()->values();},
                           [](I* v){return &v->style()->values();}))
```

This time, `VList` is the visualization type used to create the wrapped visualization item. The difference of this call to the previous one, is the addition of the second argument. It is a *lambda-function* taking an argument of type `VReturnStatement*`, and returning a *pointer to a model node*. Whenever this node is different from the one returned previously, the item wrapper automatically creates a new item out of the model node, with the given visualization type and style. Note that the item will be `nullptr`, if no node is returned.

To look at the item wrapper with the *model node only* (1), we can change the element argument in listing 3 on lines 8-10 to:

```
item<I>(&I::values_, [](I* v){return v->node()->values();})
```

Once we remove the style, the first template argument is no longer needed. Additionally, the definition of member `values_` needs to be changed from `VList* values_{}` to `Item* values_{}`. This is because the model node will be rendered with the default visualization, of which the type is not known.
After those changes, the return values are no longer visualized as in figure 2a, but rather as in figure 2b.

5.3 Layouts

Layouts can be used to *arrange a set of form elements* in a specific way. In general those layouts are implemented as form elements, holding a list of other form elements, and rules on how to arrange them. The exception is the sequential layout, which on every redraw gets a list of visualization items or model nodes to display.

5.3.1 Sequential Layout

The class `SequentialLayoutFormElement` lets us define sequences of elements as defined in section 2.2. The sequential layout gets a list of visualization items to display on every redraw. There are three ways to provide it with those items:

(1) Specify a lambda-function returning a *list of visualization items*.

(2) Specify a lambda-function returning a *list of model nodes*.

(3) Specify a lambda-function returning a *model node of type* `Model::List`.

Note that the layout can transform the node of type `List` into a list of nodes, and each node from the list can be rendered with its default visualization, yielding a visualization item. Finally, those can be accumulated to a list of visualization items.

Next, we describe how those methods can be specified, using the sequential layouts in the visualization of the class (`VClass`) as an example.

```
auto fieldContainerElement = (new GridLayoutFormElement())
  ->setVerticalSpacing(3)
  ->put(0, 0, (new SequentialLayoutFormElement())->setVertical()
    ->setListOfNodes(
            [](Item* i){return (static_cast<VClass*>(i))->publicFields_;}))
  ->put(0, 1, (new SequentialLayoutFormElement())->setVertical()
    ->setListOfNodes(
            [](Item* i){return (static_cast<VClass*>(i))->privateFields_;}))
  ->put(0, 2, (new SequentialLayoutFormElement())->setVertical()
    ->setListOfNodes(
            [](Item* i){return (static_cast<VClass*>(i))->protectedFields_;}))
  ->put(0, 3, (new SequentialLayoutFormElement())->setVertical()
    ->setListOfNodes(
            [](Item* i){return (static_cast<VClass*>(i))->defaultFields_;}));
```

Listing 4: Definition of the header form element in `VLoopStatement::initializeForms`

In listing 4, four sequential layouts arranged using a grid layout. All those sequential layouts are of type (2), as they specify a list of nodes:

```
(new SequentialLayoutFormElement())->setVertical()
  ->setListOfNodes([](Item* i){return (static_cast<VClass*>(i))->
     publicFields_;})
```

Here, the private member `publicFields_` of `VClass` is of type `QList<Model::Node*>`.

Note, that in contrast to the item wrapper, in the sequential layout the *lambda-functions* take a general `Visualization::Item*` as an argument, which needs to be casted to `VClass*` in order to get full access to visualization object.

In a sequential layout of type (1), we need to supply the sequential layout with a list of rendered visualization items. Let us assume, that `VClass` has a private member `publicFieldItems_` of type `QList<Visualization::Item*>`. Then we can supply this list to the sequential layout as follows:

```
(new SequentialLayoutFormElement())->setVertical()
  ->setListOfItems([](Item* i){return (static_cast<VClass*>(i))->
     publicFieldItems_;})
```

Lastly, in a sequential layout of type (3), we need to supply the layout with a model node of type `Model::List`. Let us assume, that all the field nodes are accessible for an instance of `VClass` via `node()->fields()`, and that this method returns an object of type `Model::List*`. Then we can instantiate the sequential layout as follows:

```
(new SequentialLayoutFormElement())->setVertical()
  ->setListNode([](Item* i){return (static_cast<VClass*>(i))->node()->
     fields();})
```

5.3.2 Grid Layout

The class `GridLayoutFormElement` allows arranging components inside a grid as described in section 2.5, with one exception: The grid size is fixed for one instance of class `GridLayoutFormElement`.

A matrix as described in section 2.5 cannot be implemented with this layout. It can only be used for statically-sized grids. But since the grid layout is designed for arranging a statically fixed number of form elements, we do not need this additional flexibility.

In the following, we show how the grid layout can be used to arrange form elements through the example of the loop statement (`VLoopStatement`).

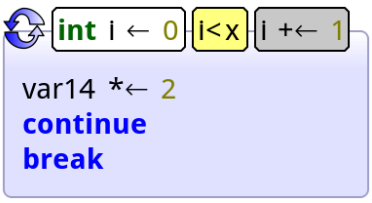


Figure 3: Visualization of the loop statement

For our demonstration of the grid layout, let us consider the grid layout defining the header part of the loop statement visualization shown in listing 5.

```
1  auto header = (new GridLayoutFormElement())
2    ->setHorizontalSpacing(3)->setColumnStretchFactor(3, 1)
3    ->setVerticalAlignment(LayoutStyle::Alignment::Center)
4    ->put(0, 0, item<Static, I>(&I::icon_, [](I* v){return &v->style()->icon();}))
5    ->put(1, 0, item<NodeWrapper, I>(&I::initStep_,
6                                     [](I* v){return v->node()->initStep();},
7                                     [](I* v){return &v->style()->initStep();}))
8    ->put(2, 0, item<NodeWrapper, I>(&I::condition_,
9                                     [](I* v){return v->node()->condition();},
10                                    [](I* v){return &v->style()->condition();}))
11   ->put(3, 0, item<NodeWrapper, I>(&I::updateStep_,
12                                    [](I* v){return v->node()->updateStep();},
13                                    [](I* v){return &v->style()->updateStep();}));
```

Listing 5: Definition of the header form element in `VLoopStatement::initializeForms`

On the first line we create a new grid layout. On the following lines we call various methods on this grid layout. The *methods can be chained* as shown, because every method modifying the grid layout returns the pointer to this grid layout again. This technique called *method chaining*, can be found throughout the declarative API. We use it to *avoid repetition of arguments* over several function calls.

Let us first consider the method `put`. This is the main method to define the grid. Every time it is called, another form element gets added to the grid at the desired position. E.g. the second call (lines 5-7), adds the form element for the initialization step at column 1 and row 0. You may observe, that we did not have to specify the size of the grid in the beginning. The size of the grid is automatically adjusted as more form elements are added.

All the other methods can be used for fine-tuning the appearance of the grid. For setting most properties, there are three kinds of methods available:

(1) Set the property for *the whole grid*.

(2) Set the property for *one column or row*.

(3) Set the property for *one cell*.

For methods of type (2), the column or row number must be specified before the parameter. However for methods of type (3) the cell does *not* have to be specified. The cell on which this parameter will be set is the *last cell modified* via the method `put`. This eliminates repeating the position of this cell in the grid.

Space Between Columns and Rows The horizontal and vertical spacing can be set separately, or as one. The following methods are available:

- `setSpacing(int spacing)`
- `setSpacing(int spaceBetweenColumns, int spaceBetweenRows)`
- `setHorizontalSpacing(int spaceBetweenColumns)`
- `setVerticalSpacing(int spaceBetweenRows)`

Alignment The horizontal and vertical alignment can be set for each cell. The default alignment is top left. The following methods are available:

- `setHorizontalAlignment(LayoutStyle::Alignment horizontalAlignment)`
- `setVerticalAlignment(LayoutStyle::Alignment verticalAlignment)`
- `setColumnHorizontalAlignment(int column, LayoutStyle::Alignment horizontalAlignment)`
- `setRowVerticalAlignment(int row, LayoutStyle::Alignment verticalAlignment)`
- `setCellHorizontalAlignment(LayoutStyle::Alignment horizontalAlignment)`
- `setCellVerticalAlignment(LayoutStyle::Alignment verticalAlignment)`
- `setCellAlignment(LayoutStyle::Alignment horizontalAlignment, LayoutStyle::Alignment verticalAlignment)`

Stretching Columns or Rows Stretch factors determine if and how additionally available space should be distributed among the columns and rows respectively.

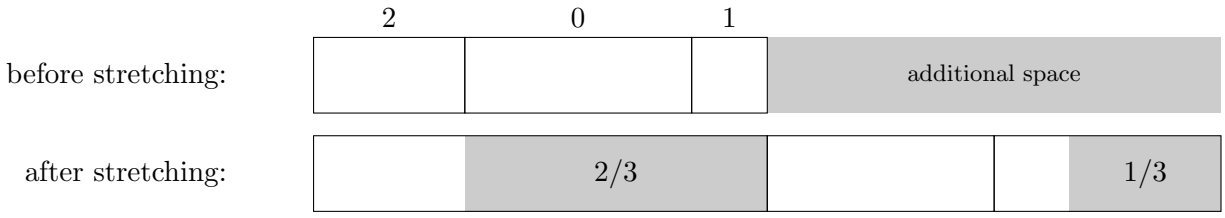


Figure 4: Stretch factor example

Consider figure 4. The upper row shows a grid layout with three columns. Their width currently meets minimum space requirements. You can see their stretch factors written above the columns, and there is some additional space available, marked on the right.
Columns with stretch factor 0 do not stretch, but each column with a higher stretch factor gets a portion of the additional space, according to their stretch factor. In our case, the first and third column have stretch factors 2 and 1 respectively. Summing them up we get a total stretch factor of 3, this means the first column gets 2/3 of the additional space, while the third column gets 1/3, as shown in the lower row of figure 4.
The following methods for setting the stretch factors are available:

- `setColumnStretchFactor(int column, float stretchFactor)`
- `setColumnStretchFactors(float stretchFactor)`
- `setRowStretchFactor(int row, float stretchFactor)`
- `setRowStretchFactors(float stretchFactor)`
- `setStretchFactors(float stretchFactor)`

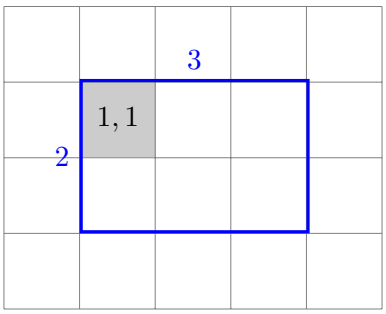


Figure 5: Merge cells in a grid layout

Merging Cells can be achieved by specifying a cell spanning different from 1×1.
Consider figure 5. Adding a form element to cell $(1, 1)$ via `put(1, 1, formElement)` (marked in gray), and then adding a spanning to this cell via `setCellSpanning(3, 2)` would result in the added form element being handled relative to the merged cell, marked in blue. If this form element is stretchable, it would be resized to be as large as the blue box.

5.3.3 Anchor Layout

The class `AnchorLayoutFormElement` allows arranging form elements by aligning their edges. It works very similar to the `QAnchorLayout` from the Qt Widgets [6], with the exception that you cannot align any component's edge to an edge of the containing layout.

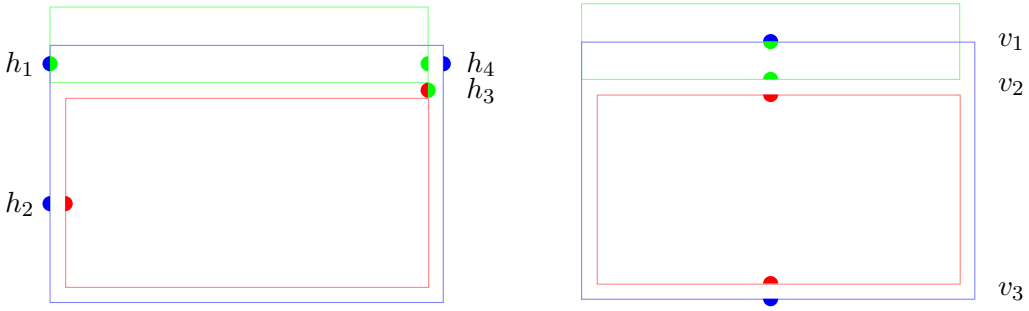
As an example, we will look at the loop statement (`VLoopStatement`) again. Consider lines (7-14) of listing 6. Those lines arrange header, body and shape in such a way, that the end result looks like figure 6c.

```
1  void VLoopStatement::initializeForms()
2  {
3    auto header = /* ... */;
4    auto body = /* ... */;
5    auto shapeElement = new ShapeFormElement();
6
7    addForm((new AnchorLayoutFormElement())
8      ->put(TheTopOf, body, 10, FromBottomOf, header)
9      ->put(TheTopOf, shapeElement, AtCenterOf, header)
10     ->put(TheLeftOf, shapeElement, AtLeftOf, header)
11     ->put(TheLeftOf, shapeElement, 10, FromLeftOf, body)
12     ->put(TheRightOf, header, AtRightOf, body)
13     ->put(TheRightOf, shapeElement, 10, FromRightOf, header)
14     ->put(TheBottomOf, shapeElement, 10, FromBottomOf, body));
15 }
```

Listing 6: Overview of `VReturnStatement::initializeForms`

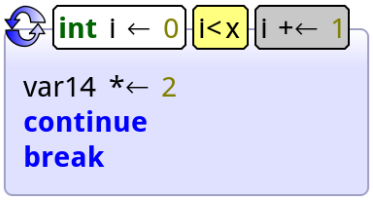
In figures 6a and 6b we used green to denote the header, red for the body, and blue for the shape. The anchors marked in figure 6a affect the form elements' widths and positions on the x-axis, while the anchors marked in figure 6b affect their heights and positions on the y-axis.
On the horizontal axis, h_1 aligns the left of the header and the shape (line 10), while h_2 aligns the left of the shape and the body with an offset (line 11). This means, the x-coordinate of header and shape will be the same, whereas the body's x-coordinate will be the previously mentioned one plus the offset. h_3 aligns the right of header and body (line 12). Together with h_1 and h_2, this means the header's width will always be longer by offset than the body's. The anchor layout *stretches* either the header or the body to satisfy this constraint. Note that no component will ever shrink. Similarly h_4 aligns the right of header and shape with an offset (line 13).

On the vertical axis v_1 aligns the top of the shape to the center of the header (line 9). v_2 places the



(a) Horizontal alignments

(b) Vertical alignments



(c) Visualization of the loop statement

Figure 6: Anchor layout explained

body some offset below the shape (line 8), while v_3 places the bottom of the body the shape within some offset from each other (line 14). v_3 forces the shape to stretch vertically, as discussed above, for the horizontal case.

Note, that if a form element is not stretchable, but it needs to stretch in order to satisfy the anchor constraints, the placement will fail, and a `VisualizationException` will be thrown.

The following methods for defining anchors are available:

(1) `put(PlaceEdge placeEdge, FormElement* placeElement, AtEdge atEdge, FormElement* fixedElement)`

(2) `put(PlaceEdge placeEdge, FormElement* placeElement, int offset, FromEdge fromEdge, FormElement* fixedElement)`

(3) `put(PlaceEdge placeEdge, FormElement* placeElement, float relativeEdgePosition, FormElement* fixedElement)`

Methods (1) and (2) were explained using the example of the loop statement. The anchors h_1, h_3, and v_1 from the example are defined using method (1), while h_2, h_4, v_2, and v_3 are defined using method (2).

Method (3) is a more general version of method (1). Both do not have an offset, and relate two form elements to each other, by specifying an edge for each element. However, in method (3) we specify the edge of the second form element as a *relative edge position*, which allows the user to specify arbitrary edges relative to this element.
In figure 7 you can see a form element. The dashed lines mark different *relative edge positions*. On each ones top, the value of this relative edge position is marked, while the corresponding `atEdge` for the horizontal case can be found at the bottom, if such a correspondence exists.

5.3.4 Position the Shape

Every visualization item may define a *shape* property in its style. This shape is displayed as the *background of the visualization*. By default, this shape encompasses the whole visualization, but this behavior can be changed.

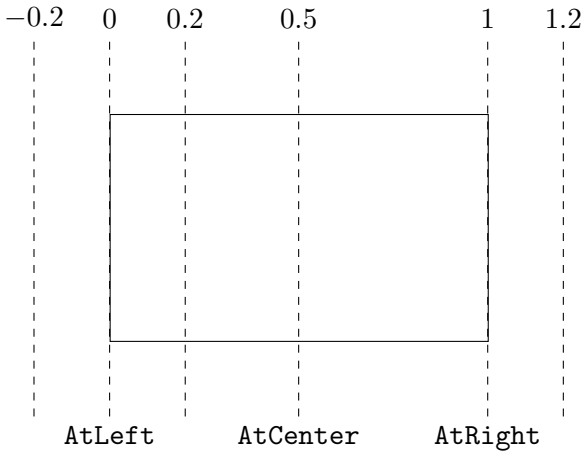


Figure 7: Relative edge positioning in the anchor layout

In the declarative API, we use the class `ShapeFormElement` to do this.

We use the loop statement as example again. In figure 6c, the shape of the loop visualization is visible as a light blue background. Let us consider lines 5-14 of listing 6:

```
auto shapeElement = new ShapeFormElement();
addForm((new AnchorLayoutElement())
  ->put(TheTopOf, body, 10, FromBottomOf, header)
  ->put(TheTopOf, shapeElement, AtCenterOf, header)
  ->put(TheLeftOf, shapeElement, AtLeftOf, header)
  ->put(TheLeftOf, shapeElement, 10, FromLeftOf, body)
  ->put(TheRightOf, header, AtRightOf, body)
  ->put(TheRightOf, shapeElement, 10, FromRightOf, header)
  ->put(TheBottomOf, shapeElement, 10, FromBottomOf, body));
```

The `ShapeFormElement` is a placeholder. It has minimum width and height of 0, respectively, but it will stretch to fill up any space that is available. In the example, the `ShapeFormElement` is inserted into an anchor layout. All its edges (left, right, top, bottom) need to be fixed in order for it to be stretched correctly. If one of those fixing anchors were missing, the shape would collapse to a line.
It is also possible to add a `ShapeFormElement` to a grid layout, where the shape would occupy the specified cell, or multiple merged cells.

6 Implementation

In this section we explain the reasons behind some of our design decisions, also in the context of the implementation.

6.1 Class Hierarchy of the Declarative Item

An important point for us to keep in mind was performance and memory efficiency. We didn't want to have to store the layout in each instance of a visualization type (e.g. `VClass`), but we wanted this information to be available in one place, for all the instances of this visualization type. The list of available forms (section 5.1) should be the same for objects of the same type, but different for each visualization type.
As Envision is written in `C++`, we implemented this behavior using templates together with static methods and fields. In figure 8 we can see that the class `DeclarativeItemBase` is directly inheriting from `Item`. `DeclarativeItemBase` which is not a template class, implements the basic functionality of any declarative item. The class `DeclarativeItem` however is a template class, such that for each visualization item it gets compiled once, and therefore its static fields are different for each type. In

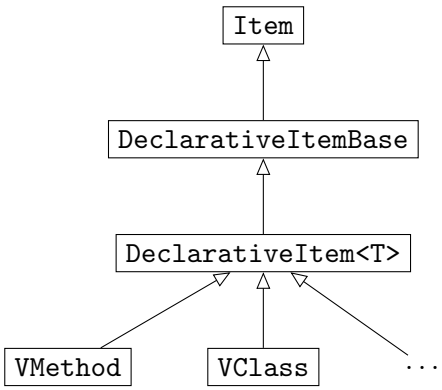


Figure 8: Declarative Item class hierarchy

this way, we ensure that each visualization type has its own static list of available forms.

Note that the use of templates and static methods and fields is the reason, why the method `initializeForms` has to be static, and implemented by each visualization type separately.

The method `determineForm` is an instance method. It can be executed by any object of the same visualization type, and decide which form to use for this instance, based on the status of the object.

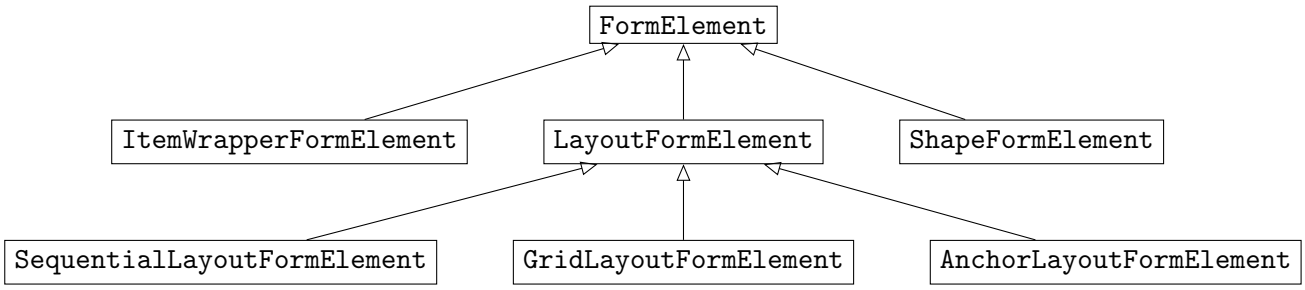


Figure 9: Form element class hierarchy

6.2 Item Wrappers

The item wrappers are implemented as a hierarchy of four template classes.

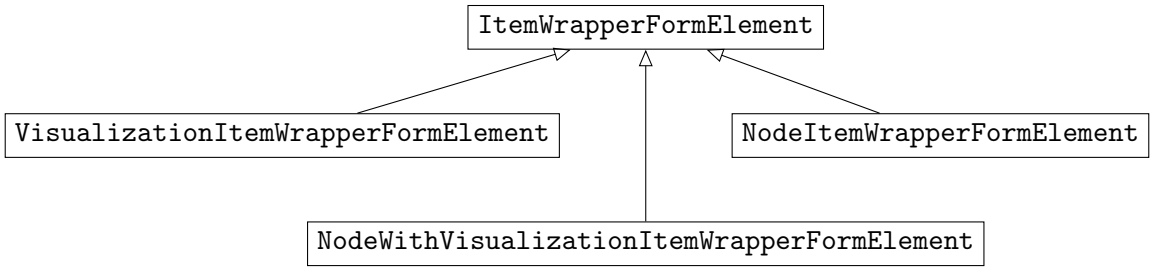


Figure 10: Item Wrapper class hierarchy

The base class (`ItemWrapperFormElement`) handles placement, resizing, and destruction of the wrapped item. For this reason, the item wrapper needs to have read and write to the private member of its containing visualization type, where the wrapped item is going to be stored. Since the item wrapper, as every other form element, is defined statically, this is not a trivial task. Supplying the item wrapper with a pointer-to-member solves this problem. If the item wrapper is now supplied with an instance of the containing visualization type, it can resolve the pointer-to-member and access the wrapped visualization item. But in order for the pointer-to-member object to be stored and accessed correctly,

the item wrapper needs to be a template with the containing visualization type and the wrapped visualization type as template arguments.

The three derived classes (see figure 10) are responsible for creating and updating the wrapped visualization item. Visualization items displaying a node have different constructors from those that don't visualize a node. Therefore, we need the three different template classes, or the code would not even compile.

As we cannot expect the user to cope with those three final classes to choose from, we added a factory method for each of those three types to the class `DeclarativeItemBase`. Now every declarative item already includes those item wrappers and can directly use the factory methods to create item wrappers, instead of the constructors of the different types. This makes the code more readable for the user, and also shroter.

6.3 The class `SequentialLayoutFormElement`

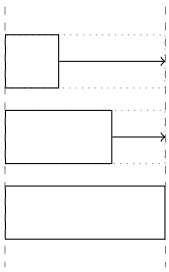
The `SequentialLayoutFormElement` is very similar to the `Visualization::SequentialLayout`, that is already part of the visualization framework of Envision.

The `SequentialLayout` is a visualization type, which can take a list of visualization items upon every redraw, and visualize them. The `SequentialLayoutFormElement` is a functionally enhanced version of the `SequentialLayout` inside the declarative API. The list of visualizations it renders, can not only be specified as a list of visualization items, but also as a list of model nodes, or even a node of type `Model::List`. This makes it easier to display lists of nodes with their default visualization, since they can be rendered automatically by the `SequentialLayoutFormElement`, instead of by the user.

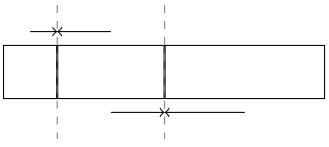
Another advantage of the `SequentialLayoutFormElement` over the `SequentialLayout` is the reduced nesting of visualization items, since the `SequentialLayoutFormElement` is used as part of a form, and not a visualization item itself.

6.4 The class `AnchorLayoutFormElement`

The anchor layout is the most flexible layout we decided to implement, but this flexibility also makes it the most complex layout. In general we do not know in advance in which order we should compute the positions and sizes of the contained form elements.



(a) Non-trivial anchoring problem



(b) Trivial anchoring problem

Figure 11: Types of anchoring problems

Consider three form elements that have their left and right edges aligned (see figure 11a). Here, the positioning depends on the current size of each one of the form elements. In general we do not know which form element will be the largest, and therefore we cannot fix an order on them, in which we could compute their positions and sizes.

This general re-sizing and positioning problem can be formulated as a linear programming problem (see appendix A). In our implementation, we use the library lpsolve [3] to compute a solution for this problem.
However it is not always necessary to use this library, as in some cases, one can find an order for positioning the form elements, that works in every case.
Our implementation detects whether we are in such a simple case, like in figure 11b. There we can always place the form elements besides one another, and no re-sizing is needed. If we are in a simple case like this, we can compute the order in which to determine the form elements' positions just after the form was defined, i.e. statically.

Our approach of detecting, whether we are in a simple case or a complex one, and using a linear programming solver, is very similar to the approach Qt uses for `QAnchorLayout` [6], although there the separation of simple and complex problems is different, and they can also dock elements to the sides of the layout, which makes `QAnchorLayout`'s linear programming problem more involved.

7 Discussion

In this section we evaluate the new declarative API based on criteria we have given earlier. We focus on the qualities of code using the API. We will compare visualization items in Envision, with and without the use of the new API.

When inspecting the code, we will judge it based on a set of criteria defined as follows.

- In section 1.1 the following important qualities were stated:

  (1) Visualizations need to be easy to add and modify.
  (2) The usage of the API should lead to readable code.
  (3) The code using the API should be as short as possible.
  (4) It should still be possible to use the underlying framework, avoiding the new API. 4)

- In addition to those, we looked at the design principles of the Qt developers [4] and [2] in section 5. The ones concerning the resulting code, and differing from the previously stated ones are:

  (5) A developer not knowing the API should understand code written using it.
  (6) Common tasks should be easy to do, rare tasks should be possible, but not the focus.

In the following, we discuss those criteria on the basis of the new declarative implementation of three visualizations, namely `VLoopStatement`, `VClass`, and `VIfStatement`.

7.1 Declarative Implementation of the Loop Statement Visualization - `VLoopStatement`

In this section, we will discuss the new declarative implementation of the class `VLoopStatement` with emphasis on the length of the code (criterion 3), ease of modification (criterion 1), how readable and intuitive the code appears (criteria 2 and 5), and possible issues of the declarative implementation.

Code Length The size of the implementation of the loop statement could be reduced from 78[2] *Source Lines of Code (SLOC)*[3] to 65[2] SLOC in the header file, and from 115[2] to 71[2] in the source file. In total, the implementation of the loop statement using the declarative API is about 30% shorter than the implementation using the original framework.
This reduced code length can be explained with the following reasons:

[2]Counted by the SLOC counter used by gitHub (https://github.com/).
[3]http://en.wikipedia.org/wiki/Source_lines_of_code

- In the version with the new API, we do not need a *destructor* anymore, because all the fields of the class are visualization items, and their destruction is managed by item wrappers.
- Using the new API, the *constructor* only calls the parent constructor, the body is empty. This is also because all the fields are visualization items, and there construction is managed by item wrappers.
- Using the new API, the method `determineChildren` does not longer need to be implemented by the user, as it is already implemented by the parent class.
- Using the new API, the new method `initializeForms` needs to be implemented by the user. Though in the case of `VLoopStatement` those new lines of code are much less, than the ones we do no longer need.
- In the version with the new API, we could remove some visualization items, that were only there, to wrap other visualizations, for the only purpose of giving them a background. Parallel to the declarative API we implemented an `NodeWrapper` item, which we can use here to replace the use of two visualization items by a single one.
  In the case of `VLoopStatement`, this was used three times.

Ease of Modification To estimate how easy it is to modify a given visualization, we count the areas of code, where of one visualization item inside `VLoopStatement` appear. We assume, that even for a simple modification, one has to change code in most of those areas, or at least visit them, to see if a modification is needed. Therefore it makes sense to consider, that if each visualization item appears in only a few areas in the code, the code will be easier to modify, than if the visualizations appear in several areas of the code.

For the purpose of this discussion, we only check the number of appearances of one visualization item in the loop statement, since the situation of the other visualization items is similar.
For the *condition* visualization item we counted seven areas for the implementation without using the declarative API, while for the implementation using the declarative API the condition only appears twice: once when the private member is defined, and once when it is placed inside the header form element. This is a great reduction of areas to modify, even more so if you take into consideration, that the member declaration is rarely changed. This leaves the user with a *single area*, where the behavior of the condition visualization can be changed.

Readability and Intuitive Use In this paragraph we analyze how each of the types of form elements is used and how readable they may appear to someone who is not familiar with Envision.

Let us first consider the *anchor layout* (listing 6, lines 7-14):

```
addForm((new AnchorLayoutElement())
  ->put(TheTopOf, body, 10, FromBottomOf, header)
  ->put(TheTopOf, shapeElement, AtCenterOf, header)
  ->put(TheLeftOf, shapeElement, AtLeftOf, header)
  ->put(TheLeftOf, shapeElement, 10, FromLeftOf, body)
  ->put(TheRightOf, header, AtRightOf, body)
  ->put(TheRightOf, shapeElement, 10, FromRightOf, header)
  ->put(TheBottomOf, shapeElement, 10, FromBottomOf, body));
```

This usage of the anchor layout may not be intuitive on first sight, but when reading out the single `put` statements one can understand the relations between the contained form elements. With the help of a piece of paper and a pencil one should be able to quickly understand the basic layout of the form elements.
A way to improve this understanding, would be to define more `put` methods, acting as shorthands for two or more of the already existing `put` methods. One such method would e.g. allow you to specify, that two form elements should have the same width. This would be a shortcut for aligning first their

left edges, and then their right edges. Another method could allow you to set a relation between two corners of the form elements. This would be a shorthand for aligning the two elements on some horizontal edges, and additionally on some vertical edges.

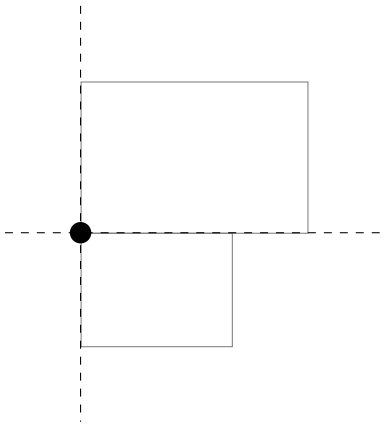


Figure 12: Illustration of how two anchor specifications could be replaced by one

Such shorthands for writing two anchor constraints in one line of code are also implemented in Qt's `QAnchorLayout` [6].

Next, we analyze the usage of the *grid layout* (listing 5):

```
auto header = (new GridLayoutElement())
        ->setHorizontalSpacing(3)->setColumnStretchFactor(3, 1)
        ->setVerticalAlignment(LayoutStyle::Alignment::Center)
        ->put(0, 0, /* icon visualization */))
        ->put(1, 0, /* init step visualization */))
        ->put(2, 0, /* condition visualization */))
        ->put(3, 0, /* update step visualization */));
```

The grid layout is much more intuitive than the anchor layout. The user can immediately see at which position of the grid any contained form element is placed. The additional settings like spacing and alignment should be clear to anyone. The only function, which might not be clear from the start, is the use of the stretch factor, as this is a more advanced concept inside a grid layout. But as a first impression, the user can see, how this grid layout arranges its components.

The last type of form element we can analyze from the perspective of the `VLoopStatement`, is the *item wrapper*:

```
item<NodeWrapper, I>(&I::condition_, [](I* v){return v->node()->condition();},
                                     [](I* v){return &v->style()->condition();})
```

The understanding of the item wrapper requires knowledge of the visualization framework underlying the API, but also knowledge of the `C++` language constructs *template*, *pointer to member*, and *lambda-functions*.
As explained in section 5.2, item wrappers are the leafs of the tree, defined by any form. Those leafs normally don't change upon modifying the layout. They stand for the visualization items that are visualized by the containing visualization, and only need to be changed if the implementation of the model, which this item is visualizing, changes.
A user, who only wants to make changes to the layout, does not need to understand how the item wrapper works, only what it stands for.
Nonetheless it would be desirable, to make the item wrappers more intuitive, though this will be a hard task. As the item wrapper is the link to the underlying framework, it cannot be avoided, that understanding the item wrapper will require some basic knowledge about that framework.

Issues In this paragraph, we discuss a limitation of the anchor layout, which may force the user to wrap certain form elements inside a grid layout.
The anchor layout creates the constraints for the library lpsolve [3] in a way (see appendix A), such that elements, that are not marked as stretchable, will never be stretched.
In the visualization of the loop statement, we want the shape, to have its right edge right of the header, and right of the body. We can only know where to put the right edge of the shape, if the right edges of header and body are at the same place, so we align header and body at their left and right, such that header and body will be resized to have the same width. The problem is, that the body is not stretchable in every case, resulting in the placement algorithm to fail.
The only solution to this problem so far, is to wrap the body inside a grid layout, which we can force to be stretchable, by adding a stretch factor to its only column.

There may be multiple possible solutions to this problem. One of them would be to retry the computation of the anchor layout, but this time handling all the contained form elements as stretchable. In this case it is not clear though, if the desired solution would be computed.
Another solution would be to give item wrappers an option to be stretchable regardless of the wrapped visualization item. In case the wrapped item were not stretchable, the item wrapper would still encompass all the available space. The wrapped visualization item could then be placed inside this space, according to some alignment options.

The second solution seems to more suitable, since it mimics the behavior of the wrapping grid layout: It tells the anchor layout, that this form element is stretchable, and aligns the wrapped visualization inside the available space.

7.2 Declarative Implementation of the Class Visualization - `VClass`

In this section, we will discuss the new declarative implementation of the class `VClass`. The points already discussed for the class `VLoopStatement` (section 7.1) are also valid here. We will therefore concentrate on different aspects here.
We will continue the discussion of how readable and intuitive the code appears (criteria 2 and 5), this time we will analyze the sequential layout. Then we will also discuss, how one can use the declarative API, but still access of the lower level mechanisms of the underlying framework (criterion 4).

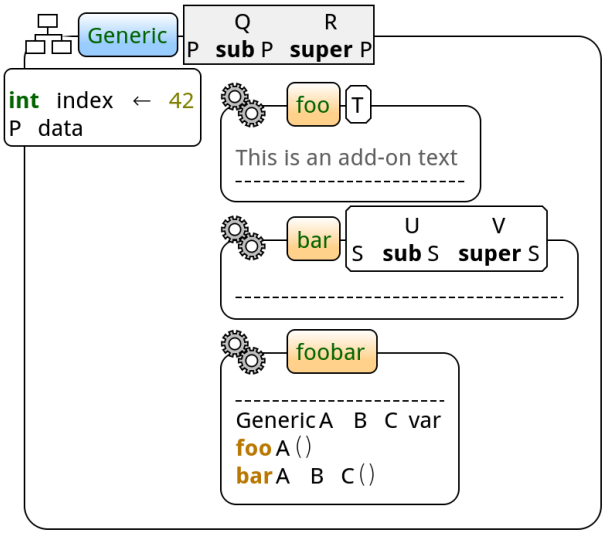


Figure 13: Visualization of the class

Readability and Intuitive Use In this paragraph, we discuss the usage of the sequential layout. Like the item wrapper, the sequential layout is a link to the underlying framework, and therefore requires some knowledge of it. But unlike the item wrapper, the sequential layout does not require

such a deep knowledge of C++ code constructs, only lambda functions are used. But when defining the lambda function, instead of getting a visualization item of type `VClass` as an argument, we get a more general pointer to an instance of class `Item`. In order to have access to the full functionality of `VClass`, we need to cast it.
It is not optimal for readability or intuitiveness, to force the user to do this cast. But in order to avoid it, we would need to introduce templates, and it is not clear which of the two options is more comfortable to use.

```
(new SequentialLayoutFormElement())
  ->setVertical()
  ->setListOfNodes([](Item* i){return (static_cast<VClass*>(i))->publicFields_;})
```

Like for the item wrapper, the sequential layout will only have to be changed, once the implementation of the model it is visualizing has changed. A user only changing the layout of the containing visualization item, will not have to touch the definition of the sequential layout.

Mixing the Declarative API and the Underlying Visualization Framework The visualization of the *body* of the class uses a special visualization item, that can not be updated in the standard way from the model it visualizes. This means, that using just an item wrapper, will not result in the desired behavior of this item.
However, we can still use the declarative API to arrange the layout, and only treat the visualization of the body differently.

First, we need to define an item wrapper for the body. This allows us position the body inside a form, just like any other visualization. In addition to that, it also sets the style of the body automatically, and we do not need to destroy the body visualization in the destructor.

```
item<PositionLayout, I>(&I::body_, [](I* v){return &v->style()->body();}))
```

We need to initialize the visualization in the constructor:

```
body_ = new PositionLayout(this, &style->body());
```

Note, that the item wrapper will never attempt to create the body visualization itself, because it is already initialized.
In contrast to a traditional visualization using the declarative API, we still need to override the `determineChildren` method. In there, we can call the required non-standard update procedures, and call the `DeclarativeItem`'s `determineChildren` method *afterwards*, since this method may rely on the state of the body visualization item.

```
1  void VClass::determineChildren()
2  {
3    // manually update the body item
4    if (body_->needsUpdate() == FullUpdate) body_->clear(true);
5    QList<Model::Node*> bodyItems = node()->classes()->nodes().toList();
6    bodyItems << node()->methods()->nodes().toList();
7    body_->synchronizeWithNodes( bodyItems, renderer());;
8
9    // call determineChildren of super class
10   BaseItemType::determineChildren();
11
12   /* ... */
13 }
```

Listing 7: Overview of `VClass::determineChildren`

7.3 Declarative Implementation of the If-Statement Visualization

In this section, we will discuss the new declarative implementation of the class `VIfStatement`. The points discussed for the classes `VLoopStatement` (section 7.1) and `VClass` (section 7.2) are also valid

for the if-statement. In this section we will concentrate on an issue with code repetition.

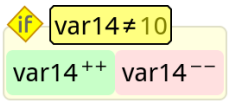


Figure 14: Visualization of the if-statement

Issues The visualization of the if-statement has two forms: the first where the then- and the else-branch are besides each other, and the second where one branch is below the other.

```
/* Define form elements */

// Form 0: then and else branch arranged horizontally
auto contentElement = (new GridLayoutFormElement())/* ... */
  ->put(0, 0, thenBranch)->put(1, 0, elseBranch);

addForm((new AnchorLayoutFormElement())
  ->put(TheLeftOf, header, AtLeftOf, contentElement)
  ->put(TheLeftOf, shapeElement, 2, FromLeftOf, contentElement)
  ->put(TheRightOf, header, AtRightOf, contentElement)
  ->put(TheRightOf, shapeElement, 2, FromRightOf, contentElement)
  ->put(TheBottomOf, header, 3, FromTopOf, contentElement)
  ->put(TheTopOf, shapeElement, AtCenterOf, header)
  ->put(TheBottomOf, shapeElement, 2, FromBottomOf, contentElement));

// Form 1: then and else branch arranged vertically
contentElement = (new GridLayoutFormElement())/* ... */
  ->put(0, 0, thenBranch)->put(0, 1, elseBranch);

addForm((new AnchorLayoutFormElement())
  ->put(TheLeftOf, header, AtLeftOf, contentElement)
  ->put(TheLeftOf, shapeElement, 2, FromLeftOf, contentElement)
  ->put(TheRightOf, header, AtRightOf, contentElement)
  ->put(TheRightOf, shapeElement, 2, FromRightOf, contentElement)
  ->put(TheBottomOf, header, 3, FromTopOf, contentElement)
  ->put(TheTopOf, shapeElement, AtCenterOf, header)
  ->put(TheBottomOf, shapeElement, 2, FromBottomOf, contentElement));
```

Listing 8: Overview of `VIfStatement::initializeForms`

The calls to the two `addForm` methods are almost the same. The difference is that in one case the `contentElement` points to a grid, where then and else branch are arranged horizontally, while in the other it points to a grid, where the branches are arranged vertically.
There is no mechanism in the declarative API, allowing the reuse of fully specified form elements, with just some of the contained form elements replaced by others.

This problem could be solved, by adding two new methods to the anchor layout. The first would be a second constructor, that takes an already existing anchor layout and copies its definition into a new anchor layout. The second method would allow the user to replace a form element inside an anchor layout by a different one.
Using this solution, the second definition of the anchor layout could be replaced by copying the first anchor layout, and then replacing the old content element with the new one, and thus avoiding the repetition.

8 Conclusion and Future Work

We have discussed how our new declarative API for visualizations in Envision can simplify the addition an modification of visualizations.
We have shown that with the declarative API, the code to define visualizations can be considerably shorter, than the code resulting in the same visual representation, but using the underlying framework directly.
We have also taken care, that code written using the declarative API is readable. A developer not knowing the API will be able to understand how the components of the visualization item will be arranged. Although to understand what exactly those components are, and how they are defined, requires more knowledge of the underlying visualization framework.
We have also demonstrated, that one can take advantage of the new declarative API, and still have low level control over the contained items. This allows the implementation of very complex visualizations, where one can still take advantage of the declarative API for all the easier and more traditional aspects of the visualization.

Still, there remains some work to be done on the declarative API for visualizations in Envision. We discussed the following issues in section 7:

- Handling of non-stretchable visualization items inside the anchor layout
- Avoid repetitions when defining two very similar forms
- Add shorthands the combination of multiple `put` methods in the anchor layout
- Improve the readability of the item wrappers

Nonetheless, using the declarative API for visualizations, we can now experiment more easily with the layout of visualizations. Also, with the existence of the new declarative API, we are much nearer goal of letting software developers write their own visualizations for Envision, and letting them visualize their own libraries and DSLs.

Appendix A The Anchor Layout as a Linear Programming Problem

In this section we discuss how to translate the positioning problem in the anchor layout into a linear programming problem. This translation allows us to use a linear programming solver to do the positioning of the elements. For our implementation we use the library lpsolve [3].

A Linear Programming Problem consists of a series of in-equations, a linear objective function to find an optimal solution, and a lower bound. It can be mathematically expressed as

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b} \\ \text{and} & \mathbf{x} \geq \mathbf{0}, \end{array}$$

where $\mathbf{x}$ represents the vector of variables, $\mathbf{c}$ and $\mathbf{b}$ are vectors of known coefficients, and A is a matrix of known coefficients.

Let $E_1, E_2, \ldots, E_n$ be the elements that are to be arranged. We consider the horizontal and vertical axes separately. Let therefore $A_1, A_2, \ldots, A_m$ be the anchors for the axis we are currently considering. An anchor A_i consists of two elements, two relative axis positions, and an offset. See figure 15 for a graphical clarification of what those components mean. We write the anchor as $A_i = (E_{i,1}, r_{i,1}, E_{i,2}, r_{i,2}, o_i)$.

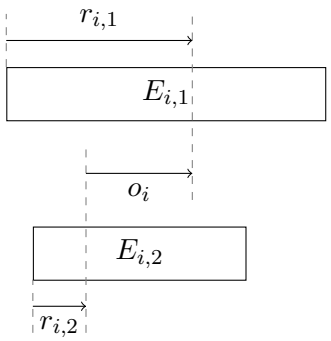


Figure 15: The components of an anchor $A_i = (E_{i,1}, r_{i,1}, E_{i,2}, r_{i,2}, o_i)$ on the horizontal axis

Variables We define two variables for each element E_i, namely $V_{i,start}$ and $V_{i,end}$, where *start* means the left/top edge position of the element and *end* the right/bottom edge position respectively. Therefore we have $2n$ variables in our linear programming problem.

Objective Function The objective function is defined to maximize a linear function, but we will define our objective function in terms of minimizing. Translating a minimizing problem to a maximizing problem is just a matter of flipping the sign.
We want to minimize the sizes of all our elements $E_1, E_2, \ldots, E_n$, so our objective function is:

$$\min \left(\sum_{i=1}^{n} V_{i,end} - V_{i,start} \right)$$

Constraints Our linear programming problem contains two sets of constraints. The first one consists of all the constraints imposing bounds on the sizes of the elements $E_1, E_2, \ldots, E_n$, while the second contains constraints for each anchor $A_1, A_2, \ldots, A_m$.

Similar to the objective function, the constraints are normally formulated in the form of $A\mathbf{x} \leq \mathbf{b}$, but we can also use $\geq$, which can be obtained from $\leq$ by flipping the signs on one side. Additionally we can also define equalities, which can be obtained by adding two inequalities to the set of constraints.

Element Constraints For each element $E_1, E_2, \ldots, E_n$ we add a lower bound for their sizes to the set of constraint. We assume that the minimum size of every element is known at this point in the computation. If an element E_i is not stretchable, we additionally add the same value as upper bound on it's size.
For every element $E_1, E_2, \ldots, E_n$ we add a constraint

$$V_{i,end} - V_{i,start} \geq minimum\_size(E_i),$$

but if the element is not stretchable, we instead add the constraint

$$V_{i,end} - V_{i,start} = minimum\_size(E_i).$$

Anchor Constraints For each anchor $A_1, A_2, \ldots, A_m$ we add an equality to the set of constraints. If we consider figure 15 again, we get the following for an anchor $A_i = (E_{i,1}, r_{i,1}, E_{i,2}, r_{i,2}, o_i)$.

$$\begin{aligned}
V_{E_{i,1},start} + r_{i,1} \cdot size(E_{i,1}) &= V_{E_{i,2},start} + r_{i,2} \cdot size(E_{i,2}) + o_i \\
V_{E_{i,1},start} + r_{i,1} \cdot (V_{E_{i,1},end} - V_{E_{i,1},start}) &= V_{E_{i,2},start} + r_{i,2} \cdot (V_{E_{i,2},end} - V_{E_{i,2},start}) + o_i \\
(1 - r_{i,1}) \cdot V_{E_{i,1},start} + r_{i,1} V_{E_{i,1},end} &= (1 - r_{i,2}) \cdot V_{E_{i,2},start} + r_{i,2} \cdot V_{E_{i,2},end} + o_i \\
\iff \quad (1 - r_{i,1}) \cdot V_{E_{i,1},start} + r_{i,1} V_{E_{i,1},end} - (1 - r_{i,2}) \cdot V_{E_{i,2},start} - r_{i,2} \cdot V_{E_{i,2},end} &= o_i
\end{aligned}$$

We can now give those $O(m+n)$ constraints and the objective function to the solver and retrieve the values $2n$ variables, representing the start and end points of the elements.

References

[1] gtk layout containers. http://developer.gnome.org/gtk3/stable/LayoutContainers.html.

[2] The little manual of api design. http://www4.in.tum.de/~blanchet/api-design.pdf.

[3] lpsolve mixed integer linear programming solver. http://lpsolve.sourceforge.net/5.5/.

[4] Qt api design principles. http://qt-project.org/wiki/API-Design-Principles.

[5] The Qt GUI module. http://qt-project.org/doc/qt-4.8/qtgui.html.

[6] Qt Widgets. http://doc.qt.digia.com/qt/widgets-and-layouts.html.

[7] Tikz manual. http://paws.wcu.edu/tsfoguel/tikzpgfmanual.pdf.

[8] Tkinter geometry managers. http://effbot.org/zone/tkinter-geometry.htm.

[9] wxWidgets' sizers. http://docs.wxwidgets.org/2.8/wx_sizeroverview.html.

[10] Dimitar Asenov. Design and implementation of Envision - a visual programming system. Master's thesis, ETH Zürich, 2010.