Semantic Zoom Support for Envision

Bachelor Thesis Report

Patrick Lüthi

Supervised by Dimitar Asenov, Prof. Dr. Peter Müller
ETH Zürich

May 22, 2014

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Table of Contents

1 Introduction

Envision is an integrated development environment (IDE) for object-oriented programming written in C++. Its development started during the master's thesis of Dimitar Asenov [1]. In contrast to most traditional IDEs Envision represents programs not only in text form but also uses various forms of visualization. Envision aims to help users increase their productivity by augmenting the code using visual elements. In this project we are adding semantic zoom support for Envision to further increase user productivity by improving code comprehension and navigation.

1.1 Motivation

A large object-oriented program can contain a huge amount of code distributed in many projects, modules and classes. The visualizations of all these elements need a certain amount of space and the available space of a screen is limited depending on resolution and screen size. Envision uses a single surface to display an entire program. For navigation purposes it is therefore important to be able to zoom in and out. It is common practice to use zoom out to fit more visual elements into a screen area. This will show more elements on a given screen and the elements themselves are displayed smaller but all the information is retained. This common notion of zoom by just geometrically zooming items is what we will refer to as geometric zoom. In practice screens have a fixed resolution and human visual perception is limited leading to an inevitable loss of information. Figure 1.1 shows how the labels of books become unreadable when geometrically zooming out.

Most of the time when geometrically zooming out the user is not interested in the elements that are getting too small to recognize. The user basically requests a higher level overview and is interested in information on that level instead. This inevitably raises the question: Why not display what the user is actually interested in instead?

Displaying something in a different way to show the most relevant information about it depending on a given situation is exactly what semantic zoom is all about.

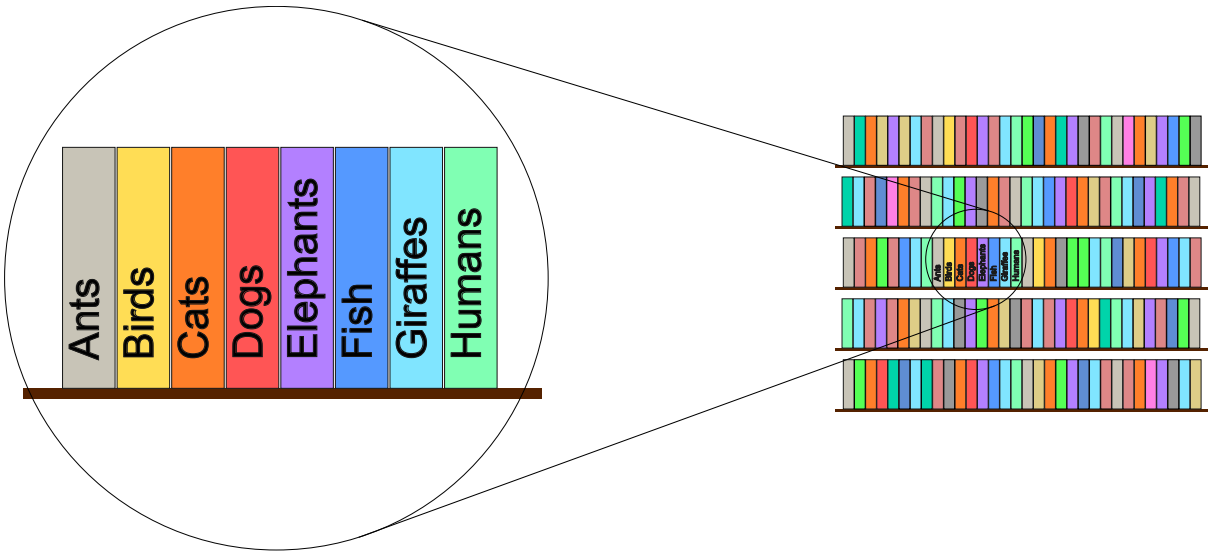

Figure 1.1: An example of geometric zoom. The captions of books become unreadable when zooming out too much.

2 Semantic Zoom

This chapter gives a brief introduction to semantic zoom. It compares semantic zoom to geometric zoom and shows why semantic zoom is useful. Additionally, we discuss common challenges that arise while working with semantic zoom.

2.1 Semantic Zoom vs. Geometric Zoom

Most people are familiar with the notion of geometric zoom. When using geometric zoom the size of all visual elements is scaled by the same amount but all the elements and their structure are retained as illustrated in Figure 1.1.

Semantic zoom on the other hand does not only vary the size of visual elements, but the amount of details shown. This may result in displaying elements in a different way, or not at all, depending on the semantic zoom level. Figure 2.1 illustrates semantic zoom using the bookshelf example used in Figure 1.1.

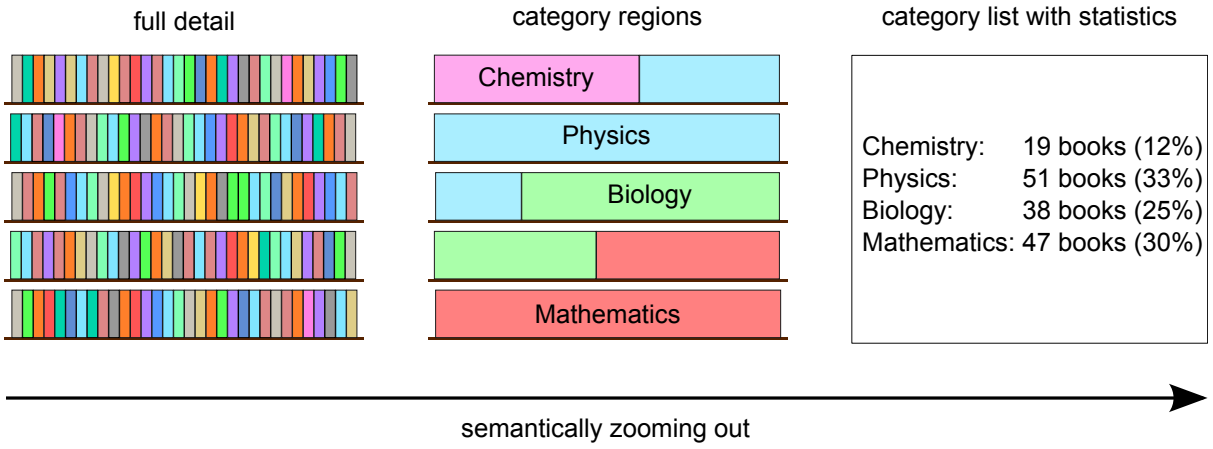

Figure 2.1: An example of different semantic zoom levels.

2.2 Challenges

In this section we are going to talk about issues that are common to all semantic zoom implementations independent of Envision's application area. Specific design choices and trade-offs in the context of Envision are discussed in the chapter Design.

2.2.1 Smooth transitions between semantic zoom levels

Semantic zoom is supposed to help the user work more efficiently. However, changing the looks of visual objects too abruptly can be confusing. The user should be able to easily relate between the appearance of an element displayed on two adjacent semantic zoom levels. This ensures that the user does not lose context information when switching the semantic zoom level.

2.2.2 Avoiding ambiguity of visualizations

It is important that visualizations of semantic zoom levels are unambiguous. If an element looks the same on a certain semantic zoom level as on another semantic zoom level in special situations the user will not be able to distinguish between the two possibilities.

For example: Assume we are visualizing the book shelf from Figure 2.1 and abstracting it on one semantic zoom level just shows the book shelf and omits the books it contains. On another semantic zoom level the whole book shelf with its books is displayed in full detail. Under these circumstances the user is unable to see the difference between a book shelf displayed on the first semantic zoom level and an empty book shelf on the second semantic zoom level.

2.2.3 Geometric consistency

When switching between semantic zoom levels it is important to preserve a certain degree of geometric consistency. For example if an element A is displayed left of element B on one semantic zoom level then A should also be on the left of B on a different semantic zoom level. Additional to the relative location of elements it is important to have a stable relation between the areas used by an item displayed on two different semantic zoom levels.

Geometric consistency forms the bridge between semantic and geometric zoom. It ensures that the user is easily able to relate between the current appearance of a scene and the appearance resulting from modifications to semantic or geometric scale.

3 Design

This chapter describes the design process as well as the final design without going into implementation details.

3.1 Visualizations in Envision

Programs written in an object-oriented language naturally exhibit a tree like structure. An example is shown in Figure 3.1. Note that Envision uses a structural component called `module` which is equivalent to a package or namespace.

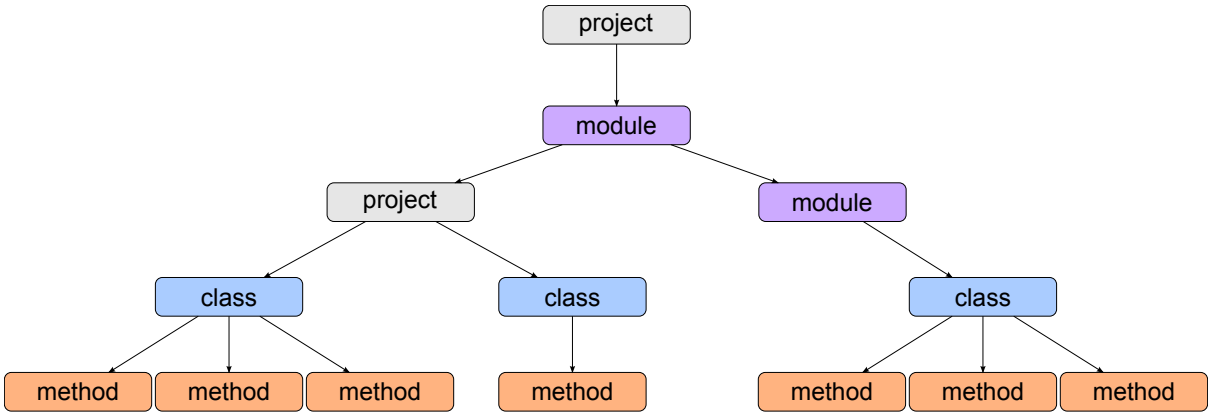

Figure 3.1: An example tree structure of an object-oriented program.

Envision stores the structure of programs in a model tree. The nodes of the tree represent program components like projects, modules, classes, methods or statements. A node may or may not be visualized and can switch visualizations depending on its content, context or other environmental conditions. The visualizations of nodes are called items organized in trees which closely resemble the structure of the model tree.

3.2 Using visualizations to achieve semantic zoom

We decided to have discrete semantic zoom levels representing different levels of abstraction (varying the amount of details shown). This means that we need items to change their appearance depending on the semantic zoom level.

In Envision the appearance of an item is determined by the visualization used. Envision supports the creation of custom visualizations. Therefore we can create all the visualizations needed for a semantic zoom level. By defining which custom visualizations belong to which semantic zoom level it is possible to change the appearance of items depending on the active semantic zoom level. For this to work Envision has to be able to choose an appropriate visualization under given circumstances. This is discussed in the next section.

3.3 Choosing a suitable visualization

Envision already has a mechanism to choose visualizations for a given item. The mechanism distinguishes between an item's *type* and a property called *purpose* to choose a suitable visualization for an item. The *purpose* is a visualization parameter describing what visual functionality the visualization exhibits. Currently Envision supports a default *purpose* and a control-flow *purpose*.

Our design of semantic zoom adds an additional property called *semantic zoom level* to extend the visualization choice mechanism. When rendering an item the visualization choice will now depend on the *semantic zoom level* as well.

If we are trying to choose a visualization given a triple of *type*, *purpose* and a *semantic zoom level* and there exists an exact match (a visualization suited for exactly those parameter values) it is used directly. However, once we are not able to match all requirements the choice gets more difficult. In that case we are interested in an existing visualization which matches the requested parameter values as closely as possible. In general there is no optimal solution to this problem.

For example if there was a visualization matching *type* and *purpose* and a different visualization matching *type* and *semantic zoom level* but no visualization matching all 3 parameters then it is not clear which of the two partially matching visualizations is better.

We decided to use a simple priority based approach. The parameters have different priorities. The type of an item was chosen to be most influential followed by the requested semantic zoom level and lastly the visualization purpose. This prioritization disambiguates the problem of choosing a suitable visualization.

For example: The visualization choice strategy that prioritizes item type over semantic zoom level over purpose first checks whether a visualization for the exact values of all 3 parameters exists. If not it will proceed in looking for a visualization that at least matches the type and semantic zoom level parameter. If there is still no match it will look for a visualization based on the type and the purpose parameter. Lastly if no visualization was found so far only the type is considered.

As this approach is very simple the visualization choice will most likely become more elaborate in the future. We decided to integrate this functionality in a way that allows it to be easily extended in the future.

3.4 Individual semantic zoom level

One could have a global semantic zoom level that determines the semantic zoom level for all items. However, there are scenarios where semantically zooming only individual items is desirable. In the context of Envision one could for example be working on one part of a program. The programmer might be interested in using other parts of the project in their own work. The programmer most likely is only interested in the public interface of those parts providing all the information needed to use them in the new code. In this example the programmer would most likely prefer seeing the necessary features of the other program parts on a different semantic zoom level than the part that is currently under construction. Therefore we decided to support individual semantic zoom levels for each item.

3.5 Arrangement of visualizations

When using a different visualization for an item to achieve semantic zoom, the visualization commonly shows less information on a coarser zoom level. As a result the visualization of an item after abstraction (showing less details about it) is usually going to be smaller compared to the size it takes to show the same item in full detail. The smaller visualization retains the position of its original representation and the freed space may stay unused, thus creating a gap between the item and its neighbours like illustrated in Figure 3.2.

In the following subsections we discuss some approaches to rearrange the items to make better use of the available space.

3.5.1 Move visualizations to the centre of the viewing region

The idea behind this first approach was to try and tackle the larger distance between visualizations directly. Based on the current viewport (the viewport is the viewing region) all visualizations in the immediate area in and around it could be moved closer together.

As a first concrete approach we considered a star-shaped visualization movement algorithm. The algorithm would go through all visualizations ordered by their ascending proximity to the centre of the viewport and move each visualization as close as possible to this centre point without overlapping different visualizations. The visualizations would be moved along straight lines joining at the centre point of the viewport resulting in a star-shaped like illustrated in Figure 3.3.

We found this approach to have one severe drawback: The resulting view can change significantly depending on the centre point of the viewport. Even if the user shifts the

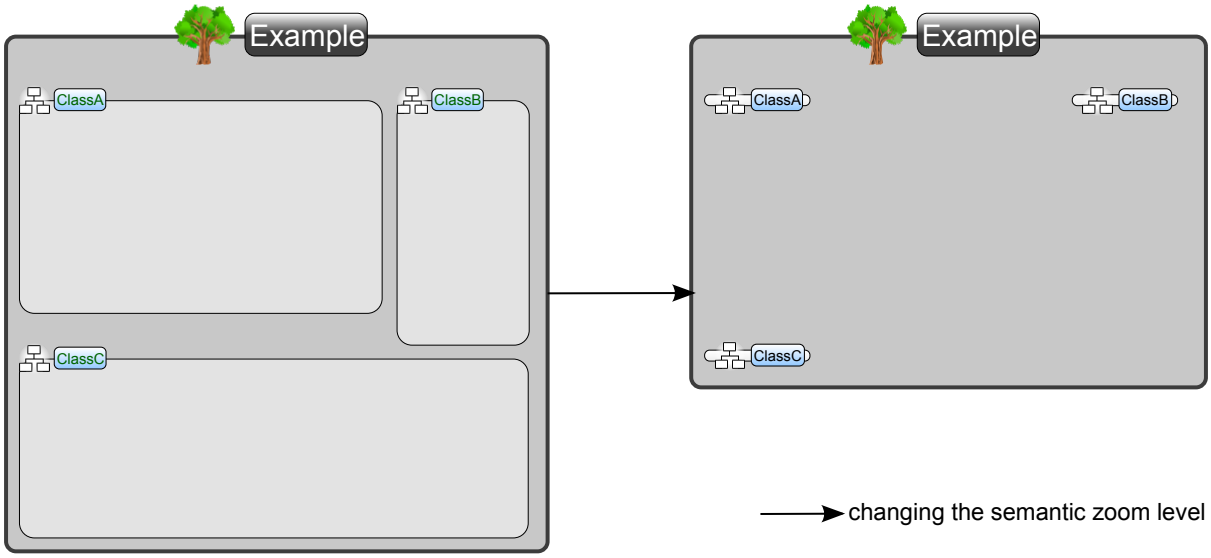

Figure 3.2: Space difference between abstracted items.

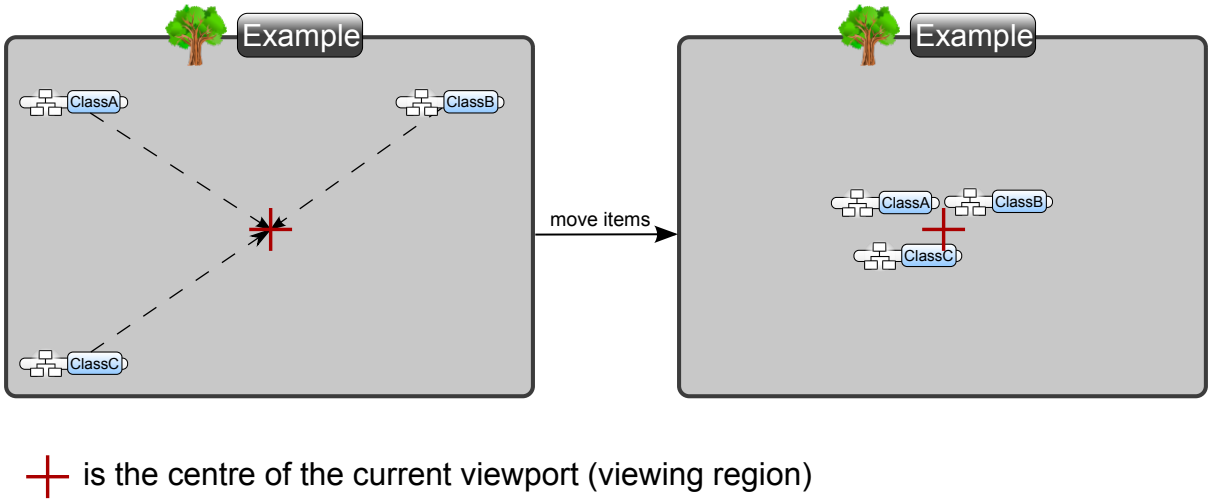

Figure 3.3: Moving items closer to viewport centre.

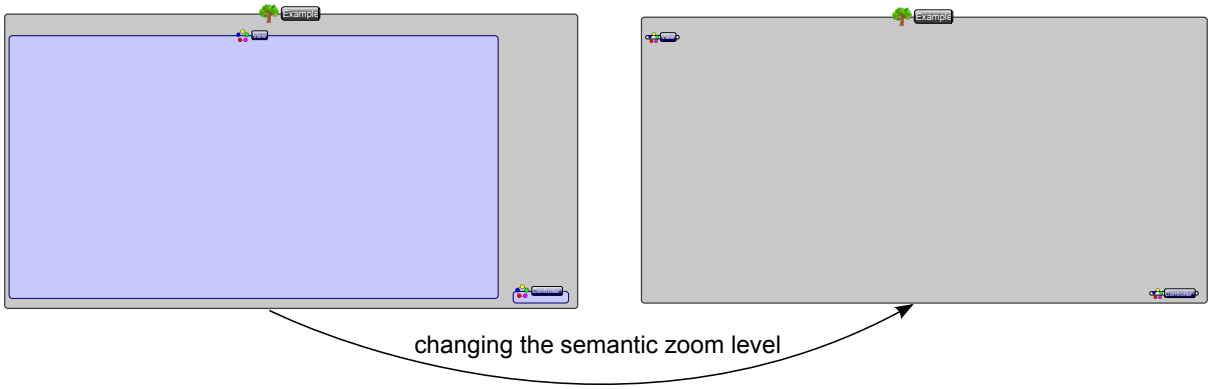

Figure 3.4: Issue of an item becoming very small and not more readable after changing the semantic zoom level.

viewport only slightly in one direction the appearance of the scene can change in a very unintuitive way. For the sake of overview and comprehensibility we argue that shifting the viewport should not lead to a completely different arrangement of visualization items. One could maybe relax the constraint and allow small, easily comprehensible arrangement differences. However, the proposed approach can not guarantee to only slightly change the arrangement of items.

Another issue we identified was the difference in size between an item visualized on different semantic zoom levels. Figure 3.4 illustrates how a huge module ends up to be very small after changing the semantic zoom level even though it would have enough space to be displayed much larger. Based on this observation we decided that not only the visualization and position of an item has to be changed during the rearrangement but its scale as well. This lead to the design idea described in the next subsection.

3.5.2 Using only the area used when shown in full detail

After the initial approach we decided to design an arrangement algorithm that is able to scale and move items while being careful about the area the transformed items would occupy. The following approach was an intermediate step in the design process. The final arrangement algorithm we used is described in the next subsection.

One of the most common uses of semantic zoom currently is encountered when using online maps. The issue we are facing is not encountered there as a map is very geometrical. What that means in general is that no matter on what geometric or semantic scale you are looking at a map piece, all information related to it will always be shown inside its own region of the total space. A region of a map owns a certain part of the total space and will always fully occupy it. However, in Envision there are no limitations to the area used by a visualization. The lack of geometric consistency between the area used by an item visualized on different semantic zoom levels is a problem. Studying how semantic zoom is used for existing online maps and why the problem we are facing is not encountered there inspired us to develop the idea of area ownership.

Area ownership makes it possible to usefully constraint the used area even if we relax the property about an item using always the whole area it owns. Each item owns a certain

area equal to the area needed when it is shown in full detail. The item would then be free to move inside this area and take any shape or size as long as the transformed items area is still contained in the area it owns.

An item can be scaled as much as possible such that its proportions are not distorted and the area the scaled item takes is still contained in its owned area. We decided it makes sense to not scale an item beyond a certain maximum scale. That maximum scale can change depending on the current geometric scale and intuitively enforces that its perceived scale is not larger than 1. The perceived scale is obtained by multiplying the geometric scale used with the scale of the item (note that the item's scale is the actual scale of the item times the parent item's scale). For example if the whole scene is currently shown using a geometric scale of 0.5 then the maximum scale an item can have is 2 as that is the maximum number that will lead to a total perceived scale of 1 or less. After scaling the item would be positioned according to the first approach but such that it still remains completely inside its owned area as illustrated in Figure 3.5.

Due to the way Envision works this approach could not be realised. To be able to use the proposed arrangement procedure we have to be able to get the area an item owns. A reasonable choice for the owned area would be the extents the item has if shown in full detail. The only way to obtain that area is to render the item in full detail. This would mean that we have to render each item that should not be shown in full detail twice. This was considered to be an unacceptable complication and therefore a different solution, as described in the next subsection, was developed.

3.5.3 Equal growth around the original area

Knowing the limitations of the framework and the properties we want the transformed items to have, the next step was to consider what can be done without having to visualize an item in full detail first. This in turn seemed to exclude the possibility of working with area ownership. However, working on the previous idea revealed an interesting correlation between the two kinds of transformations that we want to apply to items which should be rearranged. Using the idea of items being scaled dynamically depending on the geometric scale of the scene automatically takes care of the distance between different items as well. The smaller the geometric scale gets, the larger visualizations can potentially become thus leading to the empty space between items being filled by the growing items around it. We therefore decided to not have direct movement transformation. It was left to decide how the scaling should happen such that the transformed item has the most desirable properties.

Only the area of the current item visualization is known at this point. We decided to scale the item in such a way that the area an item takes on a geometric scale of 1 is always contained in the area after transformation. This guarantees that, when geometrically zooming, the relative locations of items are retained. Furthermore, this allows the transformed item to be moved by the user such that the item's new position looks reasonable on other semantic zoom levels as well. The concrete implementation of the proposed arrangement algorithm is discussed in the next chapter.

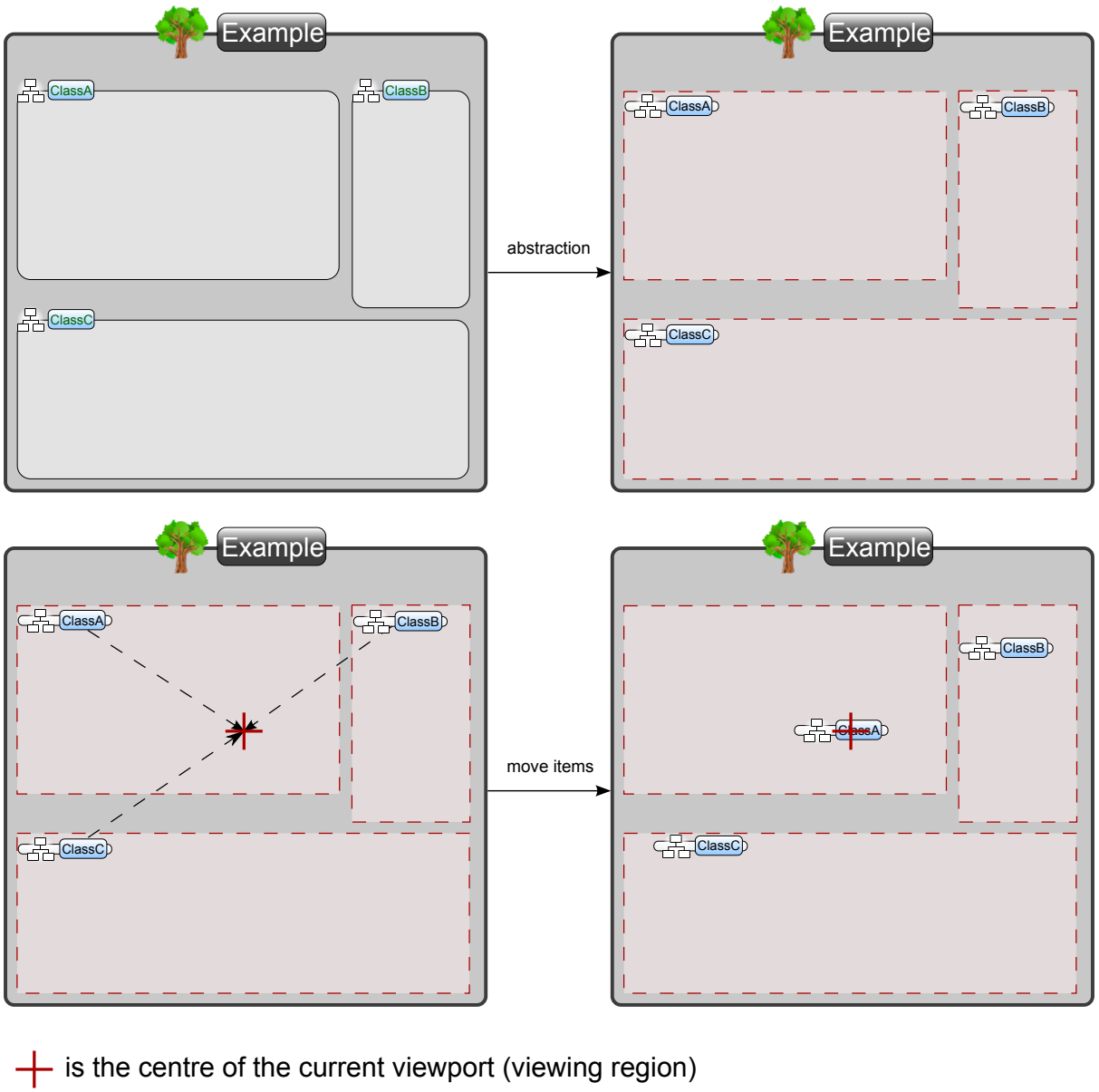

Figure 3.5: Arranging items with area ownership.

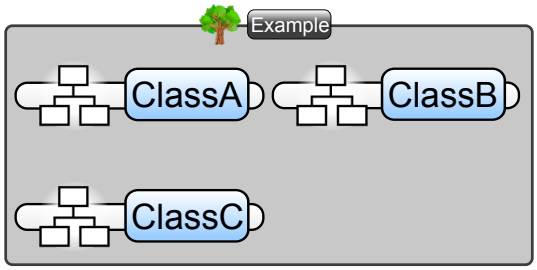

Figure 3.6: Without header scaling.

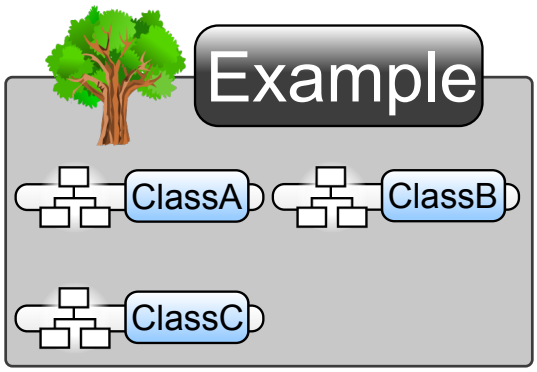

Figure 3.7: Example from Figure 3.6 with header scaling.

3.6 Header scaling

When using the arrangement algorithm at this point and geometrically zooming out on the running example the scene will eventually end up looking like shown in Figure 3.6.

Note that the structural children of the project got scaled up while the header of the project became much smaller. In this special case the project has no parent visualization. This means it has unlimited space around it. We decided to scale the header of items without a parent item in such a way that the total perceived scale of the header is always 1.

The example from Figure 3.6 with header scaling is illustrated in Figure 3.7.

3.7 Automatically changing the semantic zoom level

While it is very useful to be able to choose an item's semantic zoom level manually it makes sense to let the semantic zoom level of an item also change automatically if the user did not explicitly put a desired semantic zoom level for it. The whole idea behind semantic zoom is to display something different if the current visualization becomes inappropriate. A visualization becomes unsuitable especially if the details of it cannot be recognized any more after getting too small due to geometrically zooming out.

We decided to add an automatic mechanism, a semantic/geometric zoom mix, that

switches the semantic zoom level of items that become too small when geometrically zooming. The mechanism switches between two semantic zoom levels for all projects, modules, classes and methods. One is the fully detailed level and the other one is the declaration abstraction introduced in this project (see 4.7.2).

If an item's perceived scale falls below a certain threshold (we used 0.8) it switches from the most detailed semantic zoom level to the declaration abstraction semantic zoom level. When an item gets abstracted the geometric scale at that point is saved. The saved value is used to switch the item's semantic zoom level back to full detail when geometrically zooming in.

However, the location where we want to save the geometric zoom scales on which the items got abstracted can get lost if some parent item changes its visualization. In this case we have to come up with some kind of estimation on when to switch items to fully detailed semantic zoom level again. We decided to do this based on the unused space around an item. If there is enough space for the fully detailed visualization to have a reasonable perceived scale (ideally the threshold chosen in the last paragraph) we want to switch to the full detail semantic zoom level. So essentially we are looking for an estimation of the area an item's fully detailed visualization takes. We took the simple approach of using a just constant size as an estimation. However, this estimation was put in a separate function such that it can easily be improved in future work.

4 Implementation

4.1 Storing the semantic zoom level

We added an extra field for each item to hold its semantic zoom level. The value refers directly to a specific semantic zoom level or alternatively takes on the value `-1`. The special value `-1` is used to create a relative dependency on the parent item's semantic zoom level. The semantic zoom level property of the root item is never negative and can be used as a single point of modification to change the semantic zoom level for the whole program.

We also added a map that holds the semantic zoom levels of child nodes.

4.2 Visualization choice

The class `ModelRenderer` was extended to be able to switch between different visualization choice strategies. We introduced a field to control the visualization choice strategy. Two concrete strategies were implemented. One prioritizes type over purpose over semantic zoom level and the other prioritizes type over semantic zoom level over purpose. The second one is used in the current implementation by default.

4.3 Introducing a semantic zoom level

A semantic zoom level essentially consists of a collection of visualizations to be used when showing the scene on that semantic zoom level. So we provide a functionality to register a semantic zoom level and link it to a unique identifier. That identifier is then used to add visualizations to the semantic zoom level.

4.3.1 Registering a semantic zoom level

One can register a new semantic zoom level using the feature `registerSemanticZoomLevel` of `ModelRenderer`. The method accepts two arguments. The first argument is a string representing the name of the new semantic zoom level and is used to refer to this semantic zoom level after it was registered. The second argument takes an integer. This integer determines its place in the hierarchy of all registered semantic zoom levels.

4.3.2 Adding visualizations to a semantic zoom level

As the semantic zoom level was designed to be another parameter determining visualization choice we extended the existing visualization registration methods of `ModelRenderer` to accept a semantic zoom level identifier as well. The semantic zoom level identifier is chosen by the programmer when registering the associated semantic zoom level.

4.4 Arrangement of visualizations

This section describes the concrete implementation of the used arrangement algorithm. The arrangement algorithm was integrated into the class `PositionLayout` because all items we want to change the visualizations for when switching the semantic zoom level happen to be direct children of `PositionLayouts`. The algorithm works as an additional modification step to the existing implementation of `PositionLayout` and is located at the end of the `updateGeometry` method.

4.4.1 Scaling items

The arrangement algorithm tries to expand the area that items which are children of a `PositionLayout` can occupy. If an item has more space available it can be scaled larger in an attempt to increase its visibility. The items should be scaled in such a way that they keep their proportions exactly the same. Additionally, the arrangement algorithm will only increase the scale of items. Items have a scale of 1 before the rearrangement takes place.

As we only need to calculate the area an item can use when performing the rearrangement, we decided to optimize the process and work directly with the area of the item. In the design section we pointed out that we want the original area (the area the item takes without getting scaled) always to be completely contained in the scaled area. One further design choice was to try to let this original area be located as central in the scaled area as possible.

We distinguish between 4 different directions in which an item's area can potentially be expanded as illustrated in Figure 4.1. The area is transformed in such a way that the larger increase in one direction is equal to a certain predefined constant. This constant

can be used to control the speed with which the item gets expanded. A higher constant will lead to faster convergence but a less optimal solution in return.

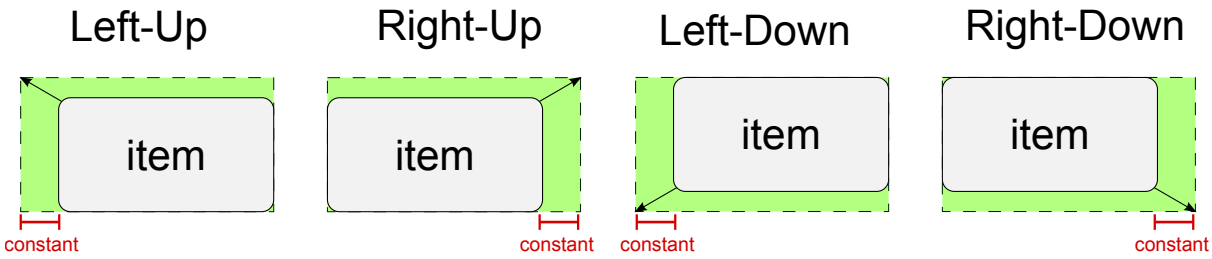

Figure 4.1: Item expansion directions.

4.4.2 Arrangement algorithm

Code 4.1: Arrangement algorithm.

```
1  const float EXPANDING_STEP
2
3  foreach (item)
4  {
5      item.expandingDirections = [LeftUp, RightUp, LeftDown, RightDown]
6      item.area = non-scaled non-moved area of the current visualization
7      item.widthHeightRatio = item.width / item.height
8  }
9
10 while (there are still items to expand)
11 {
12     foreach (item)
13         while (the item can be expanded further)
14         {
15             switch (first element of item.expandingDirections)
16             {
17             case LeftUp:
18                 expansionLeft = expansionUp = EXPANDING_STEP;
19                 complementDirection = RightDown;
20             case RightUp:
21                 expansionRight = expansionUp = EXPANDING_STEP;
22                 complementDirection = LeftDown;
23             case LeftDown:
24                 expansionRight = expansionDown = EXPANDING_STEP;
25                 complementDirection = LeftUp;
26             case RightDown:
27                 expansionLeft = expansionDown = EXPANDING_STEP;
28                 complementDirection = RightUp;
29             }
30
31             if (item.widthHeightRatio > 1)
32             {
33                 expansionUp /= item.widthHeightRatio
34                 expansionDown /= item.widthHeightRatio
```

```
35              }
36              else
37              {
38                  expansionRight *= item.widthHeightRatio
39                  expansionLeft *= item.widthHeightRatio
40              }
41
42              item.area = item.area expanded by expansionLeft/Right/Up/Down;
43
44              if (item.area does not collide with any other item area &&
45                  item.area does not collide with any border)
46              {
47                  scale the item using the space of the expanded area
48
49                  if (item has now a perceived scale of 1)
50                      set item to not need any further expansions
51                  else
52                  {
53                      put the current expanding direction at the end of item.
                            expandingDirections
54
55                      move the complementDirection to the front of item.
                            expandingDirections
56                  }
57
58                  break
59              }
60              else
61              {
62                  revert the changes to item.area
63              }
64          }
65  }
```

The algorithm starts with the information about the extents of the `PositionLayout`'s inner size (the size of the area usable by children of the `PositionLayout`), the position of the items and each item visualizations extents (note that this is the size of the current visualization of an item and not the fully detailed one).

During initialization the algorithm associates a queue of directions to each item indicating in which directions an item can still try to expand itself. It then iterates over all child items of the `PositionLayout` and keeps doing so as long as some modification has taken place. As the available space for items to expand to is limited and does not change during the iteration process it will always terminate after a finite amount of iteration steps.

In each iteration step the algorithm checks for each item whether it can be further expanded. An item can be further expanded if and only if the queue of possible directions it can grow to is not empty. If an item can be further expanded it dequeues the first possible direction and transforms the area associated to this item accordingly. The transformed area is then checked against all other item areas as well as the boundaries set by the parent `PositionLayout`. If the transformed area does not overlap with any other area and is still fully contained in the parent `PositionLayout` the items transformation is committed. However, if that was not the case the transformation is reverted and the

algorithm tries the next possible direction.

When an item was successfully expanded in one direction the used direction gets enqueued at the end of the queue and the complement direction, if existent, is pushed to the front the queue. This results in a direction being reused only after trying to expand in all other available directions first. The special case with the complement direction being pushed to the front tries to ensure that the growing is as balanced as possible. This way the original area before transformation gets centralized as much as possible in the transformed area.

4.5 Header scaling

The implementation for scaling the header of items without a parent item was put directly into the update procedure of the most common top level items: `Project` and `Module`. In the update procedure we check whether the item has no parent visualization and if so we calculate the increase in scale needed to cancel out the geometric scale. The calculated scale is then set directly for all the items that make up the header.

4.6 Automatically changing the semantic zoom level

The mechanism is implemented in `PositionLayout`. After the rearrangement of items the function `determineAutomaticSemanticZoomLevel` is invoked for each item. The return value of this function determines the semantic zoom level of the item. The function `determineAutomaticSemanticZoomLevel` gets the item for which the automatic semantic zoom level has to be determined and uses the function `estimateItemSizeFullDetail` to estimate whether there is enough empty space around the item for it to be shown in full detail.

4.7 Implemented Semantic Zoom Levels

4.7.1 Declaration abstraction

In Envision all projects, modules, classes and procedures are also declarations. It is a common base class of the items we want to change depending on their semantic scale. All the fully detailed visualizations of the declarations of interest have a common structure shown in Figure 4.2.

The goal was to create a visualization which abstracts away most details about an item. It makes sense to not vary the appearance of an item too much when abstracting it. When an item gets abstracted the user should be able to easily get a hold of what is happening and therefore should be able to relate between the abstracted and fully

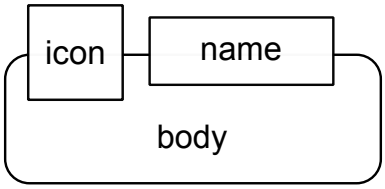

Figure 4.2: Basic structure of declarations.

detailed visualization of an item reasonably quickly. To accomplish this it was deemed important to keep the colour scheme as well as a major part of the header and outline the same. Each declaration item is equipped with an icon when shown in full detail and we decided to retain this visual cue during the abstraction as well.

When just omitting the visualization of the item's body entirely the icon and name of the abstracted visualization lacks visual connection. Usually the icon and name of a declaration seem to be attached to the background of the declaration body. Therefore we kept the background shape of a declaration. However, when just structuring the shape the same way as in the fully detailed visualization the looks of the abstraction of an item and the item having an empty body but shown in full detail are ambiguous. To resolve the ambiguity we moved the bottom of the shape up such that it still acts as binding for the icon and name but would be different from any looks attainable by using the fully detailed visualization. Figure 4.3 illustrates the result.

4.7.2 Public interface abstraction

This semantic zoom level implementation aims to show only the public interface of classes. Instead of completely abstracting away the whole body of a class it selectively only omits the non-public members of the class.

Most of the times when writing code the programmer needs to know the public interface of other classes to be able to use them in the current working process. The implementation details of those classes is not of importance at that point and displaying them essentially serves no purpose. By hiding the unnecessary details it is easier for users to find the information they are looking for.

Figure 4.4 illustrates the abstraction in comparison to the fully detailed visualization.

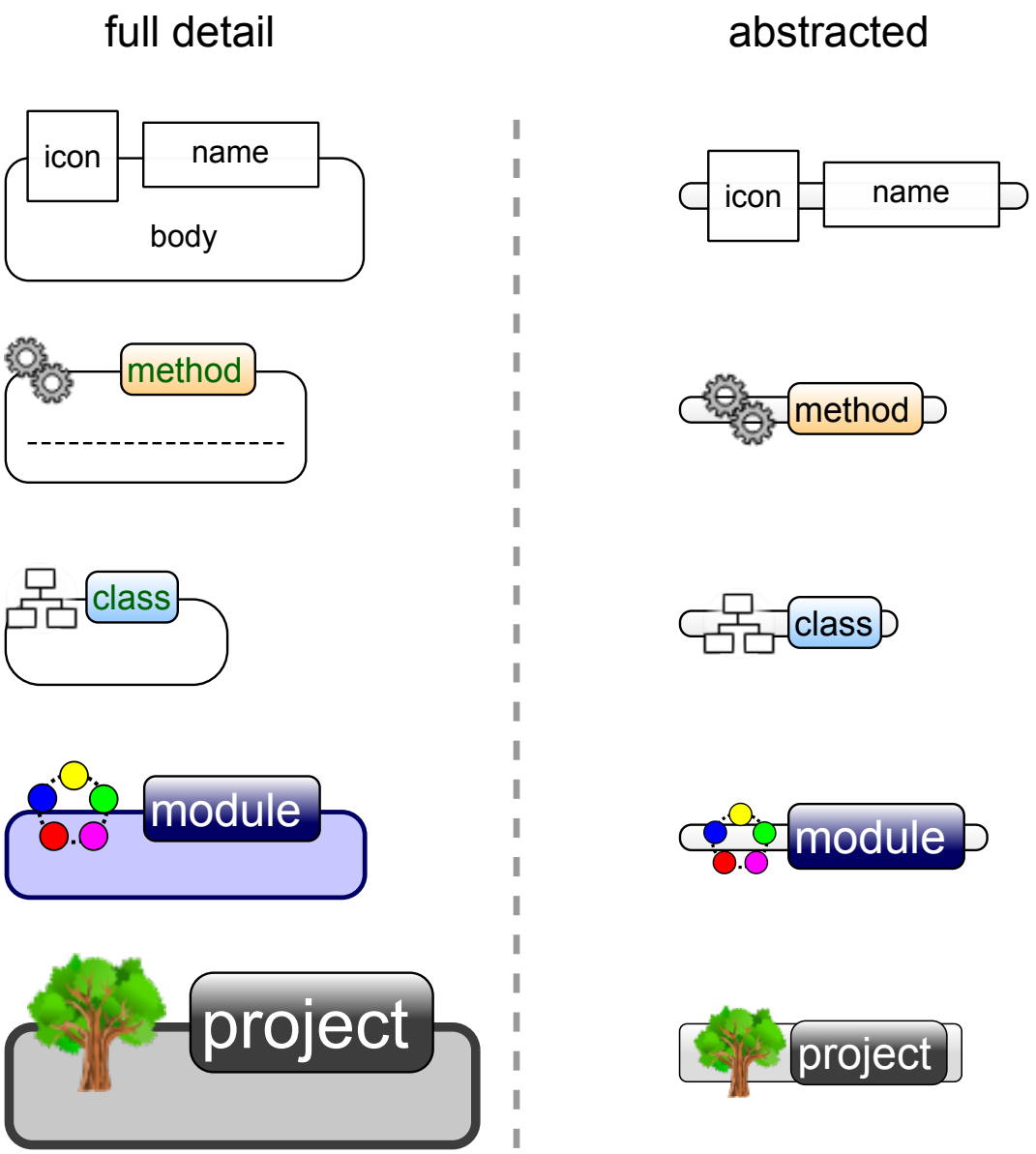

Figure 4.3: Concrete declaration abstraction.

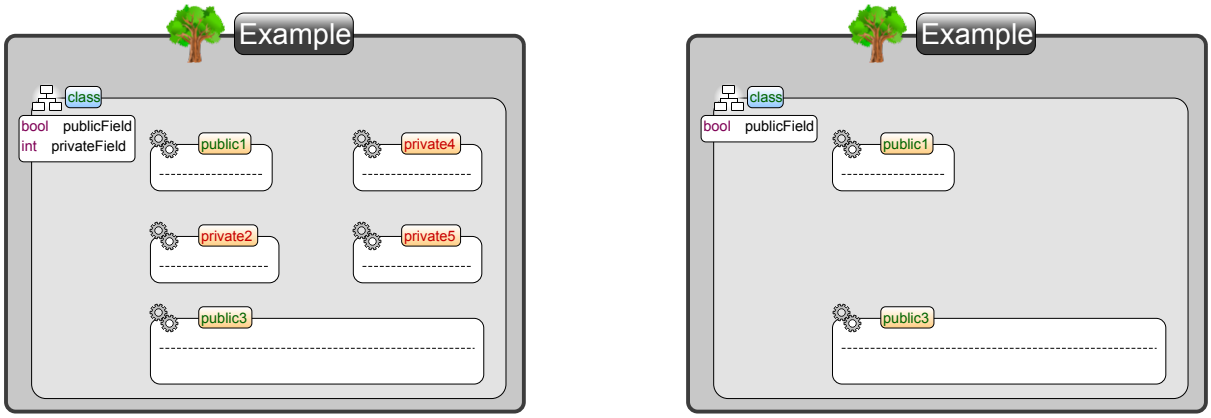

Figure 4.4: Showing only the public interface of a class.

5 Discussion and Future Work

5.1 Known issues

This section discusses known issues of the current implementation.

5.1.1 Scaling of items on geometric scale change

We decided that items should never be scaled to be larger than a perceived scale of 1. However, currently there is an issue with items being scaled over this value. This happens because the `PositionLayouts` are not updated in the necessary order. For the scaling to work correctly a `PositionLayout` higher up the structural hierarchy of a program has to be updated before a `PositionLayout` deeper down the hierarchy. If the order is not considered then it can happen that for example a `PositionLayout` sees its parent's scale to be a value A and arranges its items according to the parent scale A and one of its items reaches a perceived scale of 1. Afterwards a `PositionLayout` higher up the hierarchy gets updated and changes its scale to a value B that is greater than A. This change directly influences every item deeper down the hierarchy and therefore also increases the item formerly defined as having a perceived scale of 1. As a consequence that item's perceived scale must now be larger than 1 and thus violates the design decision of having perceived scale of at most 1.

The issue can be resolved by ensuring that the `PositionLayouts` are updated in the right order (top-down the structural hierarchy).

5.1.2 Delayed update of visualizations

The changes to the semantic zoom level of items performed by the semantic/geometric zoom mix mechanism currently do not cause the appearance of an item to be changed immediately. The appearance change is delayed until an item gets updated again at a later time.

5.1.3 Performance

Envision was designed specifically to be able to handle large projects containing a huge amount of items. This thesis introduced an arrangement algorithm that rearranges items in `PositionLayouts`. Specifically all projects, modules and classes have a `PositionLayout` in their body arranging their containing items. When testing the current implementation using a large project we discovered it to run very slow. Investigations of this performance issue showed that the source of the problem was the usage of calls to change an item's scale located in the underlying graphics framework. The performance problem was therefore not caused directly by the implementation itself. As work inside the underlying graphics framework was not considered to be inside the scope of this thesis we did not put any more efforts into resolving the issue.

5.1.4 Varying item size on different semantic zoom levels

In our implementation the size of the visualizations of an item can change when changing the semantic zoom level. In hindsight we should have probably decided to have an item's visualization size stay the same on every semantic zoom level. That design could have been less confusing for the user.

5.2 Possible extensions

In the following subsections we discuss ideas on how to get better results. We also mention possible features that were not implemented as part of this thesis.

5.2.1 Locking the semantic zoom level of an item

In the current implementation the semantic zoom level for an item can change because of the automatic semantic zoom level change mechanism on geometric zoom or because the user changed it manually. There is currently no difference between these two ways of changing the semantic zoom level of an item and therefore both ways can interfere with each other. In particular the automatic mechanism can change a manually set semantic zoom level for an item. However, a user's choice should always be more important than the decisions made by the automatic mechanism.

To resolve this one could introduce some sort of locking mechanism for the semantic zoom level of items. The user could then choose to lock the semantic zoom level of an item. If locked an item's semantic zoom level as well as the semantic zoom level of all of its parents should not be changed further by the automatic mechanism. The user should be able to unlock an item in a similar fashion resulting in the item getting modified by automatic changes again.

5.2.2 Estimation of an item's area

The performance of the semantic/geometric zoom mix feature can be improved by introducing a more elaborate method to estimate the size an item takes in full detail. Currently this method just returns a constant size for any input.

5.2.3 Arrangement of visualizations

Improving the arrangement algorithm

The current implementation of the arrangement algorithm only scales items and does not move them directly. However, while in many cases this simple functionality provides a useful result, there are scenarios where a more complex arrangement algorithm would be preferable. Especially if items in the same `PositionLayout` have vastly different sizes the algorithm will provide a poor solution.

As an example Figure 5.1 illustrates a project containing one large and two rather small modules. After abstraction the difference in size between the largest item and the two smaller items entails a larger distance between the large item and the small items than between the two small items. The large item and the upper small item have a lot more space between themselves than the two smaller items. As the algorithm uses the space around items to expand them this means that the large item and the upper small item can grow bigger. The result shows the large module as well as the upper smaller module scaled up very much while the other small item is barely visible any more.

While this behaviour may be desirable in some use cases we think that in the general case one would rather like the items to share the total space better resulting in a more equal scaling of items in the same `PositionLayout`. To achieve a more reasonable arrangement (in the sense of having better sharing the free space between items) one could consider to add a "pushing" step to the existing arrangement algorithm.

Instead of just stopping to try expanding an item in a certain direction in the arrangement algorithm after detecting a collision one could ask the items that occupy the required space to move away. The quality of this approach would heavily depend on the method used to decide where the items should be moved. Moving them away in the direction we want to expand to directly seems to be a straight forward idea but we argue that this could lead to suboptimal solutions in many cases. Additionally, it is noteworthy that the approach has to be absolutely deterministic. That is because when calculating the arrangement of an unchanged item multiple times the result must look the same.

Revisiting the previous example and assuming we have a good deterministic method to move items one might achieve a different arrangement like shown in Figure 5.2.

Assuming there was a way to know the precise size of items at full detail, we could do a better semantic zoom in a number of different ways. For example the idea presented in 3.5.2. At the very end of this project we used a workaround to make use of an item's size when shown in full detail to visualize it in a ways such that it would always have the same size even on different semantic zoom levels. The result is illustrated in Figure 5.3.

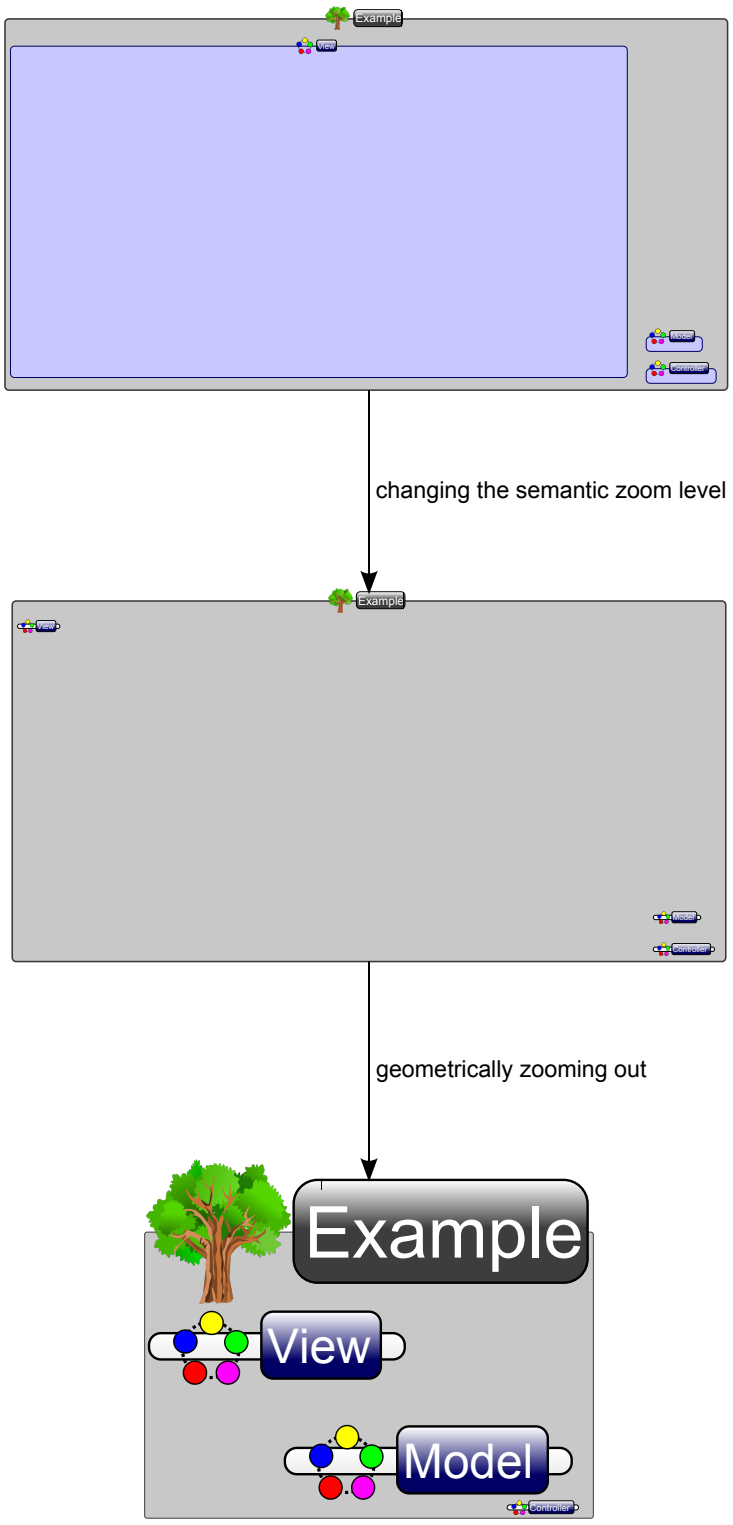

Figure 5.1: Example of the arrangement algorithm performing poorly.

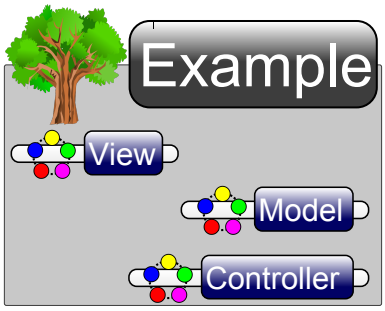

Figure 5.2: Potential result of an improved arrangement algorithm.

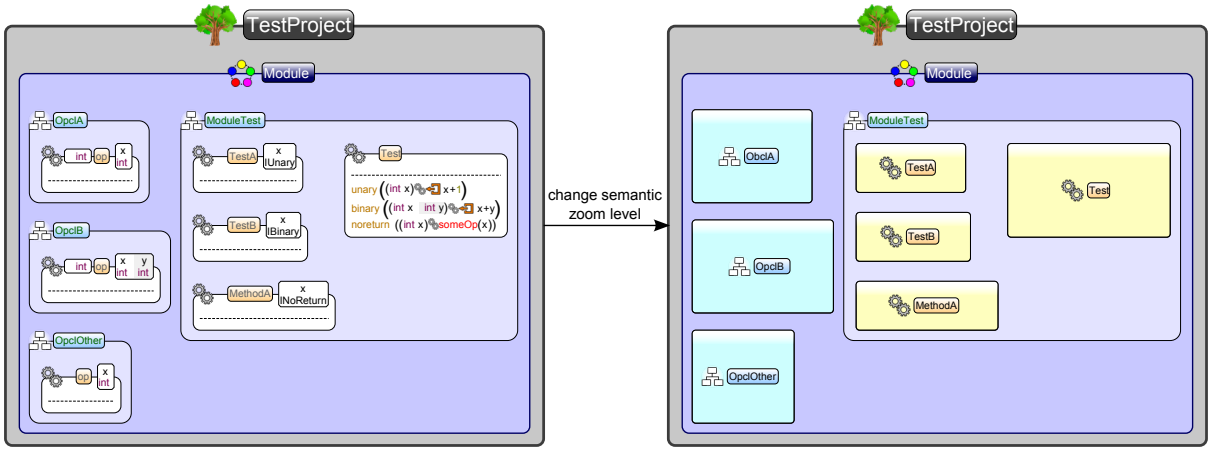

Figure 5.3: Items retain their position and size on different semantic zoom levels.

Share available space up the item hierarchy

When working with the current implementation we identified another issue. The space used by the contents of an item remains the same at all times. This means that even if not all space of a `PositionLayout` is actually needed to display all items with a perceived scale of 1 the size of the `PositionLayout` will not shrink. However, if the items could instead be compacted that would enable the parent item to use less space. When the parent item uses less space it can in turn be scaled more provided the space around it is still available. Having a method of compacting items would therefore distribute the available space to be used by the parent as well. This would potentially lead to a more equally distributed scaling over an item hierarchy. An example is shown in Figure 5.4.

5.2.4 Ambiguity in semantic zoom level hierarchy

Switching from a certain semantic zoom level to the next one in either direction can be a non trivial task. In general there exists no optimal solution and like many issues related to semantic zoom the most appropriate solution depends on the use case and user preference. Figure 5.5 illustrates the problem in an example using the semantic zoom levels implemented in the scope of this thesis.

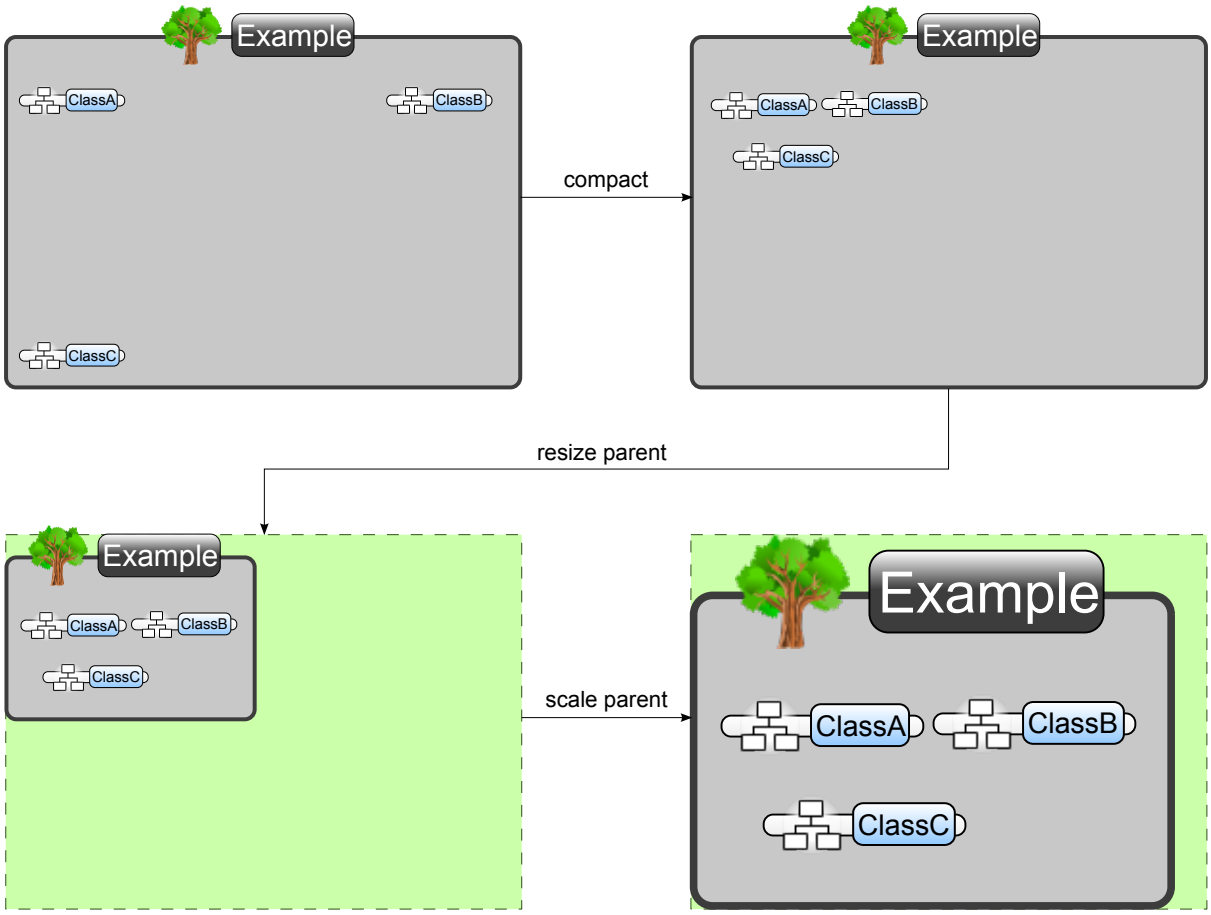

Figure 5.4: Compacting items to enable the rescaling of the parent item.

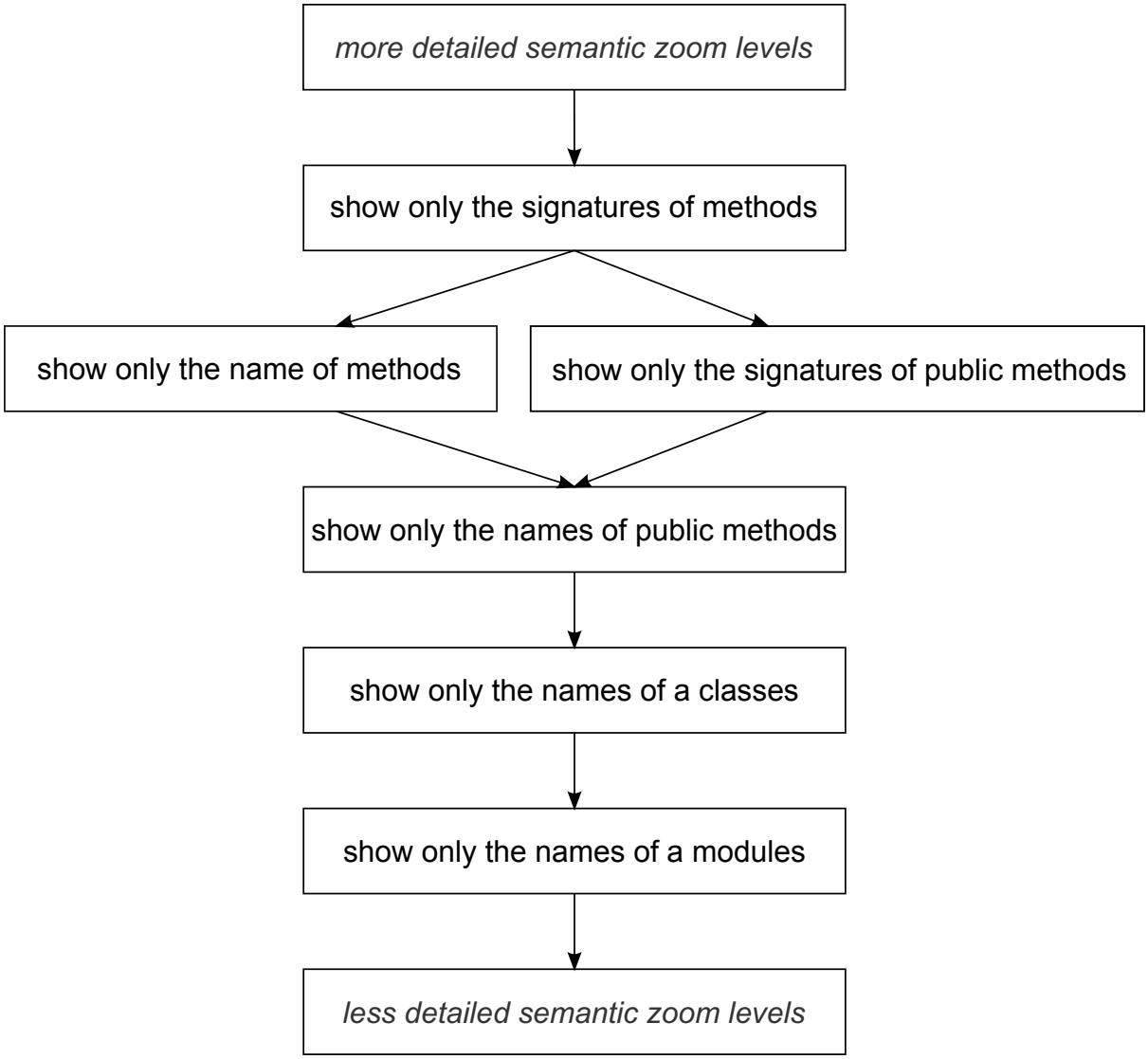

Figure 5.5: Example of ambiguity.

Here we can see that we cannot compare the semantic zoom level where we show only the names of methods of a class and the semantic zoom level where we show only the public members but with their signatures. Depending on the use case both semantic zoom levels can be of interest but they are incomparable in terms of how much information is shown.

One possibility to solve the problem is to let the user decide which way he would like to take at each junction in the illustrated graph.

5.2.5 Declarative creation of semantic zoom levels

Currently all semantic zoom levels are realised using fixed visualizations created from scratch to achieve the desired appearance. However, a visualization of a semantic zoom level is mostly just an abstraction of some original visualization. One could use this correlation to introduce a different way of creating semantic zoom levels. Instead of creating fixed visualizations for a semantic zoom level one could provide a mechanism that accepts a rule set to determine whether a certain item should be visualized. There could be an empty visualization (essentially displaying nothing at all) replaced for all items that should not appear in the abstracted visualization determined by the rule set. This would provide a fast and uniform way of achieving abstracted visualization for semantic zoom purposes.

6 Related work

6.1 Code Canvas

Code Canvas [2] was developed by Microsoft Research. It is a user interface for IDEs that replaces the standard multiple text windows by a single infinitely zoomable surface. Unlike Envision, which uses individual visualizations even on the statement level, Code Canvas uses text windows as the most fine grained visual elements. The tool supports semantic zoom when geometrically zooming to try and display as much useful information as it can on a given geometric zoom level.

6.2 PolyZoom

The idea behind PolyZoom [4] is to create multiple viewports forming a hierarchy of differently zoomed regions. The viewports are related using correlation graphics. Having this multi viewport hierarchy has the advantage of providing the user with more context information about a magnified area.

The arrangement algorithm approach in 3.5.2 was inspired by studying this paper and comparing the properties of the elements, namely a piece of a map, to custom visualizations commonly used in Envision.

6.3 Semantic Zooming for UML Diagrams

In their paper Semantic Zooming techniques for UML Diagrams [3] the authors are discussing possible usage of semantic zoom in UML diagrams as well as interaction techniques specifically adapted for interfaces using semantic zoom.

7 Conclusion

We introduced semantic zoom support for Envision. Semantic zoom is achieved by changing the visualization of items and rearranging them appropriately depending on the requested semantic zoom level. In the scope of this thesis we implemented a declaration abstraction and a public interface only visualization to modify the appearance of an item directly. An arrangement algorithm tries to make better use of the space typically created when an item's appearance uses less space when less information is shown. As a result the user can benefit from improved overview achieved by presenting more relevant information based on the requested level of detail.

The use of semantic zoom in Envision is not limited to the works in this thesis. Some ideas on how to improve or extend the existing features related to semantic zoom were introduced in the future works section. Furthermore, other works like Code Canvas [2] show how semantic zoom can be used together with common features of IDEs.

References

[1] Dimitar Asenov. Design and implementation of envision - a visual programming system. Master's thesis, ETH Zürich, 2011.

[2] Robert DeLine and Kael Rowan. Code canvas: Zooming towards better development environments. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 207–210, New York, NY, USA, 2010. ACM.

[3] Mathias Frisch, Raimund Dachselt, and Tobias Brückmann. Towards seamless semantic zooming techniques for uml diagrams. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 207–208, New York, NY, USA, 2008. ACM.

[4] Waqas Javed, Sohaib Ghani, and Niklas Elmqvist. Polyzoom: Multiscale and multifocus exploration in 2d visual spaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 287–296, New York, NY, USA, 2012. ACM.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Semantic Zoom Support for Envision

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Lüthi	Patrick

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 15.05.2014	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.